Micrium OS

- Software Manual
  - General Concepts
    - RTOS Description File
    - Platform Manager
    - Assertions
    - Memory Segments And LIB Heap
    - Stacks Initialization Methods
  - Understanding Micrium Os Internals And Optimizing Its Configuration
    - Internals Overview
    - Transitioning From Development To Production Code
  - Example Applications
    - Example Description File
    - Example Applications Documentation
  - Common
    - Common Overview
    - Integrating Common Into Your Project
    - Common Configuration
    - Common Programming Guide
    - Common Troubleshooting
  - CPU
    - CPU Overview
    - Integrating CPU Into Your Project
    - CPU Configuration
    - CPU Programming Guide
    - CPU Hardware Porting Guide
    - CPU Troubleshooting
  - IO
    - SD
    - SPI Master
  - Kernel
    - Kernel Overview
    - Integrating The Kernel Into Your Project
    - Kernel Configuration
    - Kernel Programming Guide
    - Kernel Troubleshooting
  - File System
    - File System Overview
    - File System Basic Concepts

- Integrating File System Into Your Project
- File System Example Applications
- File System Configuration
- File System Programming Guide
- File System Hardware Porting Guide
- File System Drivers
- File System Troubleshooting
- Network
  - Network Overview
  - Integrating Network Into Your Project
  - Network Core Example Applications
  - Network Core Configuration
  - Network Core Start Up
  - Network Core Programming Guide
  - Network Core Hardware Porting Guide
  - Network Core Troubleshooting
  - Application Modules
- USB Device
  - USB Device Overview
  - Integrating USB Device Into Your Project
  - USB Device Core Example Applications
  - USB Device Configuration
  - USB Device Programming Guide
  - USB Device Hardware Porting Guide
  - USB Device Classes
  - USB Device Troubleshooting
  - Migration Guide to Silicon Labs USB Device stack
- USB-Host
  - USB Host Overview
  - Integrating USB Host Into Your Project
  - USB Host Core Example Applications
  - USB Host Configuration
  - USB Host Programming Guide
  - USB Host Device Resource Needs
  - USB Host Class Drivers
  - USB Host Hardware Porting Guide
  - USB Host Troubleshooting
- CAN
  - CANopen
  - CAN Bus Hardware Porting Guide
- Hardware Porting Guide
  - Standard BSP Functions
  - Other Micrium OS Modules

API Reference

- Common API
  - Common Core API
  - Authentication API
  - Lib API
  - Logging API
  - RTOS Err API
  - Toolchain Abstraction API
  - Utilities API
  - Shell API
  - RTOS Task Cfg
- CPU API
  - CPU Core API
  - CPU Cache Management API
- Kernel API
  - Kernel Core API
  - Kernel Event Flag API
  - Kernel Message Queue API
  - Kernel Monitor API
  - Kernel Mutex API
  - Kernel Port Hooks API
  - Kernel Semaphore API
  - Kernel Statistic API
  - Kernel Task Management API
  - Kernel Time Management API
  - Kernel Timer API
- IO API
  - IO Core API
  - Serial API
  - SPI API
  - SD API
- File System API
  - File System Core API
  - File System Storage API
  - File System Posix API
- Network API
  - Network Core API
  - HTTP Client API
  - HTTP Server API
  - MQTT Client API
  - SMTP Client API
  - SNTP Client API
  - FTP Client API
  - TFTP Client API
  - TFTP Server API

- Telnet Server API
- IPerf API
- Mocana nanoSSL Certificate API
- USB Controller API
  - USB_CTRLR_HW_INFO_REG()
  - USB_CTRLR_HW_INFO_DEV_ONLY_REG()
  - USB_CTRLR_HW_INFO_HOST_ONLY_REG()
  - USB_CTRLR_HW_INFO_HOST_COMPANION_REG()
- USB Device API
  - USB Device Core API
  - USB Device CDC API
  - USB Device ACM API
  - USB Device CDC EEM API
  - USB Device HID API
  - USB Device MSC API
  - USB Device Vendor API
  - USBDev_API API
- USB Host API
  - USB Host Core API
  - USB Host PBHCI API
  - USB Host Shell Commands API
  - USB Host HUB API
  - USB Host AOAP API
  - USB Host CDC API
  - USB Host ACM API
  - USB Host HID API
  - USB Host MSC API
  - USB Host USB2SER API
- CAN API
  - CAN API Functions And Macros
  - CAN API Structures And Data Types
- Error Codes Description
  - Error Codes Description Table
- Tools
  - Segger Systemview
    - Including The Trace Recorder
    - Configuring The Trace Recorder
    - Initializing The Trace Recorder
    - Starting The Trace Recorder
- Appendices
  - Appendix A Internal Tasks
    - Micrium OS Internal Tasks
  - Appendix B Shell Commands Description
    - Help

Lsusb
File System Commands
Network Commands

General Concepts

General Concepts

This section provides details on methods that are used throughout the Micrium products. For more specific information on a given product, please consult that section. This section is more generic and is intended that way, to explain some concepts only once, not in every stack.

RTOS Description File

RTOS Description File

The rtos_description.h file is a configuration file that indicates to the whole Micrium OS which components are present. It allows components of Micrium OS to know what other components are enabled or disabled in the system. For example, knowing if the file system is present would allow the HTTP server or the MSC USB class to know that they can use it.

The rtos_description.h file is also used to indicate which CPU architecture is used (via RTOS_CPU_SEL) and which toolchain is used to build the project (via RTOS_TOOLCHAIN_SEL).The toolchain #defines has a special "automatic" values (RTOS_TOOLCHAIN_AUTO), where the system will attempt to automatically detect which to use. In some cases this automatic detection does not work, so you must specify that information yourself. Please refer to the section below to know which values can be specified for those two defines.

By default, the rtos_description.h file is located together with all other configuration files (xxxx_cfg.h). Please note that the xxxx_AVAIL defines do not need to have a value, they simply should be #define'd.

If these #defines do not correctly reflect which components are present in your project, this could lead to problems, such as modules not being initialized correctly, features not being available, or compilation errors.

The Listing - rtos description.h example file in the *RTOS Description File* page shows an rtos_description.h file for a kernel-only project on a Cortex-M4 MCU, such as Silicon Labs' Pearl Gecko series. Please note the use of both "automatic" values, which are automatically resolved by the system.

Listing - rtos_description.h example file

```
/***************************************************************************//**
 * @brief RTOS Description - Configuration Template File
 *******************************************************************************
 * # License
 * <b>Copyright 2018 Silicon Laboratories Inc. www.silabs.com</b>
 *******************************************************************************
 *
 * The licensor of this software is Silicon Laboratories Inc.  Your use of this
 * software is governed by the terms of Silicon Labs Master Software License
 * Agreement (MSLA) available at
 * www.silabs.com/about-us/legal/master-software-license-agreement.  This
 * software is distributed to you in Source Code format and is governed by the
 * sections of the MSLA applicable to Source Code.
 *
 ******************************************************************************/
/********************************************************************************************************
 ********************************************************************************************************
 *                                               MODULE
 ********************************************************************************************************
 *******************************************************************************************************/

#ifndef  _RTOS_DESCRIPTION_H_
#define  _RTOS_DESCRIPTION_H_

/********************************************************************************************************
 ********************************************************************************************************
 *                                            INCLUDE FILES
 ********************************************************************************************************
 *******************************************************************************************************/

#include  <rtos/common/include/rtos_opt_def.h>

/********************************************************************************************************
 ********************************************************************************************************
 *                                      ENVIRONMENT DESCRIPTION
 ********************************************************************************************************
 *******************************************************************************************************/

#define  RTOS_CPU_SEL                                       RTOS_CPU_SEL_SILABS_GECKO_AUTO

#define  RTOS_TOOLCHAIN_SEL                                 RTOS_TOOLCHAIN_AUTO

/********************************************************************************************************
 ********************************************************************************************************
 *                                      RTOS MODULES DESCRIPTION
 ********************************************************************************************************
 *******************************************************************************************************/

#define  RTOS_MODULE_KERNEL_AVAIL
#define  RTOS_MODULE_COMMON_AVAIL

/********************************************************************************************************
 ********************************************************************************************************
 *                                             MODULE END
 ********************************************************************************************************
 *******************************************************************************************************/

#endif  // End of rtos_description.h module include.
```

List of Possible Values

All of the possible values for the three selection defines are listed below. They can all be found in <micrium_os/common/include/rtos_opt_def.h>.

CPU Selection Define (RTOS_CPU_SEL)

- RTOS_CPU_SEL_SILABS_GECKO_AUTO
- RTOS_CPU_SEL_ARM_CORTEX_M0P
- RTOS_CPU_SEL_ARM_CORTEX_M3
- RTOS_CPU_SEL_ARM_CORTEX_M4
- RTOS_CPU_SEL_ARM_CORTEX_M33
- RTOS_CPU_SEL_EMPTY

Toolchain Selection Define (RTOS_TOOLCHAIN_SEL)

- RTOS_TOOLCHAIN_ARMCC
- RTOS_TOOLCHAIN_GNU
- RTOS_TOOLCHAIN_IAR
- RTOS_TOOLCHAIN_AUTO

Platform Manager

Platform Manager

Hardware capabilities are registered at run-time by the BSP to indicate to Micrium OS what features are available on the board. This information is used by the Micrium OS stacks to configure as much as possible automatically without requiring you to change anything. File System media, Network interfaces, and USB controllers are all referenced the same way, with a simple string, during initialization and configuration.

Registration

To be able to use an ID to add some hardware peripheral, this ID must first be registered to the platform builder. The preferred way to do this is by using the register macros. In most BSP examples and templates, it is possible to have examples of the detailed usage of these macros. The Hardware Porting Guide page also details how these macros should be used, in order to register your board's hardware capabilities to Micrium OS. The Listing - Register Macros Example in the *Platform Manager* page also provides a small example showing how to register a given hardware peripheral to the platform manager.

These calls can either be made in the BSP, typically in the BSP_OS_Init() function or in the application, after the Common module has been initialized (by calling Common_Init()), but before referencing the string ID.

Listing - Register Macros Example

```
void  BSP_OS_Init (void)
{
                                        /* Associate MACNet ether interface to "eth0".        */
    NET_CTRLR_ETHER_REG("eth0", &BSP_NetEther_MACNet_HwInfo);
                                        /* Associate Device and Host USB controller to "usb1". */
    USB_CTRLR_HW_INFO_REG("usb1", &BSP_USBD_USBHS_HwInfo, &BSP_USBH_USBHS_HwInfo);
                                        /* Associate SPI bus to "spi0".                     */
    IO_SERIAL_CTRLR_REG("spi0", &BSP_Serial_DSPI_Bus2_DrvInfo);
                                        /* Associate Wi-Fi interface to "wifi0".             */
    NET_CTRLR_WIFI_SPI_REG("wifi0", "spi0", &NetWiFi_Info_QCA400x_AddOn, &BSP_NetWiFi_K70F120M_QCA400x, 0u);
}
```

Use

After registration has been successfully completed, an ID can then be used to do certain operations on hardware capabilities registered.

The Listing - Platform Manager Use Example in the *Platform Manager* page shows how this can be done, typically in the start task.

Listing - Platform Manager Use Example

```
static void Ex_MainStartTask (void *p_arg)
{
    RTOS_ERR    err;
    NET_IF_NBR  if_nbr;

    Common_Init(&err);
    if (err.Code != RTOS_ERR_NONE) {
        /* Error handling. */
    }

    BSP_OS_Init();

    if_nbr = NetIF_Ether_Add("eth0",
                             Ex_NetEther_CfgPtr,
                             DEF_NULL,
                            &err);
    if (err.Code != RTOS_ERR_NONE) {
        /* Error handling. */
    }

    (void)USBH_HC_Add("usb1",
                      DEF_NULL,
                     &err);
    if (err.Code != RTOS_ERR_NONE) {
        /* Error handling. */
    }

    (void)SPI_BusAdd("spi0",
                    &err);
    if (err.Code != RTOS_ERR_NONE) {
        /* Error handling. */
    }
}
```

Some configuration function (in addition to the 'Add()' functions) will also take the string ID as parameter. But, in order to make often-used operations more efficient, the stacks also use some kind of handle, number, index or reference to do those functions. Therefore, each stack has a <Module>_<Peripheral><ReferenceType>GetFromName() function that takes the string ID as an argument and returns the correct handle, index or reference for that particular string ID. For example, to obtain the interface number for an Ethernet or WiFi interface, you need to call NetIF_NbrGetFromName(), for USB Host, you should call USBH_HC_HandleGetFromName(), etc. Please refer to each of the stacks's documentation or examples for more details on what these functions are and how to use them.

Assertions

Assertions

- Debug Asserts
- Critical Asserts

An assert is a check that should always be true when the program is executing correctly. Asserts are used to prevent the program from continuing to execute if some kind of problem is detected. Micrium products use asserts in several places for these reasons, and also to check the validity of several arguments passed to API functions before they are used. For information about how to use the asserts in an application context, or how to configure them, see the Micrium OS Asserts Programming Guide or RTOS Compile-Time Configuration pages. The asserts come in two distinct types, as listed below.

Debug Asserts

The debug asserts (RTOS_ASSERT_DBG or APP_RTOS_ASSERT_DBG) are used in cases where the code cannot recover from what is currently happening; most of the time, this requires changes in the application's code. These debug asserts should help during the development phase of an application. For example, the debug asserts would catch invalid parameters passed to a function, or they could detect that a given function cannot be called when the system is in a particular state or with a given configuration. The user can define a macro or function to be called in case an assert fails, as explained on the RTOS Compile-Time Configuration page.

It is possible to disable the debug asserts by setting RTOS_CFG_ASSERT_DBG_ARG_CHK_EXT_MASK to 0u. This will speed code execution, and save space in memory or in code footprint (see RTOS Compile-Time Configuration).

Critical Asserts

Critical asserts (RTOS_ASSERT_CRITICAL and APP_RTOS_ASSERT_CRITICAL) are used in cases where the code reaches a point that should normally never be reached (for example, the default case of a switch statement for which every possible value is handled). If a critical assert is triggered, it means that the system is in an unknown state and that code execution should be halted before executing erroneous operations.

This kind of problem is often due to memory corruption (for example if a task's stack overflows and corrupts some other memory location). The user can define a macro or function to be called in case an assert fails (see RTOS Compile-Time Configuration).

The 'critical' asserts cannot be disabled because they represent a failsafe. Since they are more rarely used than their debug counterpart, they should not slow the execution of the application or take too much space.

Memory Segments And LIB Heap

Memory Segments and LIB Heap

- Memory Segments
  - Memory Segment Allocation
  - Dynamic Memory Pools
- LIB Heap

Memory Segments

A memory segment is a type construct that is used to describe a memory region having particular properties that will be used to allocate data at run-time. It can be as simple as a table with no constraints, or as complex as DMA-accessible or cache-capable regions, with special considerations for cache line size, padding, and alignment.

Several functions can be used to allocate data from memory segments. These functions are all part of the LIB Mem sub-module of Common and include functions such as Mem_SegAlloc(), Mem_DynPoolCreate(), Mem_DynPoolBlkGet(), and others. Each of these functions will allocate data using specific parameters, be they parameters passed at segment creation, pool creation, or directly at the function call. Please refer to the Memory Allocation Guide page for more information.

Memory Segment Allocation

Memory segment allocation calls include Mem_SegAlloc(), Mem_SegAllocExt() and Mem_SegAllocHW(). Each of these functions works in a similar fashion where they allocate a chunk of memory from a particular memory segment that can then be used to store any kind of data. The memory allocated this way is not allowed to be freed, since this would lead to fragmentation problems, and would require to keep a list of blocks available for a re-allocation, which would increase the amount of memory required by LIB.

The three flavors of the memory segment allocation functions can be used in different situations. Mem_SegAlloc() is the simplest one. Mem_SegAllocExt(), by comparison, allows you to specify more parameters, such as the alignment of the allocated block, or a pointer to the number of bytes missing for an allocation to succeed. Mem_SegAllocHW() is a bit different from the others two because it takes into account the padding alignment specified at segment creation when allocating blocks of data. This is to make sure blocks are not located on the same cache lines, to facilitate flush, and invalidate operations when the memory allocated this way is accessed via DMA.

Several usage scenarios, and more details on memory segment allocations, are provided in the Memory Segment Allocations page. These scenarios will indicate where in a memory segment the allocations are located depending on various parameters.

Dynamic Memory Pools

Dynamic memory pools are constructs that are used to allocate and free blocks of the same size. These blocks are allocated from a memory segment. At its creation, the pool can already have an initial number of blocks allocated, and can have either a maximum number of blocks allocatable or no maximum at all, limited only by available space in the memory segment. For example, this allows pools to have a fixed amount of blocks all allocated at creation, or no blocks allocated at creation and an allocation only when needed.

There are three functions that can create a dynamic memory pool: Mem_DynPoolCreate(), Mem_DynPoolCreatePersistent() and Mem_DynPoolCreateHW(). As for the memory segment allocation, the Mem_DynPoolCreateHW() flavor takes into account the padding alignment specified at the creation of the memory segment to make sure no two blocks are on the same cache line.

One of the differences between Mem_DynPoolCreate() and Mem_DynPoolCreatePersistent() is that blocks allocated by Mem_DynPoolCreate() are not guaranteed to keep their data when freed and reallocated, while those allocated by Mem_DynPoolCreatePersistent() do keep their data. Also, blocks allocated by Mem_DynPoolCreatePersistent() can have a

callback associated with them that is called once for every newly-allocated block, allowing to data to be set at that moment.

Several usage scenarios and more details on dynamic memory pools are provided in the Dynamic Memory Pools page. These scenarios will indicate where in a memory segment the allocations are located depending on various parameters.

LIB Heap

The LIB Heap is a particular memory segment that can be provided by the LIB Mem sub-module of Common. It is a normal memory segment, which acts exactly the same way as the other memory segments detailed above. Its size, padding alignment, and base address can all be configured in common_cfg.h, as detailed in the LIB Heap configuration page. It will be used by default by various stacks and products, or if the caller passes DEF_NULL when a memory segment pointer is required as a parameter.

Dynamic Memory Pools

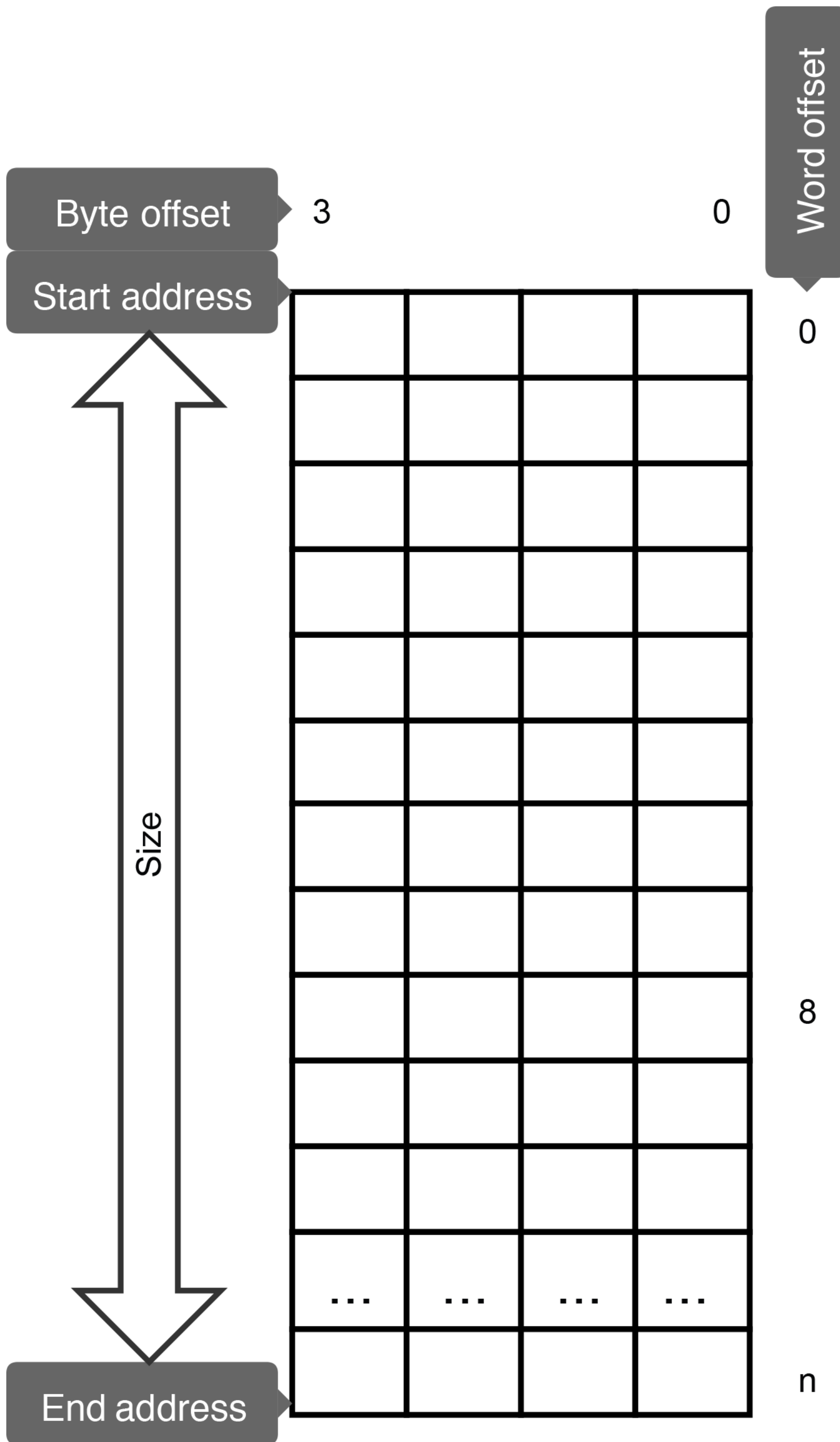
This page provided in-depth information about the various types of dynamic memory pools that you can use to allocate and free fixed-size blocks from a memory segment.

- Usage Scenarios
- Dynamic Memory Pools
  - Persistent Dynamic Memory Pools
- Hardware Dynamic Memory Pools
- Multiple Dynamic Memory Pools Calls
- Mixed Usage
- Final Words

Usage Scenarios

The Figure - Empty MEM_SEG representation in the *Dynamic Memory Pools* page represents a new, empty memory segment. We will assume a 32-bit architecture for all of the following examples. That means that the size of the CPU_ALIGN type is 32, which is used by default in some allocation cases. In this figure, the bytes increase from right to left and the memory addresses grow from top to bottom.

Figure - Empty MEM_SEG representation



*Assuming 32-bit word architecture, meaning 32-bits for CPU_ALIGN type.

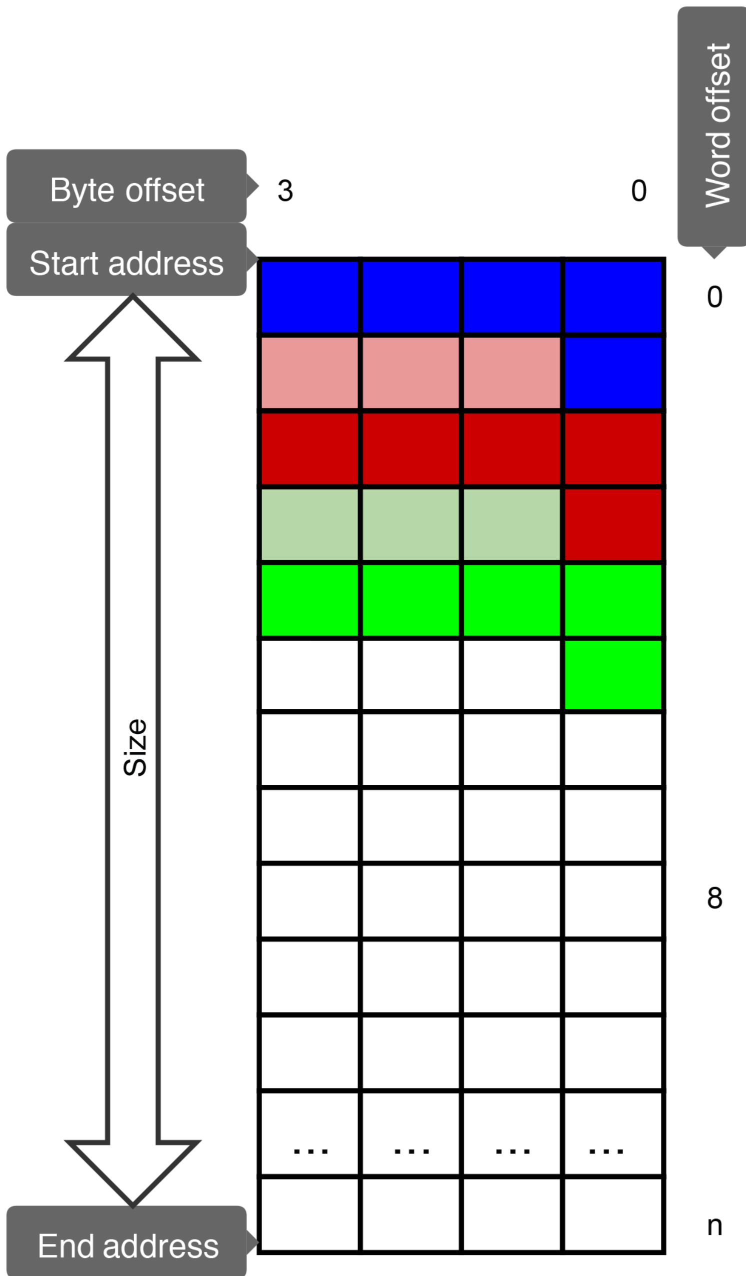
Dynamic Memory Pools

The Figure - Mem DynPoolCreate() and Mem DynPoolBlkGet() in the *Dynamic Memory Pools* page shows a call to Mem_DynPoolCreate() (or Mem_DynPoolCreatePersistent(), which works the same way) with an initial block value of 1, meaning it will allocate one single block during pool creation, followed by two calls to Mem_DynPoolBlkGet(), on that same pool. The block size is set to 5 bytes and the block alignment to 4 bytes. The initial call to Mem_DynPoolCreate() allocates

the first 5-byte block, in blue. The following Mem_DynPoolBlkGet() call will need to sacrifice three bytes to align the second block on an 8-byte boundary, and the same can be said for the third call.

This means that, in this particular case, with an alignment of 8 and no interleaved allocation calls (other pools allocating blocks or Mem_SegAlloc...() calls) on that segment, both 8-byte blocks and 5-byte blocks use up the same amount of space in the segment. Knowing that it is possible to use 3 more bytes per block without actually using more memory, it may be possible to optimize the code to make use of these three additional bytes. Or, knowing that the alignment of each block is 4, it might be possible to scale down each block to 4 bytes instead of 5, saving 3 extra bytes for every block.

Figure - Mem_DynPoolCreate() and Mem_DynPoolBlkGet()



*Assuming 32-bit word architecture.

Color	Description
Blue	Mem_DynPoolCreate(): blk_size= 5 bytes, blk_align= 4 bytes, blk_qty_init= 1
Red	Mem_DynPoolBlkGet() on that same pool.
Pink	Space lost due to Mem_DynPoolCreate() blk_align of 4 bytes.
Green	Mem_DynPoolBlkGet() on that same pool.
Light green	Space lost due to Mem_DynPoolCreate() blk_align of 4 bytes.

Listing - Dynamic memory pools code snippet

```
#define  MY_MEM_SEG_DATA_SIZE  64u

CPU_INT08U    MyMemSegData[MY_MEM_SEG_DATA_SIZE];
MEM_SEG       MyMemSeg;

MEM_DYN_POOL  MyMemDynPool;

/* ... */

static  void  MyCreationFnct (void)
{
    RTOS_ERR  err;

    Mem_SegCreate("My mem seg",
                  &MyMemSeg,
                  (CPU_ADDR)&MyMemSegData[0u],
                  MY_MEM_SEG_DATA_SIZE,
                  LIB_MEM_PADDING_ALIGN_NONE,
                  &err);
    if (RTOS_ERR_CODE_GET(err) != RTOS_ERR_NONE) {
        /* Handle error. */
    }

    Mem_DynPoolCreate("My mem dyn pool",
                      &MyMemDynPool,
                      &MyMemSeg,
                      5u,
                      4u,
                      1u,
                      LIB_MEM_BLK_QTY_UNLIMITED,
                      &err);
    if (RTOS_ERR_CODE_GET(err) != RTOS_ERR_NONE) {
        /* Handle error. */
    }
}

/* ... */

static  void  MyAlloctionFnct (void)
{
    CPU_INT08U  *p_blk_1;
    CPU_INT08U  *p_blk_2;
    RTOS_ERR     err;

    p_blk_1 = (CPU_INT08U *)Mem_DynPoolBlkGet(&MyMemDynPool, &err);
    if (RTOS_ERR_CODE_GET(err) != RTOS_ERR_NONE) {
        /* Handle error. */
    }

    p_blk_2 = (CPU_INT08U *)Mem_DynPoolBlkGet(&MyMemDynPool, &err);
    if (RTOS_ERR_CODE_GET(err) != RTOS_ERR_NONE) {
        /* Handle error. */
    }
}
```

Persistent Dynamic Memory Pools

Persistent pools, which can be created by calling Mem_DynPoolCreatePersistent(), are very similar to regular dynamic memory pools but for a few notable exceptions.

First, they require an extra pointer to be allocated *for each block*, meaning that if the caller requires a 14-byte block for a persistent pool, the actual amount of memory allocated will be 18 bytes (on a 32-bit architecture). But this requirement provides a potentially useful advantage: once allocated, the block's content will never be altered, even while it is freed.

With regular pools, between a Mem_DynPoolBlkFree() and a subsequent Mem_DynPoolBlkGet(), a block's content is not guaranteed to remain the same, since part of it is used for maintaining the "free blocks" list.

The third and final difference with persistent pools is that a callback function and an argument can be passed during the pool's creation. After this, for every *new* block created (not ones freed and then re-obtained), the callback will be executed. This is particularly useful if every block needs to hold some persistent data such as a kernel object, more allocated data, or anything difficult or impossible to free. With persistent pools, you need only allocate or create whatever data that needs to be persistent in the callback, and the block will be able to hold it for as long as required, even if the block is freed and re-obtained at some point.

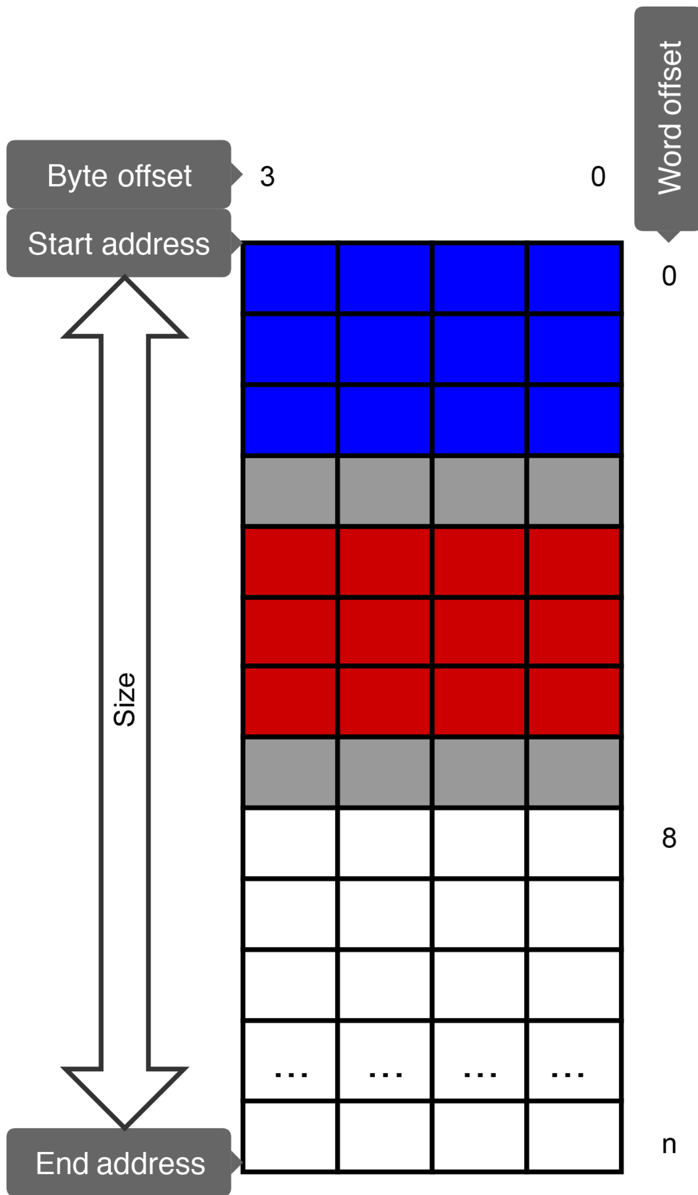
Hardware Dynamic Memory Pools

The Figure - Mem_DynPoolCreateHW() and Mem_DynPoolBlkGet() calls in the *Dynamic Memory Pools* page shows the result of two functions calls: First, a call to Mem_DynPoolCreateHW(), which takes into account the segment's padding alignment; and second, a call to Mem_DynPoolBlkGet(), which allocates another block from that same pool. The segment's padding alignment (as specified during segment creation) is set to 16 bytes, while the pool has a block size of 12, a block alignment of 4, and an initial quantity of 1.

The first block allocated uses 12 bytes, and pads for the 16-byte alignment of the segment. The subsequent call to Mem_DynPoolBlkGet() does the same thing, resulting in a total of 32 bytes used in memory for two 12-byte blocks. We can see that, since the segment's padding of 16 is an even multiple of the block alignment of 4, the next block naturally falls on a 4-byte boundary requiring no adjustment. This might not have been the case if another allocation had been made between those two calls, which is why it is important to provide the block alignment nonetheless.

Knowing how these blocks are allocated could help to optimize the code by giving each block 4 extra bytes that it always uses in order to conform to the segment's padding alignment constraint, which is a bit like the case with Mem_DynPoolCreate() mentioned above. But, contrary to the previous example, reducing the block size (if possible) would not be beneficial, since the padding alignment would still require a 16-byte boundary, meaning that the minimum size of an allocation on that segment is 16 bytes, not less.

Figure - Mem_DynPoolCreateHW() and Mem_DynPoolBlkGet() calls



*Assuming 32-bit word architecture. Segment's padding at creation is set to 16 bytes.

	Mem_DynPoolCreateHW(): blk_size= 12 bytes, blk_align= 4 bytes, blk_qty_init= 1
	Space lost due to segment's padding of 16 bytes.
	Mem_DynPoolBlkGet() on that same pool.

Listing - Hardware dynamic memory pools code snippet

```
#define  MY_MEM_SEG_DATA_SIZE  64u

CPU_INT08U    MyMemSegData[MY_MEM_SEG_DATA_SIZE];
MEM_SEG       MyMemSeg;

MEM_DYN_POOL  MyMemDynPool;

/* ... */

static  void  MyCreationFnct (void)
{
    RTOS_ERR  err;

    Mem_SegCreate("My mem seg",
                  &MyMemSeg,
                  (CPU_ADDR)&MyMemSegData[0u],
                  MY_MEM_SEG_DATA_SIZE,
                  16u,
                  &err);
    if (RTOS_ERR_CODE_GET(err) != RTOS_ERR_NONE) {
        /* Handle error. */
    }

    Mem_DynPoolCreateHW("My mem dyn pool",
                        &MyMemDynPool,
                        &MyMemSeg,
                        12u,
                        4u,
                        1u,
                        LIB_MEM_BLK_QTY_UNLIMITED,
                        &err);
    if (RTOS_ERR_CODE_GET(err) != RTOS_ERR_NONE) {
        /* Handle error. */
    }
}

/* ... */

static  void  MyAlloctionFnct (void)
{
    CPU_INT08U  *p_blk_1;
    RTOS_ERR     err;

    p_blk_1 = (CPU_INT08U *)Mem_DynPoolBlkGet(&MyMemDynPool, &err);
    if (RTOS_ERR_CODE_GET(err) != RTOS_ERR_NONE) {
        /* Handle error. */
    }
}
```

Multiple Dynamic Memory Pools Calls

The Figure - Several Mem_DynPoolCreate() and Mem_DynPoolBlkGet() in the *Dynamic Memory Pools* page shows two calls to Mem_DynPoolCreate(), followed by a series of calls to Mem_DynPoolBlkGet() on each of the two created pools. Pool A has a block size and alignment requirement of 4 bytes and an initial quantity of 1. Pool B has a block size of 6 and a block alignment requirement of 8, with two initial blocks allocated. The figure shows that the initial block for pool A is allocated first in the segment, followed by the first two blocks of pool B. Then pool A allocates another block, followed by another in pool B, and a final one in pool A.

The result is that 44 bytes of memory have been used even though the various callers asked for only 30 bytes. It is easy to see that if the pool A allocations been grouped together (or at least if the second block had been made at pool creation), no memory would have been wasted, as the block size is 4 with an alignment of 4. Every block allocated from pool B still takes at least 2 bytes more than required, since every other allocation is aligned minimally on 4 bytes and the pool B blocks have a size of 6.

If all the blocks from pool A had been grouped, followed by allocation of blocks from pool B, the amount of memory used could have been reduced to 38–42 bytes, depending on the subsequent allocation alignment. If it were also possible to allocate the second pool A block at creation, and leave the others identical, the amount of memory used would have been reduced to 36 bytes.

This example shows how it is possible to optimize some cases, when block size and alignment are known beforehand. It is not possible to know exactly in every case how many blocks are needed to be used, but by using an initial value near (but not above) the estimated amount, it is possible to reduce the amount of memory lost due to padding or alignment constraints.

Figure - Several Mem_DynPoolCreate() and Mem_DynPoolBlkGet()

Listing - Several dynamic memory pools code snippet

```
#define  MY_MEM_SEG_DATA_SIZE  64u

CPU_INT08U    MyMemSegData[MY_MEM_SEG_DATA_SIZE];
MEM_SEG       MyMemSeg;

MEM_DYN_POOL  MyMemDynPoolA;
MEM_DYN_POOL  MyMemDynPoolB;

/* ... */

static  void  MyCreationFnct (void)
{
    RTOS_ERR  err;

    Mem_SegCreate("My mem seg",
                  &MyMemSeg,
                  (CPU_ADDR)&MyMemSegData[0u],
                  MY_MEM_SEG_DATA_SIZE,
                  LIB_MEM_PADDING_ALIGN_NONE,
                  &err);
    if (RTOS_ERR_CODE_GET(err) != RTOS_ERR_NONE) {
        /* Handle error. */
    }

    Mem_DynPoolCreate("My mem dyn pool A",
                      &MyMemDynPoolA,
                      &MyMemSeg,
                      4u,
                      4u,
                      1u,
                      LIB_MEM_BLK_QTY_UNLIMITED,
                      &err);
    if (RTOS_ERR_CODE_GET(err) != RTOS_ERR_NONE) {
        /* Handle error. */
    }

    Mem_DynPoolCreate("My mem dyn pool B",
                      &MyMemDynPoolB,
                      &MyMemSeg,
                      6u,
                      8u,
                      2u,
                      LIB_MEM_BLK_QTY_UNLIMITED,
                      &err);
    if (RTOS_ERR_CODE_GET(err) != RTOS_ERR_NONE) {
        /* Handle error. */
    }
}

/* ... */

static  void  MyAlloctionFnct (void)
{
    CPU_INT08U  *p_blk_1;
    CPU_INT08U  *p_blk_2;
    CPU_INT08U  *p_blk_3;
    RTOS_ERR     err;

    p_blk_1 = (CPU_INT08U *)Mem_DynPoolBlkGet(&MyMemDynPoolA, &err);
    if (RTOS_ERR_CODE_GET(err) != RTOS_ERR_NONE) {
        /* Handle error. */
    }

    p_blk_2 = (CPU_INT08U *)Mem_DynPoolBlkGet(&MyMemDynPoolB, &err);
    if (RTOS_ERR_CODE_GET(err) != RTOS_ERR_NONE) {
        /* Handle error. */
    }
```

```
p_blk_3 = (CPU_INT08U *)Mem_DynPoolBlkGet(&MyMemDynPoolA,&err);if(RTOS_ERR_CODE_GET(err)!= RTOS_ERR_NONE){/* Handle error. */}}
```

Mixed Usage

The Figure - Mixed usage in the *Dynamic Memory Pools* page shows the same example as the one above featuring the multiple pool calls. But in this case, a call to Mem_SegAlloc() is executed after the second call to Mem_DynPoolCreate(). It shows how a segment can be used to allocate different kinds of data, and how they can all be interleaved together in a single segment.

Of course, allocating several forms of data with varying padding and alignment requirements can lead to more memory space lost due to conflicting alignment constraints. Also, this figure shows how some blocks can end up spatially disconnected. For example, the third block from pool A could end up being allocated far from the first two, which would make it less likely that they could all be cached together, leading to code that could still be optimized.

Figure - Mixed usage

Listing - Mixed usage code snippet

```
#define  MY_MEM_SEG_DATA_SIZE  64u

CPU_INT08U    MyMemSegData[MY_MEM_SEG_DATA_SIZE];
MEM_SEG       MyMemSeg;

MEM_DYN_POOL  MyMemDynPoolA;
MEM_DYN_POOL  MyMemDynPoolB;

/* ... */

static  void  MyCreationFnct (void)
{
    RTOS_ERR  err;

    Mem_SegCreate("My mem seg",
                  &MyMemSeg,
                  (CPU_ADDR)&MyMemSegData[0u],
                  MY_MEM_SEG_DATA_SIZE,
                  LIB_MEM_PADDING_ALIGN_NONE,
                  &err);
    if (RTOS_ERR_CODE_GET(err) != RTOS_ERR_NONE) {
        /* Handle error. */
    }

    Mem_DynPoolCreate("My mem dyn pool A",
                      &MyMemDynPoolA,
                      &MyMemSeg,
                      4u,
                      4u,
                      2u,
                      LIB_MEM_BLK_QTY_UNLIMITED,
                      &err);
    if (RTOS_ERR_CODE_GET(err) != RTOS_ERR_NONE) {
        /* Handle error. */
    }

    Mem_DynPoolCreate("My mem dyn pool B",
                      &MyMemDynPoolB,
                      &MyMemSeg,
                      6u,
                      8u,
                      2u,
                      LIB_MEM_BLK_QTY_UNLIMITED,
                      &err);
    if (RTOS_ERR_CODE_GET(err) != RTOS_ERR_NONE) {
        /* Handle error. */
    }
}

/* ... */

static  void  MyAlloctionFnct (void)
{
    CPU_INT08U  *p_data_1;
    CPU_INT08U  *p_blk_1;
    RTOS_ERR     err;

    p_data_1 = (CPU_INT08U *)Mem_SegAllocExt("allocation 1",
                                             &MyMemSeg,
                                             11u,
                                             4u,
                                             DEF_NULL,
                                             &err);
    if (RTOS_ERR_CODE_GET(err) != RTOS_ERR_NONE) {
        /* Handle error. */
    }
```

```
p_blk_1 = (CPU_INT08U *)Mem_DynPoolBlkGet(&MyMemDynPoolA,&err);if(RTOS_ERR_CODE_GET(err)!= RTOS_ERR_NONE){/* Handle error. */}}
```

Final Words

The various scenarios explained above provide a detailed view of the internals of allocating data using memory segments, as well as several hints about how it is possible to optimize memory usage if certain conditions are met.

Of course, run-time allocation is made during run-time simply because the amount and type of data needed is not known at compile-time. However, by planning the order of the memory segment's allocations, and adapting the size of the allocation when possible, or carefully choosing the properties of the memory segments themselves, it may be possible to optimize the amount of memory used. Or you can at least reduce the amount of memory lost, and convert it to memory that could be used for other purposes at no additional cost.

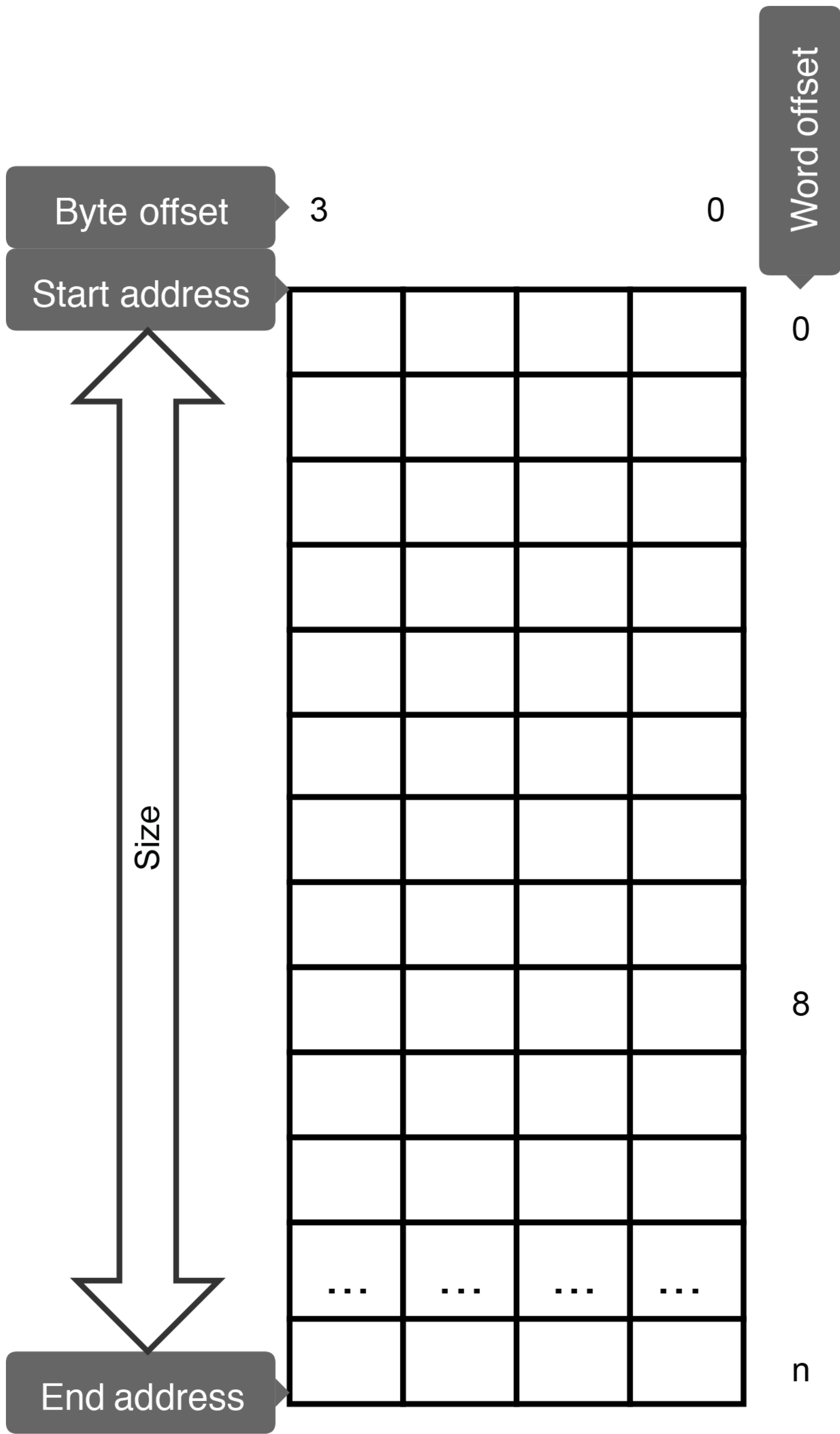
Memory Segment Allocations

This page provides in-depth information about the various methods used to allocate memory blocks from a memory segment.

Usage Scenarios Details

Figure - Empty MEM_SEG representation in the *Memory Segment Allocations* page represents a new, empty memory segment. We will assume a 32-bit architecture for all of the following examples, which also leads to a size of 32 for the CPU_ALIGN type, and which is used by default in some allocation cases. In this figure, the bytes increase from right to left and the memory addresses grow from top to bottom.

Figure - Empty MEM_SEG representation



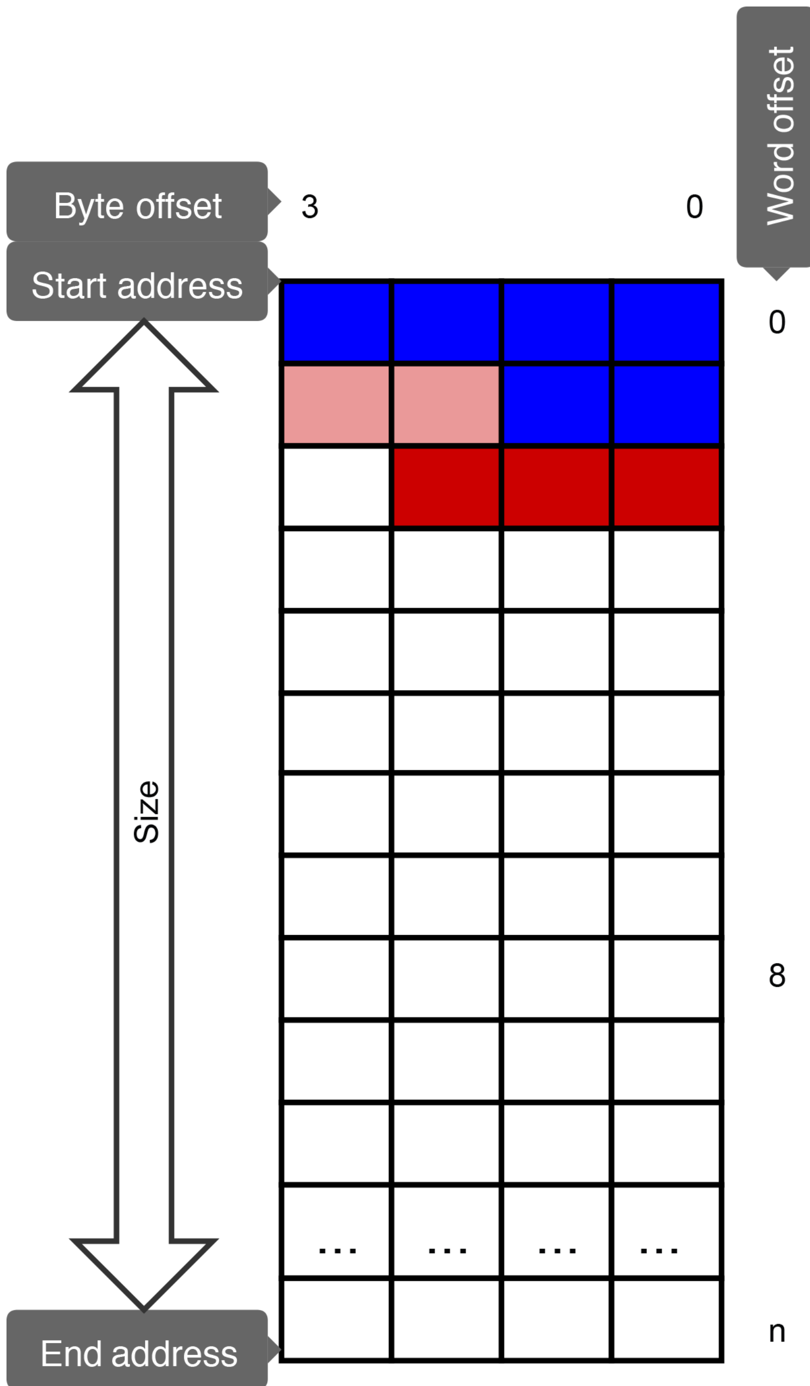
*Assuming 32-bit word architecture, meaning 32-bits for CPU_ALIGN type.

Memory Segment Allocations

Figure - Mem SegAlloc() in the *Memory Segment Allocations* page shows two calls to Mem_SegAlloc(). The first, in blue, simply allocates the needed data (6 bytes), with no alignment required. Since the implicit memory alignment used when

calling Mem_SegAlloc() is sizeof(CPU_ALIGN) (which is 4, in a 32-bit architecture), the second call (in bright red) needs to align itself on a 4-byte boundary. This means sacrificing 2 bytes (in pale red) to meet the alignment requirement. After this call, if a third call is made, it would need to sacrifice one byte in order to be aligned on a 4-byte boundary as well.

Figure - Mem_SegAlloc()



*Assuming 32-bit word architecture, meaning 32-bits for CPU_ALIGN type.

	Mem_SegAlloc(): size= 6 bytes
	Mem_SegAlloc(): size= 3 bytes
	Space lost due to following Mem_SegAlloc() implicit alignment(4 bytes).

Listing - Mem_SegAlloc() code snippet

```
#define  MY_MEM_SEG_DATA_SIZE  64u

CPU_INT08U  MyMemSegData[MY_MEM_SEG_DATA_SIZE];
MEM_SEG     MyMemSeg;

/* ... */

static  void  MyCreationFnct (void)
{
   RTOS_ERR  err;

   Mem_SegCreate("My mem seg",
                 &MyMemSeg,
                 (CPU_ADDR)&MyMemSegData[0u],
                 MY_MEM_SEG_DATA_SIZE,
                 LIB_MEM_PADDING_ALIGN_NONE,
                 &err);
   if (RTOS_ERR_CODE_GET(err) != RTOS_ERR_NONE) {
      /* Handle error. */
   }
}

/* ... */
static  void  MyAlloctionFnct (void)
{
   CPU_INT08U  *p_data_1;
   CPU_INT08U  *p_data_2;
   RTOS_ERR     err;

   p_data_1 = (CPU_INT08U *) Mem_SegAlloc("allocation 1",
                                          &MyMemSeg,
                                          6u,
                                          &err);
   if (RTOS_ERR_CODE_GET(err) != RTOS_ERR_NONE) {
      /* Handle error. */
   }

   p_data_2 = (CPU_INT08U *) Mem_SegAlloc("allocation 2",
                                          &MyMemSeg,
                                          3u,
                                          &err);
   if (RTOS_ERR_CODE_GET(err) != RTOS_ERR_NONE) {
      /* Handle error. */
   }
}
```

Extended Memory Segment Allocations

Figure - Mem_SegAllocExt() in the *Memory Segment Allocations* page shows three calls made to Mem_SegAllocExt(), which allows you to specify the memory block alignment for each call yourself. But varying the alignment value without forethought can produce unexpected results.

The first call simply allocates the data, which is 5 bytes. The second call, needing to be aligned on a 2-byte boundary, sacrifices a byte before allocating the requested data, which is 3 bytes. Finally, the third call, which requires a 16-byte alignment, sacrifices 7 bytes in order to align its allocated memory on a 16-byte boundary.

When these allocations are executed in that order, we end up using 24 bytes of memory for 16 usable, correctly-aligned bytes. When the alignment constraints vary greatly between allocation calls, you should consider the order in which you perform these calls, or try to adjust the size allocated, to save memory space. For example, by swapping the second and third calls, you would end up using only 19 bytes of memory to have 16 correctly-aligned, usable bytes.

Figure - Mem_SegAllocExt()

Listing - Mem_SegAllocExt() code snippet

```
#define  MY_MEM_SEG_DATA_SIZE  64u

CPU_INT08U  MyMemSegData[MY_MEM_SEG_DATA_SIZE];
MEM_SEG     MyMemSeg;

/* ... */

static  void  MyCreationFnct (void)
{
    RTOS_ERR  err;

    Mem_SegCreate("My mem seg",
                  &MyMemSeg,
                  (CPU_ADDR)&MyMemSegData[0u],
                  MY_MEM_SEG_DATA_SIZE,
                  LIB_MEM_PADDING_ALIGN_NONE,
                  &err);
    if (RTOS_ERR_CODE_GET(err) != RTOS_ERR_NONE) {
        /* Handle error. */
    }
}

/* ... */

static  void  MyAlloctionFnct (void)
{
    CPU_INT08U  *p_data_1;
    CPU_INT08U  *p_data_2;
    CPU_INT08U  *p_data_3;
    RTOS_ERR     err;

    p_data_1 = (CPU_INT08U *) Mem_SegAllocExt("allocation 1",
                                              &MyMemSeg,
                                              6u,
                                              4u,
                                              DEF_NULL,
                                              &err);
    if (RTOS_ERR_CODE_GET(err) != RTOS_ERR_NONE) {
        /* Handle error. */
    }

    p_data_2 = (CPU_INT08U *) Mem_SegAllocExt("allocation 2",
                                              &MyMemSeg,
                                              3u,
                                              2u,
                                              DEF_NULL,
                                              &err);
    if (RTOS_ERR_CODE_GET(err) != RTOS_ERR_NONE) {
        /* Handle error. */
    }

    p_data_3 = (CPU_INT08U *) Mem_SegAllocExt("allocation 3",
                                              &MyMemSeg,
                                              8u,
                                              16u,
                                              DEF_NULL,
                                              &err);
    if (RTOS_ERR_CODE_GET(err) != RTOS_ERR_NONE) {
        /* Handle error. */
    }
}
```

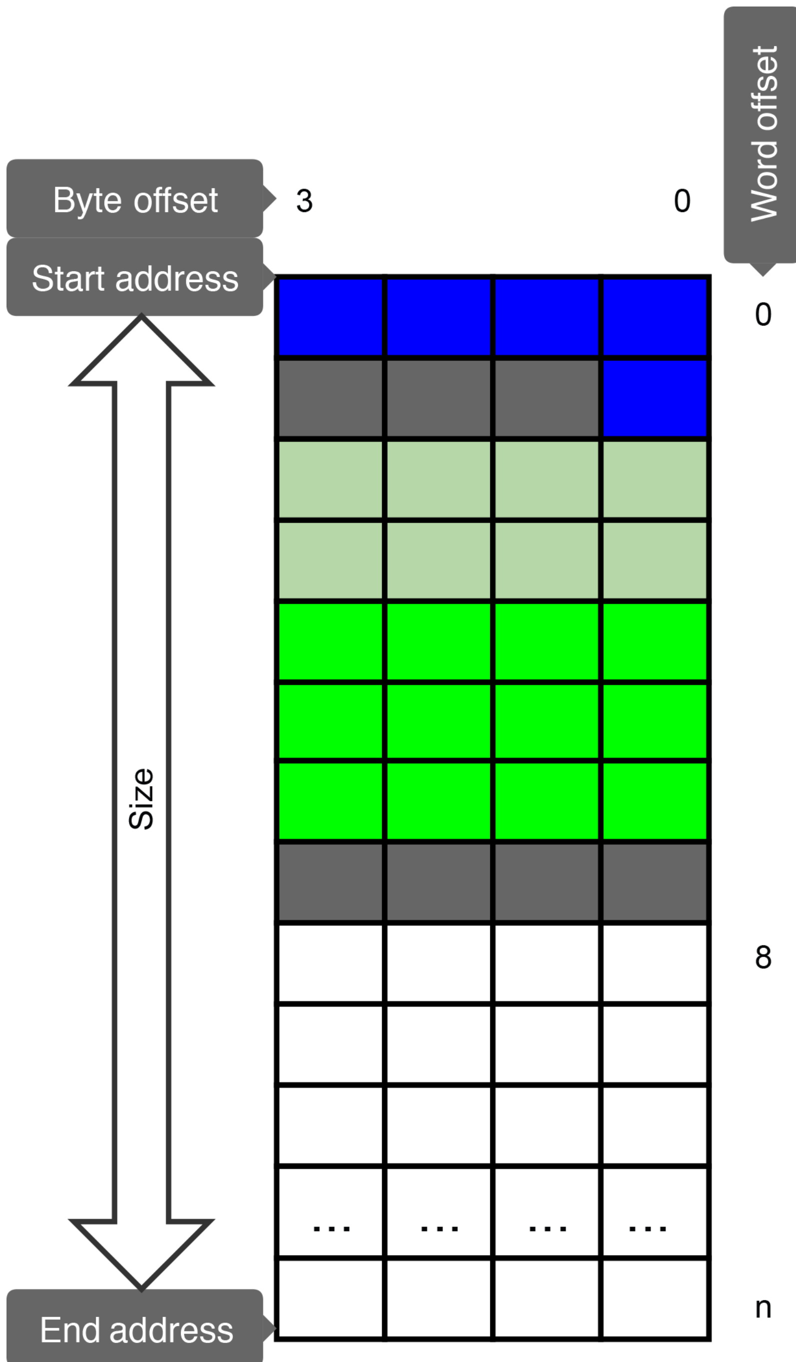
Hardware Memory Segment Allocations

The Figure - Mem_SegAllocHW() in the *Memory Segment Allocations* page shows calls to Mem_SegAllocHW(). This flavour of allocation function is a bit different from the first two because it also takes in consideration the padding alignment of the segment from which data is allocated. This padding alignment is specified at segment creation, and it can be set to zero if it is not required. Mem_SegAllocHW() should be used when you are allocating data that will be accessed via DMA, and which are located in cacheable memory. These conditions are often present when using controllers (USB, Ethernet, etc.) in DMA mode. The advantage of Mem_SegAllocHW() is that it makes it easy to ensure that allocated blocks are not within the same cache lines, which makes flush and invalidate operations easier.

In this example, the padding alignment was set to 8 bytes at segment creation.

The first call is made for 5 bytes, but the following 3 are lost, to push the next allocation outside the current 8-byte segment. The second allocation requires a 16-byte alignment, which means that 8 more bytes need to be sacrificed to correctly align this allocation. Then, a bit like for the first allocation, the segment is padded until it reaches an 8-byte boundary, where a future allocation could be made.

Figure - Mem_SegAllocHW()



*Assuming 32-bit word architecture. Segment's padding at creation is set to 8 bytes.

Color	Description
Blue	Mem_SegAllocHW(): size= 5 bytes, align= 4 bytes
Dark gray	Space lost due to segment's padding of 8 bytes.
Green	Mem_SegAllocExt(): size= 12 bytes, align= 16 bytes
Light green	Space lost due to Mem_SegAllocHW() alignment of 16 bytes.

Listing - Mem_SegAllocHW() code snippet

```
#define  MY_MEM_SEG_DATA_SIZE  64u

CPU_INT08U  MyMemSegData[MY_MEM_SEG_DATA_SIZE];
MEM_SEG     MyMemSeg;

/* ... */

static  void  MyCreationFnct (void)
{
    RTOS_ERR  err;

    Mem_SegCreate("My mem seg",
                  &MyMemSeg,
                  (CPU_ADDR)&MyMemSegData[0u],
                  MY_MEM_SEG_DATA_SIZE,
                  8u,
                  &err);
    if (RTOS_ERR_CODE_GET(err) != RTOS_ERR_NONE) {
        /* Handle error. */
    }
}

/* ... */

static  void  MyAlloctionFnct (void)
{
    CPU_INT08U  *p_data_1;
    CPU_INT08U  *p_data_2;
    RTOS_ERR     err;

    p_data_1 = (CPU_INT08U *) Mem_SegAllocHW("allocation 1",
                                             &MyMemSeg,
                                             5u,
                                             4u,
                                             DEF_NULL,
                                             &err);
    if (RTOS_ERR_CODE_GET(err) != RTOS_ERR_NONE) {
        /* Handle error. */
    }

    p_data_2 = (CPU_INT08U *) Mem_SegAllocHW("allocation 2",
                                             &MyMemSeg,
                                             12u,
                                             16u,
                                             DEF_NULL,
                                             &err);
    if (RTOS_ERR_CODE_GET(err) != RTOS_ERR_NONE) {
        /* Handle error. */
    }
}
```

Stacks Initialization Methods

Stacks Initialization Methods

- Standard Configuration Method
- Advanced Configuration Method
- Comparison Summary
- Stack-Specific Pages

Most Micrium products share the same initialization method. This page provides information about two possible methods that you can use to configure and initialize Micrium modules. We recommend that you use the standard method first during development of your application because it is easier to tailor the modules to your application's needs. Once development is done, you can use the advanced method to save memory and code space and to make the initialization faster.

Standard Configuration Method

This is the default method, and it is recommended that you start with this method.

This method consists of using default values for as many of the configuration fields as possible. These include task stack sizes and task priority, memory segment to use to allocate data, the maximum count of an item type, timeouts, etc. The application can then provide to the <Module>_Init() functions only the mandatory arguments, for which default values cannot be determined.

Using the <Module>_Configure...() function, the application can override any single parameter before the <Module>_Init() is called. These functions allow the application to set some values for the stack to use instead of the default ones. Some of these functions configure a single parameter, while others configure a group of related parameters via a structure. If the function requires a structure, you should follow these steps:

1. Copy the default configuration in a local structure.
2. Modify only the fields that the application needs to change.

It is possible to obtain the default configuration related to any <Module>_Init() function, they are defined in the same file than the <Module>_Init() function and are called <Module>_InitCfgDflt. This structure contains every default value related to that particular Init() function, so you can find the exact structure to copy in the local one (see Listing - Copy default configuration structure and call Module_Configure() function in the *Stacks Initialization Methods* page for an example). Normally, once the call to <Module>_Configure() is done, the configuration structure to which the pointer passed in parameter pointed to is no longer required to remain valid. Exceptions are indicated in the function header of the <Module>_Configure().

Listing - Copy default configuration structure and call Module_Configure() function

```
#include <rtos/module/module.h>

void  MyFunction(void)
{
    MODULE_SUB_CFG  my_local_module_sub_cfg;
    RTOS_ERR        err;

    my_local_module_sub_cfg = Module_InitCfgDflt.SubCfg; /* Copy the default values in the local struct.      */
    my_local_module_sub_cfg.Field1 = 4u;                 /* Specify new value for given field(s).             */

    Module_ConfigureSubCfg(&my_local_module_sub_cfg);    /* Call Module_Configure() for that sub-cfg.         */

    /* [...] */                                          /* Repeat, if required, for other configuration(s).  */

    Module_Init(&err);                                   /* Call Module_Init() after Configure call(s).       */
    if (RTOS_ERR_CODE_GET(err) != RTOS_ERR_NONE) {       /* Check err.                                        */
        /* Handle err. */
    }
}
```

Some properties cannot be configured pre-initialization and can only set once the initialization is completed. The functions to do this kind of operation are not called <Module>_Configure(), but should instead be called <Module>_xxxx_Set().

Advanced Configuration Method

This method requires that you provide every optional configuration structure needed by every module. For every <Module>_Init() function, a corresponding const <MODULE>_INIT_CFG called <Module>_InitCfg must be defined and compiled by the application somewhere. When using this method, the modules will assume that such a configuration exists and will declare it as extern, and will assume the application will define it. To use this method, set RTOS_CFG_EXTERNALIZE_OPTIONAL_CFG_EN to DEF_ENABLED.

Neither the <Module>_Configure...() functions nor the <Module>_InitCfgDflt will be available if this method is used. Examples and templates containing the const <MODULE>_INIT_CFG called <Module>_InitCfg for each module are available.

This method uses less memory than the standard one because no copy of the configuration structures need be kept in memory by the stacks.

Listing - Example of configuration structure externalization

```
const  MODULE_INIT_CFG  Module_InitCfg =
{
    .SubCfg =
    {
        .Field1 = 0u,
        .Field2 = 512u
    },
    .StkSizeElements = 256u,
    .StkPtr          = DEF_NULL,
    .MemSegPtr       = DEF_NULL
};
```

Comparison Summary

Operation	Standard Method	Advanced Method
Initialize the module	(Optional) Call `<Module>_Configure...()`, to change configurations from their default values. Call `<Module>_Init(...)`.	Define a `<Module>_InitCfg` for every `<Module>_Init(...)` that the application will call, specify a value for every field of the struct. Call `<Module>_Init(...)`.
Recover the default configuration structure	Global variable `<Module>_InitCfgDflt` contains it.	Not available with this method.
Specify new configuration values	Call `<Module>_Configure...()` with the relevant parameter(s) for every parameter (or structure of parameter) that needs to be configured differently than its default value.	Change the values specified in the relevant `<Module>_InitCfg`.
Change properties (task priority, timeouts, etc.) after the initialization is done.	Call the relevant `<Module>_...Set()` function.	Call the relevant `<Module>_...Set()` function.

Stack-Specific Pages

- Page:Logging Run-Time Application-Specific Configuration
- Page:Common Run-Time Configuration
- Page:File System Run-Time Configuration
- Page:Shell Run-Time Application-Specific Configurations
- Page:Authentication Run-Time Application-Specific Configurations
- Page:SPI Core Run-Time Configurations
- Page:Network Core Run-Time Configurations
- Page:SD Core Run-Time Configurations
- Page:Kernel Run-Time Application Specific Configurations
- Page:USB Device Vendor Class Run-Time Application Specific Configurations
- Page:USB Device Run-Time Application-Specific Configuration
- Page:USB Device HID Class Run-Time Application Specific Configurations
- Page:USB Device CDC ACM Class Run-Time Application Specific Configurations
- Page:USB Host Run-Time Application Specific Configurations
- Page:USB Device MSC Class Run-time Application Specific Configurations
- Page:USB Device CDC EEM Class Run-time Application Specific Configurations

Understanding Micrium Os Internals And Optimizing Its Configuration

Understanding Micrium OS Internals and Optimizing Its Configuration

The Micrium OS Kernel has up to two internal tasks that provide various services to your application. These tasks feature a large number of configurable settings that can help optimize your code for size, memory usage, or speed.

Internals Overview

Internals Overview

Internal Tasks

Timer Task

The timer task is used to implement all software-based timers in the Micrium OS Kernel. You must create these timers, and they can be either one-shot or periodic timers. If software timers are not needed in your application, this task can be disabled (through OS_CFG_TMR_EN) to save memory.

When a timer expires, your application can receive a callback from the timer. Note that this callback *must not* make any pend calls. It is also important to note that this callback is made from the timer task's stack, so the timer's stack must be sized appropriately if the callback function is rather long.

The rate at which the timer task runs, its priority, and its stack size are all configurable. To change the default values, call the following function *before* the OSInit() call in main():

```
void  OS_ConfigureTmrTask (OS_TASK_CFG  *p_tmr_task_cfg)
```

Statistics Task

The statistics task is used keep track of all the kernel objects in use in Micrium OS Kernel so they can be reported via uC/Probe, Micrium Kernel-Awareness plugins or user logs. This task runs at a fixed interval (typically 10Hz) and the rate is configurable. If the statistics task is not needed, it can be disabled to save space.

The statistics task also provides a hook to allow user code after the statistics task has run. This can be useful if a user wishes to log certain statistics. The hook is called via OSStatTaskHook().

The rate at which the statistics task runs, its priority and its stack size are all configurable. To change the default values, call the following function **before** the OSInit() call in main():

```
void  OS_ConfigureStatTask  (OS_TASK_CFG  *p_stat_task_cfg)
```

Interrupt Stack

The Cortex-M architecture allows for a separate interrupt stack, which makes it easier to write applications for a real-time operating system. Without an extra interrupt stack, you would be required to always have enough room at the end of every task's stack to handle the stack elements of the interrupt. With the Cortex-M architecture, a separate ISR stack is defined and used during an interrupt context. The default size of the interrupt stack (256 stack units/1024 bytes) is typically enough for most Micrium OS applications. All interrupts should ideally be very short, so there should not be a need for a large stack.

If it does become necessary to have a larger interrupt stack, the size of the stack can be adjusted by making the following call *before* the OSInit() call in main():

```
void  OS_ConfigureISRStk (CPU_STK       *p_stk_base_ptr,
                          CPU_STK_SIZE   stk_size)
```

Micrium Heap

Micrium OS makes use of a heap-like area to allocate certain OS objects. By default, the heap will be used for the following items: the internal tasks' TCB and stack (assuming any are enabled), the interrupt task stack, and data related to any kernel objects created. In other stacks such as Network or USB, the heap is used for service tasks related to the stack, as well as

any internal data structures that must be created. Unfortunately, because everything is so configurable, it is not possible at compile time to determine the exact size of the heap that is needed before runtime. This means it is imperative that all return values from any Micrium OS call are checked.

The size of the heap is configured in common_cfg.h. To change the size of the heap, locate the following line:

```
#define  LIB_MEM_CFG_HEAP_SIZE  XXXXuL
```

If an assert is hit with the error RTOS_ERR_SEG_OVF, this means the heap size is too small and needs to be made larger.

Note that this heap is provided by the Common module of Micrium OS, and is *not* the general-purpose C heap that would be used with the standard malloc-type functions. That general-purpose heap is configured through the linker file and is outside the scope of Micrium OS, which does not use the standard malloc(). Micrium OS applications should never use the malloc functions.

This heap is one of an indefinite number of memory segments (see Mem_SegCreate()). When a Micrium OS module, or the application, allocates space on a memory segment (see Mem_SegAlloc()), it specifies a pointer to the memory segment to use. Passing the value DEF_NULL will result in allocating the memory on the heap. By default, all Micrium OS modules will get their required memory from the heap. This, however, is configurable for each module, by calling the function <module>_ConfigureMemSeg() *before* initializing the module with <module>_Init(). Each module can therefore use isolated memory sections, either onboard the MCU or in external RAM.

Error Handling

The last parameter of any Micrium OS function is reserved for the error pointer. The error pointer is always an RTOS_ERR structure. The error structure is defined as:

```
typedef  struct  rtos_err {
    RTOS_ERR_CODE     Code;        /**< Err code enum val.                        */
#if (RTOS_ERR_CFG_EXT_EN == DEF_ENABLED)
#if (RTOS_ERR_CFG_STR_EN == DEF_ENABLED)
    CPU_CHAR const   *CodeText;    /**< Err code in string fmt.                   */
    CPU_CHAR const   *DescText;    /**< Err desc string.                          */
#endif
    CPU_CHAR         *FileName;    /**< File name where error occurred.           */
    CPU_INT32U        LineNbr;     /**< Line nbr  where error occurred.           */
#ifdef PP_C_STD_VERSION_C99_PRESENT /* Only present if C99 enabled.                */
    const  CPU_CHAR  *FnctName;    /**< Fnct name where err occurred.             */
#endif
```

After every Micrium OS API call, you should always check the return value of RTOS_ERR. Since the error return is a struct, Micrium OS provides a set of macros to access parts of the struct to future-proof code from any possible changes to RTOS_ERR in the future. These macros are:

- RTOS_ERR_CODE_GET(err_val)
- RTOS_ERR_STR_GET(err_code)
- RTOS_ERR_DESC_STR_GET(err_code)
- RTOS_ERR_SET(err_var, err_code)
- RTOS_ERR_COPY(err_dst, err_src)

As shown in the declaration of RTOS_ERR, you can enable or disable all of the fields except the code variable. This is accomplished using the following defines in rtos_err_cfg.h .

#define	Result
RTOS_ERR_CFG_EXT_EN	DEF_ENABLED: use and populate the *extended* error fields for **file name**, **line number**, and – if C99 is enabled – the **function name**, where the error occurred.
RTOS_ERR_CFG_STR_EN	DEF_ENABLED: allow for the error's string-name and descriptive text to be saved in RTOS_ERR. This is useful if you want to log the errors to a file and review it at a later time. Requires RTOS_ERR_CFG_EXT_EN to be enabled.

Assertions

Micrium OS has four assertion macros that can be used to check error conditions:

- RTOS_ASSERT_CRITICAL(expr, err_code, ret_val)
- RTOS_ASSERT_DBG(expr, err_code, ret_val)
- APP_RTOS_ASSERT_CRITICAL(expr, err_code, ret_val)
- APP_RTOS_ASSERT_DBG(expr, err_code, ret_val)

The first two macros are used only for internal Micrium OS code. The second two macros are provided for you to use in your application if you wish.

The *critical* asserts are used to check for an irrecoverable error. These can typically occur when incidences such as a stack overflow have occurred and made unexpected changes to kernel objects or stacks. The internal and application critical asserts map to another macro called CPU_SW_EXCEPTION() which will lock the system. Due to this, you should always have a watchdog enabled so it can properly restart the system if an error occurs.

The *debug* asserts typically check for conditions that are caused by invalid parameters or invalid configurations. They are used to notify you that something is not correct with the way the code is being used. These asserts can and should be disabled once development is completed. To disable the debug asserts, set RTOS_CFG_ASSERT_DBG_ARG_CHK_EXT_MASK in rtos_cfg.h to RTOS_CFG_MODULE_NONE.

Transitioning From Development To Production Code

Transitioning from Development to Production Code

While your project is in development, you will typically set:

- **Over sized configuration values**, so that you don't have to constantly increase them to fit your needs as your application grows.
- **More verbose logging**, so that you can quickly pinpoint any error or abnormal behavior.
- **Stricter error handling**, so that you don't, for example, overlook important validations.

As you move towards a production release, you may want to fine tune these values in order to reach your optimization goals.

Here are some typical values to consider.

Kernel Services Configuration

<table>
<tr><th>Setting</th><th>Development</th><th>Production</th><th>Approximate savings</th></tr>
<tr><td colspan="4">Logging</td></tr>
<tr><td>RTOS_CFG_LOG_EN</td><td>DEF_ENABLED</td><td>DEF_DISABLED</td><td></td></tr>
<tr><td colspan="4">Speed up execution and save code space by disabling logging. Can also reduce compilation time significantly.</td></tr>
<tr><td colspan="4">Externalized configuration</td></tr>
<tr><td>RTOS_CFG_EXTERNALIZE_OPTIONAL_CFG_EN</td><td>DEF_DISABLED</td><td>DEF_ENABLED</td><td></td></tr>
<tr><td colspan="4">Save RAM by making the configuration structures const. Modules will not be runtime-configurable anymore; that is, the <Module>_ConfigureXxxx() functions will be removed. This setting applies to the configuration of all modules at once.</td></tr>
<tr><td colspan="4">Argument Checks</td></tr>
<tr><td>RTOS_CFG_ASSERT_DBG_ARG_CHK_EXT_MASK</td><td>RTOS_CFG_MODULE_ALL</td><td>RTOS_CFG_MODULE_NONE</td><td>Up to 5 KB code/ROM</td></tr>
<tr><td colspan="4">Speed up execution and save code space by not validating function arguments. It is generally expected that your development cycles have afforded you the opportunity to catch these errors.
Note that it is possible to keep the argument checks for only some module, by OR'ing their #defined module ID. For example:
#define RTOS_CFG_ASSERT_DBG_ARG_CHK_EXT_MASK RTOS_CFG_MODULE_NET|RTOS_CFG_MODULE_KERNEL</td></tr>
<tr><td colspan="4">Extended Error Information</td></tr>
<tr><td>RTOS_ERR_CFG_EXT_EN</td><td>DEF_ENABLED</td><td>DEF_DISABLED</td><td>1 KB code/ROM</td></tr>
<tr><td colspan="4">Speed up execution by not logging extended errors (file name, line number, function name).</td></tr>
<tr><td colspan="4">Descriptive Error Messages</td></tr>
<tr><td>RTOS_ERR_CFG_STR_EN</td><td>DEF_ENABLED</td><td>DEF_DISABLED</td><td>9 KB code/ROM</td></tr>
<tr><td colspan="4">Speed up execution and save code space by logging only the error's enum value instead of looking up its string value.</td></tr>
</table>

Removing Unnecessary Tasks

Task	#define (os_cfg.h)	Approximate savings	Remarks
Timer Task	OS_CFG_TMR_EN	2500 bytes code/ROM 200 bytes RAM	Save code space and RAM by disabling this task if software timers are not required.
Stats Task	OS_CFG_STAT_TASK_EN	3000 bytes code/ROM 400 bytes RAM	Save code space and RAM by disabling this task if kernel statistics are not required.

Note that each of the above tasks also uses a stack size of 256 elements by default. This provides an additional **1024 bytes of RAM saving** for each disabled task.

Disabling Unnecessary Services

When scraping for small amounts of RAM or code space, some very specific services can be disabled. Among the most interesting of these small improvements are:

- OS_CFG_TASK_DEL_EN
- OS_CFG_TASK_Q_EN
- OS_CFG_TASK_STK_REDZONE_EN

Adjusting Heap Size

If every unnecessary features have been disabled and the amount of RAM available is almost reached, consider lowering the heap size. Finding the appropriate value may not be an easy task. The amount of heap space required is influenced not only by your application design, but also by other Micrium OS modules such as Net, USB, and File System. Familiarize yourself with each stack's compile-time and run-time configuration options. Look for settings related to size (e.g., stack size, number of cached items) or optional features to disable.

Adjusting Stack Sizes

Depending on how your application is constructed, it may be possible to reduce the size of one or more task stacks and therefore save RAM. Stack usage can be monitored using kernel-aware tools build into some toolchains. Alternatively, Micrium's µC/Probe can be used to display that information while subjecting the target to its known worse case conditions. µC/Probe will report (graphically) the stack usage of each task.

Example Applications

Example Applications

Micrium OS is provided with a set of example applications. Example applications are meant to be used as a starting point to develop your application and are designed to be compile-able and executable (in most cases) as-is without modification. Example application files can be found under the `SiliconLabs\SimplicityStudio\vX\developer\sdks\gecko_sdk_suite\vX.Y\app\micrium_os_example` folder. The folder is organized by module and contains one main example file called ex_main.c. This example creates a kernel task that performs the basic initialization of Micrium OS and calls the example applications you added to your project. It uses the ex_description.h file to determine which example is present or not.

Example Description File

Example Description File

The example applications rely on a file called ex_description.h. This file has multiple purposes:

- It lists all the example applications available that should be called automatically. They are listed using #define having the following format:

```
#define EX_<MODULE>_<SUBMODULE>_INIT_AVAIL
```

- It allows you to override some default configurations set in the example applications. Some example applications contain default configurations that *must* be overridden. Refer to your examples documentation to find out if you have to override a configuration.

Example Applications Documentation

Example Applications Documentation

Following is a list of pages that describes some of the provided example applications. Note that not all of the provided examples are documented.

- Page:CANopen Example Applications
- Page:File System Example Applications
- Page:FTP Client Example Application
- Page:HTTP Client Example Applications
- Page:HTTP Server Example Applications
- Page:IPerf Example Applications
- Page:MQTT Client Example Applications
- Page:Network Core Example Applications
- Page:SMTP Client Example Applications
- Page:SNTP Client Example Applications
- Page:Telnet Server Example Applications
- Page:TFTP Client Example Application
- Page:USB Device CDC ACM Class Example Applications
- Page:USB Device CDC EEM Class Example Applications
- Page:USB Device Core Example Applications
- Page:USB Device HID Class Example Applications
- Page:USB Device MSC Class Example Applications
- Page:USB Device Vendor Class Example Applications
- Page:USB Host Android Accessory Class Example Applications
- Page:USB Host CDC ACM Class Example Applications
- Page:USB Host Core Example Applications
- Page:USB Host MSC Class Example Applications
- Page:USB Host USB-to-Serial Class Example Applications

Common

Common

Micrium OS Common is a module that groups common components used by various Micrium products. It currently includes:

- An authentication module which allows you to create users and manage their access rights
- An emulation of the C standard library, including functions for memory allocation and de-allocation including:
  - Data copy, compare
  - String copy, compare, replace
  - Some character functions and a basic random number generator (RNG)
- A kernel abstraction layer (KAL), which allows products to make an abstraction of the active kernel (OS2 or OS3)
- Logging capabilities available for every Micrium OS module
- For all the Micrium OS modules, there is a common error type and several related utilities
- A shell interface for Micrium OS components and any other components that register shell functions
- A toolchain detection and abstraction component

Most parts of Micrium OS Common are expected to be present in every Micrium OS project.

Common Overview

Common Overview

- Specifications
  - LIB
  - KAL
- Features
  - Authentication
  - LIB
  - KAL
  - Logging
  - RTOS ERR
  - Shell
  - Toolchain
- Limitations
  - Auth
  - LIB

Specifications

LIB

- Provides a clean, organized, and ANSI C-compatible implementation of the most common standard library functions, macros, and constants.
- Independent of and compatible with any processor (CPU) and compiler.
- Memory footprint can be adjusted at compile time, based on the features you need and optimal run-time performance.
- Designed and implemented with safety-critical certification in mind.
- Intended for use in any high-reliability, safety-critical systems including avionics RTCA DO-178B and EUROCAE ED-12B, medical FDA 510(k), IEC 61508 industrial control systems, and EN-50128 rail transportation and nuclear systems.

  For example, the FAA (Federal Aviation Administration) requires that all the source code for an application be available in source form and conform to specific software standards in order to be certified for avionics systems. Since most standard library functions provided by compiler vendors are in an uncertifiable binary format, this module offers an alternative by providing its library functions in certifiable source-code format.

- Source available, and conforms to the Micrium's coding standard.

KAL

- Currently supports the following kernels:
  - Micrium's µC/OS-II
  - Micrium OS Kernel

Features

Authentication

- Set rights for any number of users at run-time.
- Can validate credentials (name and password) for a user before executing a specific operation.
- No hierarchy between users.

LIB

- Memory allocation and management, including:
  - Segments

  - Blocks
  - Dynamic pools
- Memory-related utilities (manage endianness conversion).
- String-related utilities (string length, copy, compare, replace, formatting, etc.).
- ASCII characters operations (lowercase/uppercase conversion, #defines, etc.).
- Bit operations (clear, set, mask, etc.).
- Basic random number generator.
- Various useful defines.

KAL

- Allows you to specify a single location where to allocate all the kernel data used by other modules.
- Facilitates the use of any kernel with any Micrium OS component, where only one abstraction layer needs to be changed.
- Factorizes every kernel abstraction layer into a single element.

Logging

- Configures multiple channels independently.
- Hierarchical configuration that can inherit configuration from a parent channel.
- Three logging levels that can apply filters to display only the required information.
- Any putchar-like function can be used to output data.
- Synchronous and asynchronous modes are available.
- In asynchronous mode, an internal ring buffer keeps the most recent logging entries and the application can choose when to output them.

RTOS_ERR

- Single, unified error type for every Micrium OS module.
- Provides a string description for every error.
- Provides several utilities to determine the files and lines where errors occur.

Shell

- Shell interface available for Micrium modules.
- Parses and executes commands via the command line.
- Adds additional commands easily.

Toolchain

- C standard version and toolchain detection.
- Macros to void unused function parameters or align specific data in memory for several toolchains.

Limitations

Auth

- The maximum number of distinct, exclusive rights that can be defined is set to 28.

LIB

- By design, variable argument library functions are not supported.

Integrating Common Into Your Project

Integrating Common Into Your Project

The Micrium OS Common module can be composed of several components, each of which is a set of files that implement specific functions. The Common module consists of one component for the core part named "Common" and one optional part called "Shell". To use Common, you must add these files to your project and populate your RTOS Description File .

Starting Common Quickly

Micrium offers a set of example applications that demonstrate some of the features of Micrium OS Common and help you start the development of your application. We recommend starting from one of these examples.

Configuring Common

Common can be configured to optimize memory usage or features. The following pages explain how Common and the whole Micrium OS can be configured at compile-time: Common Compile-Time Configuration , RTOS Compile-Time Configuration and RTOS ERR Type Configuration . The Common Run-Time Configuration page and all pages below it explain how Common and its sub-modules can be configured at run-time.

Common Configuration

Common Configuration

You can configure the Common module to define how the module will behave, what resources it requires, and so on. The configuration options take effect at compile-time.

You can also change run-time configurations that will tailor the Common module to the needs of the application, while using a minimum of resources when it not needed. The following pages describe both types of configurations.

- RTOS Compile-Time Description
- RTOS Compile-Time Configuration
- Common Compile-Time Configuration
- RTOS ERR Type Configuration
- Common Run-Time Configuration

RTOS Compile-Time Description

- RTOS Description
- CPU & Toolchain - Mandatory Constant Description
- Modules Present Description

RTOS Description

The following configuration constants can be added to the rtos_description.h file.

CPU & Toolchain - Mandatory Constant Description

Constante	Description	Possible value
RTOS_CPU_SEL	Specify the CPU architecture used.	For all Silicon Labs Gecko chip this #define shall be set to: **RTOS_CPU_SEL_SILABS_GECKO_AUTO** For other MCUs, the following is the list of possible values: RTOS_CPU_SEL_SILABS_GECKO_AUTO RTOS_CPU_SEL_ARM_CORTEX_M0 RTOS_CPU_SEL_ARM_CORTEX_M0P RTOS_CPU_SEL_ARM_CORTEX_M1 RTOS_CPU_SEL_ARM_CORTEX_M3 RTOS_CPU_SEL_ARM_CORTEX_M4 RTOS_CPU_SEL_ARM_CORTEX_M7 RTOS_CPU_SEL_ARM_V7_M RTOS_CPU_SEL_EMUL_WIN32 RTOS_CPU_SEL_EMUL_POSIX
RTOS_TOOLCHAIN_SEL	Specify the toolchain used.	Default value: **RTOS_TOOLCHAIN_AUTO** Possible values: RTOS_TOOLCHAIN_ARMCC RTOS_TOOLCHAIN_CCS RTOS_TOOLCHAIN_CROSSCORE_BLACKFIN RTOS_TOOLCHAIN_GNU RTOS_TOOLCHAIN_IAR RTOS_TOOLCHAIN_MPLAB_C30 RTOS_TOOLCHAIN_RXC RTOS_TOOLCHAIN_VISUALSTUDIO RTOS_TOOLCHAIN_VDSP

Modules Present Description

The following is a list of #define to be added to rtos_description.h to activate the modules and its sub-modules.

#define
RTOS_MODULE_KERNEL_AVAIL
RTOS_MODULE_COMMON_SHELL_AVAIL
RTOS_MODULE_USB_HOST_AVAIL
RTOS_MODULE_USB_HOST_CDC_AVAIL
RTOS_MODULE_USB_HOST_ACM_AVAIL
RTOS_MODULE_USB_HOST_AOAP_AVAIL
RTOS_MODULE_USB_HOST_USB2SER_AVAIL
RTOS_MODULE_USB_HOST_USB2SER_FTDI_AVAIL
RTOS_MODULE_USB_HOST_USB2SER_SILABS_AVAIL
RTOS_MODULE_USB_HOST_USB2SER_PROLIFIC_AVAIL
RTOS_MODULE_USB_HOST_MSC_AVAIL
RTOS_MODULE_USB_HOST_HID_AVAIL
RTOS_MODULE_USB_DEV_AVAIL
RTOS_MODULE_USB_DEV_AUDIO_AVAIL
RTOS_MODULE_USB_DEV_CDC_AVAIL
RTOS_MODULE_USB_DEV_ACM_AVAIL

#define
RTOS_MODULE_USB_DEV_EEM_AVAIL
RTOS_MODULE_USB_DEV_HID_AVAIL
RTOS_MODULE_USB_DEV_MSC_AVAIL
RTOS_MODULE_USB_DEV_VENDOR_AVAIL
RTOS_MODULE_FS_AVAIL
RTOS_MODULE_FS_STORAGE_NAND_AVAIL
RTOS_MODULE_FS_STORAGE_NOR_AVAIL
RTOS_MODULE_FS_STORAGE_RAM_DISK_AVAIL
RTOS_MODULE_FS_STORAGE_SCSI_AVAIL
RTOS_MODULE_FS_STORAGE_SD_CARD_AVAIL
RTOS_MODULE_FS_STORAGE_SD_SPI_AVAIL
RTOS_MODULE_IO_SERIAL_AVAIL
RTOS_MODULE_IO_SERIAL_SPI_AVAIL
RTOS_MODULE_NET_AVAIL
RTOS_MODULE_NET_IF_ETHER_AVAIL
RTOS_MODULE_NET_IF_WIFI_AVAIL
RTOS_MODULE_NET_SSL_TLS_AVAIL
RTOS_MODULE_NET_SSL_TLS_MOCANA_NANOSSL_AVAIL
RTOS_MODULE_NET_SSL_TLS_MBEDTLS_AVAIL
RTOS_MODULE_NET_HTTP_CLIENT_AVAIL
RTOS_MODULE_NET_HTTP_SERVER_AVAIL
RTOS_MODULE_NET_TELNET_SERVER_AVAIL
RTOS_MODULE_NET_MQTT_CLIENT_AVAIL
RTOS_MODULE_NET_SMTP_CLIENT_AVAIL
RTOS_MODULE_NET_SNTP_CLIENT_AVAIL
RTOS_MODULE_NET_IPERF_AVAIL
RTOS_MODULE_NET_FTP_CLIENT_AVAIL
RTOS_MODULE_NET_TFTP_CLIENT_AVAIL
RTOS_MODULE_NET_TFTP_SERVER_AVAIL

RTOS Compile-Time Configuration

- RTOS Configuration
  - Asserts configurations
  - RTOS_CFG_RTOS_ASSERT_DBG_FAILED_END_CALL_SEL
  - RTOS_CFG_RTOS_ASSERT_CRITICAL_FAILED_END_CALL_SEL
  - RTOS_CFG_EXTERNALIZE_OPTIONAL_CFG_EN
- RTOS Logging Configuration
  - Logging Channel Configuration
  - Major Logging Channels

RTOS Configuration

The following configuration constants are located in the rtos_cfg.h file.

Asserts configurations

RTOS *CFG_ASSERT_DBG_ARG_CHK_EXT*<module>_EN configurations define if asserts that are used in argument checking are enabled or not. These asserts check the validity of the arguments passed to any stack function. This can include checking whether a pointer is non-NULL, or if a given value is within a certain range, or if a configuration is valid, etc.

These configurations ca be set to **1** (enable asserts) or 0 (disable asserts).

There is one configuration per module. The following table lists the different configurations.

Recommendation: disable this feature as much as possible in 'release' code. This will improve performance and reduce the amount of code space required.

Configuration	Description
RTOS_CFG_ASSERT_DBG_ARG_CHK_EXT_APP_EN	Controls asserts located in sample applications.
RTOS_CFG_ASSERT_DBG_ARG_CHK_EXT_BSP_EN	Controls asserts located in BSPs.
RTOS_CFG_ASSERT_DBG_ARG_CHK_EXT_CAN_EN	Controls asserts located in CAN module.
RTOS_CFG_ASSERT_DBG_ARG_CHK_EXT_COMMON_EN	Controls asserts located in common module.
RTOS_CFG_ASSERT_DBG_ARG_CHK_EXT_CPU_EN	Controls asserts located in CPU module.
RTOS_CFG_ASSERT_DBG_ARG_CHK_EXT_FS_EN	Controls asserts located in File System module
RTOS_CFG_ASSERT_DBG_ARG_CHK_EXT_KERNEL_EN	Controls asserts located in the real-time kernel.
RTOS_CFG_ASSERT_DBG_ARG_CHK_EXT_NET_EN	Controls asserts located in network module.
RTOS_CFG_ASSERT_DBG_ARG_CHK_EXT_NET_APP_EN	Controls asserts located in network application modules.
RTOS_CFG_ASSERT_DBG_ARG_CHK_EXT_USBD_EN	Controls asserts located in USB device module.
RTOS_CFG_ASSERT_DBG_ARG_CHK_EXT_USBH_EN	Controls asserts located in USB host module.
RTOS_CFG_ASSERT_DBG_ARG_CHK_EXT_IO_EN	Controls asserts located in IO module.
RTOS_CFG_ASSERT_DBG_ARG_CHK_EXT_PROBE_EN	Controls asserts located in uc/Probe drivers.

RTOS_CFG_RTOS_ASSERT_DBG_FAILED_END_CALL_SEL

Description	Possible values
`RTOS_CFG_RTOS_ASSERT_DBG_FAILED_END_CALL_SEL` configures what operation will happen in cases where a debug assert has failed. A debug "asserts fail" reflects an invalid argument passed by the programmer to a stack function or an operation that cannot be handled in a particular context (such as an ISR). The program *should not* continue running after a debug assert has failed. However, it is possible to configure it to return (by defining it to: `return ret_val` (without parentheses or ';')), if the code is being executed in a test context. Typical operations can include breaking the CPU, looping indefinitely, calling a function or macro, etc. The default behavior is to loop indefinitely.	• **`RTOS_ASSERT_END_CALL_SEL_TRAP`** • `RTOS_ASSERT_END_CALL_SEL_RETURN` • `RTOS_ASSERT_END_CALL_SEL_CUSTOM` See table Table - Assert failed end call configuration value description for a detailed description of each possible value.

RTOS_CFG_RTOS_ASSERT_CRITICAL_FAILED_END_CALL_SEL

Description	Possible values
`RTOS_CFG_RTOS_ASSERT_CRITICAL_FAILED_END_CALL_SEL` configures what operation occurs in cases where a critical assert is failed. A failed critical assert reflects unrecoverable situations in the system such as corruption or other cases that are unexpected. The program *must not continue* to execute in those cases. The program *must not return* from this call, since the software is in an unknown and/or invalid state. Typical operations can include outputting logs or traces, dumping memory, halting and/or restarting the system. The default behavior is to call the `CPU_SW_EXCEPTION` macro.	• **`RTOS_ASSERT_END_CALL_SEL_TRAP`** • `RTOS_ASSERT_END_CALL_SEL_RETURN` • `RTOS_ASSERT_END_CALL_SEL_CUSTOM` See table Table - Assert failed end call configuration value description for a detailed description of each possible value.

Table - Assert failed end call configuration value description

Assert end call possible value	Description
RTOS_ASSERT_END_CALL_SEL_TRAP	Program enters an infinite loop. In case of a critical assert, the macro CPU_SW_EXCEPTION is used. Its default implementation is an infinite loop.
RTOS_ASSERT_END_CALL_SEL_RETURN	Function returns. If function has an argument p_err, it will return an assert related error code.
RTOS_ASSERT_END_CALL_SEL_CUSTOM	Allows to specify a custom behavior when an assert fails. In order to specify the behavior, the macro RTOS_CFG_RTOS_ASSERT_DBG_FAILED_END_CALL(ret_val) must be defined in compiler's defines list (-D). Failure to define the macro RTOS_CFG_RTOS_ASSERT_DBG_FAILED_END_CALL(ret_val) in this mode will cause a pre-processor error.

RTOS_CFG_EXTERNALIZE_OPTIONAL_CFG_EN

Description	Possible values
RTOS_CFG_EXTERNALIZE_OPTIONAL_CFG_EN determines whether the configurations can be provided via optional <Module>_Configure...() calls to the stacks or if these configurations are assumed extern by the stacks and must be provided by the application. If set to DEF_DISABLED, default configurations will be present in stacks and they can optionally be overridden with <Module>_Configure...() calls. This is the easier and preferred option to use when starting the development of an application. The default values used by the stacks are available for the application to copy, but you should only modify parts of it and set them as the new configuration (see each stack's user's manual for more information). If set to DEF_ENABLED, no default configuration will be present in the stacks and every configuration that could be set via <Module_Configure...() calls are now assumed extern by the stacks. These calls must then be defined by the application. This option is useful to reduce the amount of memory and code space used. See Stacks Initialization Methods for more details.	1 (enabled) or **0** (disabled).

RTOS Logging Configuration

Table - RTOS Logging Configuration Constants

Constant	Description	Possible values
RTOS_CFG_LOG_EN	RTOS_CFG_LOG_EN is used to enable or disable the logging module as a whole. If enabled, RTOS_CFG_LOG_ALL MUST be defined. All logging channels will inherit this configuration if not overridden by a more specific configuration as shown below.	1 (enabled) or **0** (disabled).
RTOS_CFG_LOG_ALL	See Logging Channel Configuration section.	**VRB, ASYNC, FUNC_DIS, TS_DIS, putchar** or any valid combinations of flags.
RTOS_CFG_LOG_[...]	See Logging Channel Configuration section.	

Logging Channel Configuration

The logging mechanism allows every logging channel to inherit from the parent channel, with the root ancestor being the RTOS_CFG_LOG_ALL channel. This allows any channel option to have the DFLT setting, which means it uses the inherited channel option.

The "parent-child" relationship is defined by finding the channel to where a particular module is logging. For example, the kernel logs in the KERNEL channel and the USB-Device Audio class would log to the USBD, CLASS, AUDIO channel. This would mean that the kernel channel inherits only from the root parent ALL and that the USB-Device Audio class would inherit from (in order):

- The USBD_CLASS channel if defined
- The USBD channel if defined and the root parent ALL

For the USB-Device Audio class channel, this means that if a #define named RTOS_CFG_LOG_USBD_CLASS does exist, it will take the options set via that configuration as default. If not, it could take its options from a RTOS_CFG_LOG_USBD #define. If it is also not present, it will set the options in the RTOS_CFG_LOG_ALL #define.

This allows a very fine granularity while allowing to configure several channels in a single step.

The available options for each channel #define must respect the order and be separated by commas. The order should be as follows: <lowest_level>, <timing>, <function_name_en>, <time_stamp_en>, <log_output_funct>. The possible values for each option are as follows:

Table - RTOS Logging Channel Options

Option	Description	Possible values
Lowest level to log	Sets the minimal level entries must be to be logged. For example: • If it is set to VRB, every entry will be logged • If it is set to ERR, only error-level messages will be logged	**VRB** : Log error-, debug- and verbose-level messages. DBG: Log error- and debug-level messages. ERR: Log error-level messages only. OFF: Do not log any level. DFLT: Use the inherited setting.
Timing of output of entries	Specifies when to output entries, either: • Synchronously: at the moment the entry is made • Asynchronously: it will then store it in a ring buffer and outputting it only when `Log_Output()` is called.	SYNC: Output logs synchronously, during code execution. May disrupt timing. **ASYNC** : Save logs in buffer, always keeping the most recent ones. Output will only be done when `Log_Output()` is called (see `rtos/common/include/logging.h` for more details). DFLT: Use the inherited setting.
Function name	Includes or omits the function name where the log was made for every logging entry.	FUNC_EN: Function name where logging was made will be included in entry. **FUNC_DIS** : Function name where logging was made will NOT be included in entry. DFLT: Use the inherited setting.
Time-stamp	Includes or omits the timestamp for every logged entry.	TS_EN: Timestamp of when entry was made will be included in entry. **TS_DIS** : Timestamp of when entry was made will NOT be included in entry. DFLT: Use the inherited setting.
Log output function	Outputs any logging entries.	Any function of type `int foo(int character)`, as `putchar`, for example.

Major Logging Channels

The table below presents the major logging channels that can be referenced when configuring the logging module. Several other channels exist, for example for each FS or USB driver (FS, DRV, SD; USBD, DRV, SYNOPSYS_OTG_HS or USBH, HCD, EHCI).

Comma notation (as seen in source code files)	#define used to configure the channel	Description
COMMON	RTOS_CFG_LOG_COMMON	Common's root logging channel
COMMON, LIB	RTOS_CFG_LOG_COMMON_LIB	Common's LIB module logging channel
COMMON, SHELL	RTOS_CFG_LOG_COMMON_SHELL	Common's Shell module logging channel
FS	RTOS_CFG_LOG_FS	File System's root logging channel
FS, BLK_DEV	RTOS_CFG_LOG_FS_BLK_DEV	File System's Block Device layer logging channel
FS, CORE	RTOS_CFG_LOG_FS_CORE	File System's Core layer logging channel
FS, DRV	RTOS_CFG_LOG_FS_DRV	File System's Driver layer logging channel
FS, FAT	RTOS_CFG_LOG_FS_FAT	File System's FAT layer logging channel
FS, MEDIA	RTOS_CFG_LOG_FS_MEDIA	File System's Media layer logging channel
FS, STORAGE	RTOS_CFG_LOG_FS_STORAGE	File System's Storage layer logging channel
KERNEL	RTOS_CFG_LOG_KERNEL	Kernel's logging channel
NET	RTOS_CFG_LOG_NET	Network's root logging channel
NET, DNS	RTOS_CFG_LOG_NET_DNS	Network's DNS Application logging channel
NET, DHCP	RTOS_CFG_LOG_NET_DHCP	Network's DHCP Application logging channel
NET, FTP	RTOS_CFG_LOG_NET_FTP	Network's FTP Application logging channel
NET, HTTP	RTOS_CFG_LOG_NET_HTTP	Network's HTTP Application logging channel
NET, IPERF	RTOS_CFG_LOG_NET_IPERF	Network's IPerf Application logging channel

Comma notation (as seen in source code files)	#define used to configure the channel	Description
NET, MQTT	RTOS_CFG_LOG_NET_MQTT	Network's MQTT Application logging channel
NET, SMTP	RTOS_CFG_LOG_NET_SMTP	Network's SMTP Application logging channel
NET, SNTP	RTOS_CFG_LOG_NET_SNTP	Network's SNTP Application logging channel
NET, SSL	RTOS_CFG_LOG_NET_SSL	Network's SSL layer logging channel
USBD	RTOS_CFG_LOG_USBD	USB Device's root logging channel
USBD, CLASS	RTOS *CFG_LOG* USBD_CLASS	USB Device's Classes logging channel
USBD, CLASS, AUDIO	RTOS_CFG_LOG_USBD_CLASS_AUDIO	USB Device's Audio Class logging channel
USBD, CLASS, CDC, ACM	RTOS_CFG_LOG_USBD_CLASS_CDC_ACM	USB Device's CDC-ACM logging channel
USBD, CLASS, CDC, EEM	RTOS_CFG_LOG_USBD_CLASS_CDC_EEM	USB Device's CDC-EEM logging channel
USBD, CLASS, HID	RTOS_CFG_LOG_USBD_CLASS_HID	USB Device's HID Class logging channel
USBD, CLASS, MSD	RTOS_CFG_LOG_USBD_CLASS_MSC	USB Device's MSC Class logging channel
USBD, CLASS, VENDOR	RTOS_CFG_LOG_USBD_CLASS_VENDOR	USB Device's Vendor Class logging channel
USBD, DRV	RTOS_CFG_LOG_USBD_DRV	USB Device's Driver layer logging channel
USBH	RTOS_CFG_LOG_USBH	USB Host's root logging channel
USBH, CLASS	RTOS_CFG_LOG_USBH_CLASS	USB Host's Classes logging channel
USBH, CLASS, AOAP	RTOS_CFG_LOG_USBH_CLASS_AOAP	USB Host's AOAP Class logging channel
USBH, CLASS, CDC	RTOS_CFG_LOG_USBH_CLASS_CDC	USB Host's CDC Class logging channel
USBH, CLASS, HID	RTOS_CFG_LOG_USBH_CLASS_HID	USB Host's HID Class logging channel
USBH, CLASS, MSD	RTOS_CFG_LOG_USBH_CLASS_MSC	USB Host's MSC Class logging channel
USBH, CLASS, USB2SER	RTOS_CFG_LOG_USBH_CLASS_USB2SER	USB Host's USB2Ser Class logging channel
USBH, DEV	RTOS_CFG_LOG_USBH_DEV	USB Host's Device management layer logging channel
USBH, HCD	RTOS_CFG_LOG_USBH_HCD	USB Host's Host Controller Driver layer logging channel
USBH, PBHCD	RTOS_CFG_LOG_USBH_PBHCD	USB Host's Pipe-Based Host Controller Driver layer logging channel

Common Compile-Time Configuration

- Lib Module
  - Lib Memory Configuration
  - Lib String Configuration

The following configuration constants are located in the common_cfg.h configuration file.

Lib Module

Lib Memory Configuration

Table - Lib Mem Optimization Configuration Constants

Constant	Description	Possible values
LIB_MEM_CFG_STD_C_LIB_EN	Configures LIB_MEM_CFG_STD_C_LIB_EN to enable/disable use of standard C lib (will include standard <string.h>) for the following functions: • Mem_Set()/memset() • Mem_Copy()/memcpy() • Mem_Move()/memmove() • Mem_Clr()/memset() • Mem_Cmp()/memcmp()	1 (enabled) or **0** (disabled).
LIB_MEM_CFG_MEM_COPY_OPTIMIZE_ASM_EN	Configure LIB_MEM_CFG_MEM_COPY_OPTIMIZE_ASM_EN to enable/disable assembly-optimized memory copy function.	1 (enabled) or **0** (disabled).

Table - Lib Mem Allocation Configuration Constants

Constant	Description	Possible values
LIB_MEM_CFG_DBG_INFO_EN	Configures LIB_MEM_CFG_DBG_INFO_EN to enable/disable memory allocation usage tracking that associates a name with each segment or dynamic pool allocated.	1 (enabled) or **0** (disabled).
LIB_MEM_CFG_HEAP_SIZE	Configures the desired size of the heap memory. Set to 0 to disable heap allocation features.	0 to 2^(sizeof(CPU_SIZE_T) * 8). Default value is **9216u.**
LIB_MEM_CFG_HEAP_PADDING_ALIGN	Configures the desired size of padding alignment of each buffer allocated from the heap. Particularly useful with systems that have cache memory, to avoid having more than one buffer on a given cache line. This configuration is required only if LIB_MEM_CFG_HEAP_SIZE is higher than 0.	**1** Any power of 2 values (1, 2, 4, 8, 16, 32, ...).
LIB_MEM_CFG_HEAP_BASE_ADDR	Configures the heap memory base address location. Not defining this will cause the heap to be allocated from a standard C buffer. This configuration is used only if LIB_MEM_CFG_HEAP_SIZE is greater than 0.	**Undefined**, 1 to 2^(sizeof(CPU_DATA) * 8).

Lib String Configuration

Table - Lib Str Configuration Constants

Constant	Description	Possible values
LIB_STR_CFG_FP_EN	Enables/disables the floating point to string functions.	1 (enabled) or **0** (disabled).
LIB_STR_CFG_FP_MAX_NBR_DIG_SIG	Configures the maximum number of significant digits to calculate and/or display for floating point string function(s). This configuration is required only if LIB_STR_CFG_FP_EN is set to DEF_ENABLED.	1 to 9. Default value is **LIB_STR_FP_MAX_NBR_DIG_SIG_DFLT**

RTOS_ERR Type Configuration

The following configuration constants are located in the rtos_err_cfg.h file.

Table - RTOS_ERR Type Configuration Constants

Constant	Description	Possible values
RTOS_ERR_CFG_EXT_EN	RTOS_ERR_CFG_EXT_EN allows to configure whether the RTOS_ERR type contains only an error code (0) or contains more debug information (1). If set to 1, a string containing the file name and line number where the error has been set and also the function name (if compiling in C99) will be included in the RTOS_ERR type. Setting this configuration to 1 may have an impact on performance and resource usage **Recommendation**: set to 0 once development is complete.	1 (enabled) or **0** (disabled)
RTOS_ERR_CFG_STR_EN	RTOS_ERR_CFG_STR_EN allows strings to be associated with each error code. If set to 0, the error code enum value will be outputted instead. For example, if set to 1, it would be possible to print RTOS_ERR_NONE or RTOS_ERR_INVALID_ARG as a string instead of printing the numerical value associated, which would be 0 for RTOS_ERR_NONE and higher than 0 for RTOS_ERR_INVALID_ARG.	1 (enabled) or **0** (disabled)

Common Run-Time Configuration

- General Configuration
  - Memory Segment
  - Logging Configuration Structure
  - Logging Memory Segment
- Authentication
  - Root User Configuration Structure
  - Resource Usage Configuration Structure
  - Memory Segment
- Shell
  - Command Usage
  - Memory Segment
- Configuration Summary
  - Common
  - Authentication

This section describes the application-specific configurations of the common module. These are specified at run-time.

- For more information on how default run-time configuration values can be changed or used, see Initialization of the Common Module .
- For more information on how to initialize any Micrium product, see Stacks Initialization Methods .

The following sections describe the configurations that are optional. If you do not set them in your application, the default configurations will apply.

General Configuration

Default values can be retrieved via the Common_InitCfgDflt structure.

> **Note that these configurations must be set *before* you call the function Common_Init().**

The following tables describe the configuration structure's fields, their default values and the functions used to change them. For the advanced method, it is assumed that a COMMON_INIT_CFG structure global variable called Common_InitCfg is defined by the application.

Memory Segment

Pointer to the memory segment from which the Common module's data will be allocated.

Type	Function to Call	Default	Default Configuration Structure Field
MEM_SEG*	Common_ConfigureMemSeg()	See the General-purpose heap .	.CommonMemSegPtr

Logging Configuration Structure

Only present if the Logging module is enabled, by setting RTOS_CFG_LOG_EN to DEF_ENABLED, in rtos_cfg.h. For more details about logging configuration, see Logging Usage .

Configuration structure for the Logging module.

Type	Function to Call	Default	Default Configuration Structure Field
COMMON_CFG_LOGGING	Common_ConfigureLogging()	See structure's description.	.LoggingCfg

Logging Memory Segment

Only present if the Logging module is enabled, by setting RTOS_CFG_LOG_EN to DEF_ENABLED in rtos_cfg.h . For more details about logging configuration, see Logging Usage .

Pointer to memory segment from which the Logging module's data will be allocated, including the ring buffer, if the size is not 0.

Type	Function to Call	Default	Default Configuration Structure Field
MEM_SEG*	Common_ConfigureMemSegLogging()	The General-purpose heap .	.LoggingMemSegPtr

Authentication

The default values can be retrieved via the structure Auth_InitCfgDflt.

Note that these configurations must be set *before* you call the function Auth_Init().

The following tables describe the configuration structure's fields, their default values and the functions used to change them. When using the advanced method, the application defines the AUTH_INIT_CFG structure global variable called Auth_InitCfg.

Root User Configuration Structure

Configuration structure for the Root User.

Type	Function to Call	Default	Default Configuration Structure Field
AUTH_CFG_ROOT_USER	Auth_ConfigureRootUser()	See structure's description.	.RootUserCfg

Resource Usage Configuration Structure

Configuration structure for the Resource usage.

Type	Function to Call	Default	Default Configuration Structure Field
AUTH_CFG_RESOURCE	Auth_ConfigureResource()	See structure's description.	.ResourceCfg

Memory Segment

Pointer to memory segment from which Authentication module's data will be allocated.

Type	Function to Call	Default	Default Configuration Structure Field
MEM_SEG*	Auth_ConfigureMemSeg()	The General-purpose heap .	.MemSegPtr

Shell

The following configurations will only be present if the Shell module is available.

The default values can be retrieved via the structure Shell_InitCfgDflt.

Note that these configurations must be set *before* you call the function Shell_Init().

The following tables describe the configuration structure's fields, their default values, and the functions that modify them. If the advanced method is used and Shell_Init() is called by the application, the application defines a SHELL_INIT_CFG structure global variable called Shell_InitCfg.

Command Usage

Configuration structure for the commands that will be registered in the Shell sub-module

Type	Function to Call	Default	Default Configuration Structure Field
SHELL_CFG_CMD_USAGE	Shell_ConfigureCmdUsage()	See the structure's description.	.CfgCmdUsage

Memory Segment

Pointer to the memory segment from which the Shell module's data will be allocated.

Type	Function to Call	Default	Default Configuration Structure Field
MEM_SEG*	Shell_ConfigureMemSeg()	See the General-purpose heap .	.MemSegPtr

Configuration Summary

The table below provides a summary of the configurations functions for the core and every sub-module.

Common

Operation	Functions
<Module>_Init() Function	Common_Init()
If standard method used: <Module>_Configure...() functions	Common_ConfigureLogging() Common_ConfigureMemSegLogging() Common_ConfigureMemSeg()
If standard method used: Name and location of the default configuration structure	Common_InitCfgDflt Found in rtos/include/common.h
If advanced method used: Name of the <MODULE>_INIT_CFG-type structure that must be defined	const COMMON_INIT_CFG Common_InitCfg
Post-initialization 'Set' functions, if any.	N/A

Authentication

Operation	Functions
<Module>_Init() Function	Auth_Init()
If standard method used: <Module>_Configure...() functions	Auth_ConfigureRootUser() Auth_ConfigureResource() Auth_ConfigureMemSeg()
If standard method used: Name and location of the default configuration structure	Auth_InitCfgDflt Found in rtos/include/auth.h
If advanced method used: Name of the <MODULE>_INIT_CFG-type structure that must be defined	const AUTH_INIT_CFG Auth_InitCfg
Post-initialization 'Set' functions, if any.	N/A

Shell

Operation	Functions
<Module>_Init() Function	Shell_Init()
If standard method used: <Module>_Configure...() functions	Shell_ConfigureCmdUsage() Shell_ConfigureMemSeg()
If standard method used: Name and location of the default configuration structure	Shell_InitCfgDflt Found in rtos/include/shell.h
If advanced method used: Name of the <MODULE>_INIT_CFG-type structure that must be defined	const SHELL_INIT_CFG Shell_InitCfg
Post-initialization 'Set' functions, if any.	N/A

Authentication Run-Time Application-Specific Configurations

- AUTH_CFG_ROOT_USER
- AUTH_CFG_RESOURCE

The following structures contain values that configure the authentication sub-module, which is part of the Common module. This module is optional.

AUTH_CFG_ROOT_USER

AUTH_CFG_ROOT_USER is a structure that can be used to configure the root user's properties.

Table - AUTH_CFG_ROOT_USER configuration structure in the *Authentication Run-Time Application-Specific Configurations* page describes each configuration field available for this configuration structure.

Table - AUTH_CFG_ROOT_USER configuration structure

Field	Description	Possible values
.RootUserName	Name of the root user. Must be null-terminated.	Any valid string, "admin" by default.
.RootUserPwd	Password for the root user. Must be null-terminated.	Any valid string, "admin" by default.

AUTH_CFG_RESOURCE

AUTH_CFG_RESOURCE is a structure that can be used to configure the various resources used by the authentication sub-module.

Table - AUTH_CFG_RESOURCE configuration structure in the *Authentication Run-Time Application-Specific Configurations* page describes each configuration field available for this configuration structure.

Table - AUTH_CFG_RESOURCE configuration structure

Field	Description	Possible values
.NbUserMax	Maximum number of users.	Any valid 8-bit integer, 4 u by default.
.NameLenMax	Maximum length of user names.	Any valid 8-bit integer, 10u by default.
.PwdLenMax	Maximum length of user passwords.	Any valid 8-bit integer, 10u by default.

Logging Run-Time Application-Specific Configuration

COMMON_CFG_LOGGING is a structure that can be used to configure the buffer sizes for the logging sub-module.

Table - COMMON_CFG_LOGGING structure in the *Logging Run-Time Application-Specific Configuration* page describes each field available in this configuration structure.

Table - COMMON_CFG_LOGGING structure

Field	Description	Possible values
.AsyncBufSize	Size of the ring buffer, in bytes, used for asynchronous logging. Setting this value to 0 u will disable support for asynchronous logging, as no ring buffer will be available.	Any 16-bit integer, 512 u by default.

Shell Run-Time Application-Specific Configurations

SHELL_CFG_CMD_USAGE is a structure that can be used to configure the various quantity and buffer sizes used by the Shell sub-module.

Table - SHELL_CFG_CMD_USAGE configuration structure in the *Shell Run-Time Application-Specific Configurations* page describes each field available for this configuration structure.

Table - SHELL_CFG_CMD_USAGE configuration structure

Field	Description	Possible values
Field	Description	Possible values
.CmdTblItemNbrInit	Initial number of command items allocated for the command table.	Any 16-bit integer, 10 u by default.
.CmdTblItemNbrMax	Maximum number of command items allocated for the command table.	Any 16-bit integer, 10 u by default.
.CmdArgNbrMax	Maximum number of arguments for shell commands.	Any 16-bit integer, 10 u by default.
.CmdNameLenMax	Maximum length of a shell command, including terminating NULL character.	Any 16-bit integer, 10 u by default.

Common Programming Guide

Common Programming Guide

The following pages discuss the use of every sub-module in Common. This includes information about memory allocation, logging, error management, use of the shell or authentication sub-modules, etc.

- Initialization of the Common Module
- Authentication Module Programming Guide
- Shell Module Programming Guide
- Toolchain Abstraction Utilities
- Logging Usage
- RTOS ERR Programming Guide
- Micrium OS Asserts Programming Guide
- LIB Constants, Defines and Macros Guide
- Memory Allocation Guide

Initialization of the Common Module

- Common
- Shell
- Authentication
- Initialization Example

Recommended: Read these additional pages to understand the details of how Common can be configured and initialized:

- The Common module uses the initialization and configuration method described in Stacks Initialization Methods .
- The configuration of the Common module and of its sub-module is explained in Common Run-Time Configuration .

The Common module is composed of several sub-modules. The initialization of the Common core is always required, but the other initializations (Shell and Authentication described below) are optional, and need to be performed only if your application requires the use of these sub-modules.

Common

The Common module is initialized by calling Common_Init(). This function initializes common internal sub-modules, such as LIB and logging (if present).

Shell

The Shell sub-module is initialized by calling Shell_Init(). If the Shell sub-module is present, Shell_Init() must be called before Auth_Init().

Authentication

The Authentication sub-module is initialized by calling Auth_Init().

Initialization Example

The example below shows how to initialize Common and all of its sub-modules.

Listing - Initialization of Common and its sub-modules

```
void Ex_CommonInit (void)
{
    RTOS_ERR err;

    Common_Init(&err);
    APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);

#ifdef RTOS_MODULE_COMMON_SHELL_AVAIL
    Shell_Init(&err);
    APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);
#endif

    Auth_Init(&err);
    APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);
}
```

Authentication Module Programming Guide

- Create a User Profile
- Validate a User Name/Password Combination
  - Get a User Without Validating Its Credentials
- Rights
  - Rights Definition

Create a User Profile

To create a user profile, call the Auth_CreateUser() function with a user name and password combination. The new user profile has no rights by default, but you can add them later using the Auth_GrantRight()function. The following example shows how to create a user profile.

Listing - Ex_AuthCreate

```
void Ex_AuthCreate (void)
{
    AUTH_USER_HANDLE new_user_handle;
    RTOS_ERR         err;

    new_user_handle = Auth_CreateUser("UserA", "PwdA", &err);
    APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);

    PP_UNUSED_PARAM(new_user_handle);
}
```

Validate a User Name/Password Combination

To obtain a user's handle and confirm that the credentials are valid, use the Auth_ValidateCredentials() function. This function compares the submitted credentials against the list of existing user profiles to find a match. Once the match is found, the function returns the associated user handle.

If the user handle has the rights, use this user handle to grant or revoke a right for another user. You can also use it to proceed with the next action the user was trying to take.

Listing - Ex_AuthValidate

```
void  Ex_AuthValidate (void)
{
    AUTH_USER_HANDLE  user_handle;
    RTOS_ERR          err;

    (void)Auth_CreateUser("UserB", "PwdB", &err);
    APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);

    user_handle = Auth_ValidateCredentials("UserB", "Pwd0", &err);
    if (err.Code == RTOS_ERR_INVALID_CREDENTIALS) {
        /* Invalid user name/password combination. */
        /* At this point, we would normally return or indicate an error. */
    } else if (err.Code != RTOS_ERR_NONE) {
        /* Handle error. */
    }

    user_handle = Auth_ValidateCredentials("UserB", "PwdB", &err);
    if (err.Code == RTOS_ERR_INVALID_CREDENTIALS) {
        /* Invalid user name/password combination. */
    } else if (err.Code != RTOS_ERR_NONE) {
        /* Handle error. */
    }
    /* If credentials were good, we may continue and use the 'user_handle' obtained. */

    PP_UNUSED_PARAM(user_handle);
}
```

Get a User Without Validating Its Credentials

You can obtain a user's handle without validating the credentials by using the Auth_GetUser() function instead of Auth_ValidateCredentials(). Once you recover the user handle, it only indicates that the user exists, but does not display any associated rights.

Listing - Ex_AuthGet

```
void  Ex_AuthGet (void)
{
    AUTH_USER_HANDLE  user_a_unvalidated_handle;
    RTOS_ERR          err;

                                                         /* The user must have been created before,  */
    user_a_unvalidated_handle = Auth_GetUser("UserA", &err);    /* for this call to be successful.          */
    APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);

                                                         /* If obtained, this merely means           */
    PP_UNUSED_PARAM(user_a_unvalidated_handle);           /* that the user exists.                    */
}
```

Rights

To grant or revoke rights to a user handle, use the Auth_GrantRight() and Auth_RevokeRight() functions respectively. Before you can call these two functions, you need to retrieve two user handles.

- The first user handle is for the user to which you want to grant or revoke a right. In this case, you do not normally need to have that user's password.
- The second user handle is for the user that wishes to grant or revoke the right. This user handle must either be a root user, have admin rights, or have granting/revoking rights. Verify this second user's credentials to prevent anyone from knowing only that user's name to grant rights to another user.

To check if a given user has a particular right, use the Auth_GetUserRight() function.

The following example shows how to grant rights to a user handle as an Admin. Typically, the user name and password come from an external source (such as typing it in or from a shell), but to simplify this example, the user name/password are

directly included. The example also checks if the user has the right before the operation (no rights given) and after the operation (rights now given).

Listing - Ex_AuthRights

```
void  Ex_AuthRights (void)
{
    AUTH_USER_HANDLE  user_a_unvalidated_handle;
    AUTH_USER_HANDLE  admin_validated_handle;
    AUTH_RIGHT        right;
    RTOS_ERR          err;

                                                            /* The user must have been created before,  */
    user_a_unvalidated_handle = Auth_GetUser("UserA", &err);    /* for this call to be successful.          */
    APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);
    /* If obtained, this only means that the user exists. */

    admin_validated_handle = Auth_ValidateCredentials("admin", "admin", &err);
    if (err.Code == RTOS_ERR_INVALID_CREDENTIALS) {
        /* Invalid user name/password combination. */
        /* At this point, we would normally return or indicate an error. */
    } else if (err.Code != RTOS_ERR_NONE) {
        /* Handle error. */
    }
    /* If credentials were good, we may continue and use the 'user_handle' obtained. */

    right = Auth_GetUserRight(user_a_unvalidated_handle, &err);
    APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);
    /* At this point, 'right' does not contain any bit set, since no right has been granted to that user. */

    Auth_GrantRight((AUTH_RIGHT_6 | AUTH_RIGHT_8), user_a_unvalidated_handle, admin_validated_handle, &err);
    if (err.Code == RTOS_ERR_PERMISSION) {
        /* This would mean that the 'as_user_handle' does not have the right to add. */
        /* At this point, we would normally return or indicate an error. */
    } else if (err.Code != RTOS_ERR_NONE) {
        /* Handle error. */
    }
    /* If no error, rights 6 and 8 were added, let's confirm. */

    right = Auth_GetUserRight(user_a_unvalidated_handle, &err);
    APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);
    /* At this point, 'right' has the AUTH_RIGHT_6 and AUTH_RIGHT_8 set,   */
    /* since they have been granted to that user.                          */

    Auth_RevokeRight(AUTH_RIGHT_6, user_a_unvalidated_handle, admin_validated_handle, &err);
    if (err.Code == RTOS_ERR_PERMISSION) {
        /* This would mean that the 'as_user_handle' does not have the right to revoke. */
        /* At this point, we would normally return or indicate an error.                */
    } else if (err.Code != RTOS_ERR_NONE) {
        /* Handle error. */
    }
    /* If no error, right 6 was revoked. */

    right = Auth_GetUserRight(user_a_unvalidated_handle, &err);
    APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);

    /* At this point, 'right' has ONLY the AUTH_RIGHT_8 set, since AUTH_RIGHT_6 has been revoked. */

    PP_UNUSED_PARAM(right);
}
```

Rights Definition

An application may define its own rights as follows:

Listing - Defining authentication right

```
#include  <rtos/include/common/auth.h>                        (1)

#define  APP_FILE_READ_RIGHT      AUTH_RIGHT_0                          (2)
#define  APP_FILE_WRITE_RIGHT    (AUTH_RIGHT_1 | APP_FILE_READ_RIGHT)       (3)
#define  APP_FILE_DELETE_RIGHT  (AUTH_RIGHT_2 | APP_FILE_WRITE_RIGHT)       (4)
```

(1) Include the auth.h file.

(2) The READ right can only read and is defined as Right 0.

(3) The WRITE right can write and read, it is defined as Right 0 and Right 1.

(3) The DELETE right can delete, write and read, it is defined as Right 0, Right 1 and Right 2.

If two distinct modules use the same right numbers (AUTH_RIGHT_xx), this can cause a conflict. If a user does not have the right to execute an operation, but has an "equivalent" right from another module, the authentication module will not be able to distinguish between both rights. However, this may mean that the module may allow the user to have this right if they share the same right's value. To prevent this from happening, each module should have its own AUTH_RIGHT_xx fields, without sharing them with another module.

Shell Module Programming Guide

- Executing a Command
- Commands, Callbacks and Data Types
  - Shell Commands Function
  - Output Function
- Adding a Command Table to Shell
- Removing a Command Table From the Shell Module

Executing a Command

Modules in need of a shell facility (such as TELNETs) interact with it using an application callback function. From the caller's point of view, once the commands have been developed and the initialization performed, all that is needed is to call the Shell execution function: Shell_Exec() .

This function parses the in parameter, which is a null-terminated string containing a complete command line (command name, followed by possible arguments separated by spaces). For example:

`"App_Test –a –b –c readme.txt"`

Once the command name and its arguments have been extracted, Shell searches in its command tables for a command match (in this case, App_Test is the name of the command), and invokes it. Note that the Shell_Exec() requires the in parameter to be a modifiable string, not a const one. The function also has an out_fnct argument, which is a pointer to a callback that handles the details of responding to the requester. In other words, if called by TELNETs, then TELNETs has to provide the details of the response; if called by a UART, the UART should handle the response. Finally, the p_cmd_param is a pointer to a structure containing additional parameters for the command to use. The example below shows how to call this function.

Listing - Executing a command via Shell_Exec()

```
void  Ex_CommonShellExec (void)
{
    CPU_INT16S       ret_val;
    SHELL_CMD_PARAM  cmd_param;
    CPU_CHAR         cmd_buf[25u];
    RTOS_ERR         err;

    Str_Copy(cmd_buf, "help");                   /* String passed to Shell_Exec must not be const.       */

                                                 /* Call Shell_Exec() with:                              */
    ret_val = Shell_Exec(cmd_buf,                /*     string containing the cmd to exec,               */
                         Ex_CommonShellExecOutFnct,  /*     function to use to output,                   */
                         &cmd_param,             /*     a cmd param structure that could be used by cmd.*/
                         &err);
    APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);
    APP_RTOS_ASSERT_CRITICAL(ret_val == SHELL_EXEC_ERR_NONE, ;);
}
```

Commands, Callbacks and Data Types

Shell Commands Function

Shell commands (that is, commands added to the shell module) are of the following prototype:

Listing - Shell Command Example

```
CPU_INT16S  MyShellCmd (CPU_INT16U        argc,
                        CPU_CHAR         *argv[],
                        SHELL_OUT_FNCT    out_fnct,
                        SHELL_CMD_PARAM  *p_cmd_param);
```

Based on the SHELL_CMD_FNCT type:

Listing - Shell Commands Type

```
typedef  CPU_INT16S  (*SHELL_CMD_FNCT)(CPU_INT16U        argc,
                                       CPU_CHAR         *argv[],
                                       SHELL_OUT_FNCT    out_fnct,
                                       SHELL_CMD_PARAM  *p_cmd_param);
```

Where:

- argc is a count of the arguments supplied.
- argv is an array of pointers to the strings which are those arguments.
- out_fnct is a function pointer that outputs any information requested.
- p_cmd_param passes additional information to the command.
- The return value is command-specific and will be returned by the Shell_Exec() call for this command. However, in case of an error, SHELL_EXEC_ERR should be returned.

Output Function

Each command is responsible for responding to its requester using the pointer to the output function parameter. This function uses the following prototype:

Listing - Shell Out Function Example

```
CPU_INT16S  MyShellOutFnct (CPU_CHAR    *p_buf,
                            CPU_INT16U   buf_len,
                            void        *p_opt);
```

Based on the SHELL_OUT_FNCT type:

Listing - Shell Commands Type

```
typedef CPU_INT16S (*SHELL_OUT_FNCT)(CPU_CHAR   *p_buf,
                                     CPU_INT16U  buf_len,
                                     void       *p_opt);
```

Where:

- `p_buf` is a pointer to a response buffer
- `buf_len` is the length of the response buffer
- `p_opt` is an optional argument used to provide implementation specific information (port number, UART identification, etc.)

The return value should be:

- The number of data octets transmitted, if NO error occurred
- `SHELL_OUT_RTN_CODE_CONN_CLOSED` if the link has been closed
- `SHELL_OUT_ERR` for any other error

The example below shows what could be a valid out_fnct:

Listing - Example of an out_fnct

```
static CPU_INT16S Ex_CommonShellExecOutFnct (CPU_CHAR   *p_buf,
                                             CPU_INT16U  buf_len,
                                             void       *p_opt)
{
   CPU_INT16U tx_len = buf_len;

   PP_UNUSED_PARAM(p_opt);                        /* Supplemental options not used in this command.  */

   while (tx_len != 0u) {
      putchar(*p_buf);                       /* Any putchar-like function could be used, here.  */
      tx_len--;
      p_buf++;
   }

   return (buf_len);                       /* Return nbr of tx'd characters.                  */
}

#endif /* RTOS_MODULE_COMMON_SHELL_AVAIL */
```

Adding a Command Table to Shell

To add a new command to Shell, follow these steps:

1. Make sure an output function of the proper type (SHELL_OUT_FNCT) can be used by future shell commands. This output function can be as simple as the one provided in the example above, or more complex if outputting is done via UART or TELNETs.
2. Write functions with the proper signature. The correct prototype is defined by the SHELL_CMD_FNCT type, as described above. Any data that needs to be outputted should be outputted via the out_fnct. The samples below provide example functions that implement a wrapper for LIB's random number generator (RNG).

Listing - Shell commands functions

```
static  CPU_INT16S  Ex_CommonShellRNG_Help (CPU_INT16U        argc,
                                            CPU_CHAR         *p_argv[],
                                            SHELL_OUT_FNCT    out_fnct,
                                            SHELL_CMD_PARAM  *p_cmd_param)
{
    SHELL_CMD   *p_shell_cmd;
    CPU_INT16S   ret_val;

    PP_UNUSED_PARAM(argc);
    PP_UNUSED_PARAM(p_argv);

                                                /* Iterate over all commands in cmd tbl.       */
    p_shell_cmd = Ex_CommonShellCmdAddCmdTbl;
    while (p_shell_cmd->Fnct != 0) {
                                                /* Output each cmd's name.                     */
        ret_val = out_fnct((CPU_CHAR *)p_shell_cmd->Name, Str_Len(p_shell_cmd->Name), p_cmd_param->OutputOptPtr);
        if ((ret_val == SHELL_OUT_RTN_CODE_CONN_CLOSED) ||
            (ret_val == SHELL_OUT_ERR)) {
            return (SHELL_EXEC_ERR);
        }

                                                /* Output new line.                            */
        ret_val = out_fnct((CPU_CHAR *)STR_NEW_LINE, STR_NEW_LINE_LEN, p_cmd_param->OutputOptPtr);
        if ((ret_val == SHELL_OUT_RTN_CODE_CONN_CLOSED) ||
            (ret_val == SHELL_OUT_ERR)) {
            return (SHELL_EXEC_ERR);
        }

        p_shell_cmd++;
    }

    return (SHELL_EXEC_ERR_NONE);
}
```

Listing - Shell commands functions

```
static  CPU_INT16S  Ex_CommonShellRNG_Seed (CPU_INT16U        argc,
                                            CPU_CHAR         *p_argv[],
                                            SHELL_OUT_FNCT    out_fnct,
                                            SHELL_CMD_PARAM  *p_cmd_param)
{
    CPU_INT16S  result_h;
    CPU_INT16S  result_help;
    RAND_NBR    seed;

    if (argc != 2) {                                 /* If not enough or too much args, display help. */
        (void)out_fnct((CPU_CHAR *)EX_COMMON_SHELL_RNG_SEED_INVALID_ARG,
                       Str_Len(EX_COMMON_SHELL_RNG_SEED_INVALID_ARG),
                       p_cmd_param->OutputOptPtr);

        (void)out_fnct((CPU_CHAR *)STR_NEW_LINE,
                       STR_NEW_LINE_LEN,
                       p_cmd_param->OutputOptPtr);

        (void)out_fnct((CPU_CHAR *)EX_COMMON_SHELL_RNG_SEED_HELP,
                       Str_Len(EX_COMMON_SHELL_RNG_SEED_HELP),
                       p_cmd_param->OutputOptPtr);

        (void)out_fnct((CPU_CHAR *)STR_NEW_LINE,
                       STR_NEW_LINE_LEN,
                       p_cmd_param->OutputOptPtr);

        return (SHELL_EXEC_ERR_NONE);
    }

    result_h = Str_Cmp(p_argv[1u], "-h");
    result_help = Str_Cmp(p_argv[1u], "--help");
    if ((result_h    == 0) ||
        (result_help == 0)) {                        /* Display help.                          */
        (void)out_fnct((CPU_CHAR *)EX_COMMON_SHELL_RNG_SEED_HELP,
                       Str_Len(EX_COMMON_SHELL_RNG_SEED_HELP),
                       p_cmd_param->OutputOptPtr);

        (void)out_fnct((CPU_CHAR *)STR_NEW_LINE,
                       STR_NEW_LINE_LEN,
                       p_cmd_param->OutputOptPtr);

        return (SHELL_EXEC_ERR_NONE);
    }

                                                     /* Convert string number to int.          */
    seed = (RAND_NBR)Str_ParseNbr_Int32U((const  CPU_CHAR *)p_argv[1u],
                                         DEF_NULL,
                                         10u);

    Math_RandSetSeed(seed);

    return (SHELL_EXEC_ERR_NONE);
}
```

Listing - Shell commands functions

```
static  CPU_INT16S  Ex_CommonShellRNG_Get  (CPU_INT16U        argc,
                                            CPU_CHAR         *p_argv[],
                                            SHELL_OUT_FNCT    out_fnct,
                                            SHELL_CMD_PARAM  *p_cmd_param)
{
    RAND_NBR    rand;
    CPU_INT16S  ret_val;
    CPU_CHAR    rand_str_buf[DEF_INT_32U_NBR_DIG_MAX + 1u];

    PP_UNUSED_PARAM(argc);
    PP_UNUSED_PARAM(p_argv);

    rand = Math_Rand();                                 /* Obtain random nbr from LIB Math module.  */

                                                        /* Convert int decimal number to str.       */
    (void)Str_FmtNbr_Int32U(rand,
                            DEF_INT_32U_NBR_DIG_MAX,
                            DEF_NBR_BASE_DEC,
                            '\0',
                            DEF_NO,
                            DEF_YES,
                            &rand_str_buf[0u]);

                                                        /* Output random number obtained.           */
    ret_val = out_fnct(&rand_str_buf[0u], Str_Len(&rand_str_buf[0u]), p_cmd_param->OutputOptPtr);
    if ((ret_val == SHELL_OUT_RTN_CODE_CONN_CLOSED) ||
        (ret_val == SHELL_OUT_ERR)) {
        return (SHELL_EXEC_ERR);
    }

                                                        /* Output new line.                         */
    ret_val = out_fnct((CPU_CHAR *)STR_NEW_LINE, STR_NEW_LINE_LEN, p_cmd_param->OutputOptPtr);
    if ((ret_val == SHELL_OUT_RTN_CODE_CONN_CLOSED) ||
        (ret_val == SHELL_OUT_ERR)) {
        return (SHELL_EXEC_ERR);
    }

    return (SHELL_EXEC_ERR_NONE);
}

#endif  /* RTOS_MODULE_COMMON_SHELL_AVAIL */
```

1. Once these functions have been correctly implemented and validated, build a Shell command table. This table associates a name to the functions created above. This table must be NULL-terminated by adding an empty entry at the end of the table. The example below shows how this is done for the three RNG functions.

Listing - Shell commands table

```
static  SHELL_CMD  Ex_CommonShellCmdAddCmdTbl[] =
{
    {EX_COMMON_SHELL_CMD_NAME_HELP, Ex_CommonShellRNG_Help},    /* Associate cmd str with cmd fnct.          */
    {EX_COMMON_SHELL_CMD_NAME_SET,  Ex_CommonShellRNG_Seed},
    {EX_COMMON_SHELL_CMD_NAME_GET,  Ex_CommonShellRNG_Get},
    {0,                 0}                                  /* Tbl is NULL-terminated to indicate end.    */
};
```

1. Before these commands can be called via Shell_Exec(), call Shell_CmdTblAdd() to register them to Shell. The example below illustrates how this is done.

Listing - Shell commands table

```
void  Ex_CommonShellCmdAdd (void)
{
    RTOS_ERR  err;

                                                /* Add cmds in Cmd Tbl to Shell's available cmds. */
    Shell_CmdTblAdd((CPU_CHAR *)EX_COMMON_SHELL_CMD_TBL_NAME,
                    Ex_CommonShellCmdAddCmdTbl,
                    &err);
    APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);
}
```

The table name should be the same as the commands' prefix, which is rng, in our case.

At this point, the commands are ready to be used, just like any other Shell command.

Removing a Command Table From the Shell Module

To remove a command table from the Shell module, call the Shell_CmdTblRem() function with the command table name that must be removed.

Toolchain Abstraction Utilities

- C Standard Version Detection
- Unused Parameter

The Common module offers some degree of abstraction for toolchain-specific operations such as:

- Detecting which version of the C standard is used during compilation
- Flagging any unused parameters in a function to avoid compiler warnings
- Specifying an explicit memory alignment for a given variable

To make use of the toolchain abstraction utilities, include the file rtos/common/include/toolchains.h.

C Standard Version Detection

To detect which version of the C standard is used at compile time, you can do one of the following:

- Check if specified #defines have been defined or not
- Check the PP_C_STD_VERSION define value

The code snippets below show how to use either method.

Listing - C standard version detection

```
void  Ex_CommonToolchainC_Version (void)
{
#ifdef  PP_C_STD_VERSION_C89_PRESENT
    /* This part of the code will be executed only if the C standard version used to compile is at least C89. */
    EX_TRACE("Compiling with at least C89 standard version.\r\n");
#endif

#ifdef  PP_C_STD_VERSION_C90_PRESENT
    /* This part of the code will be executed only if the C standard version used to compile is at least C90. */
    EX_TRACE("Compiling with at least C90 standard version.\r\n");
#endif

#ifdef  PP_C_STD_VERSION_C94_PRESENT
    /* This part of the code will be executed only if the C standard version used to compile is at least C94. */
    EX_TRACE("Compiling with at least C94 standard version.\r\n");
#endif

#ifdef  PP_C_STD_VERSION_C99_PRESENT
    /* This part of the code will be executed only if the C standard version used to compile is at least C99. */
    EX_TRACE("Compiling with at least C99 standard version.\r\n");
#endif

#if (PP_C_STD_VERSION == PP_C_STD_VERSION_C89)
    /* This part of the code will be executed only if the C standard version used to compile is C89. */
    EX_TRACE("Compiling with the C89 standard version.\r\n");
#endif

#if (PP_C_STD_VERSION == PP_C_STD_VERSION_C90)
    /* This part of the code will be executed only if the C standard version used to compile is C90. */
    EX_TRACE("Compiling with the C90 standard version.\r\n");
#endif

#if (PP_C_STD_VERSION == PP_C_STD_VERSION_C94)
    /* This part of the code will be executed only if the C standard version used to compile is C94. */
    EX_TRACE("Compiling with the C94 standard version.\r\n");
#endif

#if (PP_C_STD_VERSION == PP_C_STD_VERSION_C99)
    /* This part of the code will be executed only if the C standard version used to compile is C99. */
    EX_TRACE("Compiling with the C99 standard version.\r\n");
#endif
}
```

Unused Parameter

The PP_UNUSED_PARAM macro allows you to suppress any warnings associated with an unused function parameter. It is often used in template functions, or in functions that require a strict prototype but where some parameters are not useful.

The example below illustrates how to use the macro.

Listing - Removing warnings for unused function parameter

```
void  Ex_CommonToolchainUnusedParam (CPU_INT08U  unused_param)
{
    PP_UNUSED_PARAM(unused_param);                    /* Use macro to show unused param and remove warning. */

    /* [...] */

    return;
}
```

Logging Usage

- Understanding Asynchronous Logging and Ring Buffer Details
- Outputting Asynchronous Logging Entries

Understanding Asynchronous Logging and Ring Buffer Details

When a logging channel has been set to output its entries asynchronously, none of these entries will be outputted automatically via the output function specified for that given channel.

Instead, the LOG_xxx call saves the logging entries and their information in an internal ring buffer (its size is specified during the initialization). The ring buffer mechanism allows the logging module to keep the recent log entries and discard the older ones. All the channels that are outputting asynchronously share the same ring buffer, which means that a certain channel entry may be overwritten by another channel's newer entry.

The advantage of using asynchronous mode is that logging calls have a smaller impact on the program flow than in synchronous mode. The ring buffer must either be large enough to hold several logging entries, or if every entry must be kept and outputted, it must output them periodically. Since the ring buffer always keeps the recent entries, it is also possible to not output any entry. The exception to this is when an unexpected event occurs, or if an error is detected, or if an operation did not execute as expected.

Outputting Asynchronous Logging Entries

To output all the asynchronous entries that are currently in the internal ring buffer of the logging module, you must call the Log_Output() function. You can call this function in a very low-priority task (just higher than the Idle task, if present), which will execute only when no other tasks need to do anything. By calling this function in this way, there will be less of a delay for critical application tasks.

You can also call Log_Output() from the specific location where a known problem has occurred.

In the low-priority task, you can call Log_Output() continuously (see the example below). However, if this is done in the Idle task hook itself of the kernel, this method may prevent any kind of powered-down state to be reached if there is something to output.

You can also periodically call Log_Output() by adding a minimal delay after each call, to ensure that it does not run too often.

Listing - Call to Log_Output() in task

```
void  App_LoggingTaskFnct (void  *p_arg)
{
   PP_UNUSED_PARAM(p_arg);

   while (DEF_TRUE) {
      Log_Output();
   }
}
```

Also, the Log_DataIsAvail() function can indicate if there is any data available to output. This function can be called in the Idle task hook, and if there is anything that needs to be outputted, a semaphore can be posted so that the logging output task can be readied and call Log_Output() once it is allowed to run. The following example illustrates how to do this, assuming the semaphore has already been created successfully.

First, ensure that the OS_CFG_APP_HOOKS_EN kernel configuration constant is set to DEF_ENABLED and to declare the hook needed:

Listing - Declare Idle task hook

```
void  App_IdleTaskHookFnct (void);
```

You must set App_IdleTaskHookFnct as the OS_AppIdleTaskHookPtr, after calling OSInit() and before calling OSStart(). For more information on this method, see the Kernel Documentation page .

Once done, the hook will be called each time the Idle task is entered, which results in being able to call Log_DataIsAvail().

Listing - Call to Log_DataIsAvail() in Idle task hook

```
void  App_IdleTaskHookFnct (void)
{
    CPU_BOOLEAN  data_is_avail;
    RTOS_ERR     err;

    data_is_avail = Log_DataIsAvail();
    if (data_is_avail) {
        OSSemPost(&App_LoggingSem, OS_OPT_POST_1, &err);
        if (RTOS_ERR_CODE_GET(err) != RTOS_ERR_NONE) {
            /* Handle error. */
        }
    }
}
```

In the logging output task, pend on the semaphore and call Log_Output() whenever the semaphore is posted, before re-pending on it.

Listing - Call to LogOutput() in the logging output task

```
void  App_LoggingTaskFnct (void  *p_arg)
{
    RTOS_ERR  err;

    PP_UNUSED_PARAM(p_arg);

    while (DEF_TRUE) {
        OSSemPend(&App_LoggingSem, 0, OS_OPT_PEND_BLOCKING, DEF_NULL, &err);
        if (RTOS_ERR_CODE_GET(err) != RTOS_ERR_NONE) {
            /* Handle error. */
        } else {
            Log_Output();
        }
    }
}
```

RTOS_ERR Programming Guide

- Variations
  - Non-Legacy Mode
  - Legacy Mode
- Abstraction Macros
  - RTOS_ERR_CODE_GET(err_var)
  - RTOS_ERR_STR_GET(err_code) & RTOS_ERR_DESC_STR_GET(err_code)
  - RTOS_ERR_COPY(err_dst, err_src)
  - RTOS_ERR_SET(err_var, err_code)

Variations

The content of the RTOS_ERR can vary based on on the configuration constants specified in rtos_err_cfg.h. This is described in RTOS_ERR Type Configuration .

Recommendation: You should use the abstraction macros to access any RTOS_ERR variable, assuming that these macros know which configuration is currently set and how to access the required information from the data structure. If you are not using the abstraction macros, and if the RTOS_ERR changes at any point, you may need to rewrite sections of your code. For example, you may need to prevent access to certain fields that no longer exist.

Non-Legacy Mode

If *legacy mode* is not enabled (RTOS_ERR_CFG_LEGACY_EN is set to DEF_DISABLED), the RTOS_ERR will be a structure with various fields, with at least one error code, and be based on the configuration constants in the rtos_err_cfg.h file, as well as the C standard version that was used to compile the application.

The following table summarizes how each option affects the .

RTOS_ERR_CFG_LEGACY_EN is set to DEF_DISABLED;

		RTOS_ERR_CFG_EXT_EN	
		DEF_ENABLED	**DEF_DISABLED**
RTOS_ERR_CFG_STR_EN	**DEF_ENABLED**	RTOS_ERR contains: • The error code enum • A string containing the name of the file in which the error occurred • The line number at which the error occurred • A string containing the error code in string format • A string containing the error code description • A string containing the function name, if compiling with C99 or after Error and description strings are available, both directly in the RTOS_ERR and can also be obtained via: `RTOS_ERR_STR_GET(err_code)` and `RTOS_ERR_DESC_STR_GET(err_code)` , respectively.	RTOS_ERR contains **only** the error code enum. Error and description strings are available and can be obtained via: `RTOS_ERR_STR_GET(err_code)` and `RTOS_ERR_DESC_STR_GET(err_code)`, respectively.
	DEF_DISABLED	RTOS_ERR contains: • The error code enum • A string containing the name of the file in which the error occurred • The line number at which the error occurred • A string containing the function name, if compiling with C99 or after Error and description strings are **not** available.	RTOS_ERR contains **only** the error code enum. Error and description strings are **not** available.

Legacy Mode

If you are using the *legacy mode* (by having set RTOS_ERR_CFG_LEGACY_EN to DEF_ENABLED), you can continue to use RTOS_ERR as a simple enum rather than a structure. You can still use the abstraction macros to obtain the error code and have the error code strings, although they will not be part of the RTOS_ERR . When *legacy mode* is enabled, you cannot use the extended error mode.

The following table summarizes how the configuration constants affect the errors.

- RTOS_ERR_CFG_LEGACY_EN is set to DEF_ENABLED;
- RTOS_ERR_CFG_EXT_EN is necessarily DEF_DISABLED.

Compiling with any C standard version will not affect the errors.

RTOS_ERR_CFG_STR_EN	**DEF_ENABLED**	Error and description strings are available and can be obtained via: `RTOS_ERR_STR_GET(err_code)` and `RTOS_ERR_DESC_STR_GET(err_code)`, respectively.
	DEF_DISABLED	Error and description strings are not available.

Abstraction Macros

RTOS_ERR_CODE_GET(err_var)

This macro obtains the error code contained in an error variable, no matter how the configuration constants are defined.

- If the *legacy mode* is always used, you can access the RTOS_ERR variable as an enum to obtain the error code.
- If the *legacy mode* is never used, the .Code field of the structure will always be present and can be accessed directly, without using the macro.

The RTOS_ERR_CODE_GET() macro allows you to ensure that the error code is always available, regardless of the configuration.

Listing - RTOS_ERR_CODE_GET() example

```
void MyFnct(void)
{
   RTOS_ERR err;

   AnyCall(&err);
   if (RTOS_ERR_CODE_GET(err) != RTOS_ERR_NONE) {
      /* Handle error. */
   }
}
```

This code snippet will always work, regardless of the configuration of the RTOS_ERR .

RTOS_ERR_STR_GET(err_code) & RTOS_ERR_DESC_STR_GET(err_code)

These macros obtain the string associated with an error code (ex.: "RTOS_ERR_NONE" or "RTOS_ERR_INVALID_CFG") and the description string associated with an error code (ex.: "No error." or "Invalid configuration provided.").

These macros will only return valid values if RTOS_ERR_CFG_STR_EN is set to DEF_ENABLED. If not, they will return a "String not available." string.

If the RTOS_ERR_CFG_EXT_EN and the RTOS_ERR_CFG_STR_EN configuration constants always remain to DEF_ENABLED , you can access the fields directly. which are contained in the RTOS_ERR variable (see below):

Listing - Direct access to string fields of RTOS_ERR variable

```
void MyFnct(void)
{
   RTOS_ERR err;

   AnyCall(&err);
   if (RTOS_ERR_CODE_GET(err) != RTOS_ERR_NONE) {
      printf("Error is: %s\r\n", err.CodeText);
      printf("Description text is: %s\r\n", err.DescText);
   }
}
```

The example above will no longer compile if RTOS_ERR_CFG_EXT_EN or RTOS_ERR_CFG_STR_EN is toggled to DEF_DISABLED. In this case, you should use the provided macros (see below):

Listing - Obtaining strings of RTOS_ERR variable with macros

```
void MyFnct(void)
{
   RTOS_ERR err;

   AnyCall(&err);
   if (RTOS_ERR_CODE_GET(err) != RTOS_ERR_NONE) {
      printf("Error is: %s\r\n", RTOS_ERR_STR_GET(RTOS_ERR_CODE_GET(err)));
      printf("Description text is: %s\r\n", RTOS_ERR_DESC_STR_GET(RTOS_ERR_CODE_GET(err)));
   }
}
```

Note: To obtain the corresponding string, you must pass the *error code* (not the error variable) to both string macros.

RTOS_ERR_COPY(err_dst, err_src)

This macro copies an error variable's content to another variable, keeping whatever values were present in the original variable. This can be useful to report errors asynchronously. The following code caption shows a simple use case:

Listing - RTOS_ERR_COPY() example

```
void MyTwoOperationFnct (RTOS_ERR *p_err)
{
  RTOS_ERR err_one;
  RTOS_ERR err_two;

  MyOperationOneFnct(&err_one);
  MyOperationTwoFnct(&err_two);
  if (RTOS_ERR_CODE_GET(err_one) != RTOS_ERR_NONE) {
    RTOS_ERR_COPY(*p_err, err_one);
  } else {
    RTOS_ERR_COPY(*p_err, err_two);
  }
}
```

RTOS_ERR_SET(err_var, err_code)

This macro defines an error code in an error variable and defines any of the other available fields.

Note: You are *not* required to define the error variables to RTOS_ERR_NONE before passing them to any stack.

Listing - RTOS_ERR_SET() example

```
void MyFnct (RTOS_ERR *p_err)
{
  CPU_BOOLEAN success_flag;

  success_flag = AnyOperation();
  if (success_flag == DEF_OK) {
    RTOS_ERR_SET(*p_err, RTOS_ERR_NONE);
  } else {
    RTOS_ERR_SET(*p_err, RTOS_ERR_FAIL);
  }
}
```

Micrium OS Asserts Programming Guide

- Macro Calls
- Macro Fail Calls

The Common module offers assert capabilities that can be used in various situations in your application. For more information on asserts in general, please refer to Assertions .

Macro Calls

The APP_RTOS_ASSERT_CRITICAL() and APP_RTOS_ASSERT_DBG() macros check if a given expression is evaluated with a positive result. If the result is not positive, the macro will do one of the following operations:

- If RTOS_CFG_RTOS_ASSERT_CRITICAL_FAILED_END_CALL(ret_val) and/or RTOS_CFG_RTOS_ASSERT_DBG_FAILED_END_CALL(ret_val) are defined, the macro will call these.
- If they are not defined, CPU_SW_EXCEPTION(ret_val) will be called.

The debug asserts typically check for conditions that are caused by invalid parameters or invalid configurations. They are used to notify the developer that something is not correct with the way the code is being used. Those can and should be disabled once development is completed.

The critical asserts typically check for conditions from which it is practically impossible to recover at run-time. Therefore, if such a condition is detected, the program's execution should be suspended before any more damage occurs.

An example of an assert using the macros is provided below.

Listing - Assert call example

```
#include  <rtos/common/include/rtos_utils.h>

                                        /* These are the only valid values for this example. */
#define  VALUE_FIRST    1u
#define  VALUE_SECOND   2u
#define  VALUE_THIRD    3u

void  *ExAssert (CPU_INT08U  value)
{
                                        /* Make sure arg passed is one of the valid values.  */
    APP_RTOS_ASSERT_DBG(((value == VALUE_FIRST)  ||
                         (value == VALUE_SECOND) ||
                         (value == VALUE_THIRD)), DEF_NULL);
                                        /* Indicate a value to return in case of failure.    */

    /* ... */

    return (DEF_NULL);
}
```

Macro Fail Calls

These macro calls are intended to be used when an assert must be triggered without checking any additional expression results.

For example, ending in the default case of a switch statement when all known types have their specific case could be considered a critical failure. In these cases, you should use the APP_RTOS_ASSERT_CRITICAL_FAIL() and APP_RTOS_ASSERT_DBG_FAIL() macros. An example is provided below.

Listing - Assert fail call example

```
#include <rtos/common/include/rtos_utils.h>

                                          /* These are the only valid values for this example.   */
#define VALUE_FIRST   1u
#define VALUE_SECOND  2u
#define VALUE_THIRD   3u

void ExAssertFail (CPU_INT08U value)
{
    CPU_INT08U flag;

    switch (value) {
        case VALUE_FIRST:
        case VALUE_SECOND:
            flag = DEF_YES;
            break;

        case VALUE_THIRD:
            flag = DEF_NO;
            break;

        default:                          /* Dflt case reached means an invalid arg was passed.   */
                                          /* Call APP_RTOS_ASSERT_DBG_FAIL() to indicate err.     */
            APP_RTOS_ASSERT_DBG_FAIL(;);           /* Indicate ; as return value if function returns void. */
    }
    /* ... */
}
```

LIB Constants, Defines and Macros Guide

- General
  - Constants
    - Boolean Constants
    - Bit Constants
    - Octet Constants
    - Integer Constants
    - Number Base Constants
    - Time Constants
  - Macros
    - Bit Macros
    - Integer Macros
- Characters
  - Character Values Constants
  - Character Macros and Functions
- Memory
  - Memory Macros and Functions
- Strings
  - String Functions
- Math
  - Math Functions

The LIB sub-module of Common offers several #defines and constants that can be used in any type of application. These constants range from character constants to digits, time, bits, or alignment constants. The following sections details each of these categories and explain how they can be used.

General

Constants

The sections below contain a brief description of some of the defines in API LIB Def.

Boolean Constants

LIB contains many Boolean constants to configure, assign, and test Boolean values or variables. These constants can include: DEF_TRUE/DEF_FALSE, DEF_YES/DEF_NO, DEF_ON/DEF_OFF, DEF_ENABLED/DEF_DISABLED, etc.

Bit Constants

LIB contains bit constants to configure, assign, and test appropriately-sized bit-field or integer values or variables by defining values corresponding to specific bit positions. Currently, LIB supports bit constants up to 64-bits (DEF_BIT_63). These constants include: DEF_BIT_00, DEF_BIT_07, DEF_BIT_15, etc.

Octet Constants

LIB contains octet constants to configure, assign, and test appropriately-sized, octet-related integer values or variables by defining octet or octet-related values. These constants include: DEF_OCTET_NBR_BITS and DEF_OCTET_MASK.

Integer Constants

LIB contains integer constants to configure, assign, and test appropriately-sized, integer values or variables by defining integer-related values. These constants include: DEF_INT_08_MASK, DEF_INT_16U_MAX_VAL, and DEF_INT_32S_MIN_VAL.

Number Base Constants

LIB contains number base constants to configure, assign, and test number base values or variables by defining number base values. These constants include: DEF_NBR_BASE_BIN and DEF_NBR_BASE_HEX.

Time Constants

LIB contains time constants to configure, assign, and test time-related values or variables by defining time-related values. These constants include: DEF_TIME_NBR_HR_PER_DAY, DEF_TIME_NBR_SEC_PER_MIN, DEF_TIME_NBR_mS_PER_SEC, etc.

Macros

Bit Macros

LIB provides macros to perform various bit operations, such as setting or clearing a bit, checking if a bit or a group of bits is/are set or cleared, masking bits, etc. These macros are defined in API LIB Def and API LIB Utilities.

Integer Macros

LIB offers macros to check if given values fit within a range or to obtain the maximum or minimum value between two variables. These macros are defined in API LIB Def.

Characters

Character Values Constants

LIB contains many character value constants such as:

ASCII_CHAR_LATIN_DIGIT_ZERO ... ASCII_CHAR_LATIN_DIGIT_NINE

ASCII_CHAR_LATIN_UPPER_A ... ASCII_CHAR_LATIN_UPPER_Z

ASCII_CHAR_LATIN_LOWER_A ... ASCII_CHAR_LATIN_LOWER_Z

One constant exists for each ASCII character, though additional aliases are provided for some characters. These constants should be used to configure, assign, and test appropriately-sized ASCII character values or variables.

Character Macros and Functions

LIB provides macros and functions to act on characters. It is also possible to check if a character is a whitespace, a punctuation sign or an alphanumeric character. These functions and macros are defined in lib_ascii.c and lib_ascii.h .

For example, it is possible to know if a given character is upper-case or lower-case or convert it to upper- or lower-case.

Memory

Memory Macros and Functions

LIB contains functions and macros that replace standard library memory functions such as memclr(), memset(), memcpy(), memcmp(), etc; as well as generic versions of network functions, ntohl(), ntohs(), htonl(), htons(), to convert data with any endianness type to other types of endianness. These functions are defined in lib_mem.c and lib_mem.h .

Strings

String Functions

LIB contains library functions that replace standard library string functions such as strlen(), strcpy(), strcmp(), etc. These functions are defined in lib_str.c.

Math

Math Functions

LIB contains library functions that replace standard mathematics functions such as rand(), srand(), etc. These functions are defined in lib_math.c. As for the standard functions, the random number generator must first be seeded before it can provide good random numbers. This is done using Math_RandSeed() or Math_RandSetSeed() .

Memory Allocation Guide

- Memory Segments
  - Declaring Memory Segments at Compile-Time
- Memory Segment Allocation
  - Example Usage of Memory Segments
- Dynamic Memory Pools
  - Dynamic memory pool usage example

The LIB sub-module allows the user to create memory segments and use them for tasks such as allocating blocks or creating dynamic memory pools.

Memory Segments

A memory segment is an object describing a memory region. It can describe virtually any kind of memory region, with the following parameters, when needed:

- Base address of the memory region
- Size of the memory region
- Any padding required for data allocated in this memory region (to complete a cache line, for example)

For example, one memory segment can describe the following:

- All of the external RAM on a board or only part of this RAM
- Describe a region of dedicated memory for USB
- Describe a region of cacheable memory
- Describe a region that is accessible by DMA or not
- And more

Once a memory segment is created using the Mem_SegCreate () function, you can use it in various ways. The table below summarizes each action that can be done using an existing memory segment, which function(s) can be used, and the parameters that the function(s) take to perform the operation.

Table - LIB Mem Operations

Operation	Function
Obtain remaining data space in the memory segment.	Mem_SegRemSizeGet()
Allocate a single block of a given size, from either the global heap or another segment.	Mem_SegAlloc()
Allocate a single block of a given size, from either the global heap or another segment, with alignment	Mem_SegAllocExt()
Allocate a single block of a given size, from either the global heap or another segment, with alignment, for memory that needs padding specified in memory segment (cacheable or DMA-accessed memory, typically).	Mem_SegAllocHW()
Create a dynamic pool of fixed-size memory blocks.	Mem_DynPoolCreate()
Create a dynamic pool of persistent fixed-size memory blocks, with an associated callback to initialize their content, if needed.	Mem_DynPoolCreatePersistent()
Create dynamic pool of fixed-size memory blocks that needs padding specified in memory segment (cacheable or DMA-accessed memory, typically)	Mem_DynPoolCreateHW()
Obtain block from the dynamic memory pool.	Mem_DynPoolBlkGet()
Return block to the dynamic memory pool.	Mem_DynPoolBlkFree()
Know how many blocks are left in a dynamic memory pool.	Mem_DynPoolBlkNbrAvailGet()

Declaring Memory Segments at Compile-Time

It is possible for memory segments to be created at compile-time to indicate to LIB Mem the existence of that particular segment by using the MEM_SEG_INIT() macro, combined with a call to Mem_SegReg() at run-time. If the call to Mem_SegReg() is omitted, the segment will still work correctly, but its information will not be included in any debug output. When another new segment is created, this segment will not be part of the ones checked for overlap.

Memory Segment Allocation

LIB memory allocation functions provide for the allocation of memory from a general-purpose heap or particular memory segments. Single memory blocks may be allocated directly from the heap or from any segment. However, to prevent fragmentation, these memory blocks cannot be freed back.

Three different functions are available to allocate memory from a memory segment:

Table - Memory segment allocation operations

Function	Description	Use case
Mem_SegAlloc()	General-purpose allocation function that provides a buffer aligned on a CPU word boundary and with no guaranteed padding.	General purpose buffers and control data accessed only by the CPU.
Mem_SegAllocExt()	Same as Mem_SegAlloc(), except that this function allows specifying an alignment. It will also provide the number of bytes required to prevent overflow when the memory segment's size limit is exceeded.	General purpose buffers and control data accessed only by the CPU but that require specific alignment (for example, task stacks).
Mem_SegAllocHW()	Allocates a buffer using specified alignment. This function will also add padding at the end of the buffer according to the padding_align argument specified at the creation of the memory segment. This is useful on systems that use cache memory as it allows only the buffer to be mapped over all its associated cache lines.	Data buffers that can be copied via a DMA engine.

The table below summarizes where every property used for an allocation comes from.

Table - Parameters used for allocating with Mem_SegAlloc() functions

Function	Allocated data size	Allocated data alignment	Data padding alignment
Mem_SegAlloc()	Function parameter.	Defaults to sizeof(CPU_ALIGN).	Not used.
Mem SegAllocExt()	Function parameter.	Function parameter.	Not used.
Mem_SegAllocHW()	Function parameter.	Function parameter.	Passed at segment creation.

Example Usage of Memory Segments

Listing - Memory Segment Allocation Usage Example in the *Memory Allocation Guide* page gives an example usage of memory segments. In this example, we attempt to create a memory segment and to allocate three buffers of 20 bytes, using the three different segment allocation functions available. Following the listing is an explanation of the differences between these three buffers.

Listing - Memory Segment Allocation Usage Example

```
#define  CACHE_LINE_LEN                      32u

static  CPU_INT08U  MemSegData[4096u];
static  MEM_SEG     MemorySegment;

static void Mem_SegExample (void)
{
    CPU_INT08U  *p_general_purpose_buf;
    CPU_INT08U  *p_general_purpose_buf_with_align;
    CPU_INT08U  *p_hardware_buffer;
    RTOS_ERR     err_lib;

                                            /* ------------ CREATION OF MEMORY SEGMENT ------------ */  (1)
    Mem_SegCreate(        "Name",           /* Name of mem seg (for debugging purposes).           */
                      &MemorySegment,       /* Pointer to memory segment structure.                */
                  (CPU_ADDR)MemSegData,     /* Base address of memory segment data.                */
                       4096u,               /* Length, in byte, of the memory segment.             */
                       CACHE_LINE_LEN,      /* Padding alignment value.                            */
                      &err_lib);
    if (err_lib != RTOS_ERR_NONE) {         /* Validate memory segment creation is successful.     */
        /* Handle error case. */
        return;
    }

                                            /* ------- ALLOCATION OF GENERAL PURPOSE BUFFER ------- */  (2)
    p_general_purpose_buf = Mem_SegAlloc("General purpose buffer",
                                         &MemorySegment,
                                          20u,  /* Requested buffer length.                        */
                                         &err_lib);
    if (err_lib != RTOS_ERR_NONE) {         /* Validate memory segment allocation is successful.   */
        /* Handle error case. */
        return;
    }

                                            /* --- ALLOCATION OF ALIGNED GENERAL PURPOSE BUFFER --- */  (3)
    p_general_purpose_buf_with_align = Mem_SegAllocExt("General purpose buffer aligned",
                                                       &MemorySegment,
                                                        20u,
                                                        8u, /* Request 8 bytes boundary alignment.   */
                                                        DEF_NULL,
                                                       &err_lib);
    if (err_lib != RTOS_ERR_NONE) {         /* Validate memory segment allocation is successful.   */
        /* Handle error case. */
        return;
    }

                                            /* ---------- ALLOCATION OF HARDWARE BUFFER ----------- */  (4)
    p_hardware_buffer = Mem_SegAllocHW("Buffer read/written via DMA engine",
                                       &MemorySegment,
                                        20u,
                                        1024u,  /* DMA engine required alignment.                  */
                                        DEF_NULL,
                                       &err_lib);
    if (err_lib != RTOS_ERR_NONE) {         /* Validate memory segment allocation is successful.   */
        /* Handle error case. */
        return;
    }

    /* ... */
}
```

(1) **Creation of the memory segment**. The memory segment will be created on a statically allocated buffer of 4096 bytes. The base address of the memory segment data can also point to a controller dedicated memory.

(2) **Allocation of a general-purpose buffer**. p_general_purpose_buf is intended to be a control buffer and is only read/written by the CPU. This buffer will automatically be aligned on CPU word boundary.

(3) **Allocation of a general-purpose buffer with specified alignment**. p_general_purpose_buf_with_align will be allocated from the memory segment but will be aligned on an 8-byte boundary.

(4) **Allocation of a hardware buffer** (a buffer that can be read/written from a DMA engine). p_hardware_buffer will be allocated from the memory segment and will be aligned on a 1024 bytes boundary. The difference with p_general_purpose_buf_with_align is that this buffer will have a real length of 32 bytes as it will be padded using the padding alignment specified at the time of the memory segment creation at (1). The padding_align argument was set to a cache line length. This buffer is guaranteed to not share its memory cache line with other buffer hence preventing cache incoherence.

Dynamic Memory Pools

Memory pool blocks can be allocated from either the general-purpose heap or from dedicated memory specified by the application. Memory pool blocks can be dynamically allocated and freed during application execution because memory pool blocks are fixed-size, which prevents possible fragmentation.

Dynamic memory pools are a pool of memory blocks that can be dynamically allocated from either the general-purpose heap or a specific memory segment. Since the blocks have a fixed size, it is possible to return (free) them to their pool at run-time, without any chance of fragmentation. They also have the particularity that if there are no blocks available when attempting to get one, it will be allocated from free space on the memory segment.

Each pool has an initial number of allocated blocks that will be allocated at pool creation from the memory segment passed. Each pool also has a maximum number of blocks that can be allocated. This maximum can either be a finite or infinite amount; every time there are no available blocks, pool blocks will be allocated from the memory segment until the memory segment is full.

The dynamic memory pools can allocate the following:

- General-purpose memory blocks
- Persistent blocks that keep the data stored in them even when freed
- Hardware memory blocks

This means that there are three different functions available to create dynamic memory pools:

Table - Mem_DynPoolCreate() functions

Function	Description	Use case
Mem_DynPoolCreate()	Creates a standard memory pool and allows to specify the memory alignment of each memory block.	General-purpose memory block.
Mem_DynPoolCreatePersistent()	Creates a standard memory pool that guarantees the data integrity of each block even when those blocks are freed and re-obtained.	Memory blocks that contain data that must never be lost; for example, containing other dynamically allocated data, or run-time created objects such as Kernel objects.
Mem_DynPoolCreateHW()	Creates a hardware memory pool. The memory blocks will be aligned as specified and padded as per memory segment properties.	Memory blocks that can be read/written via a DMA engine.

The table below summarizes where every property used for an allocation comes from.

Table - Parameters used for allocating with Mem_SegAlloc() functions

Function	Allocated block size	Allocated block alignment	Block padding alignment
Mem_DynPoolCreate()	Function parameter.	Function parameter.	Not used.

Function	Allocated block size	Allocated block alignment	Block padding alignment
Mem_DynPoolCreatePersistent()	Function parameter.	Function parameter.	Not used.
Mem_DynPoolCreateHW()	Function parameter.	Function parameter.	Passed at segment creation.
Mem_DynPoolBlkGet() on a pool created with Mem_DynPoolCreate()	Passed at pool creation.	Passed at pool creation.	Not used.
Mem_DynPoolBlkGet() on a pool created with Mem_DynPoolCreatePersistent()	Passed at pool creation.	Passed at pool creation.	Not used.
Mem_DynPoolBlkGet() on a pool created with Mem_DynPoolCreateHW()	Passed at pool creation.	Passed at pool creation.	Passed at segment creation.

Dynamic memory pool usage example

Listing - Dynamic Memory Pool Usage Example in the *Memory Allocation Guide* page gives an usage example of dynamic memory pools.

Listing - Dynamic Memory Pool Usage Example

```
#define  CACHE_LINE_LEN                          32u

static  CPU_INT08U    MemSegData[4096u];
static  MEM_SEG       MemorySegment;
static  MEM_DYN_POOL  DynamicMemPool;
static  MEM_DYN_POOL  DynamicMemPoolHW;

 static void Mem_DynPoolExample (void)
{
   CPU_INT08U  *p_blk;
   CPU_INT08U  *p_blk_hw;
   RTOS_ERR     err_lib;

                                           /* -------- CREATION OF MEMORY SEGMENT ------ */ (1)
   Mem_SegCreate(        "Segment name",                 /* Name of mem seg (for debugging purposes).  */
                  &MemorySegment,               /* Pointer to memory segment structure.       */
             (CPU_ADDR)MemSegData,                 /* Base address of memory segment data.       */
                   4096u,                     /* Length, in byte, of the memory segment.    */
                   CACHE_LINE_LEN,               /* Padding alignment value.                 */
                  &err_lib);
   if (err_lib != RTOS_ERR_NONE) {                    /* Check memory segment creation.            */
      /* Handle error case. */
      return;
   }

                                           /* CREATE GENERAL-PURPOSE DYNAMIC MEMORY POOL */ (2)
   Mem_DynPoolCreate("General-purpose dynamic memory pool",/* Name of dynamic pool (for debugging).     */
               &DynamicMemPool,                   /* Pointer to dynamic memory pool data.      */
               &MemorySegment,                    /* Pointer to memory segment to use.         */
                20u,                        /* Block size, in bytes.                     */
                sizeof(CPU_ALIGN),               /* Block alignment, in bytes.                */
                10u,                        /* Initial number of blocks.                 */ (3)
                10u,                        /* Maximum number of blocks.                 */ (4)
               &err_lib);
   if (err_lib != RTOS_ERR_NONE) {                    /* Validate dynamic memory pools creation.   */
      /* Handle error case. */
      return;
   }

                                           /*  CREATION OF HARDWARE DYNAMIC MEMORY POOL  */ (5)
   Mem_DynPoolCreateHW("Hardware dynamic memory pool",     /* Name of dynamic pool (for debugging).     */
```

```
&MemorySegment,/* Pointer to memory segment to use.         */20u,/* Block size, in bytes.                    */8u,/* Block alignment, in bytes.
*/10u,/* Initial number of blocks.                  */
                         LIB_MEM_BLK_QTY_UNLIMITED,/* Maximum number of blocks.               */(6)&err_lib);if(err_lib != RTOS_ERR_NONE){/* Check
dynamic memory pools creation.       *//* Handle error case. */return;}/* ---------- BLOCK GET OPERATION ----------- */(7)
    p_blk = (CPU_INT08U *)Mem_DynPoolBlkGet(&DynamicMemPool,&err_lib);if(err_lib != RTOS_ERR_NONE){/* Validate block get operation.
*//* Handle error case. */return;}/* ------ HARDWARE BLOCK GET OPERATION ------ */(8)
    p_blk_hw = (CPU_INT08U *)Mem_DynPoolBlkGet(&DynamicMemPoolHW,&err_lib);if(err_lib != RTOS_ERR_NONE){/* Check block get operation is
successful.   *//* Handle error case. */return;}/* ... *//* ---------- BLOCK FREE OPERATION ---------- */(9)Mem_DynPoolBlkFree(&DynamicMemPool,
(void *)p_blk,&err_lib);if(err_lib != RTOS_ERR_NONE){/* Check block free operation is successful.  *//* Handle error case. */return;}/* -----
HARDWARE BLOCK FREE OPERATION ------ */(10)Mem_DynPoolBlkFree(&DynamicMemPoolHW,(void *)p_blk_hw,&err_lib);if(err_lib !=
RTOS_ERR_NONE){/* Check block free operation is successful.  *//* Handle error case. */return;}}
```

(1) Creation of the memory segment from which memory blocks will be allocated.

(2) Creation of the dynamic memory pool. This memory pool will allocate general-purpose memory blocks of 20 bytes and aligned on CPU word boundaries.

(3) At creation 10 blocks will be allocated and available.

(4) A maximum of 10 blocks can be allocated from this dynamic memory pools. Since the initial number of block equals the maximum number of blocks, this will create a static memory pool.

(5) Creation of a hardware dynamic memory pool. The requested alignment is 8 bytes, but since the padding alignment specified at the time of the memory segment creation is 32 bytes, the memory blocks will be aligned on a 32-byte boundary and will have a length of 32 bytes.

(6) No limit of memory block quantity is specified for this dynamic memory pool. When more than 10 blocks are taken from the pool, the dynamic memory pool will start allocating blocks from the free space of the memory segment. Once a memory block is freed, it will be available for the next allocation. It is possible to allocate memory blocks until the memory segment overflows.

(7) A general-purpose memory block is taken from the pool. The block is aligned on a CPU word boundary and is at least 20 bytes long.

(8) A hardware memory block is taken from the pool. The block is aligned on a 32-byte boundary and is padded to meet the padding alignment requirement of the memory segment. It is then safe to be read/written via a DMA engine.

(9) (10) Memory blocks are freed back to their respective dynamic memory pools. They are available for the next allocation. Note that the data that was present on these memory blocks WILL BE altered when freed.

Common Troubleshooting

Common Troubleshooting

- Authentication
  - Unable to create a user.
  - Unable to get a user handle.
  - Unable to get user right.
  - Unable to grant or revoke user right.
- LIB
  - LIB Memory Allocation
    - A segment is not in the debug list when calling Mem_OutputUsage() .
    - A call to Mem_SegClr() does not clear the memory segment's data.
    - Memory segment should have enough space to allocate data but returns RTOS_ERR_SEG_OVF.
    - How can the memory usage be optimized?
  - LIB String
    - String function not working as expected.
- RTOS ERR
  - The string obtained from an error using RTOS_ERR_STR_GET() or RTOS_ERR_DESC_STR_GET() is always "Error string unavailable."
  - RTOS_ERR_CODE_GET() causes error when building a project.
  - Unable to access a field within a RTOS_ERR variable.
- Utilities
  - Asserts should be enabled/disabled and are not for a particular source file.
- Shell
  - Unable to add a command to the Shell module.
  - A command has been added but fails to be executed.

Authentication

Unable to create a user.

- Check that the maximum number of users defined in Auth_Init() by Auth_InitCfg.ResourceCfg.NbUserMax has not been reached.
- Check that the user name and password fit in Auth_InitCfg. ResourceCfg.NameLenMax and Auth_InitCfg. ResourceCfg.PwdLenMax.
- Check that the user name being submitted is unique: that there is no matching duplicate user name in the system.

Unable to get a user handle.

- Check if a user with that name has already been created.

Unable to get user right.

- Check that a valid AUTH_USER_HANDLE is passed. This can be obtained by using either Auth_GetUser() or Auth_ValidateCredentials().

Unable to grant or revoke user right.

- Check that the valid AUTH_USER_HANDLE is passed. This can be obtained by using either Auth_GetUser() or Auth_ValidateCredentials().
- Check that the user described by the as_user_handle has the right(s) that need to be granted or revoked.

LIB

LIB Memory Allocation

A segment is not in the debug list when calling Mem_OutputUsage().

- If the segment has been created at compile-time using MEM_SEG_INIT(), make sure a corresponding Mem_SegReg() has been made.

A call to Mem_SegClr() does not clear the memory segment's data.

- Mem_SegClr() does not erase any data. It merely marks the memory segment as empty so that all of its data can be re-used by future calls. To erase data please use Mem_Clr().

Memory segment should have enough space to allocate data but returns RTOS_ERR_SEG_OVF .

- Make sure the right segment is referenced and that LIB's heap is not used by default.
- See if the specified alignment is correct; it might cause blocks to need more data than expected.
- If a padding alignment was specified, and some 'hardware' calls are made on it (Mem_SegAllocHW() or Mem_DynPoolCreateHW()), make sure to account for that padding.
- Regularly call Mem_SegRemSizeGet() to see at which point the space used starts to be incorrect.
- Call Mem_OutputUsage() to output information about every allocation of every memory segment, and diagnose which allocation creates a problem.

How can the memory usage be optimized?

- Please refer to the Memory Segments and LIB Heap page for more information.

LIB String

String function not working as expected.

- Check if the string needs to be NULL-terminated and if it really is.

RTOS_ERR

The string obtained from an error using RTOS_ERR_STR_GET() or RTOS_ERR_DESC_STR_GET() is always "Error string unavailable."

- Check that the error strings are enabled. If not, set RTOS_ERR_CFG_STR_EN to DEF_ENABLED and re-build your application.

RTOS_ERR_CODE_GET() causes error when building a project.

- Check that the argument passed to RTOS_ERR_CODE_GET() is an error *variable*, and not an error *code*. For example, the argument should be my_err and not RTOS_ERR_NULL_PTR.

Unable to access a field within a RTOS_ERR variable.

- Macros are the recommended way of accessing the internal fields of a RTOS_ERR variable when they are available for a given field. Check if it is possible to use them instead of accessing the internal field directly.
- If accessing a field other than the Code field, check that RTOS_ERR_CFG_EXT_EN is set to DEF_ENABLED.
- Check if the corresponding configuration constant is correctly set:
  - RTOS_ERR_CFG_STR_EN must be set to DEF_ENABLED so that the CodeText and DescText are present in the RTOS_ERR structure.
  - For the FnctName field to be available, the application must be compiled in C99 or later.

Utilities

Asserts should be enabled/disabled and are not for a particular source file.

- Make sure RTOS_MODULE_CUR is #define'd *before* including the <rtos/common/include/rtos_utils.h> to indicate which module the source file belongs to.

Shell

Unable to add a command to the Shell module.

- Check the error returned by the Shell_CmdTblAdd() function:
  - If RTOS_ERR_NO_MORE_RSRC is returned, there is no more space left to add another command in the Shell module. That limit will need to be increased by calling Shell_ConfigureCmdUsage() before Shell_Init().
  - If RTOS_ERR_INVALID_ARG is returned, there is a problem with the command table name. See the Shell command add example (called ex_common_shell_cmd_add.c under examples/common/shell/) for a working example.
  - If RTOS_ERR_ALREADY_EXISTS is returned, either the module name or the command name are already part of the Shell module commands list.

A command has been added but fails to be executed.

- Check the error returned by the Shell_Exec() function:
  - If RTOS_ERR_SHELL_CMD_EXEC, the command was executed but indicated an error during execution. See that command's documentation.
  - If RTOS_ERR_NOT_FOUND is returned, check the error code returned by the call to Shell_CmdTblAdd() which was made to add the failing command and make sure it is RTOS_ERR_NONE, the command may not have been correctly added to Shell.
  - If RTOS_ERR_NO_MORE_RSRC is returned, make sure the command is well formed and that the number of argument is not higher to the maximum number of arguments passed to Shell_ConfigureCmdUsage(). If needed, this number could be increased, although memory usage will also increase.

CPU

CPU

The Micrium OS CPU module offers abstractions for common features found on most microcontrollers. These features include critical sections, interrupt management, cache controller management, and timestamping facilities. The abstractions allow you to focus on making use of the feature instead of its device-specific implementation.

CPU Overview

CPU Overview

The Micrium OS CPU module offers toolchain-agnostic abstractions of common microcontroller features. These include: data types, optimized functions, and critical sections. The subsections below document the various abstractions, and the Configuration Manual explains how to configure this module to better fit your needs.

- Data Types
- Core Features
  - Name and Optimized Operations
  - Timestamp
  - Miscellaneous

Data Types

To help you write reusable code, the CPU module offers all the data type definitions your application may need. The table below lists the supported data types:

Table - Micrium OS CPU Supported Data Types

Data Type	Description
CPU_VOID	The void type.
CPU_CHAR	8-bit character.
CPU_BOOLEAN	8-bit Boolean value.
CPU_INT08U	8-bit unsigned integer.
CPU_INT08S	8-bit signed integer.
CPU_INT16U	16-bit unsigned integer.
CPU_INT16S	16-bit signed integer.
CPU_INT32U	32-bit unsigned integer.
CPU_INT32S	32-bit signed integer.
CPU_INT64U	64-bit unsigned integer.
CPU_INT64S	64-bit signed integer.
CPU_FP32	Single precision floating point.
CPU_FP64	Double precision floating point.
CPU_REG08	8-bit volatile value.
CPU_REG16	16-bit volatile value.
CPU_REG32	32-bit volatile value.
CPU_REG64	64-bit volatile value.
CPU_ADDR	Sufficiently wide type used to address the whole address space.
CPU_DATA	Type that matches the microcontroller's data bus width.
CPU_STK	Type that matches the width of the values present in the microcontroller's stack.
CPU_FNCT_VOID	Convenient pointer to void func() functions.
CPU_FNCT_PTR	Convenient pointer to void func(void *arg) functions, like callbacks.

Core Features

Beyond data types, the CPU module offers various services useful for the development of your application.

Name and Optimized Operations

This module offers your application basic features such as identification and optimized functions. The CPU Name section of the programming guide contains more information on how to use the identification system of the CPU module.

The optimized functions created by this module allow your application to use frequently-used microcontroller instructions. These instructions include: count leading zeros and count trailing zeros. All of these functions usually process a single word, and are used often by the kernel. The Optimized Operations section of the Programming Guide contains more information on the optimized functions offered by the CPU module.

Timestamp

In order to aid the profiling of your application, and determine where most of the CPU time is spent, the Micrium OS CPU module offers timestamping functions to abstract away the details of initializing and handling a timestamp counter. Furthermore, the timestamp features offered by the CPU module are used by the kernel to measure the time it took to apply certain operations on its objects. The Timestamp section of the Programming Guide illustrates how to use the timestamp features of the CPU module.

Miscellaneous

Some microcontrollers offer the possibility to create software breakpoints and exceptions. The CPU module provides your application with a generic way to create such breakpoints and exceptions. See the related section in the Programming Guide for more details.

Integrating CPU Into Your Project

Integrating CPU Into Your Project

The Micrium OS CPU module is composed of several components, each of which is a set of files that implement specific functions. To use CPU, you must add these files to your project and populate your RTOS description file .The CPU module consists of one component for the core part named "CPU" and one component for the port called "Port".

Starting CPU

Micrium offers a set of example applications that demonstrate some of the features of Micrium OS CPU and help you start the development of your application. It is recommended to start from one of these examples.

Configuring CPU

Micrium OS CPU can be configured better fit your needs. The following pages explain how CPU and the whole Micrium OS can be configured at compile-time: CPU Configuration , RTOS Compile-Time Configuration and RTOS ERR Type Configuration .

CPU Configuration

CPU Configuration

Compile-time configuration allows you to enable or disable specific CPU features. Disabling unused features allows the CPU module to use less code and data space, increasing performance.

To configure the compile-time parameters, you must set the #define constants in the cpu_cfg.h file provided by the application. These #define constants are listed below as they are listed in the cpu_cfg.h file (default values are in **bold**).

CPU Core Options

Table - CPU Core Configuration Constants

Constant	Description	Possible values
CPU_CFG_NAME_EN	When set to 1, your application has access to functions and storage that can be used to identify the CPU. When set to 0, the CPU_CFG_NAME_SIZE configuration constant is ignored.	1 (enabled) or **0** (disabled)
CPU_CFG_NAME_SIZE	If your application needs to identify a CPU, this configuration sets the size (in characters) of the CPU's name.	Integer, **16u**
CPU_CFG_TS_32_EN	When set to 1, the 32-bit timestamp counter offered by the CPU becomes available. If set to 0, the CPU_CFG_TS_TMR_SIZE configuration is ignored. **Note** : To use the timestamp facilities, the Micrium OS CPU module Board Support Package or Port should implement the functions described in the CPU Port Board Support Package .	1 (enabled) or **0** (disabled)
CPU_CFG_TS_64_EN	When set to DEF_ENABLED, the 64-bit timestamp counter offered by the CPU becomes available. If set to DEF_DISABLED, the CPU_CFG_TS_TMR_SIZE configuration is ignored. **Note** : To use the timestamp facilities, the Micrium OS CPU module Board Support Package or Port should implement the functions described in the CPU Port Board Support Package .	1 (enabled) or **0** (disabled)
CPU_CFG_TS_TMR_SIZE	If a timestamp counter is used, this configuration option will set the size of variables used to contain the timestamp count.	CPU_WORD_SIZE_08, CPU_WORD_SIZE_16 , **CPU_WORD_SIZE_32** or CPU_WORD_SIZE_64
CPU_CFG_ENDIAN_TYPE	If the CPU supports configurable endianness, set this to the appropriate endian type.	CPU_ENDIAN_TYPE_BIG or **CPU_ENDIAN_TYPE_LITTLE**
CPU_CFG_CACHE_MGMT_EN	If the CPU offers an L1 and/or L2 cache, set this to 1 to allow your application to manipulate the data cache.	1 (enabled) or **0** (disabled)

CPU Programming Guide

CPU Programming Guide

This section contains the following topics:

- Using CPU Core Features

Using CPU Core Features

- CPU Name
- Optimized Operations
- Timestamp
- Miscellaneous

Before using the other functions, the CPU module needs to be initialized using CPU_Init() (see the example below).

Listing - Example of call to CPU_Init()

```
void  main (void)
{
           :
           :
   CPU_Init();  /* Initialize the CPU Module.                              */
           :
           :
}
```

CPU Name

For debugging (or other) purposes, an application can set a name for the CPU currently executing the code. An application can clear the CPU name with CPU_NameClr(), set it with CPU_NameSet() and read it with CPU_NameGet(). See the example below for a usage sample.

Listing - Example of call to CPU_NameClr(), CPU_NameSet() and CPU_NameGet()

```
const CPU_CHAR  App_CoreName[] = "MMIX";
          :
          :
void  App_SomeTask (void  *p_arg)
{
   RTOS_ERR  err;
          :
          :
   CPU_NameClr();               /* Clear previously used CPU name.                */

                         /* Set CPU Name.                             */
   CPU_NameSet(&App_CoreName[0],  /*   Set name to 'MMIX'.                     */
            &err);
   if (err.Code != RTOS_ERR_NONE) {
      /* Handle error on CPU name set. */
   }
          :
          :
}

void  App_SomeOtherTask (void  *p_arg)
{
   RTOS_ERR  err;
   CPU_CHAR  cpu_name[CPU_CFG_NAME_SIZE];
          :
          :
                      /* Get CPU Name.                              */
   CPU_NameGet(&cpu_name[0],  /*   Pointer to user allocated character buffer.        */
            &err);
   if (err.Code != RTOS_ERR_NONE) {
      /* Handle error on CPU name get. */
   }
          :
          :
}
```

Optimized Operations

The CPU module offers a series of functions frequently used by real-time kernels, and the CPU port can implement these functions to optimize their usage. An application can use CPU_CntLeadZeros() to count the number of most-significant bits set to zero. Conversely, CPU_CntTrailZeros() obtains the number of least-significant bits set to zero.

Listing - Example of call to CPU_CntLeadZeros() and CPU_CntTrailZeros()

```
void  App_SomeTask (void  *p_arg)
{
   CPU_DATA    value;
   CPU_DATA    zeroes;
          :
          :
   zeroes = 0x1FFFFFFF;            /* Count number of leading zeros.             */
   value  = CPU_CntLeadZeros(zeros);  /*   Should be 3.                         */
          :
          :
   zeroes = 0xFFFFFFF8;            /* Count number of trailing zeros.            */
   value  = CPU_CntTrailZeros(zeros); /*   Should be 3.                         */
          :
          :
}
```

Timestamp

The CPU module offers services that use a built-in timestamp counter (if available), which feature a number of time value types. Therefore, the CPU module has 32-bit and 64-bit width operations.

- The CPU_TS_Get32() function returns the current value of the timestamp counter, in timestamp ticks.
- The CPU_TS_Get64() function can be used instead if the counter is more than 32-bits wide.

An application can convert the value read in timestamp ticks to microseconds using CPU_TS32_to_uSec() or CPU_TS64_to_uSec(), depending on the type of the timestamp counter. An application can use CPU_TS_TmrFreqGet() to obtain the frequency at which the timestamp counter is currently ticking.

The following example shows how to use the 32-bit width functions. However, the same method can be applied to 64-bit functions.

Listing - Example of call to CPU_TS_Get32(), CPU_TS32_to_uSec() and CPU_TS_TmrFreqGet().

```
void  App_SomeTask (void  *p_arg)
{
    CPU_TS32         timestamp;
    CPU_INT64U       usecs;
    CPU_TS_TMR_FREQ  freq;
    RTOS_ERR         err;
              :
              :
    timestamp = CPU_TS_Get32();           /* Read current timestamp counter value.      */
    usecs = CPU_TS32_to_uSec(timestamp);  /* Convert timestamp counter to microseconds. */
              :
              :
    freq = CPU_TS_TmrFreqGet(&err);       /* Get the timestamp counter's frequency.     */
    if (err.Code != RTOS_ERR_NONE) {
        /* Handle error on timestamp frequency get. */
    }
              :
              :
}
```

Miscellaneous

The CPU module uses abstractions to insert a break instruction and to trap software exceptions.

- CPU_BREAK() function inserts a break.
- CPU_SW_EXCEPTION() traps software exceptions.

Listing - Example of call to CPU_BREAK() and CPU_SW_EXCEPTION().

```
void  App_SomeTask (void  *p_arg)
{
              :
              :
    if (/* Some debug event*/) {
        CPU_BREAK();
    }
              :
              :
}

void  App_SomeOtherTask (void  *p_arg)
{
              :
              :
    if (/* Some unrecoverable error */) {
        CPU_SW_EXCEPTION();
    }
              :
              :
}
```

CPU Hardware Porting Guide

CPU Hardware Porting Guide

A Micrium OS CPU port implements all of the abstractions described in the CPU Overview page.

- CPU Port Board Support Package

CPU Port Board Support Package

The only board-specific feature offered by the Micrium OS CPU module (as opposed to architecture- or microcontroller-specific) is the timestamping feature. The section below describes which functions the CPU Board Support Package should implement.

Timestamp

If you want to use a different timestamp counter than the one built into the CPU, you should implement your own bsp_cpu.c file. The functions to be implemented are described below:

Table - CPU BSP Timestamp Functions

Function Prototype	Description
`void CPU_TS_TmrInit (void);`	Initialize and start the timestamp counter. This function should also inform the CPU module about the timestamp counter's operating frequency, see `CPU_TS_TmrFreqSet()`.
`CPU_TS_TMR CPU_TS_TmrRd (void);`	Read and return the current timestamp count.
`CPU_INT64U CPU_TS32_to_uSec()(CPU_TS32 ts_cnts);`	Based on the operating timestamp counter frequency, convert the timestamp count to microseconds. **Note**: Only implement this function if the timestamp counter has a 32-bit wide counter.
`CPU_INT64U CPU_TS64_to_uSec()(CPU_TS64 ts_cnts);`	Based on the operating timestamp counter frequency, convert the timestamp count to microseconds. **Note**: Only implement this function if the timestamp counter has a 64-bit wide counter.

CPU Troubleshooting

CPU Troubleshooting

General advice.

If an expected feature does not seem to work, like the host name or timestamping features, make sure that your configuration file (cpu_cfg.h) reflects what you need. Consult the CPU Configuration Guide for more information on the CPU module's configuration.

My critical sections don't work.

Verify that interrupts are enabled before execution reaches a critical section. In a critical section, the interrupts' enable state is preserved before the interrupts are disabled; then once the critical section ends, the preserved state is restored. If the interrupts were never enabled in the first place, simply ending a critical section will not re-enable them.

My time measurements are off.

If you use the CPU_TS32_to_uSec() or CPU_TS64_to_uSec() functions to measure the time it takes (in microseconds) for an event to occur, make sure that the CPU module is aware of the operating frequency of the timestamp counter. If you change the operating frequency of the microcontroller or the timer itself once the CPU module is initialized, make sure you call the CPU_TS_TmrFreqSet() function to update the operating frequency.

IO

IO

The Micrium OS IO module contains APIs for basic communication interfaces such as SPI.

This module is designed specifically for embedded systems, and is built from the ground up with Micrium's quality, scalability, and reliability.

This section describes how to initialize, start, and use the Micrium OS IO module. It explains the various configuration values and their uses, as well as providing a porting guide for your hardware. This section also includes an overview of the technology, types of configuration possibilities, implementation procedures, etc.

SD

SD

- Integrating SD Into Your Project
- SD Configuration
- SD Programming Guide
- SD Hardware Porting Guide

Integrating SD Into Your Project

Micrium OS IO-SD is composed of several components, each of which is a set of files that implement specific functions. To use IO-SD, you must add these files to your project and populate your RTOS Description File .

Starting the IO SD Module Quickly

Micrium offers a quick example application that demonstrates how to initialize, add a controller, and start the Micrium OS IO SD module. We highly recommend that you start with one of these examples.

Configuring IO-SD

Micrium OS IO-SD module can be configured to optimize memory usage and features. See SD Core Run-Time Configurations for more information about configuring the SD core module at run-time.

SD Configuration

This section describes the configurations related to Micrium OS IO-SD.

- SD Core Run-Time Configurations

SD Core Run-Time Configurations

- Optional Pre-Init Configuration
  - Memory Segment
  - Maximum Number of SDIO Function Handles
  - Maximum Number of SD Events
  - Maximum Number of SD Data Transfers
  - Event Functions
  - Core Task Stack
  - Async Task Stack
  - Optional Post-Init Configurations
  - Standard Requests Timeout
  - Core Task Priority
  - Async Task Priority

This section describes Micrium OS IO-SD core run-time configurations.These configurations are optional and can usually be ignored until the very late stages of your application design.

Optional Pre-Init Configuration

Pre-init configuration can be performed by calling dedicated configuration functions before IO_Init() is called. If no explicit pre-init configuration is performed, default values will be used. These default values are stored in SD_InitCfgDflt, which is defined in sd.c.

Memory Segment

This module allocates control data and buffers used for data transfers with the cards. It has the ability to use different memory segments for the control data and the data buffers.

Type	Function to call	Default	Field from default configuration structure
MEM_SEG*	SD_ConfigureMemSeg()	General-purpose heap	.MemSegPtr .MemSegBufPtr

Maximum Number of SDIO Function Handles

Configures the maximum number of SDIO functions can be used simultaneously.

Type	Function to call	Default	Field from default configuration structure
CPU_SIZE_T	SD_ConfigureIO_FnctHandleQty()	Unlimited number of SDIO function handles	.IO_FnctQtyTot

Maximum Number of SD Events

Configures the maximum number of SD events that can be queued. Events include card connection and disconnection, SDIO card interruptions, and data transfer completions.

Type	Function to call	Default	Field from default configuration structure
CPU_SIZE_T	SD_ConfigureEventQty()	Unlimited number of SD events	.EventQtyTot

Maximum Number of SD Data Transfers

Configures the maximum number of SD data transfers that can be submitted simultaneously.

Type	Function to call	Default	Field from default configuration structure
CPU_SIZE_T	SD_ConfigureXferQty()	Unlimited number of SD data transfers	.XferQtyTot

Event Functions

Configures a set of application callbacks to be called on certain SD events (card insertion, card removal, etc.).

Type	Function to call	Default	Field from default configuration structure
SD_EVENT_FNCTS	SD_ConfigureEventFncts()	None.	.EventFnctsPtr

Core Task Stack

The core task handles all the card-related events such as card insertion, removal, etc. This configuration allows you to set the stack pointer and the stack size (in quantity of elements).

Type	Function to call	Default	Field from default configuration structure
CPU_INT32Uvoid *	SD_ConfigureCoreTaskStk()	A stack of 512 elements allocated on Common 's memory segment.	.CoreTaskStkPtr .CoreTaskStkSizeElements

Async Task Stack

The async task handles all the data transfers. This configuration allows you to set the stack pointer and the stack size (in quantity of elements).

Type	Function to call	Default	Field from default configuration structure
CPU_INT32Uvoid *	SD_ConfigureAsyncTaskStk()	A stack of 512 elements allocated on Common 's memory segment.	.AsyncTaskStkPtr .AsyncTaskStkSizeElements

Optional Post-Init Configurations

This section describes the configurations that can be set at any time during execution *after* you called the function IO_Init().

These configurations are optional. If you do not set them in your application, the default configurations will apply.

Standard Requests Timeout

Timeout, in milliseconds, for the SD card operations before timing out.The SD controller implements a hardware timeout. This timeout has been implemented only as a safety feature, as it should never expire. You should not set this timeout to a value lower than 200 ms.

Type	Function to call	Default
CPU_INT32U	SD_OperationsTimeoutSet()	Infinite

Core Task Priority

The IO-SD module will create a task that handles the SD card-related events. You can change the priority of the created task at any time.

Type	Function to call	Default
RTOS_TASK_PRIO	SD_CoreTaskPrioSet()	See Appendix A - Internal Tasks .

Async Task Priority

The IO-SD module will create a task that handles the SD data transfers. You can change the priority of the created task at any time.

Type	Function to call	Default
RTOS_TASK_PRIO	SD_AsyncTaskPrioSet()	See Appendix A - Internal Tasks .

SD Programming Guide

- Initial Setup of SD Module

Initial Setup of SD Module

This section describes the basic steps required to initialize the SD module and to add an SD controller.

- Initializing the IO-SD Module
- Adding Your SD Bus Controller(s)
- Starting Your SD Bus Controller(s)

Initializing the IO-SD Module

The first step is to initialize the SD Bus module core. This is done by calling the function IO_Init().

Note that the function IO_Init() will initialize all the IO sub-modules you have in your project, so there is no need to call it for each sub-module. For instance, if you have an SPI master in your project as well, you must not call IO_Init() twice.

Listing - Example of Call to IO_Init() in the *Initial Setup of SD Module* page shows an example of a call to IO_Init().

Listing - Example of Call to IO_Init()

```
RTOS_ERR  err;

IO_Init(&err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

Adding Your SD Bus Controller(s)

Once you have successfully initialized the IO-SD module, you can start adding your SD bus controller(s). This is done by calling the function SD_BusAdd(). This function must be called for each SD bus controller you want to add.

Listing - Example of Call to SD_BusAdd() in the *Initial Setup of SD Module* page shows an example of a call to SD_BusAdd() using default arguments. In this example, SD bus controller "sd0" is added to the IO-SD module. For more information on how to register an SD Bus controller (if you need to write your own BSP), see SD Bus Controller Registration to the Platform Manager .

Listing - Example of Call to SD_BusAdd()

```
RTOS_ERR       err;
SD_BUS_HANDLE  sd_handle;

sd_handle = SD_BusAdd("sd0",
                      &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

Starting Your SD Bus Controller(s)

Once you have successfully added your SD Bus controller(s), you must start it/them. This is done by calling the function SD_BusStart(). This function must be called for each SD Bus controller you have added.

Listing - Example of Call to SD_BusStart() in the *Initial Setup of SD Module* page shows an example of call to SD_BusStart().

Listing - Example of Call to SD_BusStart()

```
RTOS_ERR  err;

SD_BusStart(sd_handle,
            &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

SD Hardware Porting Guide

The Micrium OS SD Bus module uses a hardware driver that can be customized for any SD Bus controller using a *Board Support Package* (BSP). The BSP has two purposes:

- It initializes and configures any resources needed by the SD Bus controller, but which are provided by an external module.
- It provides hardware information to the SD Bus driver.

We provide example BSPs for some popular platforms. If one is available for your platform, we recommend that you use it as a starting point. However, if no example BSP is available for your platform, the information in this section will help you understand how to correctly port the SD Bus module to your platform.

Note that each SD Bus controller (that you are planning to use) will require a BSP, and all the steps described in this section must be performed for each of them.

- SD BSP Functions Guide
- SD Hardware Information
- SD Controller Registration to the Platform Manager

SD BSP Functions Guide

A Board Support Package contains a set of functions that support your specific hardware platform on Micrium OS. These functions are called by the SD Bus driver to perform hardware configuration, or initialization of IO pins, interrupts, and so on. It is your responsibility to either create these functions from scratch or to modify example code to fit your needs and the specifics of your hardware.

Each of these functions is described below.

- Initialize
- Clock Enable
- I/O Configuration
- Interrupt Configuration
- Power Configuration
- Start
- Stop
- Clock Frequency Get
- Signal Volt Set
- Capabilities Get
- ISR Handling

Initialize

The first function you should implement is a generic initialization function. This function is called by the driver only once at initialization. Listing - Init Function Signature in the *SD BSP Functions Guide* page shows the signature of this function.

Listing - Init Function Signature

```
CPU_BOOLEAN  BSP_SD_SDHC_Init (SD_CARD_CTRLR_ISR_HANDLE_FNCT   isr_fnct,
                               SD_CARD_DRV                   *p_sd_card_drv);
```

Its purpose is to initialize and allocate resources that will be necessary to the SD Bus controller.

The initialization function receives two arguments: isr_fnct and p_sd_card_drv, which are used when handling interrupts from the SD Bus controller. The argument isr_fnct is a pointer to a driver function that must be called when an SD Bus interrupt occurs, and this driver function takes p_sd_card_drv as an argument. So it is important to save these as global variables in your BSP.

You must return DEF_OK from this function if it executes successfully, or DEF_FAIL, otherwise.

Clock Enable

The Clock Enable function is called by the driver when the SD Bus controller is started. Listing - Clock Enable Function Signature in the *SD BSP Functions Guide* page shows the signature of the function.

Listing - Clock Enable Function Signature

```
CPU_BOOLEAN  BSP_SD_SDHC_ClkEn (void);
```

Its purpose is to enable the source clock of the SD Bus controller.

You must return DEF_OK from this function if it executes successfully, or DEF_FAIL, otherwise.

SILICON LABS

I/O Configuration

The I/O Configuration function is called by the driver when the SD Bus controller is started. Implementing this function is not mandatory if the SD Bus controller I/O does not require configuration. Listing - IO Configure Function Signature in the *SD BSP Functions Guide* page shows the signature of the function.

Listing - IO Enable Function Signature

```
CPU_BOOLEAN  BSP_SD_SDHC_IO_Cfg (void);
```

Its purpose is to configure the I/O pins for the SD Bus controller.

You must return DEF_OK from this function if it executes successfully, or DEF_FAIL, otherwise.

Interrupt Configuration

The Interrupt Configuration function is called by the driver when the SD Bus controller is started. Listing - Interrupt Configure Function Signature in the *SD BSP Functions Guide* page shows the signature of the function.

Listing - Interrupt Configure Function Signature

```
CPU_BOOLEAN  BSP_SD_SDHC_IntCfg (void);
```

Its purpose is to configure the interrupt(s) of the SD Bus controller.

You must return DEF_OK from this function if it executes successfully, or DEF_FAIL, otherwise.

Power Configuration

The Power Configuration function is called by the driver when the SD Bus controller is started. Implementing this function is not mandatory if the SD Bus controller power supply does not require configuration. Listing - Power Configure Function Signature in the *SD BSP Functions Guide* page shows the signature of the function.

Listing - Power Configure Function Signature

```
CPU_BOOLEAN  BSP_SD_SDHC_PwrCfg (void);
```

Its purpose is to configure the power of the SD Bus controller.

You must return DEF_OK from this function if it executes successfully, or DEF_FAIL, otherwise.

Start

The Start function is called by the driver each time the SD Bus controller is started. It is always called after all the "configure" functions. Implementing this function is not mandatory if there is nothing further to do when the controller is started. Listing - Start Function Signature in the *SD BSP Functions Guide* page shows the signature of the function.

Listing - Start Function Signature

```
CPU_BOOLEAN  BSP_SD_SDHC_Start (void);
```

Its main purpose is to perform any operation required when the SD Bus controller is started.

You must return DEF_OK from this function if it executes successfully, or DEF_FAIL, otherwise.

Stop

The Stop function is called by the driver when the SD Bus controller is stopped. Implementing this function is not mandatory if there is nothing further to do when the controller is stopped. Listing - Stop Function Signature in the *SD BSP Functions Guide* page shows the signature of the function.

Listing - Stop Function Signature

```
CPU_BOOLEAN  BSP_SD_SDHC_Stop (void);
```

Its purpose is to perform any operation required when the SD Bus controller is stopped.

You must return DEF_OK from this function if it executes successfully, or DEF_FAIL, otherwise.

Clock Frequency Get

The Clock Frequency Get function is called by the driver when the SD bus clock is configured. It is used to determine the clock divider that must be used to achieve the desired SD bus clock. The function must always return the clock frequency, in hertz, of the clock that feeds the SD Bus controller.

Listing - Clock Frequency Get Function Signature in the *SD BSP Functions Guide* page shows the signature of the function.

Listing - Clock Frequency Get Function Signature

```
CPU_INT32U  BSP_SD_SDHC_ClkFreqGet (void);
```

Signal Volt Set

The Signal Volt Set function is used to set the level of voltage for the signaling lines. The SD_CARD_SIGNAL_VOLT enum contains the voltage level to be set.

SD_CARD_SIGNAL_VOLT can take this value:

- SD_CARD_SIGNAL_VOLT_1_8
- SD_CARD_SIGNAL_VOLT_3_3
- SD_CARD_SIGNAL_VOLT_AUTO

Listing - Signal Volt Set Function Signature in the *SD BSP Functions Guide* page shows the signature of the function.

Listing - Signal Volt Set Function Signature

```
CPU_BOOLEAN  BSP_SD_SDHC_SignalVoltSet  (SD_CARD_SIGNAL_VOLT  volt);
```

You must return DEF_OK from this function if it executes successfully, or DEF_FAIL, otherwise.

Capabilities Get

The Capabilities Get function is called by the driver when the SD Bus is started. It is used to determine the capabilities of the hardware. The driver will compute the capabilities based on the SD Host controller's reported capabilities. However, if for some reason, some capabilities are not available on your board, this function allows you to override the capabilities reported by the SD host controller. The function must always return the host capabilities as a bitmap of capabilities. This bitmap contains the host controller voltage operation and it bus width.

SD_HOST_CAPABILITIES can take this value :

Bus Voltage

- SD_CAP_BUS_VOLT_3_3
- SD_CAP_BUS_VOLT_3
- SD_CAP_BUS_VOLT_1_8

Bus Signal Voltage

- SD_CAP_BUS_SIGNAL_VOLT_3_3
- SD_CAP_BUS_SIGNAL_VOLT_1_8

Bus Width

- SD_CAP_BUS_WIDTH_1_BIT
- SD_CAP_BUS_WIDTH_4_BIT
- SD_CAP_BUS_WIDTH_8_BIT

Listing - Capabilities Get Function Signature in the *SD BSP Functions Guide* page shows the signature of the function.

Listing - Capabilities Get Function Signature

```
void  BSP_SD_SDHC_CapabilitiesGet (SD_HOST_CAPABILITIES  *p_capabilities);
```

ISR Handling

Each SD Bus controller driver has an ISR handler function that must be called each time an SD Bus interrupt is triggered. However, for most platforms, it will be necessary to implement an intermediate ISR handler in the BSP. This BSP ISR will then call the driver's ISR handler. This is necessary, as some interrupt controllers may require the interrupt status to be cleared each time it is triggered. It is also necessary if your interrupt controller does not support passing an argument to the interrupt vector, as the driver's ISR handler takes the p_sd_card_drv received in the Init() function of the BSP as an argument.

Listing - Example of BSP ISR Implementation in the *SD BSP Functions Guide* page shows an example of an ISR handler implemented in the BSP if using the CMSIS standard.

Listing - Example of BSP ISR Implementation

```
void SDIO_IRQHandler (void)
{
   OSIntEnter();
   BSP_SD_SDHC_ISR_Fnct(BSP_SD_SDHC_DrvPtr);
   OSIntExit();
}
```

SD Hardware Information

- Hardware Driver Information
- BSP API Structure
- Device Hardware Information

Hardware Driver Information

The SD Bus driver requires information about the SD Bus controller on your MCU, which you can provide using a structure of type SD_CARD_CTRLR_DRV_INFO. This information can be found in the manual for your MCU.

Table - SD_CARD_CTRLR_HW_INFO Structure in the *SD Hardware Information* page describes each configuration field available in this structure.

Table - SD_CARD_CTRLR_HW_INFO Structure

<table>
<tr><th>Field</th><th colspan="2">Description</th></tr>
<tr><td>.BaseAddr</td><td colspan="2">Base address of the SD Bus controller registers set. This corresponds to the address of the first register.</td></tr>
<tr><td rowspan="5">.CardSignalVolt</td><td colspan="2">Signal voltage required by SD card. Can be one of following values:</td></tr>
<tr><th>Value</th><th>Description</th></tr>
<tr><td>SD_CARD_BUS_SIGNAL_VOLT_AUTO</td><td>Signal voltage starts at 3.3V. Signaling voltage will be switched to 1.8V during the initialization sequence if the card supports it. This is the default value.</td></tr>
<tr><td>SD_CARD_BUS_SIGNAL_VOLT_3_3</td><td>Signal voltage starts and remains at 3.3V regardless if the card supports 1.8V signaling or not.</td></tr>
<tr><td>SD_CARD_BUS_SIGNAL_VOLT_1_8</td><td>Signal voltage starts and remains at 1.8V.</td></tr>
<tr><td rowspan="5">.CardDetectMode</td><td colspan="2">Mode to detect card connection or disconnection. There are three possible modes:</td></tr>
<tr><th>Value</th><th>Description</th></tr>
<tr><td>SD_CARD_DETECT_MODE_INTERRUPT</td><td>In this mode, the card connection/disconnection is reported to the IO-SD core task using the SD Host controller interrupt. Use this mode when your Card Detect (CD) pin is properly connected to the SD Host controller.</td></tr>
<tr><td>SD_CARD_DETECT_MODE_BSP_EVENT</td><td>In this mode, the card connection/disconnection is reported to the IO-SD core task by calling explicitly specific event functions, SD_BSP_BusCardDetectEvent() and SD_BSP_BusCardRemoveEvent() . These functions are generally called from the BSP. Use this mode when your CD pin is not properly connected but you can detect the card insertion/removal via a GPIO. The GPIO could be configured to trigger an interrupt to which you can associate an interrupt handler that will call a specific event function. Or you may have your application polling this GPIO to determine if a card is inserted or not.</td></tr>
<tr><td>SD_CARD_DETECT_MODE_WIRED</td><td>In this mode, the card connection/disconnection is not reported to the IO-SD core task since the card is permanently wired to the SD Host controller. Instead upon bus start (SD_BusStart()) and stop (SD_BusStop()) operations, the card will be respectively initialized and removed accordingly.</td></tr>
<tr><td>.InfoExtPtr</td><td colspan="2">Pointer to an extended hardware information structure. Some drivers may require extra information, so the format of this structure is specific to your driver. Most of the time this can be set to DEF_NULL.</td></tr>
</table>

BSP API Structure

In order to provide a pointer to the BSP functions for the SD Bus controller driver, you must create a structure of type SD_CARD_CTRLR_BSP_API.

Table - SD_CARD_CTRLR_BSP_API Structure in the *SD Hardware Information* page describes each field available in this structure.

Table - SD_CARD_CTRLR_BSP_API Structure

Field	Description
.Init	Pointer to the BSP initialization function.
.ClkEn	Pointer to the BSP clock enable function.
.IO_Cfg	Pointer to the BSP I/O configure function.
.IntCfg	Pointer to the BSP interrupt configure function.
.PwrCfg	Pointer to the BSP power configure function.
.Start	Pointer to the BSP start function.
.Stop	Pointer to the BSP stop function.
.ClkFreqGet	Pointer to the BSP clock frequency get function.
.SignalVoltSet	Pointer to the BSP signal voltage set function.
.CapabilitiesGet	Pointer to the BSP capabilities get function.
.BSP_API_ExtPtr	Pointer to extended BSP API functions that could be required by your driver. Most of the time this can be set to DEF_NULL.

Listing - Example of BSP API Structure in the *SD Hardware Information* page shows an example of a BSP API structure.

Listing - Example of BSP API Structure

```
const SD_CARD_CTRLR_BSP_API BSP_SD_SDHC_BSP_API = {
  .Init            = BSP_SD_SDHC_Init,
  .ClkEn           = BSP_SD_SDHC_ClkEn,
  .IO_Cfg          = BSP_SD_SDHC_IO_Cfg,
  .IntCfg          = BSP_SD_SDHC_IntCfg,
  .PwrCfg          = BSP_SD_SDHC_PwrCfg,
  .Start           = BSP_SD_SDHC_Start,
  .Stop            = BSP_SD_SDHC_Stop,
  .ClkFreqGet      = BSP_SD_SDHC_ClkFreqGet,
```

```
.CapabilitiesGet = BSP_SD_SDHC_CapabilitiesGet,.BSP_API_ExtPtr = DEF_NULL
};
```

Device Hardware Information

The last step is to create the main device hardware information structure.

Table - SD_CARD_CTRLR_DRV_INFO Structure in the *SD Hardware Information* page describes each configuration field available in this structure.

Table - SD_CARD_CTRLR_DRV_INFO Structure

Field	Description
.HW_Info	Structure explained above in Hardware Driver Information .
.DrvAPI_Ptr	Pointer to the driver API structure you are using with your driver. Some drivers may provide more than one API structure.
.BSP_API_Ptr	Pointer to the BSP API structure as described above in BSP API Structure .

SD Controller Registration to the Platform Manager

Once the hardware information structure for your SD Bus controller is ready, it must be registered with the Platform Manager . This should typically be done using the BSP_OS_Init() function that is located in the file bsp_os.c.

There is a macro located in the file sd.h that you can call to register an SD Bus controller. The macro is named IO_SD_CARD_CTRLR_REG().

Listing - Example of SD Bus Controller Registration in the *SD Controller Registration to the Platform Manager* page shows an example of how to register an SD Bus controller.

Listing - Example of SD Bus Controller Registration

```
#include  <rtos_description.h>
#if defined(RTOS_MODULE_IO_SD_AVAIL)
#include  <rtos/io/include/sd.h>
#include  <rtos/io/include/sd_card.h>
#endif

#if defined(RTOS_MODULE_IO_SD_AVAIL)                (1)
BSP_HW_INFO_EXT(const  SD_CARD_CTRLR_DRV_INFO, BSP_SD_SDHC_BSP_DrvInfo);
#endif

void  BSP_OS_Init (void)
{

    /* ... */

                                        /* ------------- REGISTER SD CONTROLLERS ------------- */
#if defined(RTOS_MODULE_IO_SD_AVAIL)
    IO_SD_CARD_CTRLR_REG("sd0", &BSP_SD_SDHC_BSP_DrvInfo);
#endif
}
```

(1) Since the hardware information global variables are declared in another file, you must declare them as external in your bsp_os.c file. Always use the BSP_HW_INFO_EXT() macro.

SPI Master

SPI Master

- Integrating SPI Into Your Project
- SPI Configuration
- SPI Programming Guide
- SPI Hardware Porting Guide

Integrating SPI Into Your Project

Micrium OS IO-SPI is composed of several components, each of which is a set of files that implement specific functions. To use IO-SPI, you must add these files to your project and populate your RTOS description file .

Starting the IO SPI Module Quickly

Micrium offers quick example applications that demonstrate how to initialize, add a controller, and start the Micrium OS IO SPI module. We highly recommend that you start with one of these examples.

SPI Configuration

This section describes the configurations related to Micrium OS IO-SPI.

- SPI Core Run-Time Configurations
- SPI Slave Configurations

SPI Core Run-Time Configurations

- Core Configuration
  - Optional Pre-Init Configuration

This section describes Micrium OS IO-SPI core run-time configurations.These configurations are optional and can usually be ignored until the very late stages of your application design.

Core Configuration

Optional Pre-Init Configuration

Pre-init configuration can be performed by calling dedicated configuration functions before IO_Init() is called. If no explicit pre-init configuration is performed, default values will be used. These default values are stored in SPI_InitCfgDflt, which is defined in spi.c.

Table - IO-SPI Optional Pre-Init Configurations

Configurations	Description	Type	Function to call	Default	Field from default configuration structure
Memory segment	Configures the memory segment where the core internal data structures will be allocated.	MEM_SEG*	SPI_ConfigureMemSeg()	General-purpose heap	.MemSegPtr
Maximum number of slave handles	Configures the maximum number of SPI slaves that will be opened simultaneously.	CPU_SIZE_T	SPI_ConfigureSlaveHandleQty()	Unlimited number of slave handles	.SlaveHandleQty

SPI Slave Configurations

Opening an SPI slave is done by calling the function SPI_SlaveOpen(). This function takes one configuration argument which is described below.

p_slave_info

p_slave_info is a pointer to a configuration structure of type SPI_SLAVE_INFO. Its purpose is to provide the SPI module with basic information regarding the slave, such as clock frequency, frame size, etc.

Table - SPI_SLAVE_INFO Configuration Structure in the *SPI Slave Configurations* page describes each configuration field available in this configuration structure.

Table - SPI_SLAVE_INFO Configuration Structure

Field	Description
.Mode	SPI communication mode. Allows you to configure the CPHA and CPOL mode. This field is a bitmap. Possible values are: - SERIAL_SPI_BUS_MODE_CPHA_BIT - SERIAL_SPI_BUS_MODE_CPOL_BIT
.BitsPerFrame	Number of bits per frame.
.LSbFirst	Indicates if Least Significant Bits (LSB) must be transferred first.
.SClkFreqMax	Clock frequency, in hertz.
.SlaveID	ID of the slave. Refer to your SPI BSP to find out the ID of the SPI slave you want to use.
.TxDummyByte	When calling SPI_SlaveRx(), dummy bytes must be transferred to the slave to generate a clock. This configures the value of the dummy byte.
.ActiveLow	Indicates if the Slave Select (or Chip Select) signal must be low when the SPI slave is selected. Most SPI slaves are active low so this should generally be set to DEF_YES.

SPI Programming Guide

- Initial Setup of SPI Module
- SPI Slaves Management

Initial Setup of SPI Module

This section describes the basic steps required to initialize the SPI module and to add an SPI controller.

- Initializing the IO-SPI Module
- Adding Your SPI Bus Controller(s)
- Starting Your SPI Bus Controller(s)

Initializing the IO-SPI Module

The first step is to initialize the SPI module core. This is done by calling the function IO_Init().

Listing - Example of Call to SPI_Init() in the *Initial Setup of SPI Module* page shows an example of a call to SPI_Init().

Listing - Example of Call to SPI_Init()

```
RTOS_ERR  err;

SPI_Init(&err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

Adding Your SPI Bus Controller(s)

Once you successfully initialized the IO-SPI module, you can start adding your SPI Bus controller(s). This is done by calling the function SPI_BusAdd(). This function must be called for each SPI Bus controller you want to add.

Listing - Example of Call to SPI_BusAdd() in the *Initial Setup of SPI Module* page shows an example of a call to SPI_BusAdd() using default arguments. In this example, SPI controller "spi0" is added to the IO-SPI module. For more information on how to register an SPI Bus controller (if you have to write your own BSP), see SPI Controller Registration to the Platform Manager .

Listing - Example of Call to SPI_BusAdd()

```
RTOS_ERR        err;
SPI_BUS_HANDLE  spi_handle;

spi_handle = SPI_BusAdd("spi0",
                        &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

Starting Your SPI Bus Controller(s)

Once you successfully added your SPI Bus controller(s), you must start it/them. This is done by calling the function SPI_BusStart(). This function must be called for each SPI Bus controller you added.

Listing - Example of Call to SPI_BusStart() in the *Initial Setup of SPI Module* page gives an example of call to SPI_BusStart().

Listing - Example of Call to SPI_BusStart()

```
RTOS_ERR  err;

SPI_BusStart(spi_handle,
             &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

SPI Slaves Management

This section describes the different operations related to SPI slaves.

- Adding a Slave
- Selecting/Deselecting a Slave
- Communicating with a Slave
    - Receiving Data
    - Transmitting Data
    - Transferring Data

Adding a Slave

Before communicating with a slave, the slave must be opened on the bus that was previously added. Opening a slave is done by calling the function SPI_SlaveOpen().

Listing - Example of Call to SPI_SlaveOpen() in the SPI Slaves Management page shows an example of a call to SPI_SlaveOpen() that will open a slave. For more information on the configuration arguments to pass to SPI_SlaveOpen(), see SPI Slave Configurations .

Listing - Example of Call to SPI_SlaveOpen()

```
RTOS_ERR         err;
SPI_SLAVE_HANDLE  slave_handle;
SPI_SLAVE_INFO    slave_info;

slave_info.Mode         = SERIAL_SPI_BUS_MODE_CPHA_BIT | SERIAL_SPI_BUS_MODE_CPOL_BIT;
slave_info.BitsPerFrame = 8u;
slave_info.LSbFirst     = DEF_YES;
slave_info.SClkFreqMax  = 400000u;
slave_info.SlaveID      = 0u;
slave_info.TxDummyByte  = 0xFFu;
slave_info.ActiveLow    = DEF_YES;
slave_handle = SPI_SlaveOpen(bus_handle,
                             &slave_info,
                             &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occured. Error handling should be added here. */
}
```

Selecting/Deselecting a Slave

It is possible to explicitly select and deselect a slave. This is done by calling the functions SPI_SlaveSel() and SPI_SlaveDesel().

While a slave is selected, it is not possible to select another slave on the bus, and only the task that selected the slave can communicate with it.

> **Note that it is not mandatory to explicitly select and deselect a slave via the functions SPI_SlaveSel() and SPI_SlaveDesel() before communicating with it. If the slave has not been selected before, it will automatically be selected temporarily for the time of the communication operation.**

Listing - Example of Call to SPI_SlaveSel() and SPI_SlaveDesel() in the *SPI Slaves Management* page shows an example of a call to SPI_SlaveSel() and SPI_SlaveDesel().

Listing - Example of Call to SPI_SlaveSel() and SPI_SlaveDesel()

```
RTOS_ERR  err;

SPI_SlaveSel(slave_handle,
             5000u,                 /* 5 sec timeout. 0 for infinite.          */
             SPI_OPT_NONE,
            &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}

/* ... Communication operations with the slave...   */

SPI_SlaveDesel(slave_handle,
               &err);
if (err.Code != RTOS_ERR_NONE) {
```

```
/* An error occurred. Error handling should be added here. */}
```

Communicating with a Slave

Receiving Data

Receiving data from a slave is done by calling the function SPI_SlaveRx(). Listing - Example of Call to SPI_SlaveRx() in the *SPI Slaves Management* page shows an example of a call to SPI_SlaveRx().

Listing - Example of Call to SPI_SlaveRx()

```
RTOS_ERR    err;
CPU_INT08U  rx_buf[64u];

SPI_SlaveRx(slave_handle,
            rx_buf,
            64u,
            5000u,
           &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

Transmitting Data

Transmitting data to a slave is done by calling the function SPI_SlaveTx(). Listing - Example of Call to SPI_SlaveTx() in the *SPI Slaves Management* page shows an example of a call to SPI_SlaveTx().

Listing - Example of Call to SPI_SlaveTx()

```
RTOS_ERR    err;
CPU_INT08U  tx_buf[] = "Data to transmit\r\n";

SPI_SlaveTx(slave_handle,
            tx_buf,
            sizeof(tx_buf),
            5000u,
           &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

Transferring Data

Transferring data with a slave (transmit and receive data simultaneously) is done by calling the function SPI_SlaveXfer(). Listing - Example of Call to SPI_SlaveXfer() in the *SPI Slaves Management* page shows an example of a call to SPI_SlaveXfer().

Listing - Example of Call to SPI_SlaveXfer()

```
RTOS_ERR    err;
CPU_INT08U  rx_buf[18u];
CPU_INT08U  tx_buf[] = "Data to transmit\r\n";

SPI_SlaveXfer(slave_handle,
              rx_buf,
              tx_buf,
              sizeof(tx_buf),
              5000u,
             &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

SPI Hardware Porting Guide

The Micrium OS IO-SPI module uses a hardware driver that can be customized for any SPI controller using a *Board Support Package* (BSP). The BSP has two purposes:

- It initializes and configures any resources needed by the SPI controller, but which are provided by an external module.
- It provides hardware information to the SPI driver.

We provide example BSPs for some popular platforms. If one is available for your platform, we recommend that you use it as a starting point. However, if no example BSP is available for your platform, the information in this section will help you understand how to correctly port the SPI module to your platform.

Note that each SPI controller (that you are planning to use) will require a BSP, and all the steps described in this section must be performed for each of them.

- SPI BSP Functions Guide
- SPI Hardware Information
- SPI Controller Registration to the Platform Manager

SPI BSP Functions Guide

A Board Support Package contains a set of functions that support your specific hardware platform on Micrium OS. These functions are called by the SPI driver to perform hardware configuration or initialization of IO pins, interrupts, etc. It is your responsibility to either create these functions from scratch, or to modify example code to fit your needs and the specifics of your hardware.

Each of these functions is described below.

- Initialize
- Clock Configuration
- I/O Configuration
- Interrupt Configuration
- Power Configuration
- Start
- Stop
- Slave Select
- Slave Deselect
- Clock Frequency Get
- ISR Handling

Initialize

The first function you should implement is a generic initialization function. This function is called by the driver only once at initialization. Listing - Init Function Signature in the *SPI BSP Functions Guide* page shows the signature of this function.

Listing - Init Function Signature

```
CPU_BOOLEAN  BSP_Serial_<ctrlr>_Init (SERIAL_CTRLR_ISR_HANDLE_FNCT   isr_fnct,
                                      SERIAL_DRV                    *p_ser_drv);
```

Its purpose is to initialize and allocate resources that will be necessary to the SPI controller.

The initialization function receives two arguments: isr_fnct and p_ser_drv, which are used when handling interrupts from the SPI controller. The argument isr_fnct is a pointer to a driver function that must be called when an SPI interrupt occurs, and this driver function takes p_ser_drv as an argument. So it is important to save these as global variables in your BSP.

You must return DEF_OK from this function if it executes successfully, or DEF_FAIL, otherwise.

Clock Configuration

The clock configure function is called by the driver when the SPI controller is started. Implementing this function is not mandatory if the SPI controller clock does not require configuration. Listing - Clock Configure Function Signature in the *SPI BSP Functions Guide* page shows the signature of the function.

Listing - Clock Configure Function Signature

```
CPU_BOOLEAN  BSP_Serial_<ctrlr>_ClkCfg (void);
```

Its purpose is to configure the source clock of the SPI controller.

You must return DEF_OK from this function if it executes successfully, or DEF_FAIL, otherwise.

I/O Configuration

The I/O configure function is called by the driver when the SPI controller is started. Implementing this function is not mandatory if the SPI controller I/O does not require configuration. Listing - IO Configure Function Signature in the *SPI BSP Functions Guide* page shows the signature of the function.

Listing - IO Configure Function Signature

```
CPU_BOOLEAN  BSP_Serial_<ctrlr>_IO_Cfg (void);
```

Its purpose is to configure the I/O pins for the SPI controller.

You must return DEF_OK from this function if it executes successfully, or DEF_FAIL, otherwise.

Interrupt Configuration

The interrupt configure function is called by the driver when the SPI controller is started. Listing - Interrupt Configure Function Signature in the *SPI BSP Functions Guide* page shows the signature of the function.

Listing - Interrupt Configure Function Signature

```
CPU_BOOLEAN  BSP_Serial_<ctrlr>_IntCfg (void);
```

Its purpose is to configure the interrupt(s) of the SPI controller.

You must return DEF_OK from this function if it executes successfully, or DEF_FAIL, otherwise.

Power Configuration

The power configure function is called by the driver when the SPI controller is started. Implementing this function is not mandatory if the SPI controller power supply does not require configuration. Listing - Power Configure Function Signature in the *SPI BSP Functions Guide* page shows the signature of the function.

Listing - Power Configure Function Signature

```
CPU_BOOLEAN  BSP_Serial_<ctrlr>_PwrCfg (void);
```

Its purpose is to configure the power of the SPI controller.

You must return DEF_OK from this function if it executes successfully, or DEF_FAIL, otherwise.

Start

The start function is called by the driver each time the SPI controller is started. It is always called after all the "configure" functions. Implementing this function is not mandatory if there is nothing further to do when the controller is started. Listing - Start Function Signature in the *SPI BSP Functions Guide* page shows the signature of the function.

Listing - Start Function Signature

```
CPU_BOOLEAN  BSP_Serial_<ctrlr>_Start (void);
```

Its main purpose is to perform any operation required when the SPI controller is started.

You must return DEF_OK from this function if it executes successfully, or DEF_FAIL, otherwise.

Stop

The stop function is called by the driver when the SPI controller is stopped. Implementing this function is not mandatory if there is nothing further to do when the controller is stopped. Listing - Stop Function Signature in the *SPI BSP Functions Guide* page shows the signature of the function.

Listing - Stop Function Signature

```
CPU_BOOLEAN  BSP_Serial_<ctrlr>_Stop (void);
```

Its purpose is to perform any operation required when the SPI controller is stopped.

You must return DEF_OK from this function if it executes successfully, or DEF_FAIL, otherwise.

Slave Select

The slave select function is called by the driver when a slave is selected by the application. Listing - Slave Select Function Signature in the *SPI BSP Functions Guide* page shows the signature of the function.

Listing - Slave Select Function Signature

```
CPU_BOOLEAN  BSP_Serial_<ctrlr>_SlaveSel (const  SERIAL_SLAVE_INFO  *p_slave_info);
```

Its purpose is to perform any operation required when a slave is selected.

Implementing this function is not mandatory if the SPI's driver supports built-in slave select lines. Otherwise, you have to implement it yourself; for example, by using the GPIO interface to control the slave selection.

The function receives one argument of type SERIAL_SLAVE_INFO which provide information regarding the slave. Table below shows a description of each filed available in this structure.

Field	Description
.Addr	Address/ID of the slave.
.ActiveLow	Flag that indicates if the slave is active when the Slave select line is low.

You must return DEF_OK from this function if it executes successfully, or DEF_FAIL, otherwise.

Slave Deselect

The slave select function is called by the driver when a slave is deselected by the application. It has the same signature as the slave select function. Refer to the slave select function for more information.

Clock Frequency Get

The clock frequency get function is called by the driver when a slave clock is configured. It is used to determine the clock divider that must be used to achieve the desired slave clock. The function must always return the clock frequency, in hertz, of the clock that feeds the SPI controller.

Listing - Clock Frequency Get Function Signature in the *SPI BSP Functions Guide* page shows the signature of the function.

Listing - Clock Frequency Get Function Signature

```
CPU_INT32U  BSP_Serial_<ctrlr>_ClkFreqGet (void);
```

ISR Handling

Each SPI controller driver has an ISR handler function that must be called each time a SPI interrupt is triggered. However, for most platforms, it will be necessary to implement an intermediate ISR handler in the BSP. This BSP ISR will then call the driver's ISR handler. This is necessary, as some interrupt controllers may require the interrupt status to be cleared each time it is triggered. It is also necessary if your interrupt controller does not support passing an argument to the interrupt vector, as the driver's ISR handler takes the p_ser_drv received in the Init () function of the BSP as an argument.

Also, some driver will require the interrupt event source to be passed to the ISR handler as the second argument. Here are the possible event values:

- SERIAL_CTRLR_ISR_SRC_NONE
- SERIAL_CTRLR_ISR_SRC_RX
- SERIAL_CTRLR_ISR_SRC_TX
- SERIAL_CTRLR_ISR_SRC_ERR
- SERIAL_CTRLR_ISR_SRC_DMA_CMPL_RX
- SERIAL_CTRLR_ISR_SRC_DMA_CMPL_TX

You may browse a Gecko SDK SPI driver demo in Simplicity Studio to view an example of how the ISR handler source event should be handled.

Listing - Example of BSP ISR Implementation in the *SPI BSP Functions Guide* page shows an example of an ISR handler implemented in the BSP if using a Silicon Labs chip (that follows the CMSIS naming standard).

Listing - Example of BSP ISR Implementation

```
void DMA_IRQHandler(void)
{
    /* TODO: Clear interrupt status, if needed. */

    OSIntEnter();
    BSP_Serial_<ctrlr>_DrvISR_Handler(BSP_Serial_<ctrlr>_DrvPtr,
                                      SERIAL_CTRLR_ISR_SRC_NONE);
    OSIntExit();
}
```

SPI Hardware Information

- Hardware Driver Information
- BSP API Structure
- Device Hardware Information

The SPI driver requires information about the SPI controller on your MCU, which you provide using a structure of type SERIAL_CTRLR_DRV_INFO. The controller information can be found in the manual for your MCU.

This structure is used by the macro IO_SERIAL_CTRLR_REG() to register the SPI controller with the Platform Manager . For more details about using the macro IO_SERIAL_CTRLR_REG(), refer to the page SPI Controller Registration to the Platform Manager .

The main SPI hardware information structure contain other structures which are explained in the sections below.

Hardware Driver Information

The structure SERIAL_CTRLR_HW_INFO provides information about the SPI controller's characteristics. This structure is referenced from the main structure SERIAL_CTRLR_DRV_INFO.

Table - SERIAL_CTRLR_HW_INFO Structure in the *SPI Hardware Information* page describes each configuration field available in this structure.

Table - SERIAL_CTRLR_HW_INFO Structure

Field	Description
.SupportedMode	Communication mode(s) supported by the controller. Should be set to SERIAL_CTRLR_MODE_SPI.
.BaseAddr	Base address of the SPI controller registers set.
.InfoExtPtr	Pointer to an extended hardware information structure. Some driver may require extra information. The format of this structure is hence specific to your driver. Most of the time this can be set to DEF_NULL.

If need to create an additional information structure, you can browse Simplicity Studio for an example of a Gecko SDK SPI.

For example, the Gecko SDK SPI driver requires that you select the port where the SPI controller is routed. The port selection is set in a custom hardware information structure of type SERIAL_CTRLR_HW_INFO_EXT_SILICONLABS. The structure itself is defined in serial_drv.h. You should refer to your chip's reference manual and to your board's schematic to find out which port you should use.

Below is an example of initializing and assigning an additional information structure.

```
const SERIAL_CTRLR_HW_INFO_EXT_SILICONLABS BSP_Serial_USART0_SPI_ExtHwInfo = {
 .PortLocationRx = 1u,
 .PortLocationTx = 1u,
 .PortLocationClk = 1u,
 .PortLocationCs = 1u
};

const SERIAL_CTRLR_DRV_INFO BSP_Serial_SiliconLabs_Bus0_DrvInfo = {
   .HW_Info.SupportedMode =  SERIAL_CTRLR_MODE_SPI,
   .HW_Info.BaseAddr      =  0x4000C000u,
   .HW_Info.InfoExtPtr    = &BSP_Serial_USART0_SPI_ExtHwInfo,
   .BSP_API_Ptr           = &BSP_Serial_USART0_SPI_API,
   .DrvAPI_Ptr            = &Serial_CtrlrDrv_API_SiliconLabsGeckoSDK
};
```

BSP API Structure

In order to provide a pointer to the BSP functions for the SPI controller driver, you must create a structure of type SERIAL_CTRLR_BSP_API.

Table - SERIAL_CTRLR_BSP_API Structure in the *SPI Hardware Information* page describes each field available in this structure.

Table - SERIAL_CTRLR_BSP_API Structure

Field	Description
.Init	Pointer to the BSP initialization function.
.ClkEn	Pointer to the BSP clock enable function.
.IO_Cfg	Pointer to the BSP I/O configure function.
.IntCfg	Pointer to the BSP interrupt configure function.
.PwrCfg	Pointer to the BSP power configure function.
.Start	Pointer to the BSP start function.
.Stop	Pointer to the BSP stop function.
.SlaveSel	Pointer to the BSP slave select function.
.SlaveDesel	Pointer to the BSP slave deselect function.
.ClkFreqGet	Pointer to the BSP clock frequency get function.
.DMA_API_Ptr	For future considerations. Must always be set to DEF_NULL.
.BSP_API_ExtPtr	Pointer to extended BSP API functions that could be required by your driver. Most of the time this can be set to DEF_NULL .

Listing - Example of BSP API Structure in the *SPI Hardware Information* page shows an example of a BSP API structure.

Listing - Example of BSP API Structure

```
const  SERIAL_CTRLR_BSP_API  BSP_Serial_<ctrlr>_API = {
    .Init           = BSP_Serial_<ctrlr>_Init,
    .ClkEn          = BSP_Serial_<ctrlr>_ClkEn,
    .IO_Cfg         = BSP_Serial_<ctrlr>_IO_Cfg,
    .IntCfg         = BSP_Serial_<ctrlr>_IntCfg,
    .PwrCfg         = BSP_Serial_<ctrlr>_PwrCfg,
    .Start          = BSP_Serial_<ctrlr>_Start,
    .Stop           = BSP_Serial_<ctrlr>_Stop,
    .SlaveSel       = BSP_Serial_<ctrlr>_SlaveSel,
    .SlaveDesel     = BSP_Serial_<ctrlr>_SlaveDesel,
    .ClkFreqGet     = BSP_Serial_<ctrlr>_ClkFreqGet,
    .DMA_API_Ptr    = DEF_NULL,
    .BSP_API_ExtPtr = DEF_NULL
};
```

Device Hardware Information

The last step is to create the main device hardware information structure.

Table - SERIAL_CTRLR_DRV_INFO Structure in the *SPI Hardware Information* page describes each configuration field available in this structure.

Table - SERIAL_CTRLR_DRV_INFO Structure

Field	Description
.HW_Info	Structure explained at step hardware driver information .
.DrvAPI_Ptr	Pointer to the driver API structure you are using with your driver. Some drivers may provide more than one API structure.
.BSP_API_Ptr	Pointer to the BSP API structure you created at step BSP API Structure .

SPI Controller Registration to the Platform Manager

Once the hardware information structure for your SPI controller is ready, it must be registered with the Platform Manager . This should typically be done using the BSP_OS_Init() function that is located in the file bsp_os.c.

There is a macro located in the file serial.h that you can call to register an SPI controller. The macro is named IO_SERIAL_CTRLR_REG().

Listing - Example of SPI Controller Registration in the *SPI Controller Registration to the Platform Manager* page shows an example of how to register a SPI Device controller.

Listing - Example of SPI Controller Registration

```
#include  <rtos_description.h>
#ifdef RTOS_MODULE_IO_SERIAL_AVAIL
#include  <rtos/io/include/serial.h>
#endif

#if defined(RTOS_MODULE_IO_SERIAL_AVAIL)                        (1)
BSP_HW_INFO_EXT(const  SERIAL_CTRLR_DRV_INFO, BSP_Serial_<ctrlr>_DrvInfo);
#endif

void  BSP_OS_Init (void)
{

    /* ... */

                                        /* ------------- REGISTER SPI CONTROLLERS ------------- */
#if defined(RTOS_MODULE_IO_SERIAL_AVAIL)
    IO_SERIAL_CTRLR_REG("spi0",
                    &BSP_Serial_<ctrlr>_DrvInfo);
#endif
}
```

(1) Since the hardware information global variables are declared in another file, you must declare them as external in your bsp_os.c file. Always use the BSP_HW_INFO_EXT() macro.

Kernel

Kernel

The Micrium OS Kernel, uC/OS 5, is a real-time preemptive kernel. Like the other Micrium OS modules and stacks, the kernel is written for resource-constrained microntrollers, while allowing the full potential of more capable processors. it currently features most services you would expect from a real-time kernel such as resource access synchronization, task management, and memory management. You can find more information about the basic concepts and theory behind a real-time kernel in the Kernel section of the Technologies Overview Manual.

Through the Integration and Configuration pages, you will learn how to integrate the kernel in your project and fine-tune its feature set to better fit your needs. The Programming Guide provides examples on how to use the kernel features. Common problems and their solution are discussed in the Troubleshooting Guide.

Kernel Overview

Kernel Overview

The Micrium OS Kernel is written to the highest coding standards, making its implementation easy to read and understand. The kernel itself is a real-time preemptive kernel and offers many features, such as:

- Task management
- Time management
- Resource synchronization through semaphores, monitors and intelligent mutexes
- Message queues
- Software timers
- Application hooks to extend the features of the kernel
- Instrumentation, such as Tracing and resource utilization tracking

The kernel has support for the more commonly-used microcontroller architectures, such as ARMv6-M, ARMv7-M and ARMv8-M.

Integrating The Kernel Into Your Project

Integrating the Kernel Into Your Project

The Micrium OS Kernel module is composed of several components, each of which is a set of files that implement specific functions. The kernel itself consists of a core component named "Kernel" and a port component named "Port". To use the Kernel, you must add these files to your project and populate your RTOS Description File .

Using Example Projects

Micrium offers a set of example applications that demonstrate some of the features of the Micrium OS kernel, and allow you to start developing your application. We recommend that you start from one of these examples.

Configuring the Kernel

Micrium OS Kernel can be configured to optimize memory usage and overhead. The following pages explain how the kernel and the whole Micrium OS can be configured at compile-time:

- Kernel Compile-Time and Data Type Configurations
- RTOS Compile-Time Configuration
- RTOS ERR Type Configuration

The Kernel Run-Time Application Specific Configurations page and its subpages show how to fine-tune the application-specific features of the Kernel at run-time.

Kernel Configuration

Kernel Configuration

The kernel can be configured to meet any needs your application may have. Most kernel features can be toggled on or off to preserve data space, code size, and execution time. Some of the kernel's features are used only while profiling and debugging and can be disabled for production builds.

There are two configurations types:

- Compile-time configuration : Allows you to enable or disable specific features.
- Run-time configuration : Allows you to specify the size of the kernel's object pools and internal task stacks.

Kernel Compile-Time and Data Type Configurations

- Compile-Time Configurations
  - Miscellaneous Options
  - Event Flag Configuration
  - Mutal Exclusion Semaphore Configuration
  - Message Queue Configuration
  - Semaphore Configuration
  - Monitor Configuration
  - Task Management Options
  - Task Local Storage Configuration
  - Timer Management Configuration

Compile-Time Configurations

Compile-time configuration allows you to specify which features of the kernel to enable. By enabling only the necessary features, you can reduce the code and data size of the kernel (i.e., its footprint).

To configure the compile-time parameters, you must set the #define constants in the os_cfg.h file provided by the application. These #define constants are listed below in the same order as in the os_cfg.h file (default values are in **bold**).

Miscellaneous Options

Table - Miscellaneous Configuration Constants

Constant	Description	Possible values
OS_CFG_APP_HOOKS_EN	Enables application-defined hooks that can be called by the kernel port hooks. For more information, see Extending the Kernel with Application Hooks .	1 (enabled) or **0** (disabled)
OS_CFG_DBG_EN	When set to 1, this configuration adds ROM constants to help support kernel-aware debuggers. Specifically, a number of named ROM variables can be queried by a debugger to check compile-time options. For example, a debugger can check the size of an OS_TCB, the kernel's version number, the size of an event flag group, and much more.	**1** (enabled) or 0 (disabled)

Constant	Description	Possible values
OS_CFG_TICK_EN	If your application does not require any form of timeouts or time keeping, either with timeouts on kernel objects or delayed execution times, set this option to 0. Doing so removes all time-keeping facilities from the kernel.	**1** (enabled) or 0 (disabled)
OS_CFG_TS_EN	When set to 1, the timestamp facilities provided by the CPU module are used to measure the time between various events. For example, the kernel will measure the time spent by a task pending on an object, the time the scheduler is locked, etc. This option is used mostly for profiling and performance measurement. To save space and processing time, set this option to 0. **Note**: To use the timestamp facilities, the CPU module Board Support Package or Port should implement the functions described in the CPU Port Board Support Package .	**1** (enabled) or 0 (disabled)
OS_CFG_PRIO_MAX	Specifies the number of task priorities available to the application. OS_CFG_PRIO_MAX should always be set to even multiples of 8 (8, 16, 32, 64, 128, 256, etc.). The higher the number of different priorities, the more RAM the kernel will consume. Task priorities can range from 0 (highest priority) to a maximum of 255 (lowest priority) when the data type OS_PRIO is defined as a CPU_INT08U. However, there is no practical limit to the number of available priorities. If OS_PRIO is defined as CPU_INT16U, there can be up to 65536 priority levels. We recommend leaving OS_PRIO defined as CPU_INT08U since 256 priority levels is sufficient for most applications. An application cannot create tasks with a priority number higher than OS_CFG_PRIO_MAX. To ensure proper operation of kernel and its services, use care when setting the priorities of other system tasks, such as the Statistics Task and the Timer Task in the run-time configuration.	Integer, 64 u
OS_CFG_SCHED_LOCK_TIME_MEAS_EN	When set to 1, this allows the kernel to use the timestamp facilities (provided OS_CFG_TS_EN is also set to 1) to measure the peak amount of time that the scheduler is locked. Use this feature to profile the application during development and testing. A deployed application should have this set to 0.	1 (enabled) or **0** (disabled)
OS_CFG_SCHED_ROUND_ROBIN_EN	When set to 1, use the Round Robin Scheduler. This is only useful when there are multiple tasks sharing the same priority. If this is not the case, set this option to 0.	**1** (enabled) or 0 (disabled)
OS_CFG_STK_SIZE_MIN	Specifies the minimum stack size (in CPU_STK elements) for each task, which the kernel uses to verify that sufficient stack space is provided when the task is created. For example, suppose the full context of a processor consists of 16 registers of 32 bits and that CPU_STK is declared as being of type CPU_INT32U. OS_CFG_STK_SIZE_MIN would then be set to 16. However, it would be wise to also account for storage of local variables, function call returns, and possibly nested ISRs. Refer to the "port" of the processor used to see how to set this minimum.	Integer, 64 u
OS_CFG_TRACE_EN	When set to 1, this adds the Trace Instrumentation to the Kernel. Note that an external tracer, like SystemView , is needed to extract the tracing information generated by the Kernel.	1 (enabled) or **0** (disabled)

Event Flag Configuration

Table - Event Flag Configuration Constants

Constant	Description	Possible values
Constant	Description	Possible values
OS_CFG_FLAG_EN	When set to 1, this enables the event flag services and data structures. If event flags are not needed, set this to 0to reduce the amount of code and data space needed by the kernel. **Note**: When OS_CFG_FLAG_EN is set to 0, it is unnecessary to enable or disable the OS_CFG_FLAG_MODE_CLR_EN option.	**1** (enabled) or 0 (disabled)
OS_CFG_FLAG_MODE_CLR_EN	If you require your application to wait until a flag is cleared, set this to 1. If not, set this to 0. Generally, you would wait for event flags to be set.	1 (enabled) or **0** (disabled)

Mutal Exclusion Semaphore Configuration

Table - Mutual Exclusion Semaphore Configuration Constants

Constant	Description	Possible values
OS_CFG_MUTEX_EN	When set to 1, this enables the mutual exclusion semaphore services and data structures. If mutual exclusion semaphores are not needed, set this to 0, which reduces the amount of code and data space needed by the kernel.	**1** (enabled) or 0 (disabled)

Message Queue Configuration

Table - Message Queue Configuration Constants

Constant	Description	Possible values
OS_CFG_Q_EN	When set to 1, this enables the message queue services and data structures. If message queues are not needed, set this to 0 to reduce the amount of code and data space needed by the kernel.	**1** (enabled) or 0 (disabled)

Semaphore Configuration

Table - Semaphore Configuration Constants

Constant	Description	Possible values
OS_CFG_SEM_EN	When set to 1, this enables the semaphore services and data structures. If semaphores are not needed, set this to 0. It reduces the amount of code and data space needed by the kernel.	**1** (enabled) or 0 (disabled)

Monitor Configuration

Table - Monitor Configuration Constants

Constant	Description	Possible values
OS_CFG_MON_EN	When set to 1, this enables the monitor services and data structures. If monitors are not needed, set this to 0 to reduce the amount of code and data space needed by the kernel.	**1** (enabled) or 0 (disabled)

Task Management Options

Table - Task Management Constants

Constant	Description	Possible values
OS_CFG_STAT_TASK_EN	When set to 1, this enables the kernel's Statistics Task. This task computes the CPU usage of an application, the stack usage of each task, the CPU usage of each task, and more at run-time. When enabled, the Statistics Task executes at a rate equal to the .RateHz member of the Statistics Task configuration structure used in the kernel's initialization function. The Statistics Task calls OSStatTaskHook() every time it executes so that you can add your own statistics calculations as needed. The priority of the Statistics Task is set by the .Prio member of the Statistics Task configuration structure used in the kernel's initialization function. For more information, see the Statistics Task configuration in the Run-Time Application Specifics page. When OS_CFG_STAT_TASK_EN is set to 0, all variables used by the statistic task are not declared. This reduces the amount of RAM and code space needed by the kernel.	**1** (enabled) or 0 (disabled)
OS_CFG_STAT_TASK_STK_CHK_EN	When set to 1, the Statistics Task computes the stack usage of each task created. In this case, the Statistics Task calls OSTaskStkChk() for each task and places the result in the task's TCB. The .StkFree and .StkUsed fields of the task's TCB represent the amount of free space (in CPU_STK elements) and amount of used space (in CPU_STK elements), respectively. **Note**: This option, OS_CFG_STAT_TASK_EN must also be set to 1.	**1** (enabled) or 0 (disabled)
OS_CFG_TASK_PROFILE_EN	To enable the performance profiling tools within the kernel, set this option to 1, which allows variables to be allocated in each task's TCB to hold performance data about each task. Each task uses variables to keep track of the number of times a task is switched in, the task execution time, the CPU usage percentage of the task relative to the other tasks, and more. The information made available with this feature is highly useful when debugging, but requires extra RAM. To save data and code space, set this option to 0 after you are certain that your application is profiled and works correctly.	**1** (enabled) or 0 (disabled)
OS_CFG_TASK_Q_EN	When set to 1, the kernel will offer task messages queues with the OSTaskQ???() functions, which send and receive messages directly to and from tasks and ISRs. Sending messages directly to a task is more efficient than sending messages using a traditional message queue because there is no pend list associated with messages sent to a task. If your application does not require task-level message queues, set this option to 0.	**1** (enabled) or 0 (disabled)

Constant	Description	Possible values
OS_CFG_TASK_REG_TBL_SIZE	This constant allows each task to have task context variables, which allow you to store such elements as “errno”, task identifiers, and other task-specific values. The number of variables that a task contains is set by this option. Each variable is identified by a unique identifier from 0 to OS_CFG_TASK_REG_TBL_SIZE-1. Also, each variable is declared as having an OS_REG data type (see Data Type Configuration). To disable the usage of task context variables, set this option to 0 u.	Integer, 3u
OS_CFG_TASK_STK_REDZONE_EN	While debugging, you should determine if a task has overflowed its stack space by setting this option to 1. Every time a task is switched in, its stack is checked. If the monitored zone located at the end of a task's stack is corrupted, an application-defined hook is called and a software exception is generated. To disable this feature, set this option to 0.	1 (enabled) or 0 (disabled)
OS_CFG_TASK_STK_REDZONE_DEPTH	The default monitored zone is 8 CPU_STK elements long (located at the end of a task's stack). Use this option to change the size of the monitored zone. If OS_CFG_TASK_STK_REDZONE_EN is set to 0, this value is ignored. **Note**: The effectively usable stack space is the task's stack size minus the OS_CFG_TASK_STK_REDZONE_DEPTH value.	Integer, 8u

Task Local Storage Configuration

Table - Task Local Storage Configuration Constant

Constant	Description	Possible values
OS_CFG_TLS_TBL_SIZE	If your application requires Task Local Storage, set this option to a positive integer value. This value determines the size of the Task Local Storage Table (.TLS_Tbl, member of OS_TCB) present in each task. To disable TLS, set this option to 0 u. Task local storage is disabled by default because none of Micrium OS's modules need it. However, if your application requires a standard C library, like newlib for example, you would need enable this feature. Most third-party libraries will indicate if they need TLS or not.	Integer, 0 u

Timer Management Configuration

Table - Timer Management Configuration Constants

Constant	Description	Possible values
OS_CFG_TMR_EN	When set to 1, this enables the timer management services and data structures. If your application does not require programmable timers, set this option to 0 to reduce the kernel's required data and code space.	1 (enabled) or 0 (disabled)

Kernel Run-Time Application Specific Configurations

- Kernel Initialization
- Optional Configurations
  - Memory Segment
  - Message Pool Size
  - Task Stack Limit
  - ISR Stack

- Statistics Task
- Timer Task

This section describes the application-specific configurations of the kernel, that are set at run-time.

Kernel Initialization

For information on the methods that can be used to initialize any Micrium product, see Stacks Initialization Methods . To initialize the kernel, OSInit() must be called.

Optional Configurations

This section describes the configurations that are optional. If you do not set them in your application, the default configurations will apply.

The default values can be retrieved via the structure OS_InitCfgDflt. For examples of how to change the run-time configuration, see the Kernel Initialization page of the Kernel Programming Guide.

Note: These configurations must be set *before* you call the function OSInit().

Memory Segment

The kernel can allocate its internal task's stack and objects on a specific memory segment.

Type	Function to call	Default	Field from default configuration structure
MEM_SEG *	OS_ConfigureMemSeg()	The General-purpose heap .	.MemSeg

Message Pool Size

The kernel can manage a variable number of messages used by both the message queues and the per-task message queues.

Type	Function to call	Default	Field from default configuration structure
OS_MSG_SIZE	OS_ConfigureMsgPoolSize()	The kernel will manage 100 messages.	.MsgPoolSize

Task Stack Limit

Some processors offer the ability to monitor the task's stack usage. This option sets the percentage to Full, at which point the processor must alert the application.

Type	Function to call	Default	Field from default configuration structure
CPU_STK_SIZE	OS_ConfigureStkLimit()	10% from Full is the limit.	.TaskStkLimit

ISR Stack

If the processor allows it, a separate stack may be used for the ISRs. This option specifies the stack used by the ISRs.

Type	Function to call	Default	Field from default configuration structure
OS_STACK_CFG	OS_ConfigureISRStk()	The ISRs will use a 256 element stack allocated in the kernel memory segment.	.ISR

Statistics Task

This option changes the Statistics Task's stack, priority, and rate of execution.

Type	Function to call	Default	Field from default configuration structure
OS_TASK_CFG	OS_ConfigureStatTask()	The Statistics Task will use a 256 element stack allocated in the kernel memory segment. It will run at rate of 10Hz with a priority of 3.	.StatTaskCfg

Timer Task

This option alters the Timer Management Task's stack, priority, and rate of execution.

Type	Function to call	Default	Field from default configuration structure
OS_TASK_CFG	OS_ConfigureTmrTask()	The Timer Management Task will use a 256 element stack allocated in the kernel memory segment. It will run at rate of 10Hz with a priority of 2 .	.TmrTaskCfg

Kernel Task Configuration Structure

- Task Configuration for Statistic task
- Task Configuration for Timer

The kernel defines structures to configure its internal tasks. The usage depends on the task being configured, which is described in detail below.

Task Configuration for Statistic task

Table - OS_TASK_CFG for the Statistics Task

Field	Description	Possible Values
.StkBasePtr	Pointer to the base of the buffer used as a stack.	Any valid pointer, or DEF_NULL to allocate the stack in the kernel's memory segment.
.StkSize	Size of the buffer, in CPU_STK elements.	Any positive integer.
.Prio	The execution priority of the Task. When choosing its priority, keep the rate of execution of this task in mind to assure proper operation of the kernel.	1 to OS_CFG_PRIO_MAX-2.
.RateHz	The rate of execution of the Task (measured in Hertz).	1 to 10.

Task Configuration for Timer

Table - OS_TASK_CFG for the Timer Management Task

Field	Description	Possible Values
.StkBasePtr	Pointer to the base of the buffer used as a stack.	Any valid pointer, or DEF_NULL to allocate the stack in the kernel's memory segment.
.StkSize	Size of the buffer, in CPU_STK elements.	Any positive integer.

Field	Description	Possible Values
.Prio	The execution priority of the Timer Management Task. When choosing its priority, keep the rate of execution of this task in mind to assure proper operation of the kernel.	1 to OS_CFG_PRIO_MAX-2.
.RateHz	The rate of execution of the Timer Management Task (in hertz). The software timers will be updated at RateHz times per second. This also specifies the minimum delay a timer can use.	10 to 1000.

Kernel Task Stack Configuration Structure

The kernel has an internal stack used for ISR handlers. To configure the stack, the following structure is used:

Table - OS_STACK_CFG for ISR handlers

Field	Description	Possible Values
.StkBasePtr	Pointer to the base of the buffer used as a stack for ISR.	Any valid pointer, or DEF_NULL to allocate the stack in the kernel's memory segment.
.StkSize	Size of the buffer, in CPU_STK elements.	Any positive integer.

Kernel Programming Guide

Kernel Programming Guide

The following sections provide an overview of the usage of the kernel's features.

- Initialization of the Kernel
- Kernel Basic Services
- Task Management Kernel Services
- Kernel Interrupt Services
- Resource Management Using the Kernel
- Synchronization with Task Semaphores
- Using Event Flags
- Using Message Queues
- Using Monitors
- Kernel Time Management
- Using the Kernel Timers
- Extending the Kernel with Application Hooks

Initialization of the Kernel

The kernel uses the initialization methods described in Stacks Initialization Methods , and it is initialized by calling OSInit(). The Kernel Run-Time Application Specific Configurations page describes the default values used by the kernel.

If the standard initialization method is used, several configuration functions are available to the application for fine-tuning certain aspects of the kernel. The Table - Kernel Configuration Functions in the *Initialization of the Kernel* page provides an overview of those functions and their purpose.

Table - Kernel Configuration Functions

Function	Description
OS_ConfigureMemSeg()	Specify the memory segment used by the kernel to allocate objects and stacks. (Optional. If not called, default values will be used.)
OS_ConfigureISRStk()	Specify the buffer used for the ISR stack.
OS_ConfigureMsgPoolSize()	Specify the number of messages managed by the kernel. (Optional. If not called, default values will be used.)
OS_ConfigureStkLimit()	Specify the stack limit, in percentage to full, before issuing an exception.
OS_ConfigureStatTask()	Specify the Statistics Task's stack, priority, and execution rate. (Optional. If not called, default values will be used.)
OS_ConfigureTmrTask()	Specify the Timer Management Task's stack, priority, and execution rate. (Optional. If not called, default values will be used.)

If the advanced initialization method is used, the kernel assumes that the global OS_InitCfg variable is available and configured to the liking of the application. When using the advanced configuration, the OS_Configure...() functions are not available.

Example

Below is an example that shows how to change the number of messages, and the configuration of the Timer Task.

Listing - Example of call to OSInit()

```
:
          :
                                          /* Application defines.                      */
#define  APP_KERNEL_MAX_MESSAGES          200u
#define  APP_KERNEL_TIMER_TASK_STK_SIZE     256u
#define  APP_KERNEL_TIMER_TASK_PRIO         10u
          :
          :
CPU_STK  App_TimerTaskStk[APP_KERNEL_TIMER_TASK_STK_SIZE];
          :
          :
int  main (void)
{
   RTOS_ERR     err;
   OS_TASK_CFG  tmr_task_cfg;
          :
          :
                                          /* Configure the number of messages             */
   OS_ConfigureMsgPoolSize(APP_KERNEL_MAX_MESSAGES);     /* managed by the kernel.                    */

                                          /* Configure the Tmr Task:                      */
   tmr_task_cfg            = OS_InitCfgDflt.TmrTaskCfg;  /*  Get default values for Timer Task.          */
   tmr_task_cfg.Prio       = APP_KERNEL_TIMER_TASK_PRIO; /*  Set the priority of the Timer Task.         */
   tmr_task_cfg.StkBasePtr = App_TimerTaskStk;           /*  Set buffer to use for Timer Task stack.     */
   tmr_task_cfg.StkSize    = APP_KERNEL_TIMER_TASK_STK_SIZE; /* Set Timer Task stack size.               */
   OS_ConfigureTmrTask(&tmr_task_cfg);
                                          /* Initialize the Kernel.                       */
   OSInit(&err);
   if (err.Code != RTOS_ERR_NONE) {
      /* Handle error on kernel init. */
   }
          :
          :
}
```

Kernel Basic Services

You can start the kernel once it has been initialized, and all other setup operations have been made. Starting the kernel also starts the preemptive scheduler and executes the highest-priority task. An example call to OSStart() is shown below.

Listing - Example of call to OSStart()

```
RTOS_ERR  err;

OSStart(&err);                 /* Start the preemptive scheduler.          */
if (err.Code != RTOS_ERR_NONE) {
   /* Should not happen, unless critical error. */
}
```

To force the kernel to recalculate the highest-priority task and switch to it pre-preemptively, call OSSched():

Listing - Example of call to OSSched()

```
/* Schedule highest-priority task for       */
                        /*  execution and switch to it.             */
OSSched();
```

Note: This function is called automatically by the kernel when the user calls a blocking function from a task (OS???Pend() functions, time delay functions, etc.).

The user can obtain the version of the currently executing kernel by calling RTOS_Version():

Listing - Example of call to OSVersion()

```
CPU_INT32U  version;

version = RTOS_Version();      /* Get Micrium OS version.              */
```

Task Management Kernel Services

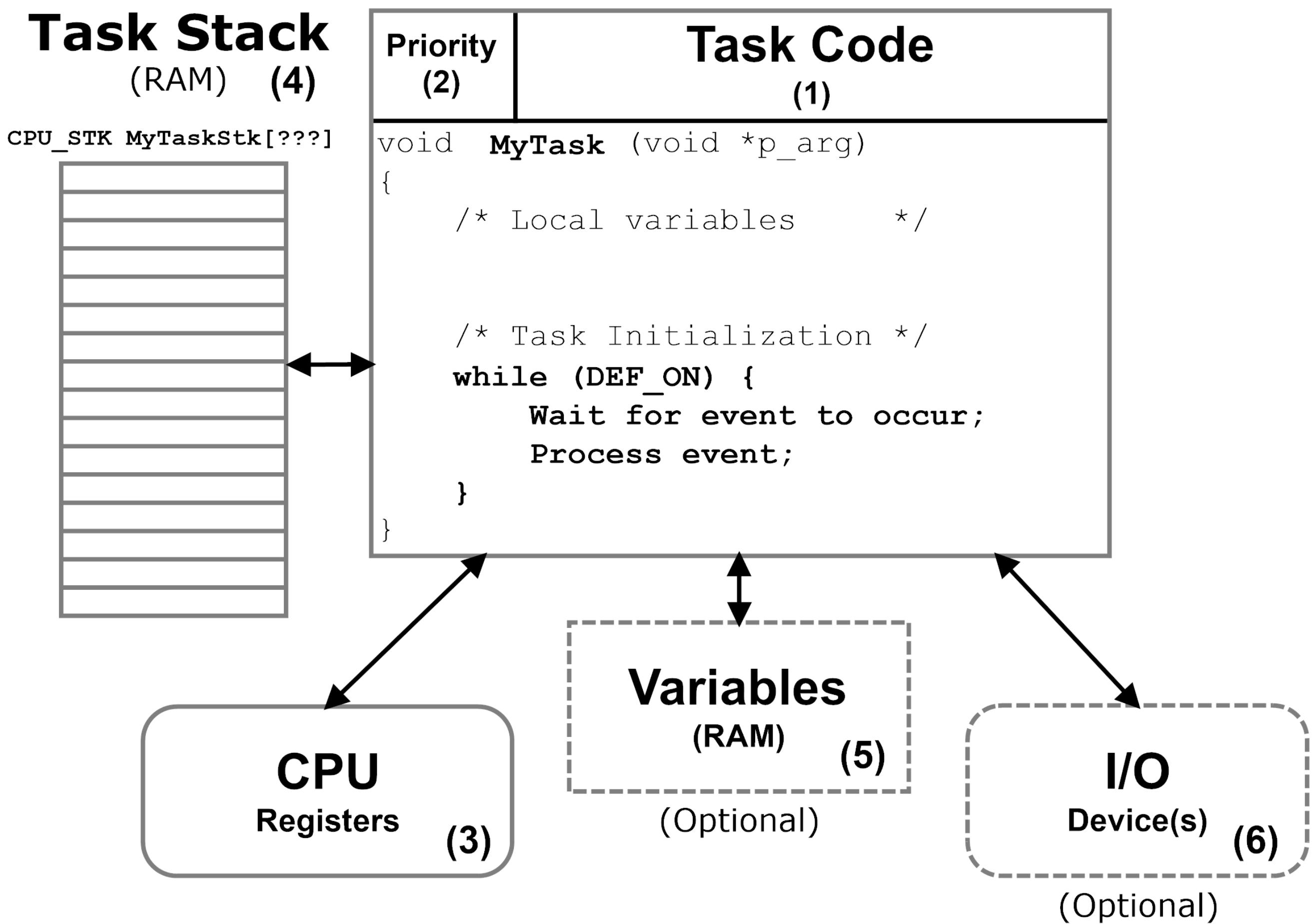
- Task Components
- Creating and Deleting Tasks
- Round-Robin Scheduler
  - Configuration
  - Usage
  - Example
- Task Registers
- Execution Control
- Stack Checking

This section showcases the kernel's task management features.

Task Components

A task usually has an implementation function, a priority and a stack. Tasks also access the CPU's registers and sometimes its peripherals, or I/O devices. The figure below showcases the interactions between a task and its components.

Figure - Interactions between a task and its components



(1) The task's implementation looks like any other C function, except that it is typically implemented as an infinite loop, and is not allowed to return.

(2) Each task is assigned a priority based on its importance within the application. For more information on how to assign a priority to a task, see Task Priority .

(3) A task has its own set of CPU registers. As far as a task is concerned, the task thinks it has the CPU to itself.

(4) Because the Kernel is preemptive, each task must have its own stack area. This stack area can be allocated at compile-time using a static buffer or at run-time using a memory manager. Care must be taken while using a dynamic memory manager in order to avoid memory segmentation and to prevent the de-allocation of a currently in-use buffer. The stack always resides in RAM and is used to keep track of local variables, function calls, and possibly ISR nesting as explained in Task Stack.

(5) A task can have access to global variables. However, code that accesses such global variables should be protected since a global variable can be shared among many tasks.

(6) A task may also have access to one or more Input/Output (I/O) devices (also known as *peripherals*). In fact, it is common practice to assign tasks to manage I/O devices.

Creating and Deleting Tasks

To create a task, you must first allocate the task's TCB and stack, then use a single function to represent a task's entry point. The snippet below shows how to create a task using OSTaskCreate().

Listing - Example of call to OSTaskCreate()

```
/* Example Task Defines:           */
#define  APP_EXAMPLE_TASK_PRIO          21u  /*   Task Priority.                */
#define  APP_EXAMPLE_TASK_STK_SIZE      256u  /*   Stack size in CPU_STK.        */

                                    /* Example Task Data:      */
OS_TCB   App_ExampleTaskTCB;                    /*   Task Control Block.   */
CPU_STK  App_ExampleTaskStk[APP_EXAMPLE_TASK_STK_SIZE]; /*   Stack.               */

                                    /* Example Task Code:      */
void  App_ExampleTask (void  *p_arg);                /*   Function.            */

void  App_TaskCreate (void)
{
   RTOS_ERR     err;

   OSTaskCreate(&App_ExampleTaskTCB,              /* Pointer to the task's TCB.  */
             "Example Task.",                  /* Name to help debugging.     */
             &App_ExampleTask,                  /* Pointer to the task's code. */
              DEF_NULL,                         /* Pointer to task's argument. */
              APP_EXAMPLE_TASK_PRIO,            /* Task's priority.            */
             &App_ExampleTaskStk[0],            /* Pointer to base of stack.   */
             (APP_EXAMPLE_TASK_STK_SIZE / 10u),  /* Stack limit, from base.     */
              APP_EXAMPLE_TASK_STK_SIZE,         /* Stack size, in CPU_STK.     */
              10u,                              /* Messages in task queue.     */
              0u,                               /* Round-Robin time quanta.    */
              DEF_NULL,                         /* External TCB data.          */
              OS_OPT_TASK_STK_CHK,              /* Task options.               */
             &err);
   if (err.Code != RTOS_ERR_NONE) {
      /* Handle error on task creation. */
   }
}

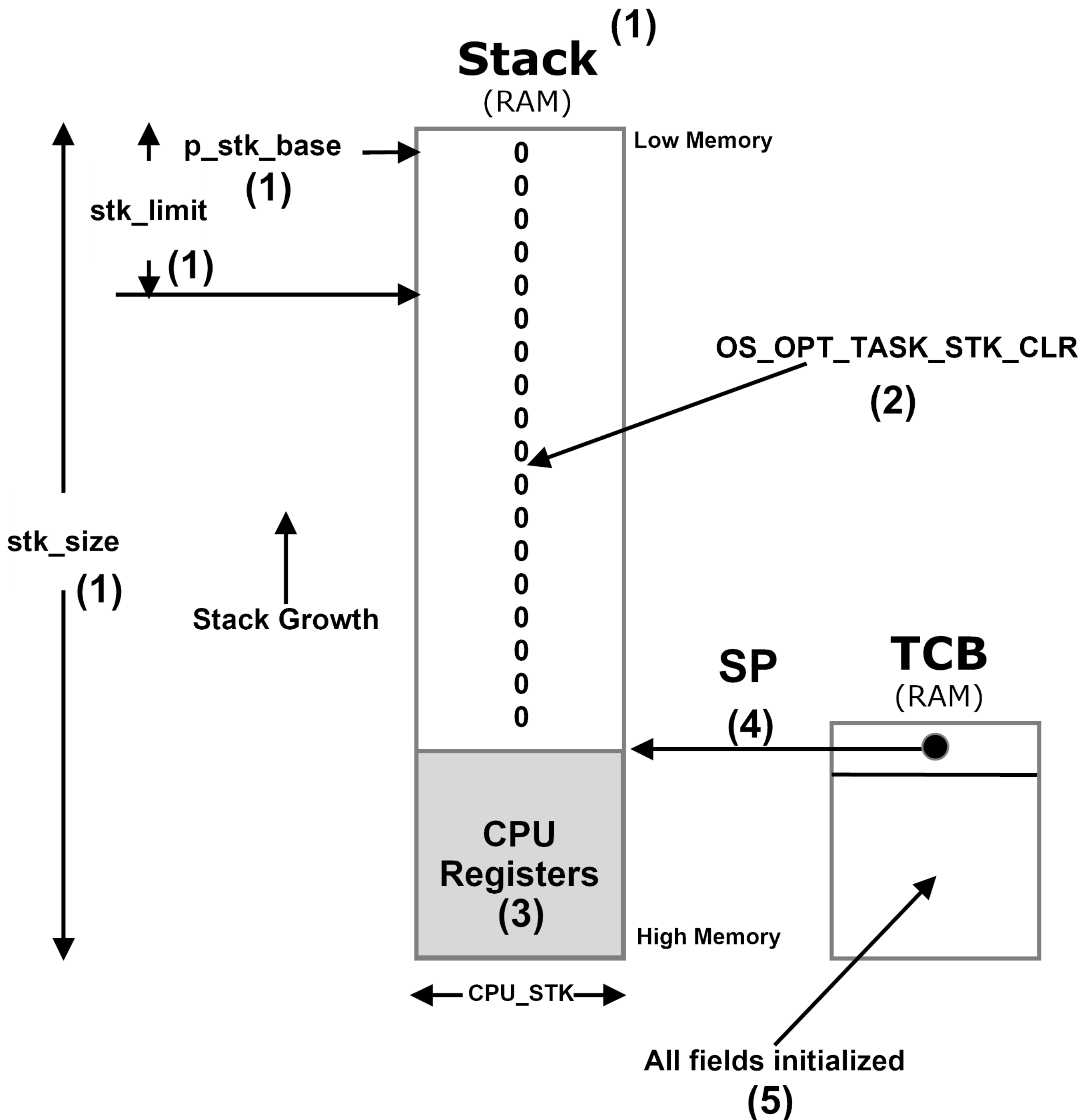
void  App_ExampleTask (void  *p_arg)
{
   /* Use argument. */
  (void)&p_arg;

   while (DEF_TRUE) {
      /* All tasks should be written as infinite loops. */
   }
}
```

The App_TaskCreate() function above creates a single task (named Example Task) with a priority of 21 and a stack of 256 CPU_STK elements.

The illustration below explains the steps taken by OSTaskCreate() to initialize both the TCB and the task's stack.

Figure - OSTaskCreate() initialization of the TCB and task's stack



(1) When calling OSTaskCreate(), you pass the base address of the stack (p_stk_base) that will be used by the task, the watermark limit for stack growth (stk_limit) which is expressed in number of CPU_STK entries before the stack is empty, and the size of stack (stk_size), also in terms of CPU_STK elements.

(2) When specifying OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR in the opt argument of OSTaskCreate(), the task's stack will be zeroed.

(3) The top of the task's stack is initialized with a copy of the CPU registers in the same stacking order as if they were saved in a context switch-out operation.

(4) The new value of the stack pointer (SP) is saved in the TCB.

(5) The remaining fields of the TCB are initialized: task priority, task name, task state, internal message queue, internal semaphore, and many others.

Assuming the previously created Example Task is used, calling OSTaskDel() with the Example Task's TCB would delete the task, after which the task's stack and TCB would be available to be re-used.

Listing - Example of call to OSTaskDel()

```
/* Example Task Data:      */
OS_TCB   App_ExampleTaskTCB;                    /*   Task Control Block.   */
                                  :
                                  :
RTOS_ERR  err;
                                  :
                                  :
OSTaskDel(&App_ExampleTaskTCB,  /* Pointer to the task's TCB.                    */
        &err);
if (err.Code != RTOS_ERR_NONE) {
    /* Handle error on task deletion. */
}
```

Round-Robin Scheduler

The kernel offers a simple Round-Robin Scheduler. The functions related to the Round-Robin Scheduler can be separated in two categories: **Configuration** and **Usage**.

Configuration

A single function, OSSchedRoundRobinCfg() configures the Round-Robin Scheduler with parameters as described below:

Listing - Example of call to OSSchedRoundRobinCfg()

```
RTOS_ERR  err;

OSSchedRoundRobinCfg( DEF_TRUE,  /* DEF_TRUE to enable, DEF_FALSE to disable        */
                          /* Round-Robin scheduling.                        */
                  100u,      /* Default time amount per task, in OS Ticks.      */
                  &err);
if (err.Code != RTOS_ERR_NONE) {
    /* Handle error on Round-Robin Scheduler configuration. */
}
```

Usage

Once the Round-Robin Scheduler is properly configured, tasks can change the amount of time they need, or change the amount of time for another task with the OSTaskTimeQuantaSet() function:

Listing - Example of call to OSTaskTimeQuantaSet()

```
RTOS_ERR  err;

OSTaskTimeQuantaSet( DEF_NULL,  /* DEF_NULL will change the time for the current    */
                         /*  task. Otherwise, specify the task's TCB.       */
                   25,      /* New time amount for the task, in OS Ticks.       */
                  &err);
if (err.Code != RTOS_ERR_NONE) {
    /* Handle error on time change. */
}
```

Tasks can also yield to the CPU with OSSchedRoundRobinYield() if they do not need their allocated time, as illustrated below:

Listing - Example of call to OSSchedRoundRobinYield()

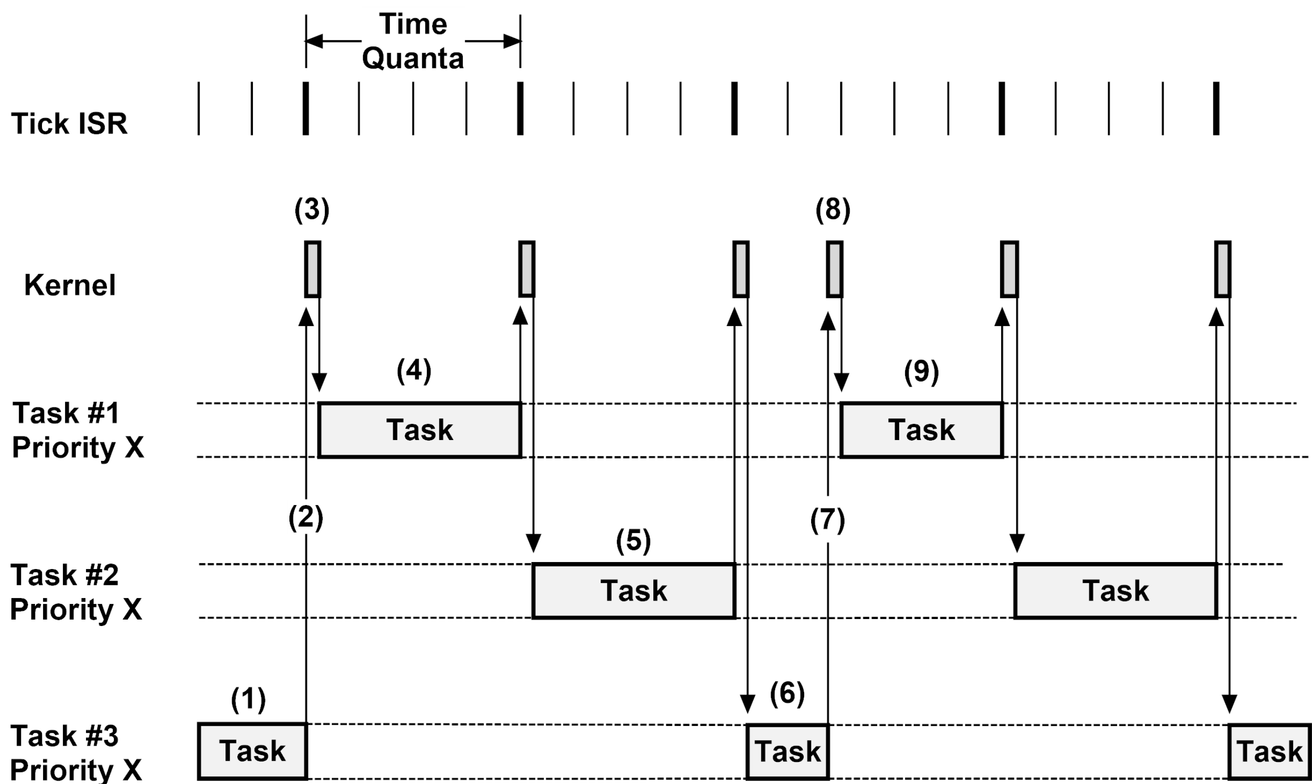
```
RTOS_ERR  err;

                    /* In any Task:                         */
OSSchedRoundRobinYield(&err);
if (err.Code != RTOS_ERR_NONE) {
    /* Handle error on yield. */
}
```

Example

The figure below shows a timing diagram with tasks running at the same priority. There are three tasks that are ready-to-run at priority X. For sake of illustration, the time quanta is 4 Kernel ticks. This is shown as a darker tick mark.

Figure - Round-Robin Scheduling



(1) Task #3 is executing. During that time, a tick interrupt occurs but the time quanta has not expired yet.

(2) On the 4th tick interrupt, the time quanta for Task #3 expires.

(3) The Kernel resumes Task #1 since it was the next task in the list of tasks at priority X that was ready-to-run.

(4) Task #1 executes until its time quanta expires, which is four ticks.

(5) After Task #1 uses its time quanta, Task #2 executes.

(6) After Task #2 uses its time quanta, Task #3 executes.

(7) Task #3 decides to give up its time quanta by calling OSSchedRoundRobinYield().

(8) This causes the Kernel to resume Task #1.

(9) Task #1 executes for its full time quanta.

Task Registers

Applications can have task-specific variables (such as errno). The kernel provides OS_CFG_TASK_REG_TBL_SIZE task registers (see Kernel Compile-Time and Data Type Configurations). To use task-specific registers, you must first request an ID using OSTaskRegGetID(). This ID is shared among all tasks so that it refers to the same variable across all created tasks. With the ID, the user can get the value of a task-specific register with OSTaskRegGet() and set the value of the task-specific register with OSTaskRegSet() as shown in the example below.

Listing - Example usage of OSTaskRegGetID(), OSTaskRegSet() and OSTaskRegGet()

```
OS_REG     value;
OS_REG_ID  id;
RTOS_ERR   err;

                          /* Request a Task Register ID number.           */
id = OSTaskRegGetID(&err);
if (err.Code != RTOS_ERR_NONE) {
   /* Handle error on task register allocate ID. */
}

value = 0xC0FFEE;
                          /* Set Task Register:                           */
OSTaskRegSet( DEF_NULL,        /*   DEF_NULL refers to the current task,         */
                          /*   otherwise specify the task's TCB.           */
           id,            /*   Task Register ID.                           */
           value,         /*   Task Register value.                        */
          &err);
if (err.Code != RTOS_ERR_NONE) {
   /* Handle error on task register set. */
}

                          /* Get Task Register:                           */
value = OSTaskRegGet( DEF_NULL, /*   DEF_NULL refers to the current task,         */
                          /*   otherwise specify the task's TCB.           */
                 id,      /*   Task Register ID.                           */
                &err);
if (err.Code != RTOS_ERR_NONE) {
   /* Handle error on task register get. */
}
```

Execution Control

The kernel offers the ability to suspend a task (which can also be nested) and resume its execution. A suspended task is effectively removed from the list of tasks that can be scheduled. Use the OSTaskSuspend() function to suspend a task and the OSTaskResume() function to resume a task. The example below shows that a task can suspend itself.

Listing - Example usage of OSTaskSuspend() and OSTaskResume()

```
/* In Example Task:                                        */
RTOS_ERR  err;

OSTaskSuspend( DEF_NULL,         /* DEF_NULL refers to current task.            */
                          /*  otherwise specify the task's TCB.          */
          &err);
                          /* This check will only be executed once Example */
                          /*  Task is resumed.                        */
if (err.Code != RTOS_ERR_NONE) {
   /* Handle error on task suspend. */
}

/* In any other Task:                                      */
RTOS_ERR  err;

OSTaskResume(&App_ExampleTaskTCB,  /* Resume Example Task using its TCB.          */
         &err);
if (err.Code != RTOS_ERR_NONE) {
   /* Handle error on task resume. */
}
```

Stack Checking

The OSTaskStkChk() function checks the current size of the free and used space of a task's stack.

Note: The task must be created with the OS_OPT_TASK_STK_CHK option in order to check its stack usage.

Listing - Example of call to OSTaskStkChk()

```
CPU_STK_SIZE  free_space;
CPU_STK_SIZE  used_space;
RTOS_ERR      err;

OSTaskStkChk( DEF_NULL,     /* DEF_NULL refers to the current task, otherwise       */
                    /*  specify the task's TCB.                         */
         &free_space,  /* Will contain the amount of free CPU_STK elements.    */
         &used_space,  /* Will contain the amount of used CPU_STK elements.     */
         &err);
if (err.Code != RTOS_ERR_NONE) {
   /* Handle error on stack checking. */
}
```

Kernel Interrupt Services

When an Interrupt Service Routine (ISR) needs to wake up a task (using the allowed OS???Post() calls), the ISR must inform the kernel that it is going to service an interrupt. The following example shows how to use the OSIntEnter() and OSIntExit() calls.

Listing - Example usage of OSIntEnter() and OSIntExit()

```
void  App_MyISR (void)
{
   OSIntEnter();

   /* Process interrupt. */

   OSIntExit();
}
```

Resource Management Using the Kernel

- Simple

  - Locking the Scheduler
- Semaphores
  - Use cases
- Mutual Exclusion Semaphore (Mutexes)

The kernel offers the following services that control the access to a shared resource among many tasks.

Simple

Critical Section (Deprecated)

Critical Section handling is no longer provided by the kernel. See the EMLIB / CORE section of your MCU documentation at https://docs.silabs.com to learn more about the types of critical sections offered.

Locking the Scheduler

A safer approach to resource arbitration is locking the scheduler. During that time, the kernel cannot schedule another task.

Be sure that the shared resource is not used in an ISR. Interrupts are not disabled while the scheduler is locked, so an ISR and a task could access the resource at the same time. The following is an example of a shared resource being accessed while the scheduler is locked with OSSchedLock() and unlocked with OSSchedUnlock().

Listing - Example of call to OSSchedLock() and OSSchedUnlock()

```
CPU_INT32U  App_GlobalCounter;
          :
          :
void  App_SomeTask (void  *p_arg)
{
   RTOS_ERR  err;
          :
          :
   OSSchedLock(&err);        /* Lock the scheduler.                        */
   if (err.Code == RTOS_ERR_NONE) {
      App_GlobalCounter++;
      OSSchedUnlock(&err);  /* Unlock the scheduler.                      */
   }
          :
          :
}
```

Semaphores

A counting semaphore can be used to protect a resource that is shared by multiple tasks. Before you can use the semaphore, you must create it with OSSemCreate(). A semaphore can be deleted with OSSemDel(), as shown below.

Listing - Example of call to OSSemCreate() and OSSemDel()

```
OS_SEM  App_Semaphore;
          :
          :
void  App_SomeTask (void  *p_arg)
{
   RTOS_ERR    err;
   OS_OBJ_QTY  qty;
          :
          :
                         /* Create the semaphore.                          */
   OSSemCreate(&App_Semaphore,    /*   Pointer to user-allocated semaphore.          */
            "App Semaphore",   /*   Name used for debugging.                     */
             1,                /*   Initial count: available in this case.        */
            &err);
   if (err.Code != RTOS_ERR_NONE) {
      /* Handle error on semaphore create. */
   }
          :
          :
          :
                         /* Delete the semaphore.                          */
   qty = OSSemDel(&App_Semaphore,       /*   Pointer to user-allocated semaphore.    */
               OS_OPT_DEL_NO_PEND,  /*   Option: delete if 0 tasks are pending.  */
              &err);
   if (err.Code != RTOS_ERR_NONE) {
      /* Handle error on semaphore delete. */
   }
          :
          :
}
```

A task can acquire the right to use the protected resource by calling OSSemPend(). Once the task is done, it can release the resource with OSSemPost().

Listing - Example of call to OSSemPend() and OSSemPost()

```
OS_SEM  App_Semaphore;
            :
            :
void  App_SomeTask (void  *p_arg)
{
    RTOS_ERR    err;
    OS_SEM_CTR  ctr;
            :
            :
                                    /* Acquire resource protected by semaphore.        */
    ctr = OSSemPend(&App_Semaphore,         /* Pointer to user-allocated semaphore.    */
                    1000,                   /* Wait for a maximum of 1000 OS Ticks.    */
                    OS_OPT_PEND_BLOCKING,   /* Task will block.                        */
                    DEF_NULL,               /* Timestamp is not used.                  */
                   &err);
    if (err.Code == RTOS_ERR_NONE) {
        /* Resource acquired. 'ctr' contains number of available resources. */
            :
            :
                                    /* Release resource protected by semaphore.        */
        ctr = OSSemPost(&App_Semaphore,     /* Pointer to user-allocated semaphore.    */
                        OS_OPT_POST_1,      /* Only wake up highest-priority task.     */
                       &err);
        if (err.Code != RTOS_ERR_NONE) {
            /* Handle error on semaphore post. */
        }
    } else {
        /* Handle error on semaphore pend. */
    }
            :
            :
}
```

If the kernel configuration permits it, a task can force the pends on a semaphore to abort with OSSemPendAbort().

Listing - Example of call to OSSemPendAbort()

```
OS_SEM  App_Semaphore;
            :
            :
void  App_SomeTask (void  *p_arg)
{
    RTOS_ERR    err;
    OS_OBJ_QTY  qty;
            :
            :
                                    /* Abort the highest-priority task's pend.     */
    qty = OSSemPendAbort(&App_Semaphore,        /* Pointer to user-allocated semaphore. */
                         OS_OPT_PEND_ABORT_1,/* Only abort the HP task's pend.        */
                        &err);
    if (err.Code != RTOS_ERR_NONE) {
        /* Handle error on semaphore pend abort. */
    }
            :
            :
}
```

Finally, a task may explicitly set the number of available resources by calling OSSemSet().

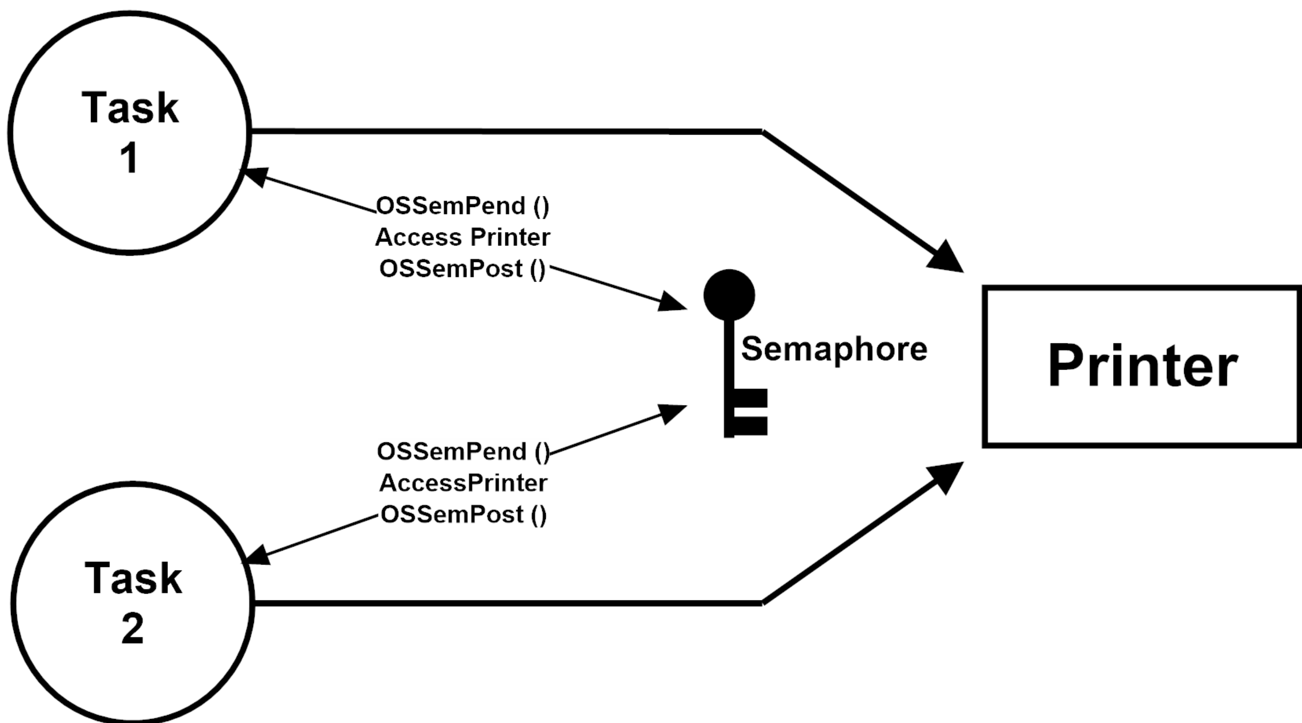
Listing - Example of call to OSSemSet()

```
OS_SEM  App_Semaphore;
            :
            :
void  App_SomeTask (void  *p_arg)
{
    RTOS_ERR  err;
            :
            :
                          /* Set the number of available resources to 10.        */
    OSSemSet(&App_Semaphore,  /*   Pointer to user-allocated semaphore.              */
             10,            /*   10 resources.                                */
             &err);
    if (err.Code != RTOS_ERR_NONE) {
        /* Handle error on semaphore set count. */
    }
            :
            :
}
```

Use cases

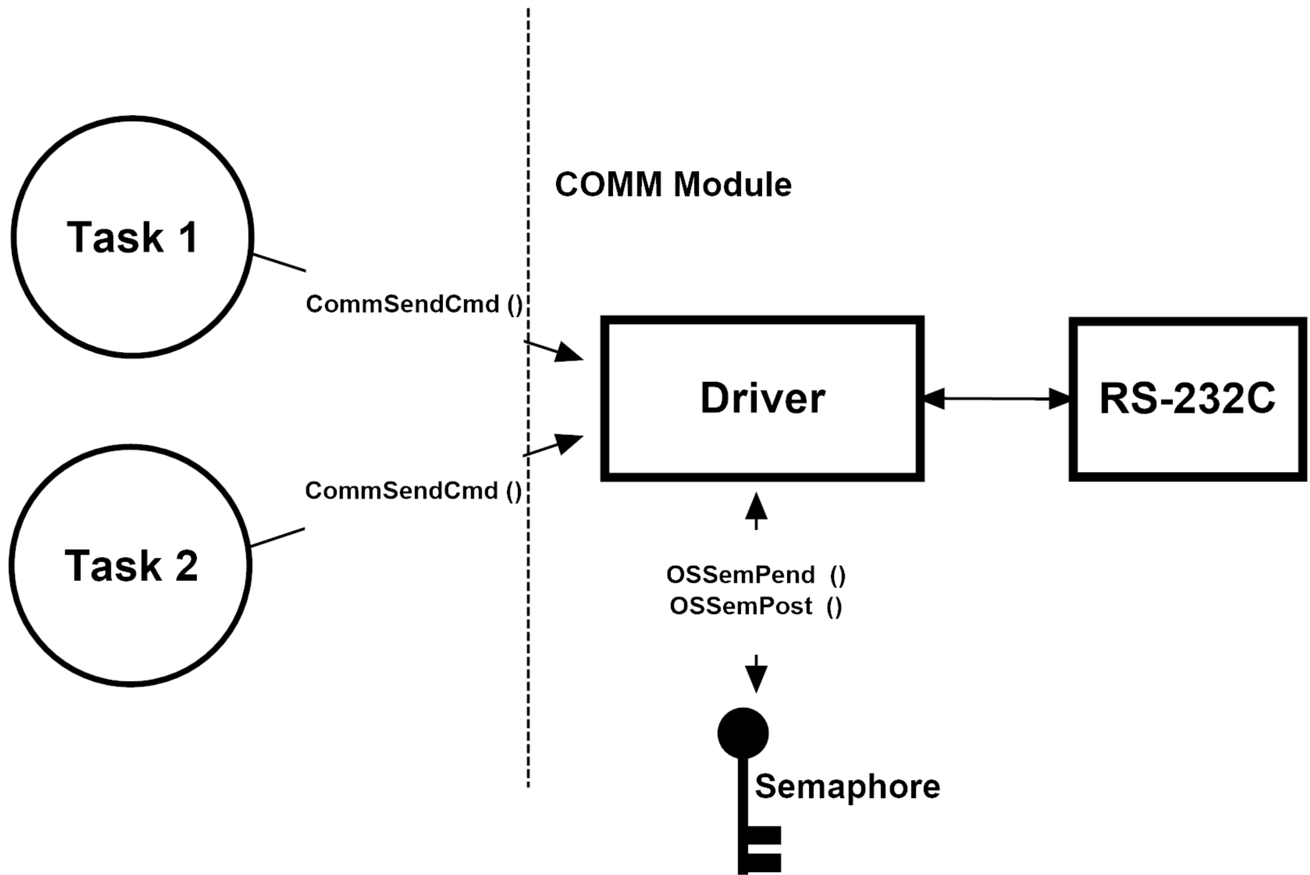
In the example below, the application uses a global semaphore object to properly share a resource, a Printer, among two tasks, Task 1 and Task 2.

Figure - Protecting a Shared Hardware Resource



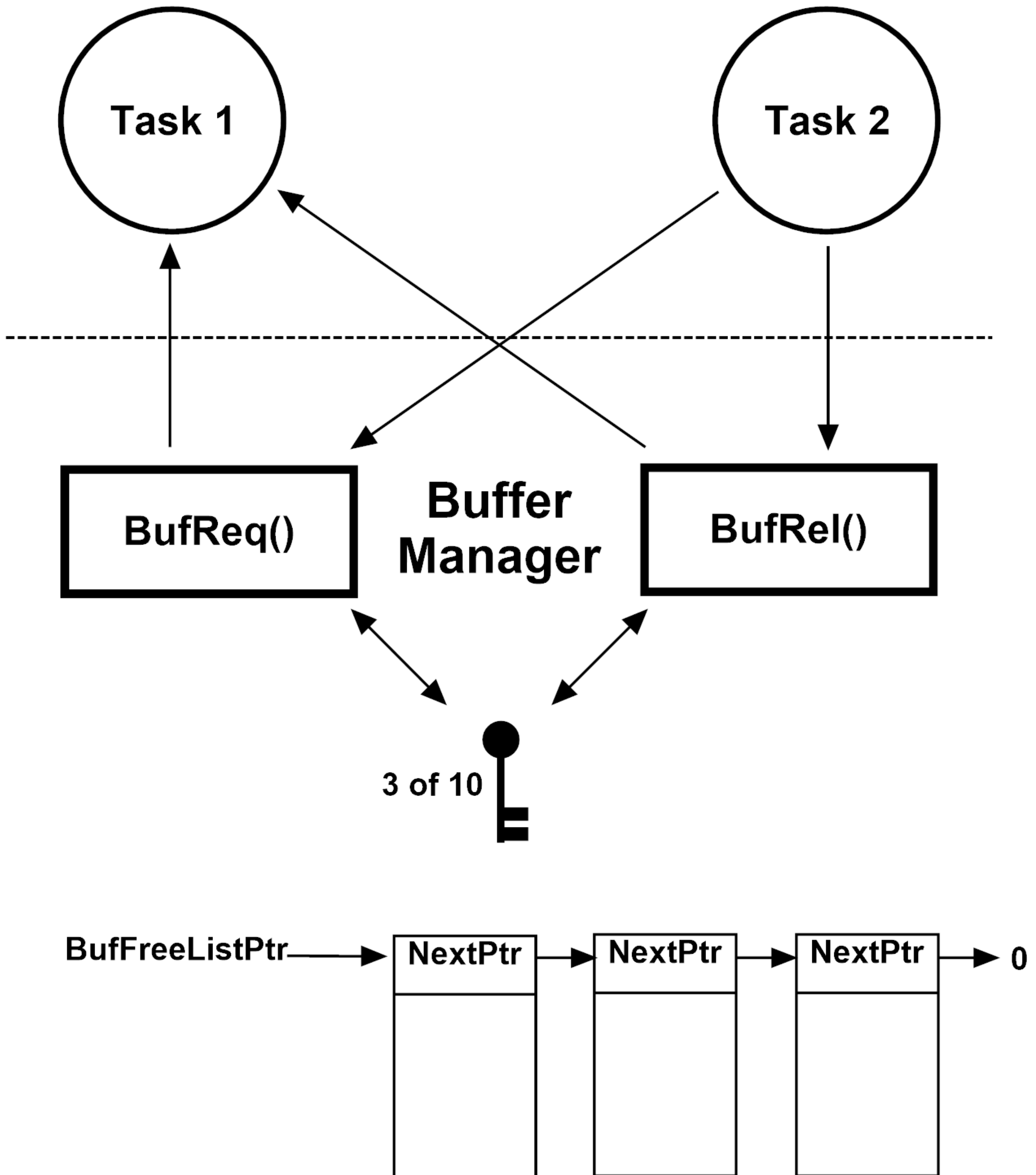
A disadvantage arises in that particular use-case. Indeed, the semaphore object itself needs to be shared and known by all tasks that need it. Furthermore, it needs to be created before any of the tasks tries to use it. It would be beneficial in terms of maintainability, and usability, if the semaphore object was encapsulated within a module that would export the services needed by the tasks. The illustration below shows a serial communications module, called COMM Module, that encapsulates the required locking. The tasks simply use the provided services to send commands and messages.

Figure - Encapsulating the Protection of a Shared Resource



A semaphore can also be used to protect a shared resource that is composed of more than one element. The semaphore is first initialized with the total number of available elements. When a task needs an element, it uses the semaphore to take an element. When it no longer needs the element, it gives it back through the semaphore. If a task tries to acquire an element and there is none left, the task will block until an element becomes available. An example of a counting semaphore can be found below.

Figure - Counting Semaphore Usage



In this example, the application defines a module called Buffer Manager that manages 10 buffers. When a task needs a buffer, it calls BufReq(). When the task no longer needs the buffer, it releases the buffer with BufRel(). Internally, the BufReq() and BufRel() methods use the Kernel's OSSemPend() service to allocate a buffer to the calling task, and the OSSemPost() service to put the buffer back in the pool.

Mutual Exclusion Semaphore (Mutexes)

If a shared resource can be used by only one task at a time, use a mutex. Because the semaphores are vulnerable to the priority-inversion problem, the mutex implementation in the kernel prevents priority inversions by using priority inheritance. You can use OSMutexCreate() to create a mutex and delete it with OSMutexDel(), as shown below.

Listing - Example of call to OSMutexCreate() and OSMutexDel()

```
OS_MUTEX  App_Mutex;
           :
           :
void  App_SomeTask (void  *p_arg)
{
   RTOS_ERR    err;
   OS_OBJ_QTY  qty;
           :
           :
                        /* Create the mutex.                          */
   OSMutexCreate(&App_Mutex,   /*   Pointer to user-allocated mutex.               */
              "App Mutex",  /*   Name used for debugging.                     */
              &err);
   if (err.Code != RTOS_ERR_NONE) {
      /* Handle error on mutex create. */
   }
           :
           :
           :
                        /* Delete the mutex.                          */
   qty = OSMutexDel(&App_Mutex,          /*   Pointer to user-allocated mutex.      */
                 OS_OPT_DEL_NO_PEND,  /*   Option: delete if 0 tasks are pending.*/
                 &err);
   if (err.Code != RTOS_ERR_NONE) {
      /* Handle error on mutex delete. */
   }
           :
           :
}
```

A task can use the protected resource by calling OSMutexPend(). Once the task is done, it can release the resource with OSMutexPost().

A mutex can be nested, which means that you can perform severals consecutive calls of OSMutexPend() on a given mutex. This will increment an internal nesting counter and set the error to RTOS_ERR_IS_OWNER. This error could actualy be used as a nesting indicator. The resource will be released after the same number of OSMutexPost() calls.

Listing - Example of call to OSMutexPend() and OSMutexPost()

```
OS_MUTEX  App_Mutex;
          :
          :
void  App_SomeTask (void  *p_arg)
{
   RTOS_ERR  err;
          :
          :
                              /* Acquire resource protected by mutex.       */
   OSMutexPend(&App_Mutex,             /*   Pointer to user-allocated mutex.         */
            1000,                /*   Wait for a maximum of 1000 OS Ticks.     */
            OS_OPT_PEND_BLOCKING,  /*   Task will block.                         */
            DEF_NULL,            /*   Timestamp is not used.                   */
           &err);
   if (err.Code == RTOS_ERR_NONE) {
      /* Resource acquired. */
          :
          :
                              /* Release resource protected by mutex.       */
      OSMutexPost(&App_Mutex,         /*   Pointer to user-allocated mutex.         */
               OS_OPT_POST_1,     /*   Only wake up highest-priority task.      */
              &err);
      if (err.Code != RTOS_ERR_NONE) {
         /* Handle error on mutex post. */
      }
   } else {
      /* Handle error on mutex pend. */
   }
          :
          :
}
```

If the kernel configuration permits it, a task can force the pends on a mutex to abort with OSMutexPendAbort().

Listing - Example of call to OSMutexPendAbort()

```
OS_MUTEX  App_Mutex;
          :
          :
void  App_SomeTask14 (void  *p_arg)
{
   RTOS_ERR    err;
   OS_OBJ_QTY  qty;
          :
          :
                                /* Abort the highest-priority task's pend. */
   qty = OSMutexPendAbort(&App_Mutex,            /* Pointer to user-allocated mutex. */
                       OS_OPT_PEND_ABORT_1,  /* Only abort the HP task's pend.   */
                      &err);
   if (err.Code != RTOS_ERR_NONE) {
      /* Handle error on mutex pend abort. */
   }
          :
          :
}
```

Synchronization with Task Semaphores

Since semaphores are sometimes used as a synchronization mechanism, the kernel allows all tasks to have a built-in semaphore. This means that instead of creating an external object and using it to synchronize, a task may pend on its built-in semaphore until some other task posts it.

The following example shows a task pending on itself using OSTaskSemPend() and another task waking it with OSTaskSemPost().

Listing - Example of call to OSTaskSemPend() and OSTaskSemPost()

```
OS_TCB  App_SomeTaskTCB;
           :
           :
void  App_SomeTask (void  *p_arg)
{
    RTOS_ERR    err;
    OS_SEM_CTR  ctr;
           :
           :
                          /* Block until another task signals this task.     */
    ctr = OSTaskSemPend( 1000,                /* Wait for a maximum of 1000 OS Ticks.*/
                         OS_OPT_PEND_BLOCKING, /* Task will block.                   */
                         DEF_NULL,
                        &err);
    if (err.Code == RTOS_ERR_NONE) {
        /* Task has been signaled by App_SomeOtherTask. */
           :
           :
    } else {
        /* Handle error on task semaphore pend. */
    }
           :
           :
}

void  App_SomeOtherTask (void  *p_arg)
{
    RTOS_ERR    err;
    OS_SEM_CTR  ctr;
           :
           :
                          /* Post the pending App_SomeTask task.           */
    ctr = OSTaskSemPost(&App_SomeTaskTCB,   /* Pointer to App_SomeTask's TCB.         */
                         OS_OPT_POST_NONE,  /* No special option.                   */
                        &err);
    if (err.Code != RTOS_ERR_NONE) {
        /* Handle error on task semaphore post. */
    }
           :
           :
}
```

Like the regular semaphore, a pend on a task semaphore can be aborted with OSTaskSemPendAbort().

Listing - Example of call to OSTaskSemPendAbort()

```
OS_TCB  App_SomeTaskTCB;
          :
          :
void  App_SomeOtherTask (void  *p_arg)
{
   RTOS_ERR    err;
   OS_OBJ_QTY  qty;
          :
          :
                            /* Abort App_SomeTask's pend.                 */
   qty = OSTaskSemPendAbort(&App_SomeTaskTCB,   /* Pointer to App_SomeTask's TCB.    */
                      OS_OPT_POST_NONE,  /* No special option.              */
                      &err);
   if (err.Code != RTOS_ERR_NONE) {
      /* Handle error on task semaphore pend abort. */
   }
          :
          :
}
```

Finally, a task can set the task's semaphore counter by calling OSTaskSemSet() to simulate the effect of calling OSTaskSemPost() multiple times without calling OSTaskSemPend().

Listing - Example of call to OSTaskSemSet()

```
OS_TCB  App_SomeTaskTCB;
          :
          :
void  App_SomeOtherTask (void  *p_arg)
{
   RTOS_ERR    err;
          :
          :
                      /* Set App_SomeTask's semaphore counter 10.            */
   OSTaskSemSet(&App_SomeTaskTCB,  /* Pointer to App_SomeTask's TCB.                */
             10,             /* 10 counts.                                */
             &err);
   if (err.Code != RTOS_ERR_NONE) {
      /* Handle error on task semaphore set count. */
   }
          :
          :
}
```

Using Event Flags

- Creating and Deleting
- Using an Event Flag

Event flags allow your application to use a bitmask, where each bit is a flag that represents a certain condition.

Creating and Deleting

Before using an event flag, it must be created with OSFlagCreate(). An event flag can be deleted with OSFlagDel() as shown in the example below.

Listing - Example of call to OSFlagCreate() and OSFlagDel()

```
OS_FLAG_GRP  App_Flags;
         :
         :
void  App_SomeTask (void  *p_arg)
{
    RTOS_ERR    err;
    OS_OBJ_QTY  qty;
         :
         :
                                    /* Create the event flag.                    */
    OSFlagCreate(&App_Flags,            /*   Pointer to user-allocated event flag.       */
                 "App Flags",           /*   Name used for debugging.                    */
                  0,                    /*   Initial flags, all cleared.                 */
                 &err);
    if (err.Code != RTOS_ERR_NONE) {
        /* Handle error on event flag create. */
    }
         :
         :
         :
                                    /* Delete the event flag.                    */
    qty = OSFlagDel(&App_Flags,         /*   Pointer to user-allocated event flag.       */
                    OS_OPT_DEL_NO_PEND, /*   Option: delete if 0 tasks are pending.      */
                   &err);
    if (err.Code != RTOS_ERR_NONE) {
        /* Handle error on event flag delete. */
    }
         :
         :
}
```

Using an Event Flag

A task can wait for a certain flag combination before doing any work. The example below shows three tasks: a task that waits with OSFlagPend() and two other tasks that set a flag with OSFlagPost().

Listing - Example of call to OSFlagPend() and OSFlagPost()

```
OS_FLAG_GRP  App_Flags;
            :
            :
                                            /* Application Flags.                */
#define  APP_FLAG_A        (1u << 0)
#define  APP_FLAG_B        (1u << 1)
#define  APP_FLAG_ALL      (APP_FLAG_A | APP_FLAG_B)
            :
            :
void  App_SomeTask (void  *p_arg)
{
    RTOS_ERR  err;
    OS_FLAGS  flags;
            :
            :
                                        /* Wait until all flags are set.           */
    flags = OSFlagPend(&App_Flags,                /*   Pointer to user-allocated event flag. */
                       APP_FLAG_ALL,             /*   Flag bitmask to match.               */
                       100,                      /*   Wait for 100 OS Ticks maximum.       */
                       OS_OPT_PEND_FLAG_SET_ALL |/*   Wait until all flags are set and     */
                       OS_OPT_PEND_BLOCKING     |/*    task will block and                 */
                       OS_OPT_PEND_FLAG_CONSUME, /*    function will clear the flags.      */
                       DEF_NULL,                 /*   Timestamp is not used.               */
                      &err);
    if (err.Code == RTOS_ERR_NONE) {
        /* Flags were set by the other tasks. */
            :
            :
    } else {
        /* Handle error on flag pend. */
    }
            :
            :
}

void  App_SomeOtherTaskA (void  *p_arg)
{
    RTOS_ERR  err;
    OS_FLAGS  flags;
            :
            :
                                        /* Set Application Flag A.                  */
    flags = OSFlagPost(&App_Flags,            /*   Pointer to user-allocated event flag.    */
                       APP_FLAG_A,           /*   Application Flag A bitmask.              */
                       OS_OPT_POST_FLAG_SET, /*   Set the flag.                            */
                      &err);
    if (err.Code != RTOS_ERR_NONE) {
        /* Handle error on flag post. */
    }
            :
            :
}

void  App_SomeOtherTaskB (void  *p_arg)
{
    RTOS_ERR  err;
    OS_FLAGS  flags;
            :
            :
                                        /* Set Application Flag B.                  */
    flags = OSFlagPost(&App_Flags,            /*   Pointer to user-allocated event flag.    */
                       APP_FLAG_B,           /*   Application Flag B bitmask.              */
                       OS_OPT_POST_FLAG_SET, /*   Set the flag.                            */
                      &err);
    if (err.Code != RTOS_ERR_NONE) {
        /* Handle error on flag post. */
```

```
::}
```

If the kernel configuration allows it, the pend on a flag combination can be aborted with OSFlagPendAbort().

Listing - Example of call to OSFlagPendAbort()

```
OS_FLAG_GRP  App_Flags;
          :
          :
void  App_SomeTask (void  *p_arg)
{
   RTOS_ERR    err;
   OS_OBJ_QTY  qty;
          :
          :
                                    /* Abort the highest-priority task's pend.  */
   qty = OSFlagPendAbort(&App_Flags,          /*   Pointer to user-allocated event flag.  */
                    OS_OPT_PEND_ABORT_1,  /*   Only abort the HP task's pend.        */
                   &err);
   if (err.Code != RTOS_ERR_NONE) {
      /* Handle error on flag pend abort. */
   }
          :
          :
}
```

A task can use OSFlagPendGetFlagsRdy() to get the currently active flags.

Listing - Example of call to OSFlagPendGetFlagsRdy()

```
OS_FLAG_GRP  App_Flags;
          :
          :
void  App_SomeTask (void  *p_arg)
{
   RTOS_ERR  err;
   OS_FLAGS  flags;
          :
          :
                                    /* Get flags that readied this task.        */
   flags = OSFlagPendGetFlagsRdy(&err);
   if (err.Code == RTOS_ERR_NONE) {
      /* 'flags' contains the flags that woke up this task. */
   } else {
      /* Handle error on get ready flags. */
   }
          :
          :
}
```

Using Message Queues

- Creating and Deleting
- Using Message Queues
- Using Task Message Queues

Message queues allow your application to send messages between tasks.

Creating and Deleting

Before using a message queue, it must first be created with OSQCreate(). A message queue can be deleted with OSQDel(). For examples of both, see the example below.

Listing - Example of call to OSQCreate() and OSQDel()

```
OS_Q  App_MessageQ;
          :
          :
void  App_SomeTask (void  *p_arg)
{
   RTOS_ERR    err;
   OS_OBJ_QTY  qty;
          :
          :
                              /* Create the message queue.                        */
   OSQCreate(&App_MessageQ,          /*   Pointer to user-allocated message queue.          */
          "App MessageQ",         /*   Name used for debugging.                         */
           10,                   /*   Queue will have 10 messages maximum.             */
          &err);
   if (err.Code != RTOS_ERR_NONE) {
      /* Handle error on message queue create. */
   }
          :
          :
          :
                              /* Delete the message queue.                        */
   qty = OSQDel(&App_MessageQ,       /*   Pointer to user-allocated message queue.          */
             OS_OPT_DEL_NO_PEND,  /*   Option: delete if 0 tasks are pending.           */
            &err);
   if (err.Code != RTOS_ERR_NONE) {
      /* Handle error on message queue delete. */
   }
          :
          :
}
```

Using Message Queues

A task can wait for a message to be added to the queue by calling OSQPend(). Tasks can send messages by calling OSQPost(). The example below shows one task waiting for a message, while another task posts a message.

Note: The content of the message is allocated in the sending task's stack. Only a pointer to the message is sent to the waiting task, rather than the message itself.

Listing - Example of call to OSQPend() and OSQPost()

```
typedef struct app_message {
    CPU_INT08U   type;
    CPU_INT08U  *p_buf;
} APP_MESSAGE;
            :
            :
OS_Q  App_MessageQ;
            :
            :
void  App_SomeTask (void  *p_arg)
{
    RTOS_ERR      err;
    void         *p_raw_msg;
    OS_MSG_SIZE   msg_size;
    APP_MESSAGE  *p_msg;
            :
            :
                                        /* Wait until task receives a message.        */
    p_raw_msg = OSQPend(&App_MessageQ,          /*   Pointer to user-allocated message queue. */
                        100,                  /*   Wait for 100 OS Ticks maximum.           */
                        OS_OPT_PEND_BLOCKING,  /*   Task will block.                        */
                       &msg_size,             /*   Will contain size of message in bytes.   */
                        DEF_NULL,             /*   Timestamp is not used.                   */
                       &err);
    if (err.Code == RTOS_ERR_NONE) {
        /* Got message, handle. */
        p_msg = (APP_MESSAGE *)p_raw_msg;
    } else {
        /* Handle error on message queue pend. */
    }
            :
            :
}

void  App_SomeOtherTask (void  *p_arg)
{
    RTOS_ERR     err;
    APP_MESSAGE  msg;
            :
            :
                                    /* Fill test message.                             */
    msg.type  = 42;
    msg.p_buf = "Test Message";
                                    /* Send message to the waiting task.              */
    OSQPost(&App_MessageQ,              /*   Pointer to user-allocated message queue.     */
            (void *)&msg,               /*   The message is a pointer to the APP_MESSAGE.   */
            (OS_MSG_SIZE)sizeof(void *),  /*   Size of the message is the size of a pointer.  */
             OS_OPT_POST_FIFO,           /*   Add message at the end of the queue.          */
            &err);
    if (err.Code != RTOS_ERR_NONE) {
        /* Handle error on message queue post. */
    }
            :
            :
}
```

If the kernel configuration allows it, the wait for a message can be aborted with OSQPendAbort().

Listing - Example of call to OSQPendAbort()

```
OS_Q  App_MessageQ;
          :
          :
void  App_SomeTask (void  *p_arg)
{
   RTOS_ERR    err;
   OS_OBJ_QTY  qty;
          :
          :
                                /* Abort the highest-priority task's pend.      */
   qty = OSQPendAbort(&App_MessageQ,        /*  Pointer to user-allocated message queue.   */
                 OS_OPT_PEND_ABORT_1,  /*  Only abort the HP task's pend.              */
                 &err);
   if (err.Code != RTOS_ERR_NONE) {
      /* Handle error on message queue pend abort. */
   }
          :
          :
}
```

An application can remove all messages from a message queue by using OSQFlush().

Listing - Example of call to OSQFlush()

```
OS_Q  App_MessageQ;
          :
          :
void  App_SomeTask (void  *p_arg)
{
   RTOS_ERR    err;
   OS_MSG_QTY  qty;
          :
          :
                        /* Flush (empty) the message queue.                      */
   qty = OSQFlush(&App_MessageQ,
              &err);
   if (err.Code != RTOS_ERR_NONE) {
      /* Handle error on message queue flush. */
   }
          :
          :
}
```

Using Task Message Queues

Since message queues are often used to send messages to a single task, a task can have its own built-in message queue. The example call OSTaskCreate() in Task Management Kernel Services shows how to create a task which can have up to 10 messages pending.

A task can wait on its built-in message queue by calling OSTaskQPend(). Another task can send a message to the waiting task by calling OSTaskQPost().

Listing - Example of call to OSTaskQPend() and OSTaskQPost()

```
typedef struct app_message {
   CPU_INT08U   type;
   CPU_INT08U  *p_buf;
} APP_MESSAGE;
          :
          :
OS_TCB  App_SomeTaskTCB;
          :
          :
void  App_SomeTask (void  *p_arg)
{
   RTOS_ERR      err;
   void         *p_raw_msg;
   OS_MSG_SIZE   msg_size;
   APP_MESSAGE  *p_msg;
          :
          :
                                    /* Wait until task receives a message.     */
   p_raw_msg = OSTaskQPend( 100,                 /*   Wait for 100 OS Ticks maximum.         */
                    OS_OPT_PEND_BLOCKING, /*   Task will block.                       */
                   &msg_size,            /*   Will contain size of message in bytes.*/
                    DEF_NULL,            /*   Timestamp is not used.                 */
                   &err);
   if (err.Code == RTOS_ERR_NONE) {
      /* Got message, handle. */
      p_msg = (APP_MESSAGE *)p_raw_msg;
   } else {
      /* Handle error on task message queue pend. */
   }
          :
          :
}

void  App_SomeOtherTask (void  *p_arg)
{
   RTOS_ERR     err;
   APP_MESSAGE  msg;
          :
          :
                                 /* Fill test message.                         */
   msg.type  = 42;
   msg.p_buf = "Test Message";
                                 /* Send message to the waiting task.            */
   OSTaskQPost(&App_SomeTaskTCB,            /*   Pointer to pending task's TCB.              */
            (void *)&msg,                /*   Message is a pointer to the APP_MESSAGE.   */
            (OS_MSG_SIZE)sizeof(void *),  /*   Size of message is the size of a pointer.  */
             OS_OPT_POST_FIFO,            /*   Add message at the end of the queue.       */
            &err);
   if (err.Code != RTOS_ERR_NONE) {
      /* Handle error on task message queue post. */
   }
          :
          :
}
```

If the kernel configuration allows it, the wait for a message can be aborted with OSTaskQPendAbort().

Listing - Example of call to OSTaskQPendAbort()

```
OS_TCB  App_SomeTaskTCB;
          :
          :
void  App_SomeOtherTask (void  *p_arg)
{
    RTOS_ERR     err;
    CPU_BOOLEAN  aborted;
          :
          :
                                          /* Abort the task's pend.                 */
    aborted = OSTaskQPendAbort(&App_SomeTaskTCB,  /*   Pointer to pending task's TCB.        */
                               OS_OPT_POST_NONE,  /*   No special options.                   */
                              &err);
    if (err.Code != RTOS_ERR_NONE) {
        /* Handle error on task message queue pend abort. */
    }
          :
          :
}
```

Your application can remove all messages from a task message queue by using OSTaskQFlush().

Listing - Example of call to OSTaskQFlush()

```
OS_TCB  App_SomeTaskTCB;
          :
          :
void  App_SomeOtherTask (void  *p_arg)
{
    RTOS_ERR    err;
    OS_MSG_QTY  qty;
          :
          :
                                    /* Flush (empty) the message queue.                */
    qty = OSTaskQFlush(&App_SomeTaskTCB,  /*   Pointer to pending task's TCB.               */
                       &err);
    if (err.Code != RTOS_ERR_NONE) {
        /* Handle error on task message queue flush. */
    }
          :
          :
}
```

Using Monitors

- Creating and Deleting
- Using a Monitor
- Callback Usage Details
  - Enter Callback
  - Evaluation Callback

As previously discussed in Advanced Task Synchronization , monitors allow a task to pend on a certain condition.

Creating and Deleting

Before using a monitor, you must create it with OSMonCreate(). You can delete a monitor with OSMonDel(), as shown in the example below.

Listing - Example of call to OSMonCreate() and OSMonDel()

```
OS_MON      App_Mon;
CPU_INT32U  App_State;
          :
          :
void  App_SomeTask (void  *p_arg)
{
   RTOS_ERR    err;
   OS_OBJ_QTY  qty;
          :
          :
                                                  /* Create Monitor.                     */
   OSMonCreate(&App_Mon,                                 /*   Pointer to user allocated monitor.     */
            "App Monitor",                        /*   Name used for debugging.             */
            (void *)&App_State,                   /*   Global monitor data: the State.        */
            &err);
   if (err.Code != RTOS_ERR_NONE) {
      /* Handle error on monitor create. */
   }
          :
          :
          :
                                                  /* Delete Monitor.                     */
   qty = OSMonDel(&App_Mon,                              /*   Pointer to user allocated monitor.     */
               OS_OPT_DEL_NO_PEND,                /*   Option: delete if 0 tasks are pending.   */
              &err);
   if (err.Code != RTOS_ERR_NONE) {
      /* Handle error on monitor delete. */
   }
          :
          :
}
```

Using a Monitor

Tasks can use OSMonOp() to operate on a monitor. This function will register two callbacks:

- One is called immediately: the Enter callback
- The other is called every time the monitor is accessed after the current call: the Evaluate callback.

Some applications only require either one of the two functions.

In the examples below, one task uses the Enter callback to change the state of the monitor, while the other task uses the Evaluate callback to determine if the current state matches the one it requires.

The last section illustrates the usage of both callbacks.

Listing - Example of call to OSMonOp(), pend-like operation

```
OS_MON  App_Mon;
          :
          :
                                        /* Application States.                */
#define  APP_MON_PEND_ON_STATE                 0xC0FFEEu
          :
          :
OS_MON_RES  App_MonPend (OS_MON  *p_mon,
                   void    *p_eval_data,
                   void    *p_arg)
{
   CPU_INT32U  *p_state;
   OS_MON_RES   res;

                                        /* Prevent compiler warning.          */
   PP_UNUSED_PARAM(p_arg);
                                        /* Get State variable.                */
   p_state = (CPU_INT32U *)p_mon->MonDataPtr;
                                        /* If new state matches, wake up the pending task. */
   if (*p_state == (CPU_INT32U)p_eval_data) {
      res = OS_MON_RES_ALLOW;
   } else {
      res = OS_MON_RES_BLOCK;
   }

   return (res);
}
          :
          :
void  App_SomeTask (void  *p_arg)
{
   RTOS_ERR  err;
          :
          :
                                        /* Wait for State.                    */
   OSMonOp(&App_Mon,                    /*   Pointer to user allocated monitor.         */
           500,                         /*   Timeout: 500 OS Ticks.                    */
          (void *)APP_MON_PEND_ON_STATE,   /*   Wait for this state.                  */
           DEF_NULL,                    /*   Function used on entering the monitor.    */
          &App_MonPend,                 /*   Function used for evaluation.             */
           0,                           /*   Option: none.                            */
          &err);
   if (err.Code == RTOS_ERR_NONE) {
      /* Operation successful. */
   } else {
      /* Handle error on monitor operation. */
   }
          :
          :
}
```

Listing - Example of call to OSMonOp(), post-like operation

```
OS_MON  App_Mon;
         :
         :
                                          /* Application States.                */
#define  APP_MON_PEND_ON_STATE                 0xC0FFEEu
         :
         :
OS_MON_RES  App_MonPost (OS_MON  *p_mon,
                  void    *p_arg)
{
   CPU_INT32U *p_state;

                                          /* Get State variable.                */
   p_state = (CPU_INT32U *)p_mon->MonDataPtr;
                                          /* Change state.                      */
   *p_state = (CPU_INT32U)p_arg;

   return (OS_MON_RES_ALLOW);
}
         :
         :
void  App_SomeOtherTask (void  *p_arg)
{
   RTOS_ERR  err;
         :
         :
                                          /* Change State.                      */
   OSMonOp(&App_Mon,                      /*  Pointer to user allocated monitor.      */
         0,                        /*   Timeout: none, this is a post operation.  */
        (void *)APP_MON_PEND_ON_STATE,         /*   Change state to new value.            */
        &App_MonPost,                      /*  Function used on entering the monitor.   */
         DEF_NULL,                     /*   Function used for evaluation.          */
         0,                        /*   Option: none.                       */
        &err);
   if (err.Code != RTOS_ERR_NONE) {
      /* Handle error on monitor operation. */
   }
         :
         :
}
```

Callback Usage Details

The kernel offers a generic framework for creating monitors. Your application must implement both an **enter** and an **eval** callback to create the desired effect. The following sub-sections describe the interactions between both callbacks. Note that the kernel example contains an example usage of a monitor that illustrates how tasks could wait until a Finite-State Machine is in a specific state.

Enter Callback

The **enter** callback has the following prototype:

Listing - Enter callback prototype

```
OS_MON_RES  EnterCallback(OS_MON  *p_mon,
                  void    *p_data);
```

When calling OSMonOp(), the **enter** callback is called before anything else. The first parameter is a pointer to the monitor that is being operated on and the second parameter is the p_arg parameter passed to OSMonOp() .

Depending on the callback's return value, different actions will be taken. The following table describes the return values for the **enter** callback.

Return Value	Description
OS_MON_RES_ALLOW	The calling task will not be blocked.
OS_MON_RES_BLOCK	The calling task will be blocked. The p_on_eval argument passed to OSMonOp() will be saved as the task's **evaluation** callback. OSMonOp() 's p_arg argument will be saved as the task's evaluation data argument.
OS_MON_RES_STOP_EVAL	Returning this will prevent the **evaluation** callback of the waiting tasks from being executed. Can be ORed with the previous return values.

If the p_enter argument of the OSMonOp() call is DEF_NULL, the default return value of OS_MON_RES_BLOCK \| OS_MON_RES_STOP_EVAL will be used instead.

Evaluation Callback

The **evaluation** callback has the following prototype:

Listing - Evaluation callback prototype

```
OS_MON_RES  EvaluationCallback(OS_MON  *p_mon,
                               void    *p_eval_data,
                               void    *p_arg);
```

After a monitor operation, if the **enter** callback does not prevent the evaluation from taking place, the monitor calls the **evaluation** callback of every task waiting for a specific condition. The callback's first parameter, p_mon, is a pointer to the monitor that is being operated on. The second parameter, p_eval_data, is the p_arg parameter that was passed to the OSMonOp() call that blocked the task, and the third parameter, p_arg, is the the p_arg parameter passed to the OSMonOp() call that caused the **evaluation** callback to be executed.

Like the **enter** callback, depending on the **evaluation** callback's return value, different actions will be taken. The following table describes the return values for the **evaluation** callback.

Return Value	Description
OS_MON_RES_ALLOW	It was determined, based on p_eval_data, p_arg and the monitor's data that the waiting task's specific condition is met, it must be readied.
OS_MON_RES_BLOCK	The waiting task cannot be resumed because its specific condition was not met.
OS_MON_RES_STOP_EVAL	Prevent the evaluation of the monitor by the other waiting tasks. Can be ORed with the previous return values.

The OS_MON_RES_STOP_EVAL return value is used to prevent multiple tasks from becoming ready at the same time.

Kernel Time Management

- Time Base Management
- Time Delays
  - Basic Delays
  - Resuming a Task
  - Periodic Delays

Time Base Management

An application can request the current time (measured in OS Ticks) from the kernel by calling OSTimeGet().

Listing - Example of call to OSTimeGet() and OSTimeSet()

```
void  App_SomeTask (void  *p_arg)
{
    RTOS_ERR  err;
    OS_TICK   time;
```

```
:/* Get kernel time, in OS Ticks.                */
  time =OSTimeGet(&err);if(err.Code != RTOS_ERR_NONE){/* Handle error on time get. */}}
```

Time Delays

Basic Delays

A task can pause itself for a specified amount of time, specified either in OS Ticks or in Wall Clock time. The snippet below showcases the usage of OSTimeDly() and OSTimeDlyHMSM().

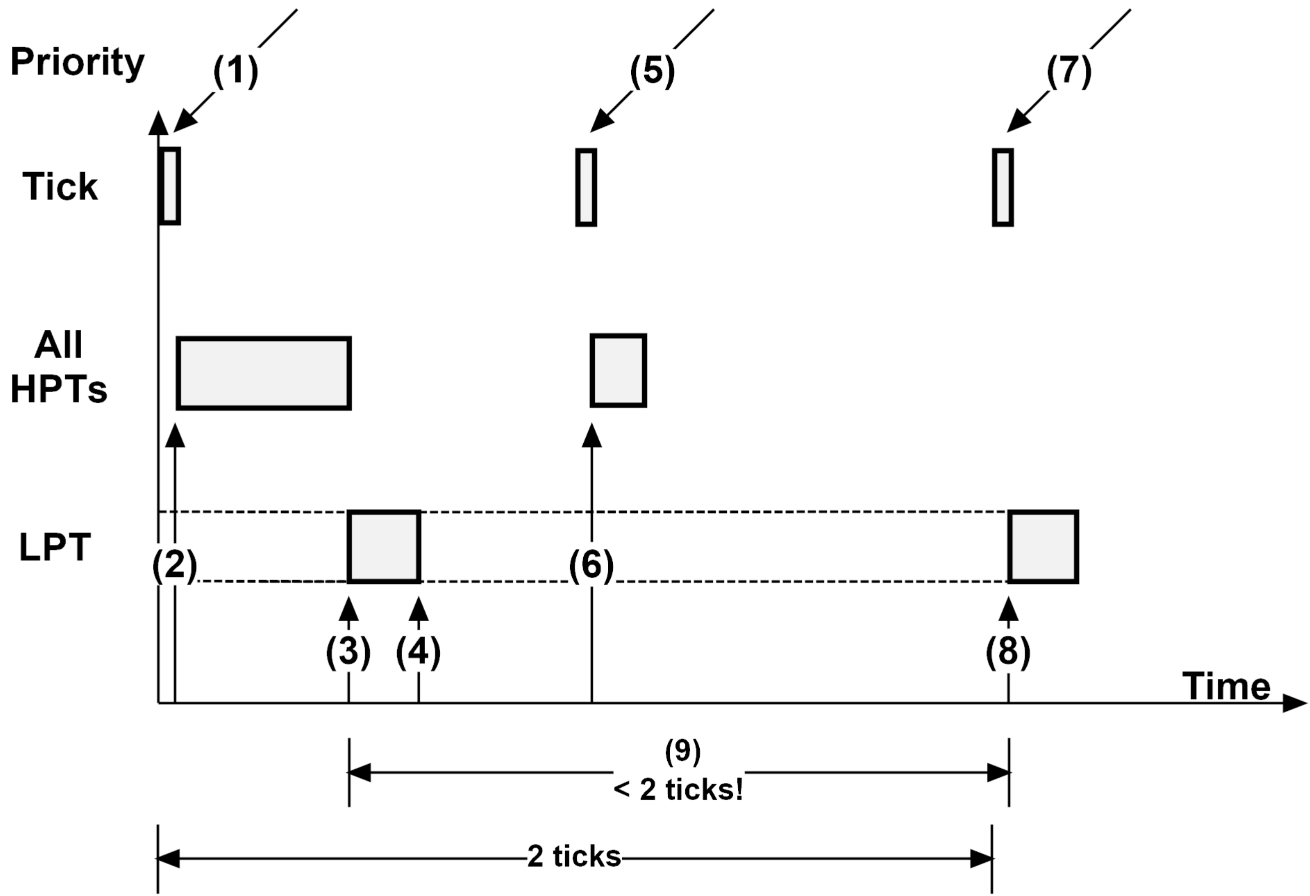
Listing - Example of call to OSTimeDly() and OSTimeDlyHMSM()

```
void  App_SomeTask (void  *p_arg)
{
    RTOS_ERR  err;
           :
           :
                            /* Delay task for 100 OS Ticks.              */
    OSTimeDly( 100,              /*   Delay the task for 100 OS Ticks.        */
              OS_OPT_TIME_DLY,  /*   Delay is relative to current time.      */
             &err);
    if (err.Code != RTOS_ERR_NONE) {
        /* Handle error on time delay. */
    }
           :
           :
}

void  App_SomeTask2 (void  *p_arg)
{
    RTOS_ERR  err;
           :
           :
                               /* Delay task for 1:03:04.250.           */
    OSTimeDlyHMSM(   1,             /*   1 hour,                        */
                    3,             /*   3 minutes,                      */
                    4,             /*   4 seconds and                    */
                  250,             /*   250 milliseconds.                */
                  OS_OPT_TIME_DLY,  /*   Delay is relative to current time.   */
                 &err);
    if (err.Code != RTOS_ERR_NONE) {
        /* Handle error on time delay in HMSm format. */
    }
           :
           :
}
```

Care must be taken when using relative delays. Indeed, these delays are not accurate since they begin counting down at the next Kernel tick which could occur immediately. The figure below illustrates the imprecision caused by relative delays.

Figure - OSTimeDly() Accuracy



(1) A tick interrupt occurs and the Kernel services the ISR.

(2) At the end of the ISR, all Higher Priority Tasks (HPTs) execute. The execution time of HPTs is unknown and can vary.

(3) Once all HPTs have executed, the Kernel runs the task that has called OSTimeDly(). For the sake of discussion, it is assumed that the task is a lower priority task (LPT).

(4) The task calls OSTimeDly() and specifies to delay for two ticks in "relative" mode. At this point, the Kernel places the calling task in the tick list where it will wait for two kernel ticks to expire. The delayed task consumes zero CPU time while waiting for the time to expire.

(5) The next tick occurs. If there are HPTs waiting for this particular tick, the Kernel will schedule them to run at the end of the ISR.

(6) The HPTs execute.

(7) The next tick interrupt occurs. This is the tick that the LPT was waiting for and will now be made ready-to-run by the Kernel.

(8) Since there are no HPTs to execute on this tick, the Kernel switches to the LPT.

(9) Given the execution time of the HPTs, the time delay is not exactly two ticks, as requested. In fact, it is virtually impossible to obtain a delay of exactly the desired number of ticks. You might ask for a delay of two ticks, but the very next tick could occur almost immediately after calling OSTimeDly(). Imagine what might happen if all HPTs took longer to execute and pushed (3) and (4) further to the right. In that case, the delay would actually appear as one tick instead of two.

Resuming a Task

If the kernel configuration allows it, a task can abort the delay requested by another task with OSTimeDlyResume().

Listing - Example of call to OSTimeDlyResume()

```
OS_TCB  App_SomeTaskTCB;
           :
           :
void  App_SomeOtherTask (void  *p_arg)
{
    RTOS_ERR  err;
           :
           :
                         /* Cancel the requested delay.          */
    OSTimeDlyResume(&App_SomeTaskTCB,  /*   Pointer to the waiting task's TCB. */
                    &err);
    if (err.Code != RTOS_ERR_NONE) {
        /* Handle error on time delay abort. */
    }
           :
           :
}
```

Periodic Delays

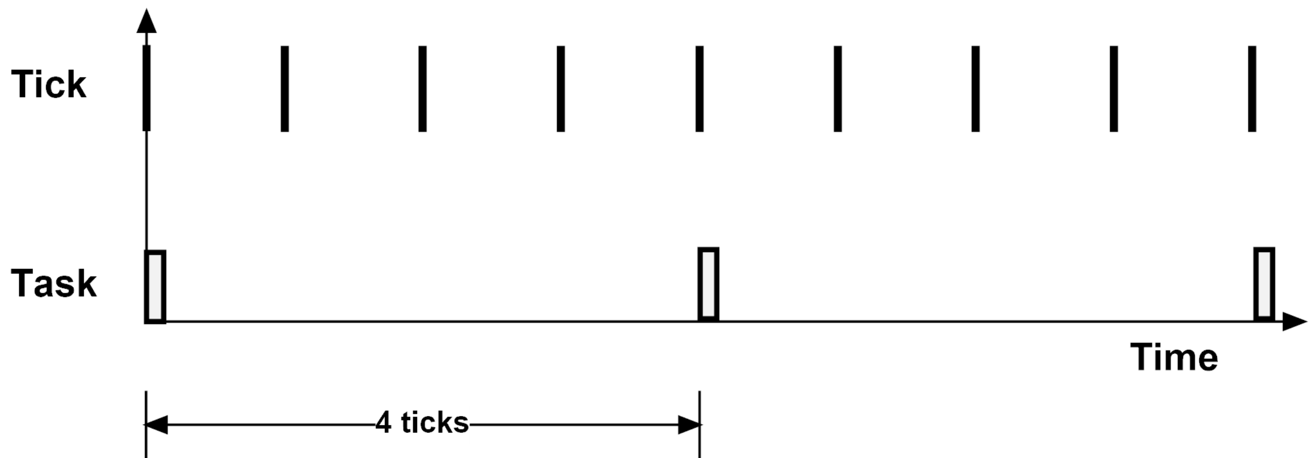
A task can specify that the delay will be periodic. The listing below shows an example call to OSTimeDly() with a periodic delay.

Listing - Example of call to OSTimeDly() and OSTimeDlyHMSM()

```
void  App_SomeTask3 (void  *p_arg)
{
    RTOS_ERR  err;
           :
           :
    while (DEF_ON) {
                                   /* Delay task for 4 OS Ticks.              */
        OSTimeDly( 4,                   /*   Delay the task for 4 OS Ticks.        */
                   OS_OPT_TIME_PERIODIC,  /*   Delay is periodic since last call.   */
                  &err);
        if (err.Code != RTOS_ERR_NONE) {
            /* Handle error on time delay. */
        }
          :
          :
    }
}
```

This will cause the Kernel to schedule the task when at least 4 Kernel ticks have occurred since the last time the task called OSTimeDly(). The image below illustrates a periodic delay of 4 Kernel ticks.

Figure - Periodic Delay



When using periodic delays, the Kernel will determine the precise OS Tick value at which the task must wake up to ensure the needed period elapsed. At the same time, the Kernel will account for the variance in CPU usage

Using the Kernel Timers

OSTimeTick

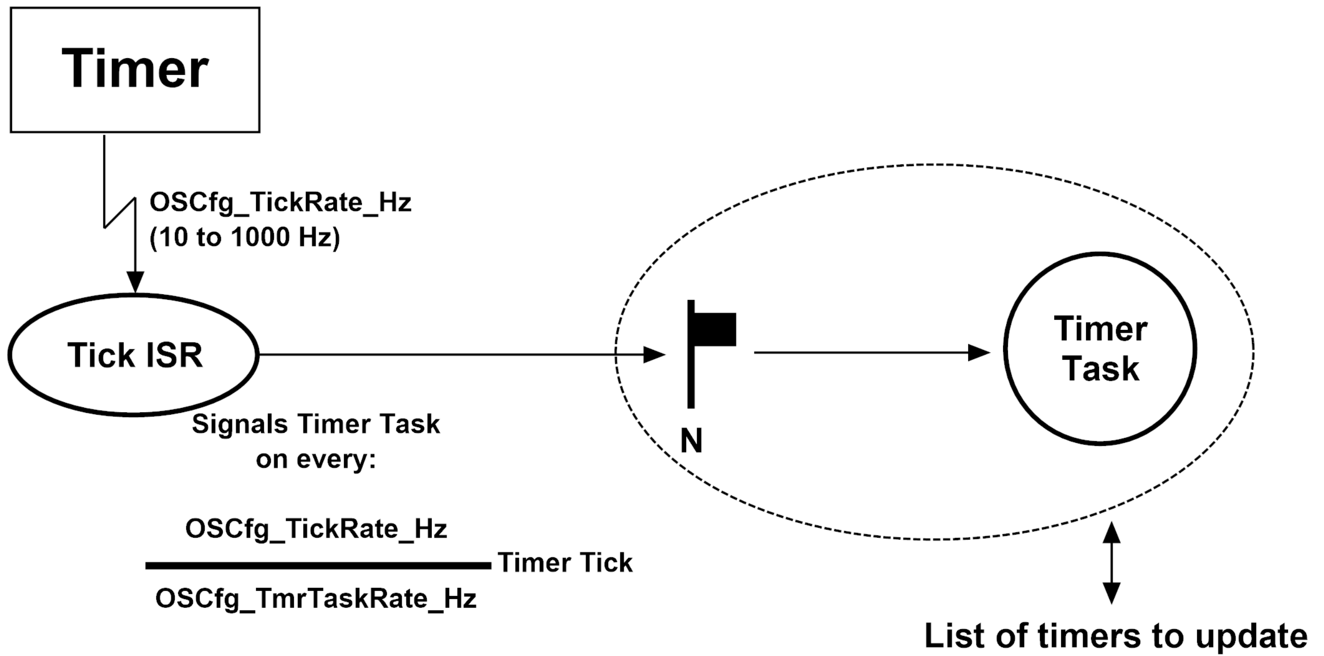
- Overview
- Creating and Deleting Timers
- Using Timers
  - Timer Modes of Operation
    - One-Shot
    - Periodic Without an Initial Delay
    - Periodic With an Initial Delay
- Getting State Information
- Timer Processing
  - Notes on Timer Processing

The kernel allows your application to define software countdown timers that call a user-defined function once the timer expires.

Overview

The Kernel timers are handled by a task, which is known as the Timer Task. This task manages a list of user-defined software timers and calls the user-registered callback when the timer expires. By default, the Timer Task will execute 10 times per second, or at 10 Hz. This allows the software timers to use a timebase of 100 ms. This rate, known as the Timer Task Rate can be changed at run-time. See the Timer Task row in the Kernel Run-Time Application Specific Configurations page for more information. The software timers use this timebase to define periods and delays. For example, a one second, one-shot software timer would use a delay of 10 Timer ticks, because 10*100 ms equals one second.

Figure - The Timer Task



As seen in the figure above, the Timer Task is executed every OSCfg_TickRate_Hz/OSCfg_TmrTaskRate_Hz Kernel tick. By default, the Tick Rate is 1000 Hz and the Timer Tick Rate is 10 Hz, this means that the Timer Task is signaled once every 100 Kernel tick.

Creating and Deleting Timers

Your application can create a periodic timer or a one-shot timer with OSTmrCreate(). The timer can be deleted with OSTmrDel() (see the example below).

Listing - Example of call to OSTmrCreate() and OSTmrDel()

```
OS_TMR  App_Timer;
          :
          :
void  App_TimerCallback (void  *p_tmr,
                         void  *p_arg)
{
   /* Called when timer expires:                            */
   /*   'p_tmr' is pointer to the user-allocated timer.     */
   /*   'p_arg' is argument passed when creating the timer. */
}
          :
          :
void  App_SomeTask (void  *p_arg)
{
   RTOS_ERR     err;
   CPU_BOOLEAN  deleted;
          :
          :
                                /* Create a periodic timer.               */
   OSTmrCreate(&App_Timer,           /*   Pointer to user-allocated timer.     */
               "App Timer",          /*   Name used for debugging.             */
                 0,                  /*     0 initial delay.                   */
                100,                 /*   100 Timer Ticks period.              */
                OS_OPT_TMR_PERIODIC, /*   Timer is periodic.                   */
               &App_TimerCallback,   /*   Called when timer expires.           */
                DEF_NULL,            /*   No arguments to callback.            */
               &err);
   if (err.Code != RTOS_ERR_NONE) {
      /* Handle error on timer create. */
   }
          :
          :
          :
                                /* Delete the periodic timer.              */
   deleted = OSTmrDel(&App_Timer,  /*   Pointer to user-allocated timer.       */
                      &err);
   if (err.Code != RTOS_ERR_NONE) {
      /* Handle error on timer delete. */
   }
          :
          :
}
```

Using Timers

Once a timer is created, it can be started with OSTmrStart().

Listing - Example of call to OSTmrStart()

```
OS_TMR  App_Timer;
          :
          :
void  App_SomeTask (void  *p_arg)
{
   RTOS_ERR     err;
   CPU_BOOLEAN  started;
          :
          :
                            /* Start the timer.                    */
   started = OSTmrStart(&App_Timer,  /*   Pointer to user-allocated timer.      */
                   &err);
   if (err.Code != RTOS_ERR_NONE) {
      /* Handle error on timer start. */
   }
          :
          :
}
```

A running timer can be stopped by calling OSTmrStop().

Listing - Example of call to OSTmrStop()

```
OS_TMR  App_Timer;
          :
          :
void  App_SomeTask (void  *p_arg)
{
   RTOS_ERR     err;
   CPU_BOOLEAN  stopped;
          :
          :
                               /* Stop the timer.                   */
   stopped = OSTmrStop(&App_Timer,       /*   Pointer to user-allocated timer. */
                   OS_OPT_TMR_NONE,  /*   Do not execute callback.         */
                   DEF_NULL,         /*   No arguments to callback.        */
                  &err);
   if (err.Code != RTOS_ERR_NONE) {
      /* Handle error on timer stop. */
   }
          :
          :
}
```

Finally, your application can use OSTmrSet() to reuse an existing timer and change its delay, period, or callback function.

Listing - Example of call to OSTmrSet()

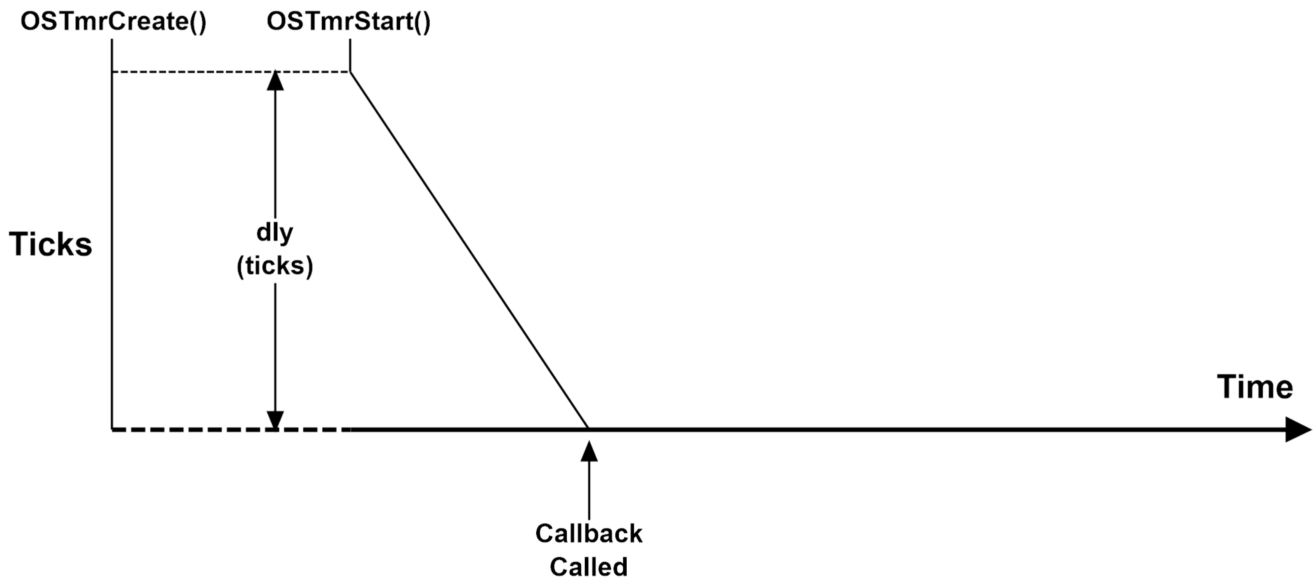
```
OS_TMR  App_Timer;
          :
          :
void  App_NewTimerCallback (void  *p_tmr,
                            void  *p_arg)
{
    /* Called when timer expires:                            */
    /*   'p_tmr' is pointer to the user-allocated timer.     */
    /*   'p_arg' is argument passed when creating the timer. */
}
          :
          :
void  App_SomeTask (void  *p_arg)
{
    RTOS_ERR  err;
          :
          :
                              /* Change callback and set period to 250.   */
    OSTmrSet(&App_Timer,            /*   Pointer to user-allocated timer.       */
              0,                    /*   0 initial delay.                       */
             250,                   /*   250 Timer Ticks period.                */
            &App_NewTimerCallback,  /*   Called when timer expires.             */
             DEF_NULL,              /*   No argument to callback.               */
            &err);
    if (err.Code != RTOS_ERR_NONE) {
        /* Handle error on timer set. */
    }
          :
          :
}
```

Timer Modes of Operation

One-Shot

As said before, a software timer can be either one-shot, periodic with no initial delay or periodic with an initial delay. A one-shot timer is created when the OS_OPT_TMR_ONE_SHOT option is used when creating the timer. The timeline of a one-shot timer is illustrated below.

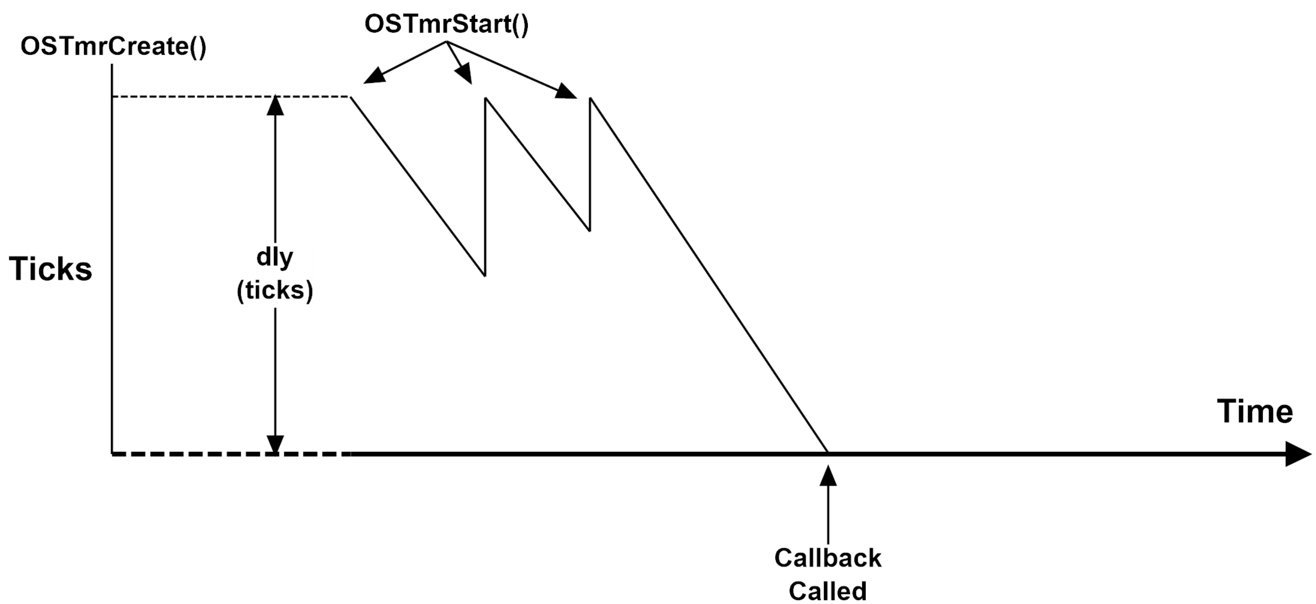
Figure - One-Shot Timer



This timer will expire after *dly* Timer Ticks have elapsed. When the timer expires, it will call the callback once.

You can reset the elapsed time of a running timer. For example, if your application calls OSTmrStart() on a currently running timer, it will reset the delay to the value set when the timer was created. The figure below shows the state of a one-shot timer being retriggered two times after the initial start.

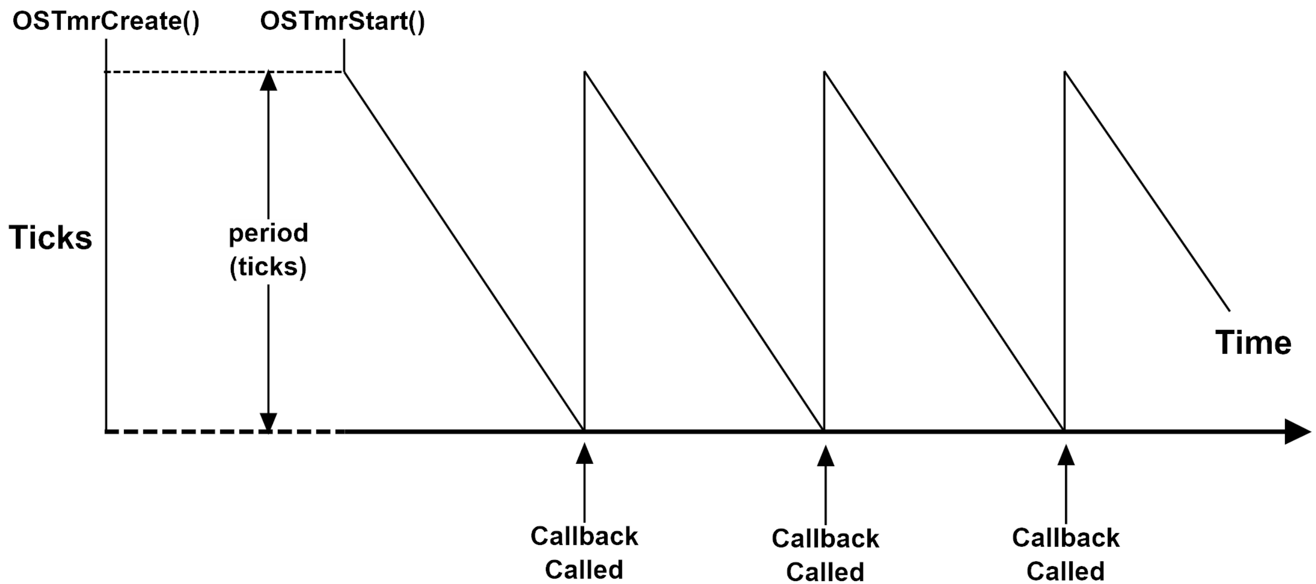
Figure - Retriggering a One-Shot Timer



Periodic Without an Initial Delay

A timer created with the OS_OPT_TMR_PERIODIC option is a periodic timer. When the timer's period has expired, the timer's callback will be called. Once the callback returns, the timer will be reset to trigger the callback again once the same period has elapsed. This mode of operation is shown below. The callback is indeed called when *period* Timer Ticks have elapsed.

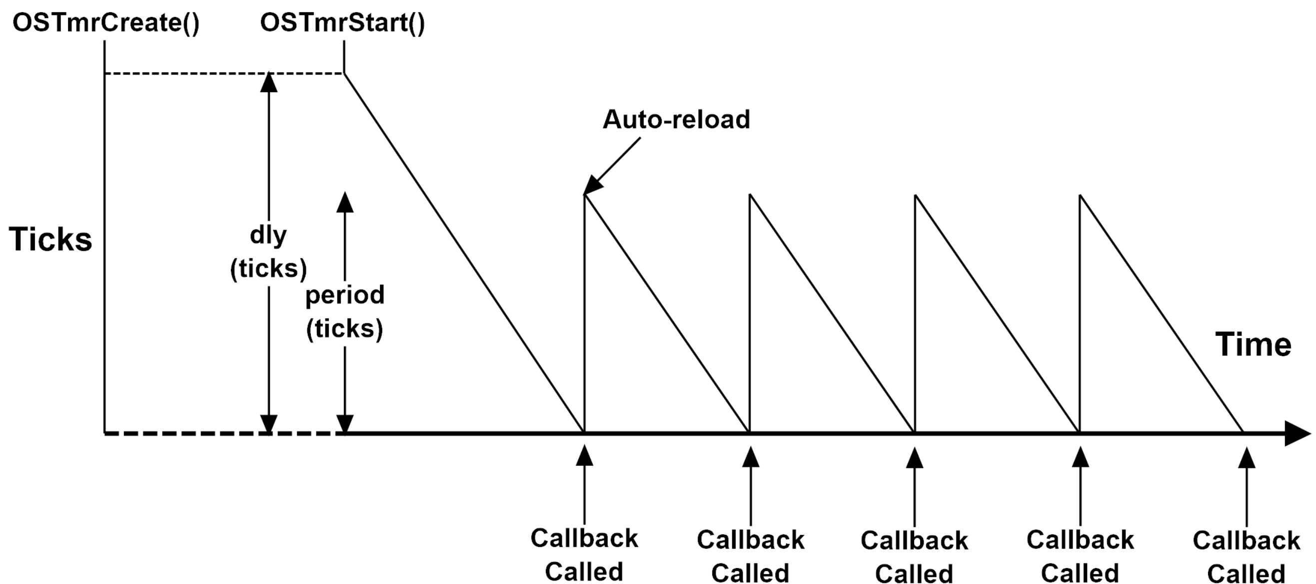
Figure - Periodic Timer Without Initial Delay



Periodic With an Initial Delay

A periodic timer can also be created with an initial delay. This means that the timer will wait until a specified amount of time has elapsed before starting the timer in the previously described periodic mode. The figure below shows the usage of a periodic timer with an initial delay of *dly* Timer Ticks and a period of *period* Timer Ticks. The first call to the callback will be made once *dly* Timer Ticks have elapsed. All subsequent calls to the callback will be made when *period* Timer Ticks have elapsed.

Figure - Periodic Timer With Initial Delay



Getting State Information

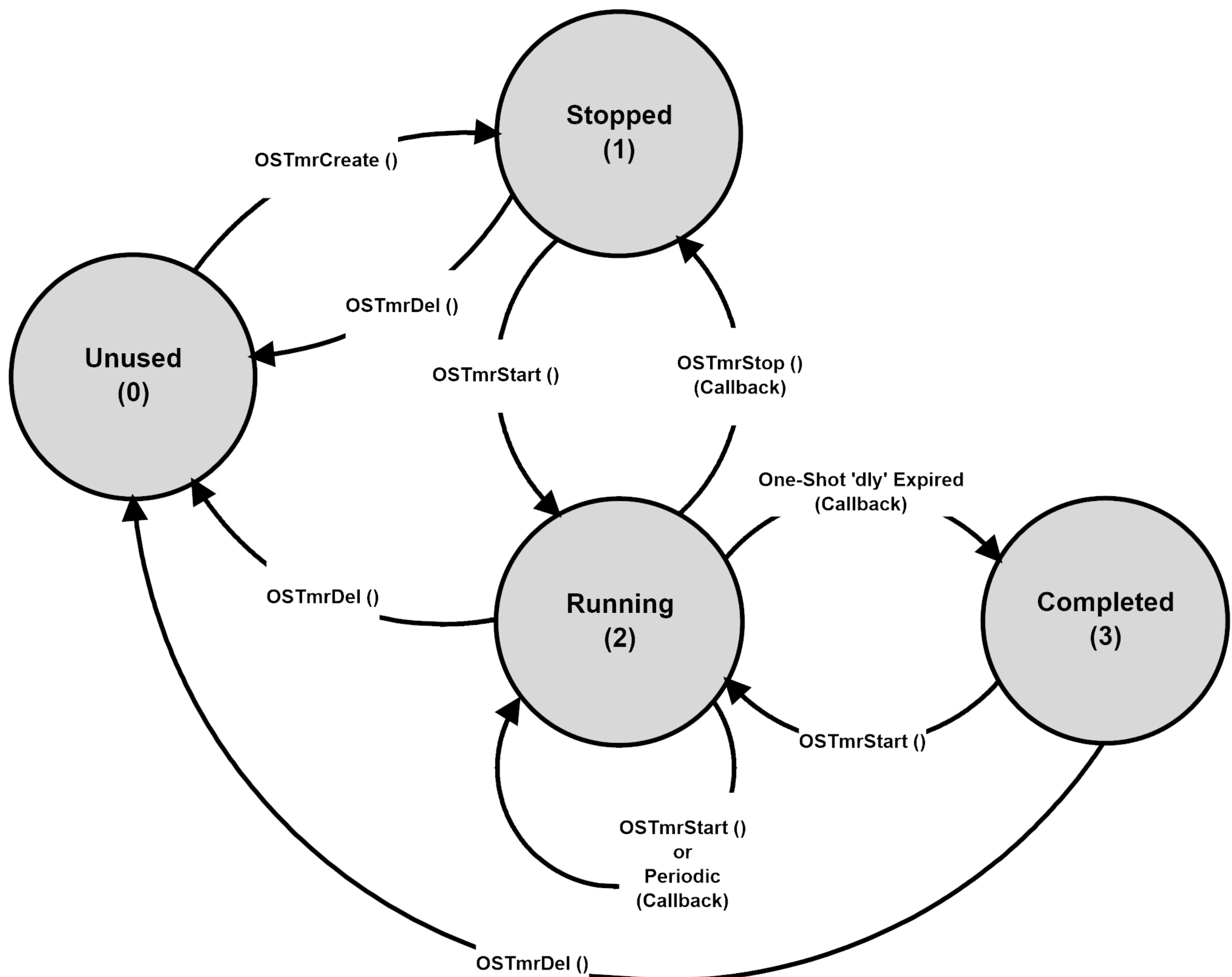
Your application can query the state of a timer by calling OSTmrStateGet().

Listing - Example of call to OSTmrStateGet()

```
OS_TMR  App_Timer;
          :
          :
void  App_SomeTask (void  *p_arg)
{
   RTOS_ERR  err;
   OS_STATE  state;
          :
          :
                           /* Get timer state.                    */
   state = OSTmrStateGet(&App_Timer,  /*   Pointer to user-allocated timer.     */
                         &err);
   if (err.Code != RTOS_ERR_NONE) {
      /* Handle error on timer state get. */
   }
          :
          :
}
```

The figure below and its listing explain the different states used by the Kernel software timers.

Figure - Timer States



(0) The “Unused” state is used by a timer that has not been created or that has been deleted. In other words, the Kernel does not manage this timer.

(1) When creating a timer or calling OSTmrStop(), the timer is placed in the “Stopped” state.

(2) A timer is placed in the "Running" state when calling OSTmrStart(). The timer stays in that state unless it’s stopped, deleted, or completes its one shot.

(3) The “Completed” state is the state of a one-shot timer once its delay expires.

To get the number of Timer Ticks before a timer expires, your application can use OSTmrRemainGet().

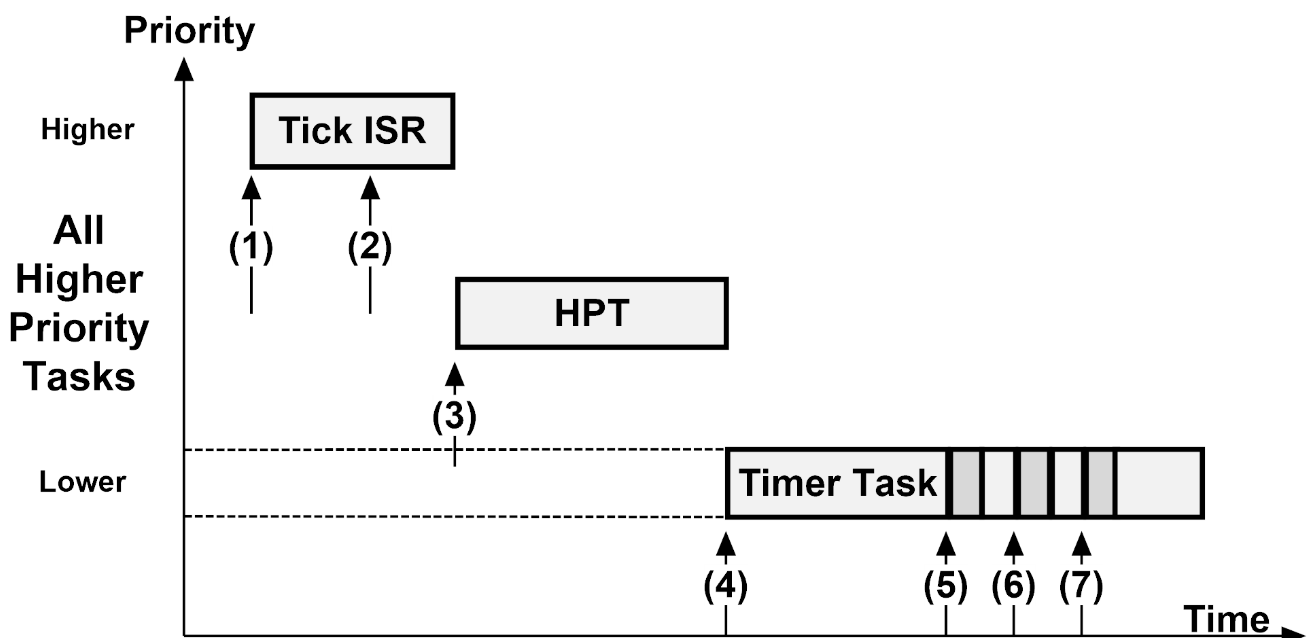
Listing - Example of call to OSTmrRemainGet()

```
OS_TMR  App_Timer;
          :
          :
void  App_SomeTask (void  *p_arg)
{
    RTOS_ERR  err;
    OS_TICK   remain;
          :
          :
                                  /* Get remaining time, in Timer Ticks.  */
    remain = OSTmrRemainGet(&App_Timer,  /*  Pointer to user-allocated timer.  */
                            &err);
    if (err.Code != RTOS_ERR_NONE) {
        /* Handle error on timer time remaining. */
    }
          :
          :
}
```

Timer Processing

As explained earlier in the Overview section, the Kernel's Timer Task is used to manage the software timers created by your application. The illustration below shows the interaction between the Tick ISR, the Timer Task and three software timers.

Figure - Software Timer Management



(1) A Tick occurs and its corresponding ISR is executed.

(2) The Tick ISR signals the Timer Task that it is time for it to update the software timers.

(3) The Tick ISR terminates, however there might be higher priority tasks that need to execute since the Timer Task may have a lower priority. The Kernel then runs the higher priority task(s).

(4) When all higher priority tasks have executed, the Kernel switches to the Timer Task and determines that there are three timers that expired.

(5) The callback for the first timer is executed.

(6) Once the callback for the first timer returns, the callback for the second expired timer is executed.

(7) When the callback for the second timer returns, the callback for the third expired timer is executed.

Notes on Timer Processing

The Timer Task is responsible for calling the application-defined callbacks. As such, a few points are worth noting:

- Execution of the callback functions is performed within the context of the Timer Task. This means that the application code will need to make sure that there is sufficient stack space for the Timer Task to handle these callbacks.
- The callback functions are executed one after the other based on the order they are found in the Timer processing list.
- The execution time of the Timer Task depends on how many timers expire and how long each of the callback functions take to execute.
- The timer callback functions must never modify the timer object, or wait on events or other Kernel objects, or make any blocking calls, because this would delay the Timer Task for excessive amounts of time – or worse, forever.
- Since the execution time of the Timer Task mostly depends on the application-defined callbacks, they should execute as quickly as possible.

Extending the Kernel with Application Hooks

Application hooks allow your application code to extend the functionality of the kernel.

Table - Kernel Hooks

The Kernel Port Hook	Calls the Application-defined Hook
OSTaskCreateHook()	OS_AppTaskCreateHookPtr
OSTaskDelHook()	OS_AppTaskDelHookPtr
OSTaskReturnHook()	OS_AppTaskReturnHookPtr
OSRedzoneHitHook()	OS_AppRedzoneHitHookPtr
OSStatTaskHook()	OS_AppStatTaskHookPtr
OSTaskSwHook()	OS_AppTaskSwHookPtr

Application hook functions can be declared as shown in the code below.

```
void  App_OS_TaskCreateHook (OS_TCB  *p_tcb)
{
   /* Your code here */
}

void  App_OS_TaskDelHook (OS_TCB  *p_tcb)
{
   /* Your code here */
}

void  App_OS_TaskReturnHook (OS_TCB  *p_tcb)
{
   /* Your code here */
}

void  App_OS_RedzoneHitHook (OS_TCB  *p_tcb)
```

```
{/* Your code here */}

void  App_OS_StatTaskHook (void){/* Your code here */}

void  App_OS_TaskSwHook (void){/* Your code here */}
```

You must define the value of the pointers so that they point to the appropriate functions (as shown below). The pointers do not have to be set in main(), but they must be set after you called OSInit() and before OSStart().

```
void  main (void)
{
    RTOS_ERR  err;

    OSInit(..., &err);
    :
    :
    OS_AppTaskCreateHookPtr = (OS_APP_HOOK_TCB )App_OS_TaskCreateHook;
    OS_AppTaskDelHookPtr    = (OS_APP_HOOK_TCB )App_OS_TaskDelHook;
    OS_AppTaskReturnHookPtr = (OS_APP_HOOK_TCB )App_OS_TaskReturnHook;
    OS_AppRedzoneHitHookPtr = (OS_APP_HOOK_VOID)App_OS_RedzoneHitHook;
    OS_AppStatTaskHookPtr   = (OS_APP_HOOK_VOID)App_OS_StatTaskHook;
    OS_AppTaskSwHookPtr     = (OS_APP_HOOK_VOID)App_OS_TaskSwHook;
    :
    :
    OSStart(&err);
}
```

Note: There is no need to define every hook function; you can define only the ones that are placed in your application code.

If you do not intend to extend the kernel through these application hooks, you can set OS_CFG_APP_HOOKS_EN to DEF_DISABLED to save RAM (i.e., the pointers).

Kernel Troubleshooting

Kernel Troubleshooting

- General Considerations
- My Application Deadlocks
- My Application Behaves Erratically
- An Important Task in My Application Does Not Run
- My Application Is Not Responsive

General Considerations

While developing your application, care must be taken to ensure that every call to a service that pends has a call that posts. Furthermore, the OSTaskSuspend() and OSTaskResume() functions and the OSSchedLock() and OSSchedUnlock() functions should be called in pairs. Indeed, the suspend and lock levels can be nested, and there should be an equal number of calls to both function pairs.

The Kernel offers multiple services that allow you to control the access to shared resources. As a general rule, if the resource can be shared by more than one task at a time, use a semaphore. However, if the resource can be used by only one task at a time, use a mutex. The mutex in the Kernel is implemented in such a way as to prevent priority inversion by using the priority ceiling protocol.

My Application Deadlocks

Make sure that at any one point in time, at least one of the four rules in the deadlocks section of Resource Management is false.

My Application Behaves Erratically

Sometimes, tasks can overflow their stack space if the required size for the stack was underestimated when the task was first created.

This situation can go undetected if the stack overflows into an unused region of memory. But that region can sometimes be used by other parts of the code, which can cause the application to fail. Worse, a overflowed stack can cause an application failure at a seemingly random point in time, which complicates the debugging effort. In those cases, the general behavior of your application would appear erratic.

To verify if one of your tasks is behaving badly, you can use the Redzone Stack Checker, which can be enabled in the compile-time configuration . Another option (also in the compile-time configuration) is to enable both the Statistics task and the Stack checker, which monitor the stack usage of the tasks. When you find that a task is overflowing its stack, you should adjust the stack size so that it does not overflow. Review the stack section of the Tasks page for more information.

An Important Task in My Application Does Not Run

If you feel that a task that should run more frequently than it is doing, verify the effective task priorities and ensure that there is not a priority inversion caused by the use of a semaphore. First, correct any priority inversion issues by using mutexes instead of semaphores. Then, verify the assigned task priorities. A good starting point in assigning task priorities can be found on the Tasks page.

My Application Is Not Responsive

This may be a sign that your application is blocking external stimuli for an extended period of time. If your application has long critical sections, the microcontroller you are using may be executing the various ISRs with a high latency. This would in turn cause your tasks that depend on external events to be executed with an even longer latency.

File System

File System

Micrium OS File System is a complete file system stack that includes:

- A virtual file system (VFS) with support for various file system formats (though only FAT is currently supported)
- A raw block device interface
- A set of drivers allowing for the support of various media types including NOR and NAND flash memories, SD cards, eMCC memories and MSC devices

Like the rest of the Micrium OS ecosystem, Micrium OS File System is targeted at embedded devices with severe resource constraints. But many of its features, such as fine-grained locking and scalable ordered write-back cache, have been designed with more capable application processors in mind. Moreover, the addition of a journaling module to the FAT file system sub-module allows for fully interoperable data storage without jeopardizing on-disk integrity and overall safety.

File System Overview

File System Overview

- Specifications
  - Core
  - Drivers
- Features
  - Core
  - Drivers
- Limitations

Specifications

Core

- Compatible with Windows FAT. All standard FAT variants and features are supported, including FAT12/FAT16/FAT32.
- Supports the Unicode standard with UTF-8 encoding for FAT Long File Name (LFN) also known as Virtual FAT (VFAT).
- Standard POSIX-like API is provided. Micrium OS File System implements a subset of the functions defined in the POSIX.1-2008 standard.

Drivers

SD

- Complies with the "Physical Layer Simplified Specification, Version 2.00" published by the SD Association.
- Complies with the "MultiMediaCard (MMC) Electrical Standard, High Capacity" (JESD84-B42) published by the JEDEC Solid State Technology Association.

NAND

- Complies with the "Open NAND Flash Interface Specification, Revision 3.0, 9-March-2011" published by ONFI industry working group.

Features

- Scalable to contain only required features and minimize memory footprint.
- Support for read-only API.

Core

- FAT Long File Name (LFN).
- Optional FAT journaling module that provides total power fail-safety to the FAT system driver.
- DOS partitions are supported, so more than one volume can be located on a device.
- Scalable ordered write-back cache module. Possibility to assign a cache instance to a specific media or a media group permitting better RAM usage.
- Shell support for file system-oriented commands such as fs_cat, fs_cd, fs_lsblk, fs_ls, fs_mkdir, fs_pwd, fs_rm, etc.
- File buffers to increase file access speeds.
- File lock to allow multi-threaded applications.

Drivers

- Storage layer decoupled from the Core layer to allow raw accesses to media. The Storage layer can be compiled without the File System Core layer, allowing a possible integration with third-party file systems. Micrium OS USB Device MSC class interfaces to the File System Storage layer enabling the creation of USB MSC devices.
- SCSI media driver (previously known as MSC driver requiring a USB-Host stack). SCSI driver is agnostic of the lower transport layer. It interfaces to Micrium OS USB Host. It could interface to other transport stacks supporting SCSI protocol.
- Media polling task in the Storage layer to allow removable media (for instance SD cards and SCSI devices) insertion/removal detection. Callbacks are available to notify application upon removable media detection.
- Some media drivers (for instance, NOR and NAND drivers) allows a single volume to span several (typically identical) devices, such as a bank of flash chips.
- Support for parallel and serial NOR flash devices.
- NAND flash media driver has the following features:
  - Dynamic wear-leveling: using logical block addressing, the driver is able to change the physical location of written data on the NAND flash, so that a single memory location does not wear early while other locations are not used.
  - Error Correction Codes (ECC) management: error correction codes are used to eliminate the bit read errors typical to NAND flash. The NAND flash driver offers a software ECC based on the Hamming code or can use the built-in hardware ECC found on some Micron flash devices.
  - Flexible controller layer: a generic controller driver that is compatible with most parallel NAND flash devices and microcontrollers is provided.
  - Fail-safe to unexpected power-loss: the NAND flash driver was designed so that write transactions are atomic. After an unexpected power-down, the NAND flash's low-level format will still be consistent, the device will be remounted as if the transaction never occurred.

Limitations

- No support for extended partition creation. Thus only 4 partitions can be created on a given block device. Micrium OS File System is anyhow able to read extended partitions.
- No support for serial NAND flash devices.
- High capacity MMCPlus cards (> 2GB) are not supported by the File System SD SPI driver.
- MMC (eMMC) v4.3 and superior dropped support for the SPI interface and as such cannot be used with the File System SD SPI driver. SPI support is optional for MMCplus and MMCmobile cards.

File System Basic Concepts

File System Basic Concepts

This section introduces important architectural aspects of Micrium OS File System and defines the terminology that will be used throughout the rest of the documentation. Although most of the terms and concepts presented here are widely used in the literature in general, their exact meaning and usage are specific to Micrium OS File System. So even if you are familiar with file systems, we suggest you at least skim through the following content before you jump to the programming guide and start actual coding.

- Top-Level Architecture
- Object Handles
- File System Objects
  - Core Objects
    - File and Directory Nodes
    - File and Directory Descriptors
    - Working Directories
    - Volumes
  - Relation Between Core Object Hierarchy and On-disk Layout
  - Storage Objects
    - Block Devices
    - Media

Top-Level Architecture

Figure - File System Top-Level Architecture in the *File System Basic Concepts* page shows a schematic view of the file system's main sub-modules and objects, as well as their relation to the application. Micrium OS File System consists of two main sub-modules: core and storage.

The core sub-module has two parts: The first is a **virtual file system (VFS)**, which is responsible for providing the application with a uniform (i.e., file system driver-agnostic) interface to file, directory, volume and working directory manipulation. The second is a set of **file system drivers** which are responsible for maintaining consistent on-disk metadata according to specific (and most often standardized) layouts (e.g., FAT32).

The storage sub-module is also divided into two parts: The first is a **generic storage layer,** which is responsible for providing the application (and the core) with a uniform (i.e., media-agnostic) interface to block device and media manipulation. The second is a set of **storage drivers**, which are responsible for emulating the behavior of a block device when needed (this is the case of the NOR and NAND flash memories), and implementing the specifics of performing media I/O.

Figure - File System Top-Level Architecture

Object Handles

The application can interact with specific file system objects (files, directories, block devices, etc) through the use of object handles. Object handles are obtained through dedicated open() functions and released through corresponding close() functions. A file handle, for instance, is obtained through a call to FSFile_Open() and is released through a call to FSFile_Close(). Similarly, a block device handle is obtained through a call to FSBlkDev_Open() and is released through a call to FSBlkDev_Close(). The media object is slightly different in that it is instantiated internally by the storage sub-module itself. A media handle can be retrieved in a few different ways, as explained in Media .

After an object is closed, the corresponding handle cannot be reused. An error is returned whenever the application attempts to reuse an old handle, which effectively prevents any unwanted access to a stale object.

File System Objects

Although it is not necessary to understand the details of Micrium OS FS internal implementation to use it, it is nonetheless useful to have at least a schematic mental representation of how common file system concepts such as files, directories, volumes are represented internally. This section describes the main Micrium OS FS objects and their relation to the on-disk file system concepts.

Core Objects

File and Directory Nodes

File and directory nodes are the in-RAM representation of on-disk files and directories. They basically serve as small distributed caches for on-disk metadata such as content start position, file size and other file and directory attributes. A single instance of a node exists for each opened file or directory (no matter how many times the file or directory is opened). A node is created when a file or directory is opened for the first time and destroyed when the last associated descriptor is destroyed.

File and Directory Descriptors

Unlike file and directory nodes, file and directory descriptors do not have corresponding on-disk data structures. This is because nodes represent the state of actual on-disk entries, whereas descriptors represent the state of read and write operations on those entries. Each time a file or directory is opened, a new descriptor is created, and a handle to this descriptor is returned. This handle can then be used to perform various read and write operations. The same file or directory can be opened many times, returning as many handles to the application. Since each descriptor carries its own internal operation state, many "threads" of read and write operations may be performed concurrently without interfering with each other.

Working Directories

Working directory objects are lightweight objects containing an on-disk directory position. They can be used as the first argument of many file-level APIs to specify the base location from where a new file or directory can be referenced.

Volumes

Volume objects are the in-RAM representation of on-disk partitions. In contrast to files and directories, a volume can be opened only once (using FSVol_Open()). Upon opening, the newly created volume is given a name, which can be later used as part of an entry path.

Relation Between Core Object Hierarchy and On-disk Layout

Consider the file system on-disk layout shown in Figure - Example of a File System On-Disk Layout in the *File System Basic Concepts* page. There are two partitions (possibly formatted using a different file system), each containing its own file and directory tree. Now consider the object hierarchy shown in Figure - Example of a File System Core Object Hierarchy in the *File System Basic Concepts* page, which is created after performing the following operations:

- Open (mount) partition 1 as "vol1"

- Open (mount) partition 2 as "vol2"
- Open "data" directory
- Open "archive" as a working directory
- Open "sys.log" file
- Open "sys_00.log" file
- Open "sys.log" file again

You can see that the directory "log", the file "user.db" and the file "sys_01.log", which are present on-disk, do not have corresponding in-RAM file or directory nodes. They have not been opened. All other files and directories have nodes in-RAM. Also, you can notice that the "sys.log" file has been opened twice resulting in two different file descriptors being created and two distinct file handles being returned to the application.

Figure - Example of a File System On-Disk Layout

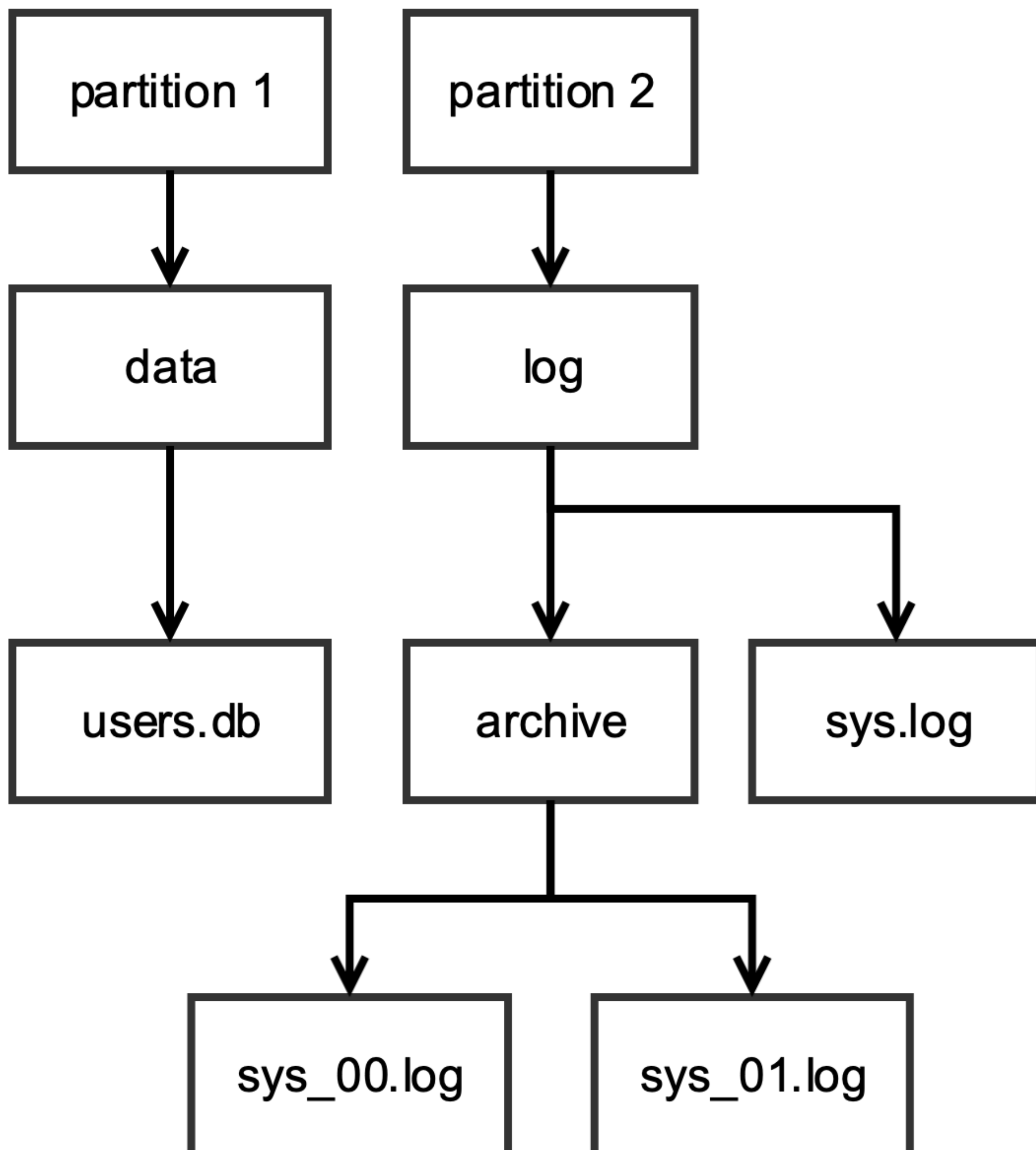
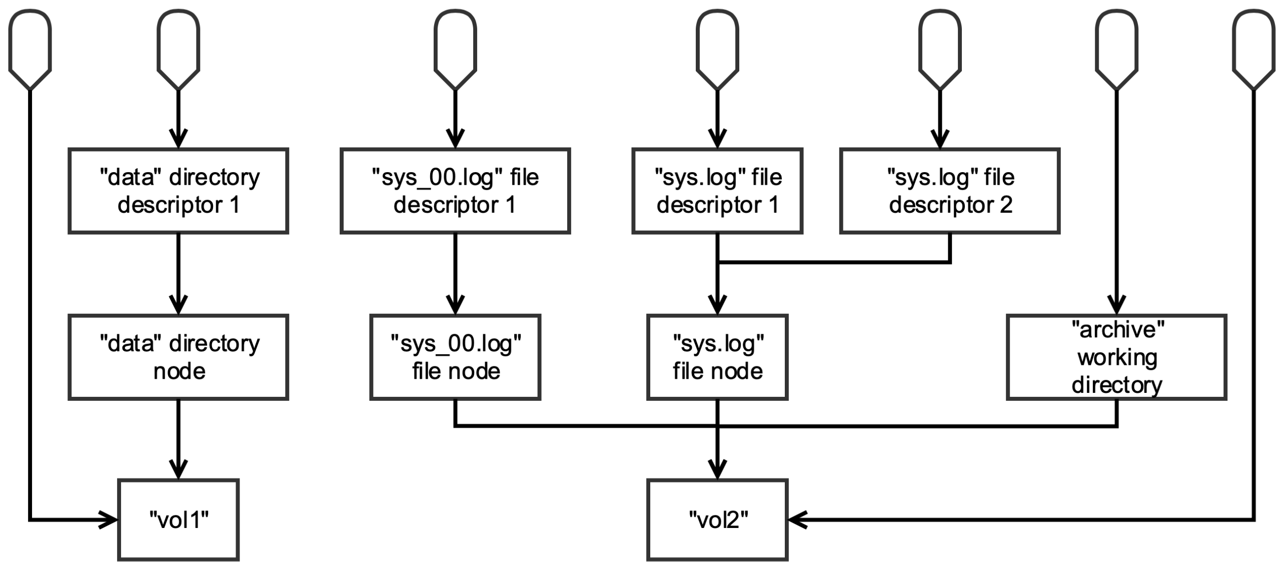


Figure - Example of a File System Core Object Hierarchy



Storage Objects

Block Devices

Block device objects represent storage devices whose smallest addressable unit is a block of a fixed and uniform size (typically 512, 1024, 2048 or 4096 bytes). Physical media may or may not exhibit such a property. For instance, NAND flash memories are not typically made of continuous blocks of uniform size that can be addressed individually: a special translation layer (called flash translation layer (FTL)) must be used in order to make the NAND flash appear like a block device to the upper layer (either the file system core or the application).

A block device is opened on top of an existing media and can be opened only once. A block device cannot exist without its underlying media, but a media may be used on its own, without a corresponding block device being opened on it.

Media

Media objects represent physical media. A media can be categorized according to two distinct criteria: its persistence and its ability to be disconnected and reconnected. Following these criteria a media can be:

1. Persistent and fixed (or non-removable): the media's existence begins as soon as the storage initialization ends, and it cannot be disconnected.
2. Persistent and removable: the media's existence begins as soon as the storage initialization ends, and it can be disconnected and reconnected anytime.
3. Non-persistent (and thus removable): the media can appear and vanish anytime.

Table - Possible media states per media type

Media type	Category
RAM disk	persistent / fixed
NOR	persistent / fixed
NAND	persistent / fixed
eMMC	persistent / fixed
SD card	persistent / removable
SCSI logical unit (USB MSC device)	non-persistent / removable

Unlike other file system objects, media are not opened explicitly, but rather instantiated internally by the storage sub-module. Since there are no such functions as FSMedia_Open() and FSMedia_Close(), media handles cannot be retrieved in the same way as the other file system object handles. Instead, media handles are retrieved using FSMedia_Get(), which takes the media name as its unique parameter and returns the corresponding media handle. The media name is statically assigned to a given media in the BSP_OS_Init() for NAND, NOR and SD when using the memory controller register macros (refer to Table - Memory Controller Register Macros in the *File System Memory Controller Registration to the Platform Manager* page for more details about the macros). For RAM disks, the media name is assigned by the application when calling FS_RAM_Disk_Add(). For SCSI, a name is internally assigned to each newly connected SCSI device (refer to section Automatic Media Naming for more details).

Integrating File System Into Your Project

Integrating File System Into Your Project

Micrium OS File System contains several components, each of which implements specific functions. The core of the File System module is named "File System." This component is mandatory and must always be part of your project. The File System module also includes one component for each storage driver. To use File System, you must add these files to your project and populate your RTOS description file .

Starting the File System Module Quickly

Micrium offers a set of example applications that demonstrate some of the features of the Micrium OS File System to help you start the development of your application. We highly recommend that you start from one of these examples.

The section File System Example Applications describes each example application.

Configuring File System

Micrium OS File System can be configured to optimize memory usage or features. The page File System Compile-Time Configuration explains how the File System Core and Storage layers can be configured at compile-time. The page File System Run-Time Configuration explains how the File System core and storage can be configured at run-time.

File System Example Applications

File System Example Applications

This section describes the examples that are related to the Micrium OS File System stack.

- File System Module Initialization Example
  - Description
  - Configuration
  - Location
  - API
- File Read/Write Example
  - Description
  - Location
  - API
- File Read/Write with Posix API Example
  - Description
  - Location
  - API
- File Multi-Descriptors Example
  - Description
  - Location
  - API
- Entry Path Example
  - Description
  - Configuration
  - Location
  - API
- Block Device Read/Write Example
  - Description
  - Location
  - API
- Media Polling Example
  - Description
  - Location
  - API

File System Module Initialization Example

Description

This example shows how to initialize the File System stack. It will allow you to accomplish the following tasks:

- Optionally configure the media polling callbacks
- Initialize the Storage layer
- Initialize the Core layer
- Optionally Create a RAM Disk region
- Optionally low-level format NAND or NOR media
- Create a cache instance
- Optionally format the RAM Disk region as a FAT volume
- Optionally initialize the Media Polling example
- Optionally high-level format any media (that is NAND, NOR or SD) as a FAT volume

Configuration

EX_CFG_FS_ACTIVE_MEDIA_NAME

The file ex_fs.h offers a global configuration that allows you to choose the media on which any File System examples will run: EX_CFG_FS_ACTIVE_MEDIA_NAME.

The media name must have the following format: "<media-name><number>". The configuration can have the following values:

- "nand0" for NAND
- "nor0" for NOR
- "ram0" for RAM Disk
- "sd0" for SD Card or SPI

The media name could be any name, in fact. But the File System examples requires that the defined media name matches the one defined in the BSP when the media is registered. Most of the time the media name used in the BSP will be "<media-name>0".

> **For SCSI devices, you do not need to specify for instance "scsi0". SCSI devices are used only by the Media Polling example and do not require a media name defined. The media name for SCSI device is automatically managed by the Media Polling example.**

The default configuration value depends on the media that are available in rtos_description.h. EX_CFG_FS_ACTIVE_MEDIA_NAME default value is determined as shown in listing Listing - EX CFG FS ACTIVE MEDIA NAME Default Configuration Value in the *File System Example Applications* page.

Listing - EX CFG FS ACTIVE MEDIA NAME Default Configuration Value

```
#ifndef  EX_CFG_FS_ACTIVE_MEDIA_NAME                         (1)
#if (defined(RTOS_MODULE_FS_STORAGE_RAM_DISK_AVAIL))
#define  EX_CFG_FS_ACTIVE_MEDIA_NAME           "ram0"
#elif (defined(RTOS_MODULE_FS_STORAGE_NAND_AVAIL))
#define  EX_CFG_FS_ACTIVE_MEDIA_NAME           "nand0"
#elif (defined(RTOS_MODULE_FS_STORAGE_NOR_AVAIL))
#define  EX_CFG_FS_ACTIVE_MEDIA_NAME           "nor0"
#elif (defined(RTOS_MODULE_FS_STORAGE_SD_CARD_AVAIL) || defined(RTOS_MODULE_FS_STORAGE_SD_SPI_AVAIL))
#define  EX_CFG_FS_ACTIVE_MEDIA_NAME           "sd0"
#endif
#endif
```

(1) If you want to run the File System examples on a specific media, you can define EX_CFG_FS_ACTIVE_MEDIA_NAME before the #ifndef to overwrite the default value. For instance:

```
#define  EX_CFG_FS_ACTIVE_MEDIA_NAME             "nand0"
```

EX_CFG_FS_MEDIA_LOW_LEVEL_FMT_EN

The file ex_fs.c offers a local configuration allowing your application to low-level format a NAND or NOR chip: EX_CFG_FS_MEDIA_LOW_LEVEL_FMT_EN.

If you have a blank NAND or NOR chip, you must low-level format the memory chip prior to accessing the media by the high-level file system API. You may want to low-level format explicitly even if the NAND or NOR chip is already formatted. After a low-level format, you must format the NAND or NOR chip with a high-level format; for instance, high-level formatting as a FAT volume. High-level formatting is controlled by the configuration EX_CFG_FS_MEDIA_HIGH_LEVEL_FMT_EN. Low-level formating does not apply to RAM, SD and SCSI devices.

The configuration can have the following values: DEF_DISABLED or DEF_ENABLED. By default, this configuration is set to DEF_DISABLED.

EX_CFG_FS_MEDIA_HIGH_LEVEL_FMT_EN

The file ex_fs.c offers a local configuration allowing your application to high-level format any media (NAND, NOR, RAM Disk, SD): EX_CFG_FS_MEDIA_HIGH_LEVEL_FMT_EN.

When enabled, the high-level format will apply to the unique partition that composes your media. If a NAND or NOR chip has been low-level formatted, the high-level format is mandatory.

The configuration can have the following values: DEF_DISABLED or DEF_ENABLED. By default, this configuration is set to DEF_DISABLED.

Location

The example implementation is located in:

- /examples/fs/ex_fs.c
- /examples/fs/ex_fs.h
- /examples/fs/ex_fs_utils.h

API

This example offers only one API named Ex_FS_Init(). This function performs the different example steps mentioned in the section Description . The function must be called by your application task prior to calling any other file system File System examples.

File Read/Write Example

Description

This example shows how to perform a file write and a file read on a given media. This example will allow you to accomplish the following tasks:

- Set file position at the beginning of the file
- Write some known data to the specified file
- Set again file position at the beginning of the file
- Read data from the specified file
- Verify the read data integrity

Location

The example implementation is located in:

- /examples/fs/ex_fs_file_rd_wr.c
- /examples/fs/ex_fs_file_rd_wr.h.

The following files are also required:

- /examples/fs/ex_fs.c
- /examples/fs/ex_fs.h
- /examples/fs/ex_fs_utils.h.

API

This example offers only one API named Ex_FS_FileRdWr(). This function performs the different example steps mentioned in the section Description . The function can be called by your application task by including the associated header file ex_fs_file_rd_wr.h.

File Read/Write with Posix API Example

Description

This example shows how to perform a file write and a file read on a given media using the Posix API. This example will allow you to accomplish the following tasks:

- Set file position at the beginning of the file
- Write some known data to the specified file
- Set again file position at the beginning of the file
- Read data from the specified file
- Verify the read data integrity

Location

The example implementation is located in:

- /examples/fs/ex_fs_file_rd_wr_posix.c
- /examples/fs/ex_fs_file_rd_wr_posix.h

The following files are also required:

- /examples/fs/ex_fs.c
- /examples/fs/ex_fs.h
- /examples/fs/ex_fs_utils.h

API

This example offers only one API named Ex_FS_FileRdWr_Posix(). This function performs the different example steps mentioned in the section Description . The function can be called by your application task by including the associated header file ex_fs_file_rd_wr_posix.h.

File Multi-Descriptors Example

Description

This example shows how to open the same file with two different file descriptors: one file descriptor has a write access and the other as a read-only access. This example will allow you to accomplish the following tasks:

- Create a Reader task
- Writer task writes N logical blocks with a known data pattern to the specified file. The Writer task is represented by the function Ex_FS_FileMultiDesc() called from your application task
- Writer task writes a few logical blocks to the file and starts the Reader task
- Reader task reads N logical blocks from the specified file and verifies the read data integrity

Location

The example implementation is located in:

- /examples/fs/ex_fs_file_multi_desc.c
- /examples/fs/ex_fs_file_multi_desc.h

The following files are also required:

- /examples/fs/ex_fs.c
- /examples/fs/ex_fs.h
- /examples/fs/ex_fs_utils.h

API

This example offers only one API named Ex_FS_FileMultiDesc(). This function performs the different example steps mentioned in the section Description . The function can be called by your application task by including the associated header file ex_fs_file_multi_desc.h.

Entry Path Example

Description

This example shows how to open directories and working directories and to use them to navigate a folder tree structure. This example will allow you to accomplish the following tasks:

- Create a predefined directory/file tree structure
- Read each entry (sub-directory or file) of each directory of the predefined tree structure using absolute paths
- Create a working directory associated with a specific directory of the predefined tree structure
- Read each entry in the directory pointed to by the working directory
- Create a working directory pointing to the virtual root directory
- Read each entry of virtual root directory

When using working directories, any path specified with a working directory handle is considered as relative. This is in contrast to absolute paths, which use a NULL working directory handle.

A virtual root directory refers to the media name level. Let's assume for instance the following generic path to a directory:

"nand0/<vol-name>/<root-dir-name>/<sub-dir-name>/<sub-dir-name>/<sub-dir-name>/"

The virtual root directory is "nand0". Thus when reading the virtual directory with FSDir_Rd() , the entry obtained will be a volume name. The volume must have been open with FSVol_Open() prior to reading the virtual root directory. Otherwise no entry is available from the virtual root directory. If several volumes have been open on the same media because it has multiple partitions, several calls to FSDir_Rd() will return all the open volumes.

Configuration

EX_CFG_FS_ENTRY_PATH_MORE_INFO_EN

The file ex_fs_entry_path.c offers a local configuration, allowing more messages to be displayed when executing the example: EX_CFG_FS_ENTRY_PATH_MORE_INFO_EN.

When enabled, the full entry path will be displayed for each directory entry read with FSDir_Rd() . If the media read contains plenty of files and folders in the root directory, the output console displaying messages may be flooded.

The configuration can have the following values: DEF_DISABLED or DEF_ENABLED. By default, this configuration is set to DEF_DISABLED to keep the number of messages to a minimum.

Location

The example implementation is located in:

- /examples/fs/ex_fs_entry_path.c
- /examples/fs/ex_fs_entry_path.h

The following files are also required:

- /examples/fs/ex_fs.c
- /examples/fs/ex_fs.h
- /examples/fs/ex_fs_utils.h

API

This example offers only one API named Ex_FS_EntryPath(). This function performs the different example steps mentioned in the section Description . The function can be called by your application task by including the associated header file ex_fs_entry_path.h.

Block Device Read/Write Example

Description

This example shows how to perform raw media accesses using the Block Device API. This example does not pass through the Core layer; it goes directly via the Storage layer. It will allow you to accomplish the following tasks:

- Write some known data in the N first sectors
- Read data back from the N first sectors
- Verify the read data integrity

> You should run this example with extra care. The Storage layer is *not* aware of the file system formatting on the media. The example writes in the first media sectors. Thus any file system formatting information contained in some sectors may be corrupted. The directories and files may also be corrupted. If the media was formatted and the example is executed, you must re-format the media so that other examples (for instance file read/write, file multi-descriptors, etc.) can be run. Refer to section Configuration for more details about low-level and high-level formats in case of media reformatting.

Location

The example implementation is located in:

- /examples/fs/ex_fs_blk_dev_rd_wr.c
- /examples/fs/ex_fs_blk_dev_rd_wr.h

The following files are also required:

- /examples/fs/ex_fs.c
- /examples/fs/ex_fs.h
- /examples/fs/ex_fs_utils.h

API

This example offers only one API named Ex_FS_BlkDevRdWr(). This function performs the different example steps mentioned in the section Description . The function can be called by your application task by including the associated header file ex_fs_blk_dev_rd_wr.h.

Media Polling Example

Description

This example shows how removable media are detected by the Media Poll task (part of the Storage layer), and how an application connection or disconnection callback is called. This example will allow you to accomplish the following tasks:

- Display information about a removable media upon its connection
- Notify an application task about the removable media connection. The task will:
  - Open a block device and a FAT volume on the removable media
  - Perform once the File Read/Write example. The steps of the File Read/Write example are described in section Description
  - Close the volume and the block device
- Display information about the removable media upon its disconnection

The Media Polling example defines the following application callbacks:

- Ex_FS_MediaPollOnConn() called by the File System stack upon connection of a removable media
- Ex_FS_MediaPollOnDisonn() called by the File System stack upon disconnection of a removable media

This example targets the SD cards and SCSI devices considered as removable devices. Note that if some fixed media such NAND, NOR and RAM are available, the connection callback will be called only once for those devices. The disconnection callback will never be called for fixed media. When the connection callback is called, some useful information about SD or SCSI devices is displayed.

This example uses the asynchronous handling of the removable media connection/disconnection. You can refer to the page Managing Removable Media for more details about removable media management within Micrium OS File System.

Location

The example implementation is located in:

- /examples/fs/ex_fs_media_poll.c
- /examples/fs/ex_fs_media_poll.h

The following files are also required:

- /examples/fs/ex_fs.c
- /examples/fs/ex_fs.h
- /examples/fs/ex_fs_utils.h

/examples/fs/ex_fs_file_rd_wr.c
- /examples/fs/ex_fs_file_rd_wr.h

API

This example does not offer any public API. You don't have to call the example from your application task. When calling the function Ex_FS_Init() corresponding to the File System Module Initialization example , some pieces of code will help initialize the Media Polling example. From that point on, the example is independent of any application function, as opposed to the other examples. Thanks to the Media Poll task, the application connection/disconnection callbacks are called appropriately, and the File Read/Write example is executed on the connected media. The Media Poll task is activated via FS_STORAGE_CFG_MEDIA_POLL_TASK_EN defined in fs_storage_cfg.h. If the Media Poll task is disabled, the application connection/disconnection callbacks will still be called when a SCSI device connects. Refer to section SCSI for more details about why SCSI does not need the Media Poll task to work.

> If SCSI is available, the Micrium OS USB Host stack and the MSC class must also be present. In the file rtos_description.h, the following #defines must be defined: RTOS_MODULE_USB_HOST_AVAIL and RTOS_MODULE_USB_HOST_MSC_AVAIL. The SCSI storage driver works with a Transport layer that transfer SCSI commands to the device. The recommended Transport layer for File System SCSI is the USB Host MSC class.

File System Configuration

File System Configuration

To configure the Micrium OS File System, there are two groups of configuration parameters:

- File System Compile-Time Configuration
- File System Run-Time Configuration

This section explains how to setup these configuration groups.

File System Compile-Time Configuration

- Core Configuration
  - Generic Options
- Storage Configuration
  - Generic Options
  - SD-Specific Options

The Micrium OS File System is configurable at compile time through a set of #defines located in fs_core_cfg.h and fs_storage_cfg.h. Through these #defines, the Micrium OS File System feature set can be tailored to match your application's requirements while keeping ROM and RAM footprints as low as possible.

We recommend that you begin the configuration process with the default configuration values, which are shown in **bold** in the next sections.

The sections below are organized following the order in Micrium OS File System template configuration files, fs_core_cfg.h and fs_storage_cfg.h.

Core Configuration

The Core layer options are found in the file fs_core_cfg.h.

Generic Options

Table - Generic Options

Constant	Description	Possible values
FS_CORE_CFG_FAT_EN	Enables or disables support for the FAT file system driver.Since FAT is the only file system driver currently supported, this option must be enabled.	DEF_ENABLED or DEF_DISABLED
FS_CORE_CFG_POSIX_EN	Enables or disables support for the POSIX compatibility layer.	DEF_ENABLED or DEF_DISABLED
FS_CORE_CFG_DIR_EN	Enables or disables support for directories. If this option is disabled, only files residing in a volume's root directory can be accessed. Furthermore, no entries can be created outside of a volume's root directory. All code paths dealing with directories are thus excluded.	DEF_ENABLED or DEF_DISABLED
FS_CORE_CFG_FILE_BUF_EN	Enables or disables support for file buffers. Disabling this option removes all code paths and data structures related to file buffers handling.This option must be enabled when FS_CORE_CFG_POSIX_EN is enabled.	DEF_ENABLED or DEF_DISABLED

Constant	Description	Possible values
FS_CORE_CFG_FILE_LOCK_EN	Enables or disables support for file locks.	DEF_ENABLED or DEF_DISABLED
FS_CORE_CFG_PARTITION_EN	Enables or disables support for partitions. If this option is disabled, only the first partition can be accessed and partition table creation in the Master Boot Record (MBR) is disabled.	DEF_ENABLED or DEF_DISABLED
FS_CORE_CFG_TASK_WORKING_DIR_EN	Enables or disables support for task bound working directories.This feature can be enabled only if the kernel abstraction layer implementation (KAL) being used provides task registers (see the subsection Task Registers in the Task Management Kernel Services section).	DEF_ENABLED or DEF_DISABLED
FS_CORE_CFG_UTF8_EN	Enables or disables support for entry names containing UTF-8 characters. If this option is disabled, only ASCII characters may be used in an entry name.	DEF_ENABLED or DEF_DISABLED
FS_CORE_CFG_THREAD_SAFETY_EN	Enables or disables thread safety at the core level. When disabled, this option may result in improved CPU time, RAM and ROM usage.It is not safe to disable this option when the file system is accessed by concurrent tasks. When in doubt, leave this option to its default value (DEF_ENABLED).	DEF_ENABLED or DEF_DISABLED
FS_CORE_CFG_ORDERED_WR_EN	Enables or disables ordered write operations. If this option is enabled, write operations will be performed on the underlying media in a known order (typically one that does not jeopardize file system integrity in case of an untimely power failure). This option can be safely disabled when caches being used have only one block.It is not safe to disable this option when at least one cache instance has more than one block. When in doubt, leave this option to its default value (DEF_ENABLED).	DEF_ENABLED or DEF_DISABLED
FS_CORE_CFG_FILE_COPY_EN	Enables or disables file copy. If this option is enabled, a dedicated API is provided for copying files. While it is possible to perform a copy using File System Compile-Time Configuration and File System Compile-Time Configuration API functions, this approach requires more RAM: one buffer for reading / writing and at least one cache block, as opposed to only one cache block in the case of File System Compile-Time Configuration.	DEF_ENABLED or DEF_DISABLED
FS_CORE_CFG_RD_ONLY_EN	Enables or disables the read-only mode. If this option is enabled, all write operations are disallowed, and all the associated code paths and data structures are excluded.	DEF_ENABLED or DEF_DISABLED
FS_CORE_CFG_POSIX_PUTCHAR	Configures the putchar()-like callback that is used internally by the perror() function to output an error message. The provided function must have the following prototype: int my_putchar_func(int c). If FS_CORE_CFG_POSIX_PUTCHAR is not defined, the perror() function will not be available.	Any putchar()-like function (default is **putchar**)
FS_CORE_CFG_MAX_VOL_NAME_LEN	Configures the maximum number of characters contained in a volume name (e.g., the name given to File System Compile-Time Configuration) not including the terminating null character.	Must be > 1 (default is **20**)
FS_CORE_CFG_DBG_MEM_CLR_EN	Enables or disables the memory clear debug feature.	DEF_ENABLED or DEF_DISABLED

Constant	Description	Possible values
FS_CORE_CFG_CTR_STAT_EN	Enables or disables core statistics counters.	DEF_ENABLED or DEF_DISABLED
FS_CORE_CFG_CTR_ERR_EN	Enables or disables core error counters.	DEF_ENABLED or DEF_DISABLED

FAT-Specific Options

Table - FAT-Specific Options

Constant	Description	Possible values
FS_FAT_CFG_LFN_EN	Enables or disables the long file name support. If this option is disabled, LFN entries cannot be accessed nor created.	DEF_ENABLED or DEF_DISABLED
FS_FAT_CFG_FAT12_EN	Enables or disables the FAT12 file system driver.	DEF_ENABLED or DEF_DISABLED
FS_FAT_CFG_FAT16_EN	Enables or disables the FAT16 file system driver.	DEF_ENABLED or DEF_DISABLED
FS_FAT_CFG_FAT32_EN	Enables or disables the FAT32 file system driver.	DEF_ENABLED or DEF_DISABLED
FS_FAT_CFG_JOURNAL_EN	Enables or disables the FAT journaling module.	DEF_ENABLED or DEF_DISABLED

Storage Configuration

The Storage layer options are found in the file fs_storage_cfg.h.

Generic Options

Table - Generic Options

Constant	Description	Possible values
FS_STORAGE_CFG_DBG_WR_VERIFY_EN	Enables or disables the written data check. If this option is enabled, a write access to the media is verified by reading back the written data integrity by performing a CRC check.	DEF_ENABLED or DEF_DISABLED
FS_STORAGE_CFG_CTR_STAT_EN	Enables or disables storage statistics counters.	DEF_ENABLED or DEF_DISABLED
FS_STORAGE_CFG_CTR_ERR_EN	Enables or disables storage error counters.	DEF_ENABLED or DEF_DISABLED
FS_STORAGE_CFG_MEDIA_POLL_TASK_EN	Enables or disables media polling task.	DEF_ENABLED or DEF_DISABLED
FS_STORAGE_CFG_RD_ONLY_EN	Enables or disables the read-only mode. If this option is enabled, all write operations are disallowed, and all the associated code paths and data structures are excluded.	DEF_ENABLED or DEF_DISABLED
FS_STORAGE_CFG_CRC_OPTIMIZE_ASM_EN	Enables or disables CRC calculation assembler optimization.	DEF_ENABLED or DEF_DISABLED

NAND-Specific Options

Table - NAND-Specific Options

Constant	Description	Possible values
FS_NAND_CFG_AUTO_SYNC_EN	Determines, for each operation on the device (i.e., each call to the device's API), if the metadata should be synchronized. Synchronizing at the end of each operation is safer; it ensures the device can be remounted and appear exactly as it should. Disabling automatic synchronization will result in a large write speed increase, as the metadata won't be committed automatically, unless done in the application. If a power down occurs between a device operation and a sync operation, the device will appear as it was in a prior state when remounted. Device synchronization can be forced with a call to FSBlkDev_Sync().Note that using large write buffers will reduce the metadata synchronization performance hit, as fewer calls to the device API will be needed.	DEF_ENABLED or DEF_DISABLED
FS_NAND_CFG_UB_META_CACHE_EN	Determines, for each update block, if the metadata will be cached. Enabling this will allow searching for a specific updated sector through data in RAM instead of accessing the device, which would require additional read page operations. More RAM will be consumed if this option is enabled, but write/read speed will be improved. RAM usage = (<Nbr update blks> x (log2(<Max associativity>) + log2(<Nbr secs per blk>)) / 8) octets (rounded up).	DEF_ENABLED or DEF_DISABLED
FS_NAND_CFG_DIRTY_MAP_CACHE_EN	Determines if the dirty blocks map will be cached. With this feature enabled, a copy of the dirty blocks map on the device is cached. It is possible then to determine if the state "dirty" of a block is committed on the device without the need to actually read the device.With this feature enabled, overall write and read speed should be improved. Also, robustness will be improved for specific cases. However, more RAM will be consumed. RAM usage = (<Nbr of blks on device> / 8) octets (rounded up)	DEF_ENABLED or DEF_DISABLED
FS_NAND_CFG_UB_TBL_SUBSET_SIZE	Controls the size of the subsets of sectors pointed by each entry of the update block tables. The value must be a power of 2 (or 0). If, for example, the value is 4, each time a specific updated sector is requested, the NAND Flash Translation layer must find the sector from a group of 4 sectors. Thus, if the cache is disabled, 4 sectors must be read from the device. Otherwise, the 4 entries will be searched in the cache. If the value is set to 0, the table will be disabled completely, meaning that all sectors of the block might have to be be read before the specified sector is found. If the value is 1, the table completely specifies the location of the sector, and thus no search must be performed. In that case, enabling the update blocks metadata cache will yield to no performance benefit. RAM usage = (<Nbr update blks> x (log2(Nbr secs per blk>) - log2(<Subset size>) x <Max associativity> / 8) octets (rounded up).	Any power of 2 or 0 (default is 1)

Constant	Description	Possible values
FS_NAND_CFG_RSVD_AVAIL_BLK_CNT	Indicates the number of blocks in the available blocks table that are reserved for metadata block folding. Since this operation is critical and must be done before adding blocks to the available blocks table, the driver needs enough reserved blocks to make sure at least one of them is not bad so that the metadata can be folded successfully. When set to 3, probability for the metadata folding operation to fail is low. This value should be sufficient for most applications.	Must be > 1 (default is **3**)
FS_NAND_CFG_MAX_RD_RETRIES	Indicates the maximum number of retries performed when a read operation fails. It is recommended by most manufacturers to retry reading a page if it fails, as successive read operations might be successful. This number should be at least set to 2 for smooth operation, but might be set higher to improve reliability.	Must be > 2 (default is **10**)
FS_NAND_CFG_MAX_SUB_PCT	Indicates the maximum number of update blocks that can be sequential update blocks (SUB). This value is set as a percentage of the total number of update blocks.	Integer between 0 and 100 (default is **30**)
FS_NAND_CFG_DUMP_SUPPORT_EN	Enables or disables support for the NAND dump facility.	DEF_ENABLED or DEF_DISABLED

NOR-Specific Options

Table - NOR-Specific Options

Constant	Description	Possible values
FS_NOR_CFG_WR_CHK_EN	Enables or disables checking for data writes. When enabled, all NOR operations implying a write operation such as page program or block erase are verified by reading back the data from the NOR chip and comparing it to the data written. When erasing a block, the check ensures that the read data bits are all one because after an erase operation, the bits are all set to one. This configuration is useful during the development phase and should not be left enabled as it impacts the write performances.	DEF_ENABLED or DEF_DISABLED

SD-Specific Options

Table - SD-Specific Options

Constant	Description	Possible values
FS_SD_SPI_CFG_CRC_EN	Enables or disables CRC and checking for data writes and reads. When enabled, a CRC will be generated for data written to the card, and the CRC of received data will be checked. When disabled, no CRC will be generated for data written to the card, and the CRC received data will not be checked.	DEF_ENABLED or DEF_DISABLED

File System Run-Time Configuration

- Core Configuration
  - Optional Pre-Init Configuration
    - Memory Segment
    - Maximum Number of Core Objects
    - Maximum Number of FAT Objects
- Storage Configuration
  - Optional Pre-Init Configuration
    - Memory Segment

Media Connection Callbacks
- Media Poll Task Stack
- Maximum Number of SCSI Logical Units
- Optional Post-Init Configuration
  - Media Poll Task Period
  - Media Poll Task Priority

This section describes the Micrium OS File System run-time configurations. These configurations are optional and can usually be ignored until the very late stages of an application design.

Core Configuration

Optional Pre-Init Configuration

To perform pre-init configuration, you must call the dedicated configuration functions before calling FSCore_Init(). If no explicit pre-init configuration is performed, default values will be used. These default values are stored in FSCore_InitCfgDflt defined in fs_core.c.

Memory Segment

Configures the memory segment where the core internal data structures will be allocated.

Type	Function to call	Default	Field from default configuration structure
MEM_SEG*	FSCore_ConfigureMemSeg()	General-purpose heap	.MemSegPtr

Maximum Number of Core Objects

Configures the maximum number of file system driver-agnostic core objects.

Type	Function to call	Default	Field from default configuration structure
FS_CORE_CFG_MAX_OBJ_CNT	FSCore_ConfigureMaxObjCnt()	Unlimited number of objects	.MaxCoreObjCnt.WrkDirCnt .MaxCoreObjCnt.RootDirDescCnt

Maximum Number of FAT Objects

Configures the maximum number of FAT objects.

Type	Function to call	Default	Field from default configuration structure
FS_CORE_CFG_MAX_FAT_OBJ_CNT	FSCore_ConfigureMaxFatObjCnt()	Unlimited number of objects	.MaxFatObjCnt.FileDescCnt
FS_CORE_CFG_MAX_FAT_OBJ_CNT	FSCore_ConfigureMaxFatObjCnt()	Unlimited number of objects	.MaxFatObjCnt.FileNodeCnt
FS_CORE_CFG_MAX_FAT_OBJ_CNT	FSCore_ConfigureMaxFatObjCnt()	Unlimited number of objects	.MaxFatObjCnt.DirDescCnt
FS_CORE_CFG_MAX_FAT_OBJ_CNT	FSCore_ConfigureMaxFatObjCnt()	Unlimited number of objects	.MaxFatObjCnt.DirNodeCnt
FS_CORE_CFG_MAX_FAT_OBJ_CNT	FSCore_ConfigureMaxFatObjCnt()	Unlimited number of objects	.MaxFatObjCnt.VolCnt

Storage Configuration

Optional Pre-Init Configuration

To perform pre-init configuration, you must call the dedicated configuration functions before calling FSStorage_Init(). If no explicit pre-init configuration is performed, default values will be used. These default values are stored in FSStorage_InitCfgDflt defined in fs_storage.c.

Memory Segment

Configures the memory segment where the storage sub-module's internal data structures will be allocated.

Type	Function to call	Default	Field from default configuration structure
MEM_SEG*	FSStorage_ConfigureMemSeg()	General-purpose heap	.MemSegPtr

Media Connection Callbacks

Configures the media connection and disconnection callbacks.

Type	Function to call	Default	Field from default configuration structure
FS_MEDIA_CONN_CB	FSStorage_ConfigureMediaConnCallback()	DEF_NULL	.OnConn
FS_MEDIA_CONN_CB	FSStorage_ConfigureMediaConnCallback()	DEF_NULL	.Disconn

Media Poll Task Stack

Configures the size and the start address of the Media Poll task stack.

Type	Function to call	Default	Field from default configuration structure
CPU_INT32U	FSStorage_ConfigureMediaPollTaskStk()	A stack of 512 elements allocated on Common 's memory segment	.PollTaskStkSizeElements
void *	FSStorage_ConfigureMediaPollTaskStk()	A stack of 512 elements allocated on Common 's memory segment	.PollTaskStkPtr

Maximum Number of SCSI Logical Units

Configures the maximum number of SCSI logical units that may co-exist at a given time.

Type	Function to call	Default	Field from default configuration structure
CPU_SIZE_T	FSStorage_ConfigureMaxSCSILogicalUnitCnt()	Unlimited	.MaxSCSILuCnt

Optional Post-Init Configuration

This section describes the configurations that can be set at any time during execution after FSStorage_Init() has been called. These configurations are optional. If you do not set them in your application, the default configurations will apply.

Media Poll Task Period

Configures the interval of time (in milliseconds) between two media polls.

Type	Function to call	Default
CPU_INT32U	FSStorage_PollTaskPeriodSet()	500

Media Poll Task Priority

Configures the priority of the Media Poll task.

Type	Function to call	Default
RTOS_TASK_PRIO	FSStorage_PollTaskPrioSet()	See Appendix A - Internal Tasks

File System Run-time Configuration Details for Core

The FS_CORE_CFG_MAX_OBJ_CNT structure allows you to configure the number of objects needed by the Core layer.

Table - FS_CORE_CFG_MAX_OBJ_CNT Configuration Structure in the *File System Run-time Configuration Details for Core* page describes each configuration field available in this configuration structure.

Table - FS_CORE_CFG_MAX_OBJ_CNT Configuration Structure

Field	Description
.WrkDirCnt	The maximum number of working directories. A working directory object is allocated each time a working directory is opened with FSWrkDir_Open().
.RootDirDescCnt	The maximum number of root directory descriptors. A root directory descriptor object is allocated e ach time a directory is opened with FSDir_Open() .

File System Run-time Configuration Details for FAT

The FS_CORE_CFG_MAX_FAT_OBJ_CNT structure allows you to configure the number of objects needed by the FAT sub-module of the Core layer.

Table - FS_CORE_CFG_MAX_FAT_OBJ_CNT Configuration Structure in the *File System Run-time Configuration Details for FAT* page describes each configuration field available in this configuration structure.

Table - FS CORE CFG MAX FAT OBJ CNT Configuration Structure

Field	Description
.FileDescCnt	The maximum number of file descriptors. A file descriptor object is allocated each time a file is opened with FSFile_Open(),
.FileNodeCnt	The maximum number of file nodes. A file node object is allocated for each different file opened with FSFile_Open(). If the same file is opened several times with FSFile_Open(), only one file node is created. Refer to the section Core Objects for more details about file descriptors versus file nodes.
.DirDescCnt	The maximum number of directory descriptors. Each time a directory is opened with FSWrkDir_Open() , a directory descriptor object is allocated.
.DirNodeCnt	The maximum number of directory nodes. A directory node object is allocated for each different directory opened with FSDir_Open(). If the same directory is opened several times with FSDir_Open(), only one directory node is created. Refer to the section Core Objects for more details about directory descriptors versus directory nodes.
.VolCnt	The maximum number of volumes. A volume object is allocated each time a volume is opened with FSVol_Open().

File System Programming Guide

File System Programming Guide

The following sections explain how to use different aspects of the file system, such as handling block devices, managing removable media, creating a cache, etc.

- Setting Up and Opening a Block Device
- Performing Raw Block Device Operations
- Creating and Assigning a Cache
- Creating and Formatting Partitions
- Volume Operations
- Creating and Accessing Files and Directories
- Managing Removable Media
- Using File Buffers
- Using Working Directories
- Posix
- Optional FAT Journaling
- Media-Specific Operations
- Shell Commands

Setting Up and Opening a Block Device

This section describes how to set up and open a block device instance. The procedure is the same no matter which underlying media type is used, so the provided examples can be used to set up a block device on a variety of media types, including SD card, SCSI logical unit, RAM disk or NOR/NAND flash memory. There are, however, advanced configurations and APIs that are media-specific. Refer to the NAND, NOR and SD-specific options in the subsection Storage Configuration for more details about media-specific advanced configurations. Refer also to Media-Specific Operations for more details about media-specific functions.

- Initializing the Storage Sub-Module
- Opening and Closing a Block Device
- Low-Level Formatting Storage Media

Initializing the Storage Sub-Module

Listing - Initializing the Block Device Sub-Module in the *Setting Up and Opening a Block Device* page illustrates how to initialize the file system's storage sub-module. The function FSStorage_Init() allocates and initializes internal data structures, and should be called only once. The storage sub-module can be configured using the optional advanced configuration API covered in File System Run-time Configuration .

Listing - Initializing the Block Device Sub-Module

```
RTOS_ERR  err;

FSStorage_Init(&err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

Opening and Closing a Block Device

Once the storage sub-module is initialized, the block device is ready to be opened. A block device can be opened using FSBlkDev_Open() and closed using FSBlkDev_Close(), as shown in Listing - Opening a Block Device in the *Setting Up and Opening a Block Device* page. Notice the call to FSMedia_Get() embedded in the FSBlkDev_Open() function call. Though it is

not the only way of retrieving and providing a media handle to FSBlkDev_Open(), this is a compact and convenient way of doing so, based on the media name (see File System Basic Concepts for more details on media and media names).

Listing - Opening a Block Device

```
FS_BLK_DEV_HANDLE  blk_dev_handle;
RTOS_ERR           err;

blk_dev_handle = FSBlkDev_Open(FSMedia_Get("sd0"), &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}

/* The block device handle can be used here to perform block device operations. */

FSBlkDev_Close(blk_dev_handle, &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

Low-Level Formatting Storage Media

NOR and NAND flash memories need to be low-level formatted before a block device can be opened on top of them. This can be done using FSMedia_LowFmt() as shown in Listing - Low-Level Formatting a Media in the *Setting Up and Opening a Block Device* page. The first example (version 1) shows the situation where you want to low-level format the media explicitly, regardless of any existing on-disk formatting information. Version 2 shows a generic way to low-level format only if the media has an invalid format. For media other than NOR and NAND flash memories, FSMedia_LowFmt() does nothing.

Listing - Low-Level Formatting a Media

```
FS_MEDIA_HANDLE    media_handle;
FS_BLK_DEV_HANDLE  blk_dev_handle;
RTOS_ERR           err;
                                            /* ---------------- VERSION 1 ---------------- */
media_handle = FSMedia_Get("nand0");                          (1)
FSMedia_LowFmt(media_handle, &err);                           (2)
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}

blk_dev_handle = FSBlkDev_Open(media_handle, &err);           (3)
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
                                            /* ---------------- VERSION 2 ---------------- */
media_handle   = FSMedia_Get("nand0");                        (4)
blk_dev_handle = FSBlkDev_Open(media_handle, &err);           (5)
if (err.Code != RTOS_ERR_NONE) {
    if (err.Code == RTOS_ERR_BLK_DEV_FMT_INVALID) {           (6)
        FSMedia_LowFmt(media_handle, &err);
        if (err.Code != RTOS_ERR_NONE) {
            /* An error occurred. Error handling should be added here. */
        }
    } else {
        /* An error occurred. Error handling should be added here. */
    }
    blk_dev_handle = FSBlkDev_Open(media_handle, &err);       (7)
    if (err.Code != RTOS_ERR_NONE) {
        /* An error occurred. Error handling should be added here. */
    }
}
```

(1) In Version 1, the media handle associated with the string ID *nand0* is obtained.

(2) Low-level format is performed explicitly regardless of the media on-disk formatting information.

(3) A block device is open on the media. A block device handle is obtained and can be used to open a volume on the media.

(4) In Version 2, once again the media handle associated to the string ID *nand0* is retrieved.

(5) This time an attempt to open a block device on the media is done. If the open operation succeeds, it means that the media has a valid format and you can continue by opening a volume for instance.

(6) If the block device open operation fails, the error type is checked. If it matches RTOS_ERR_BLK_DEV_FMT_INVALID meaning invalid format, a low-level format is performed by calling FSMedia_LowFmt().The low-level format is required only for NAND and NOR devices. Other media types such as SD or RAM disk do not need a low-level format. If FSMedia_LowFmt() is called for SD or RAM disk, the function does nothing and returns successfully.

(7) Upon success of FSMedia_LowFmt(), a new attempt to open the block device on the media is done. This block device opening should be successful.

Performing Raw Block Device Operations

This section describes how to perform raw block device operations. If you are interested only in dealing with the file and directory abstraction, you can ignore this section and jump directly to Creating and Accessing Files and Directories . Please note that although operations performed at the block device level are lower-level that file operations, these operations do not necessarily target physical blocks, pages or sectors of the underlying media. They target logical blocks as opposed to physical blocks/pages/sectors. This distinction is very important in the case of flash memories.

- Reading From and Writing to a Block Device
- Querying the Block Size and Block Count
- Syncing a Block Device

Reading From and Writing to a Block Device

Assuming that a block device handle has been acquired by calling FSBlkDev_Open(), you can read from and write to the block device using FSBlkDev_Rd() and FSBlkDev_Wr() respectively. Listing - Reading from and Writing to a Block Device in the *Performing Raw Block Device Operations* page shows how this is done.

Listing - Reading from and Writing to a Block Device

```
CPU_INT08U  RdWrBuf[MY_BLK_DEV_LB_SIZE];
RTOS_ERR    err;

                                        /* Initialize write buffer with known pattern.        */
for (CPU_INT32U k = 0u; k < MY_BLK_DEV_LB_SIZE; k++) {
    RdWrBuf[k] = (CPU_INT08U)k;
}
                                        /* Write block 0 (first block).                       */
FSBlkDev_Wr(blk_dev_handle,                         /* Block device handle identifying media to write to. */
            &RdWrBuf[0],
            0u,                                     /* Block number from where starting writing to.       */
            1u,                                     /* Number of blocks to write.                         */
            &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
                                        /* Clear read buffer.                                 */
Mem_Clr((void *)&RdWrBuf[0], sizeof(RdWrBuf));

                                        /* Read block 0 back.                                 */
FSBlkDev_Rd(blk_dev_handle,                         /* Block device handle identifying media to read from.*/
            &RdWrBuf[0],
            0u,                                     /* Block number from where starting reading from.     */
            1u,                                     /* Number of blocks to read.                          */
            &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}

                                        /* Check that read and write data match.              */
for (CPU_INT32U k = 0u; k < MY_BLK_DEV_LB_SIZE; k++) {
    if (RdWrBuf[k] != (CPU_INT08U)k) {
        printf("Mismatch at index %u.\r\n", k);
        while (1);
    }
}
```

Querying the Block Size and Block Count

In Listing - Reading from and Writing to a Block Device in the *Performing Raw Block Device Operations* page the logical block size is already known, so that the read/write buffer is allocated statically. In other contexts, such as when dealing with removable media, it might not be possible to know the block size beforehand, and so a run-time mechanism must be used to retrieve block device information dynamically. The FSBlkDev_LbSizeGet() and FSBlkDev_LbCntGet() functions return the logical block size and the logical block count, respectively. Their usage is illustrated in Listing - Querying Block Device Block Size and Block Count in the *Performing Raw Block Device Operations* page.

Listing - Querying Block Device Block Size and Block Count

```
FS_LB_SIZE  lb_size;
FS_LB_QTY   lb_cnt;
RTOS_ERR    err;

lb_size = FSBlkDev_LbSizeGet(blk_dev_handle, &err);
if (err.Code != RTOS_ERR_NONE) {
   /* An error occurred. Error handling should be added here. */
}

lb_cnt = FSBlkDev_LbCntGet(blk_dev_handle, &err);
if (err.Code != RTOS_ERR_NONE) {
   /* An error occurred. Error handling should be added here. */
}

printf("The block device has %u blocks of %u octets.", lb_cnt, lb_size);
```

Syncing a Block Device

Although the write buffer provided to the FSBlkDev_Wr() function may be reused as soon as the function returns, there is no guarantee that the written data has yet reached its final destination on the physical media. This is because the data may be buffered/cached either by the media driver or by the media itself. To force the data on the physical media explicitly, your application can use the function FSBlkDev_Sync() as demonstrated in Listing - Syncing a Block Device in the *Performing Raw Block Device Operations* page.

> When using the file write API , you can use FSBlkDev_Sync() to force the data to be synced to the physical media. However, you are not obliged to perform an explicit sync after each file write. When opening a volume with FSVol_Open(), you can specify an option that controls the auto-sync on the volume. By default, auto sync is disabled to maximize write performance. When auto-sync is enabled, a media sync is done at the end of each write access (for instance file write, file rename, file truncate, etc.) to the media. When auto-sync is off, the sync points are performed less frequently; for example, only when the cache buffers containing the data to be written on the physical media are full. In this case, you might have performed multiple write accesses between two sync points.

Listing - Syncing a Block Device

```
CPU_INT08U  RdWrBuf[MY_BLK_DEV_LB_SIZE] = {0};
RTOS_ERR    err;

                                                  /* Zero out block 0 (first block).           */
FSBlkDev_Wr(blk_dev_handle,
            &RdWrBuf[0],
            0u,                                   /* Block number from where starting writing to.  */
            1u,                                   /* Number of blocks to write.                */
            &err);
if (err.Code != RTOS_ERR_NONE) {
   /* An error occurred. Error handling should be added here. */
}

/* At this point the data may NOT have reached the physical media yet. */
/* If a power failure occurs here, the data may be lost.             */

                                                  /* Force media syncing.                      */
FSBlkDev_Sync(blk_dev_handle, &err);
if (err.Code != RTOS_ERR_NONE) {
   /* An error occurred. Error handling should be added here. */
}

/* From this point on, data is guaranteed to have reached the physical media. */
```

Creating and Assigning a Cache

Before a block device can be accessed through the file system core interfaces, a cache must be assigned.

In its most basic form, the cache is a single buffer tied to a given block device. In more advanced usage schemes, the cache may contain many blocks shared across several block devices. In both scenarios, the cache is used for reading and writing file system metadata as well as performing read/modify/write accesses on user data. Moreover, cache blocks can be used as intermediate buffers to accommodate alignment mismatches between user-provided buffers and low-level drivers requirements.

- Assigning a Cache on a Per-Device Basis
- Sharing a Cache Across Multiple Block Devices
- Cache Flush Points

Assigning a Cache on a Per-Device Basis

The simplest way of assigning a cache to a block device is by using FSCache_DfltAssign(), as shown in Listing - Assigning a Cache on a Per-Device Basis in the *Creating and Assigning a Cache* page. This will assign a cache with the specified number of blocks (4 blocks in the listing) to the given block device. The function FSCache_DfltAssign() handles the calls to the FSCache_Create() and FSCache_Assign() functions described in Listing - Assigning a Cache Instance to Multiple Block Devices in the *Creating and Assigning a Cache* page. Apart from the number of blocks that you must specify, FSCache_DfltAssign() uses default values for the cache configuration that will be associated to the block device. The cache configuration is composed of:

- The memory segment from where the cache buffers will be allocated.
- The requested cache buffer alignment in case the storage media driver uses a DMA engine for the media transfers.
- The maximum logical block size to consider.
- The minimum logical block size to consider.
- The number of cache blocks allowed for the cache instance.

The default values determined by FSCache_DfltAssign() will be:

- Memory segment: most of the time, this will be the default File System Core layer memory segment. That is the heap segment, unless you have changed the memory segment using FSCore_ConfigureMemSeg().
- Buffer alignment: default alignment associated to the given block device.
- Maximum logical block size: logical block size of the given block device. In the example, the SD logical block size is almost always 512 bytes.
- Minimum logical block size: logical block size of the given block device. Note that the minimum and maximum block size are the same.
- Number of cache blocks: the value you have specified as an argument to FSCache_DfltAssign(). In the example, 4 blocks.

Note that FSCache_DfltAssign() returns a pointer to the created cache instance. This pointer can be used with the following functions:

- FSCache_Assign(): you may assign the same cache instance to another block device. The File System allows sharing of the same cache instance across different block devices as explained in Sharing a Cache across Multiple Block Devices . Be aware that the block devices must have the same logical block size if you use FSCache_DfltAssign() to share a cache instance.
- FSCache_MinBlkSizeGet(): you can verify the minimum logical block size that the function FSCache_DfltAssign() has determined.
- FSCache_MaxBlkSizeGet(): you can verify the maximum logical block size that the function FSCache_DfltAssign() has determined.

Listing - Assigning a Cache on a Per-Device Basis

```
FS_BLK_DEV_HANDLE  blk_dev_handle;

blk_dev_handle = FSBlkDev_Open(FSMedia_Get("sd0"), &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
(void)FSCache_DfltAssign(blk_dev_handle, 4u, &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}

/* From now on, file system core API's can be used on this block device. */
```

Sharing a Cache Across Multiple Block Devices

A cache instance can be shared across multiple block devices. This sharing capability allows you to achieve the best possible trade-off between RAM usage and performance. In fact, sharing a cache instance reduces the relative overhead of the cache data structures (used for internal bookkeeping) and possibly the memory footprint associated with the cache blocks themselves (a single cache block can be shared across as many devices as you want). However, sharing also means a higher probability of block trashing, that is blocks being constantly evicted and read back, which could reduce the overall performance.

Listing - Assigning a Cache Instance to Multiple Block Devices in the *Creating and Assigning a Cache* page shows how to create a cache instance that can be shared between two block devices with different logical block sizes and different alignment requirements. The approach demonstrated here is completely generic in the sense that block device parameters are determined programmatically. However, for applications that do not require this level of generality (for example, where all block sizes are known beforehand and the code will always run on the same platform), block device parameters could simply be hard coded.

Note that cache_cfg.BlkCnt determines the number of blocks of size cache_cfg.MaxBlkSize. In general, the cache will contain cache_cfg.BlkCnt * (cache_cfg.MaxBlkSize / lb_size) blocks of size lb_size.

Also note the use of DEF_MAX() to determine the alignment requirement common to both block devices: this assumes that one of the alignment requirement is a divisor of the other. Otherwise, the least common multiple should be computed.

Listing - Assigning a Cache Instance to Multiple Block Devices

```
FS_BLK_DEV_HANDLE  nor_blk_dev_handle;
FS_BLK_DEV_HANDLE  sd_blk_dev_handle;
FS_CACHE_CFG       cache_cfg;

                                                /* Open an SD block device.                     */
sd_blk_dev_handle = FSBlkDev_Open(FSMedia_Get("sd0"), &err);
if (err.Code != RTOS_ERR_NONE) {
   /* An error occurred. Error handling should be added here. */
}
                                                /* Open a NOR block device.                     */
nor_blk_dev_handle = FSBlkDev_Open(FSMedia_Get("nor0"), &err);
if (err.Code != RTOS_ERR_NONE) {
   /* An error occurred. Error handling should be added here. */
}
                                                /* Get SD block device needed alignment.        */
sd_align_req = FSMedia_AlignReqGet(FSMedia_Get("sd0"), &err);
if (err.Code != RTOS_ERR_NONE) {
   /* An error occurred. Error handling should be added here. */
}
                                                /* Get NOR block device needed alignment.       */
nor_align_req = FSMedia_AlignReqGet(FSMedia_Get("nor0"), &err);
if (err.Code != RTOS_ERR_NONE) {
   /* An error occurred. Error handling should be added here. */
}
                                                /* Get SD logical block size.                   */
sd_lb_size = FSBlkDev_LbSizeGet(sd_blk_dev_handle, &err);
if (err.Code != RTOS_ERR_NONE) {
   /* An error occurred. Error handling should be added here. */
}
                                                /* Get NOR logical block size.                  */
nor_lb_size = FSBlkDev_LbSizeGet(nor_blk_dev_handle, &err);
if (err.Code != RTOS_ERR_NONE) {
   /* An error occurred. Error handling should be added here. */
}
                                                /* Set cache configuration according to SD & NOR.    */
cache_cfg.MinBlkSize    = DEF_MIN(sd_lb_size, nor_lb_size);
cache_cfg.MaxBlkSize    = DEF_MAX(sd_lb_size, nor_lb_size);
cache_cfg.BlkCnt        = 1u;
cache_cfg.BlkMemSegPtr  = DEF_NULL;
cache_cfg.Align         = DEF_MAX(sd_align_req, nor_align_req);
                                                /* Create 1 cache instance shared by SD & NOR.    */
p_cache = FSCache_Create(&cache_cfg, &err);              /* A pointer to cache instance is returned.        */
if (err.Code != RTOS_ERR_NONE) {
   /* An error occurred. Error handling should be added here. */
}
                                                /* Assign cache to SD passing the pointer to cache.  */
FSCache_Assign(sd_blk_dev_handle, p_cache, &err);
if (err.Code != RTOS_ERR_NONE) {
   /* An error occurred. Error handling should be added here. */
}
                                                /* Assign cache to NOR passing the pointer to cache. */
FSCache_Assign(nor_blk_dev_handle, p_cache, &err);
if (err.Code != RTOS_ERR_NONE) {
   /* An error occurred. Error handling should be added here. */
}

/* From now on, file system core API's can be used on SD & NOR block devices. */
```

Cache Flush Points

Cache flush points correspond to the moment where the cache module flushes its buffers' content to the physical media. Inside the File System Core layer, there are two types of flush points:

- Implicit flush point

- Explicit flush point

Implicit flush points occur when the cache module has no more buffers available. It flushes one buffer from among all of the full buffers. The flushed buffer is reused to cache either new file system metadata or file data. Implicit flush points can happen at any moment when performing any file or directory operation. The non-exhaustive list below indicates which functions can trigger an implicit flush:

- FSPartition_Init()
- FSPartition_Add()
- FS_FAT_Fmt()
- FSVol_Open()
- FSVol_LabelSet()
- FSVol_LabelGet()
- FSFile_Open()
- FSFile_Rd()
- FSFile_Wr()
- FSFile_Truncate()
- FSFile_Copy()
- FSDir_Open()
- FSDir_Rd()
- FSEntry_Create()
- FSEntry_Rename()
- FSEntry_AttribSet()
- FSEntry_TimeSet() (this function is deprecated and will be replaced by sl_fs_entry_time_set() in a next release)

Your application can slightly influence the frequency of implicit flush points in three situations:

- By creating a cache instance with several buffers available. This is accomplished when your application calls FSCache_Create() and configures a large number of buffers (field FS_CACHE_CFG.MinBlkCnt). If more buffers are available to the cache module, it needs to flush a buffer less often. However, this influence is limited. If your application accesses different logical blocks of the device, the chances are that all these logical blocks are cached and thus occupy all of the available cache buffers. In that case, implicit flush points occur anyway at a certain frequency.
- By frequently accessing the same logical blocks of your media. If your application is often reading or writing the same sectors of a file, and the number of available cache buffers is large enough, all these accessed sectors have been cached at least once. And the cache module will not necessarily need to flush a buffer, since the sectors' data are already cached.
- By accessing the logical blocks in very small chunks. For example, if you write a few 512-byte sectors by chunks of 20 bytes, and you have four cache buffers available, the implicit flush points may become less frequent due to the numerous file writes that end up in the cache before exhausting the cache buffer pool.

Even if you can influence the frequency of implicit flush points a little bit, they remain an *asynchronous* operation because they happen when the cache module needs to free one of its buffers.

Explicit flush points occur in two cases:

- During pre-determined locations inside some file operations
- When your application syncs the volume

In these cases, an explicit flush point flushes *all* the buffers containing cached data, and they happen in addition to the implicit flush points. In contrast to implicit flush points, explicit flush points are *synchronous* operations because you control when they happen. Explicit flush points allow your application to create its own flush points to ensure that your data is synced regularly on the physical media.

The pre-determined locations for explicit flush points are activated if the volume's auto-sync option is enabled. You can enable this option via the parameter opt of the function FSVol_Open(). When the volume's auto-sync is on, an explicit flush point occurs at the end of the following functions:

- FSFile_Wr()
- FSFile_Truncate()
- FSFile_Copy()
- FSFile_Close()

- FSEntry_Create()
- FSEntry_Del()
- FSEntry_Rename()
- FSEntry_AttribSet()
- FSEntry_TimeSet() (this function is deprecated and will be replaced by sl_fs_entry_time_set() in a next release)

The volume sync is done by calling the function FSVol_Sync().

Creating and Formatting Partitions

Micrium OS File System is able to read and create DOS-like partitions. By default, support for partition handling is enabled, but it can be disabled at compile-time via FS_CORE_CFG_PARTITION_EN if it is not needed. If support for partitions is disabled, only the first partition can be read (even though more partitions may exist on the block device). The partition table is part of the Master Boot Record (MBR) sector. The block device can be used without a partition table (that is no MBR is created) In that case, all its blocks are part of an implicit large partition spanning the whole device.

Note that although Micrium OS File System is able to read extended partitions, it does not support creating extended partitions. This means that Micrium OS File System cannot create more than four partitions on a given block device.

- Creating a Partition Table
- Adding a New Partition
- Formatting an Existing Partition
- Extended Partitions

Creating a Partition Table

Creating a partition table involves deleting any partition table that may already exist in the MBR, effectively obliterating all the data that the block device may contain. Make sure that the block device on which you want to create the new partition table does not contain any valuable data before proceeding.

A partition table, part of the MBR sector, can be created using FSPartition_Init() as shown in Listing - Creating a Partition Table in the *Creating and Formatting Partitions* page(assuming that the block device has been previously opened, and a cache assigned, as shown in Setting Up and Opening a Block Device and Creating and Assigning a Cache).

In addition to creating a new partition table, FSPartition_Init() also creates the first partition on the device, hence the second parameter, which specifies the size of the first partition in logical blocks. Here, a partition containing 1 000 000 logical blocks is created. Assuming 512-byte logical blocks, this is a 512MB partition.

Listing - Creating a Partition Table

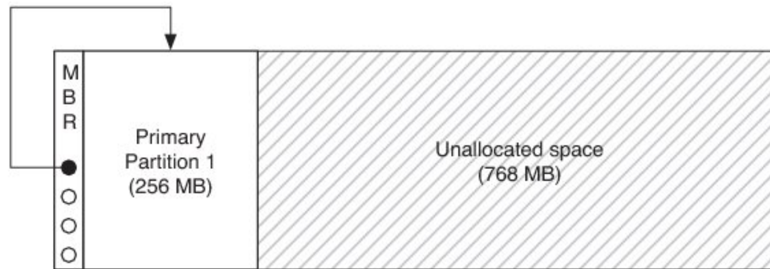
```
FS_BLK_DEV_HANDLE  blk_dev_handle;
RTOS_ERR           err;

/* Open a block device here using FSBlkDev_Open(). */

FSPartition_Init(blk_dev_handle,
        1000000u,                          /* Partition size in number of logical blocks. */
        &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

After this call, the block device will be divided as shown in Figure - Block Device after Partition Initialization in the *Creating and Formatting Partitions* page. This new partition is called a **primary partition** because its entry is in the MBR. The four circles in the MBR represent the four partition entries; the one that is now used ‘points to’ Primary Partition 1.

Figure - Block Device after Partition Initialization



Adding a New Partition

Once a partition table has been created (along with the first partition), more partitions can be added using FSPartition_Add() as illustrated in Listing - Adding Partitions in the *Creating and Formatting Partitions* page. If no partition table has been created explicitly, it is created first.

Listing - Adding Partitions

```
FS_BLK_DEV_HANDLE  blk_dev_handle;
RTOS_ERR           err;

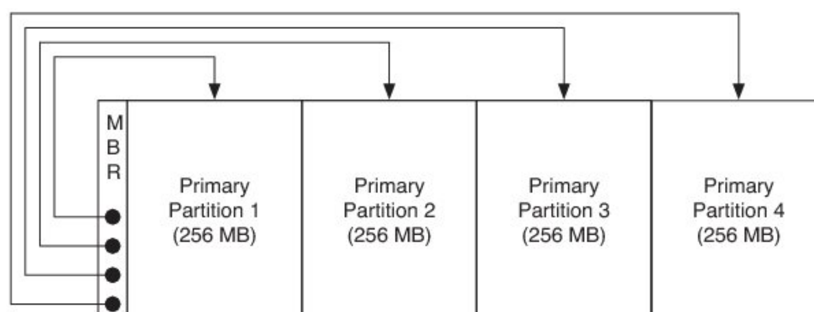
/* Open a block device here using FSBlkDev_Open(). */

FSPartition_Add(blk_dev_handle,                (1)
       100000u,                            /* Partition size in number of logical blocks. */
       &err);
if (err.Code != RTOS_ERR_NONE) {
   /* An error occurred. Error handling should be added here. */
}
```

(1) This call adds one partition besides the initial partition created if FSPartition_Init() has been previously called or create the first partition along with the MBR sector if FSPartition_Init() has not been previously called. FSPartition_Add() must be called for each partition you want to create. For example, if you need to create three more partitions, call FSPartition_Add() three times. In this example,

the device is divided as shown in Figure - Block Device after Four Partitions Have Been Created in the *Creating and Formatting Partitions* page.

Figure - Block Device after Four Partitions Have Been Created



Formatting an Existing Partition

Before a partition can be used to store actual contents, it must be populated with file system metadata. The process of writing appropriate file system metadata on disk is called formatting, and it is specific to a given file system driver. FAT is the only file system driver currently supported by Micrium OS File System, and so formatting is performed using FS_FAT_Fmt() as shown in Listing - Formatting a Partition in the *Creating and Formatting Partitions* page (assuming that at least one partition has been created, or that no partition table is used at all). The second parameter (here 1u) specifies the number of

the partition that will be formatted. The first partition has number 1, the second partition has number 2, and so on. The third parameter (here DEF_NULL) may optionally be used to specify FAT volume parameters and is covered in details in FS_FAT_Fmt().

Listing - Formatting a Partition

```
FS_BLK_DEV_HANDLE  blk_dev_handle;
RTOS_ERR           err;

/* Open a block device here using FSBlkDev_Open(). */

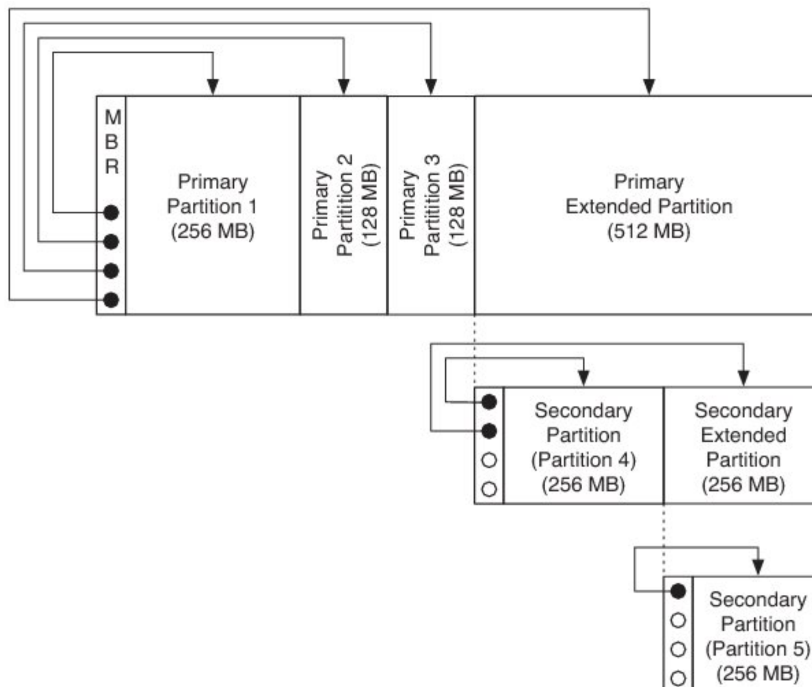
FS_FAT_Fmt(blk_dev_handle,
      1u,                              /* Partition #1 to format.      */
      DEF_NULL,                        /* Default FAT configurations.  */
     &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

Extended Partitions

When first instituted, DOS partitioning was a simple scheme allowing up to four partitions, each with an entry in the MBR. It was later extended for larger devices requiring more with **extended partitions**, partitions that contains other partitions. The **primary extended partition** is the extended partition with its entry in the MBR; it should be the last occupied entry.

An extended partition begins with a partition table that has up to two entries (typically). The first defines a **secondary partition** which may contain a file system. The second may define another extended partition; in this case, a **secondary extended partition**, which can contain yet another secondary partition and secondary extended partition. Basically, the primary extended partition heads a linked list of partitions.

Figure - Block Device with Five Partitions



Reading secondary partitions in existing pre-formatted devices is supported by Micrium OS File System. For the moment, the creation of extended and secondary partitions is not supported in Micrium OS File System.

Volume Operations

- Opening and Closing a Volume
- Syncing a Volume
- Obtaining Information Associated to a Volume

This section describes the high-level file system API that you can use to manage a volume.

There are four general operations that can be performed on a volume:

- A volume can be **opened (mounted)**. During the opening of a volume, file system control structures are read from the underlying block device, parsed and verified.
- **Files can be accessed** on a volume. A file is a linear data sequence (the file's content) that is associated with some logical, typically human-readable identifier (a file name). Additional properties, such as size, update date/time, and access mode (for example read-only, write-only, read-write) may be associated with a file. File accesses consist of reading data from files, writing data to files, creating new files, renaming files, copying files, and so on. A file access is accomplished via the file module-level functions, which are covered in Creating and Accessing Files and Directories .
- **Directories can be accessed** on a volume. A directory is a container for files and other directories. Operations include iterating through the contents of the directory, creating new directories, renaming directories, etc. A directory access is accomplished via the directory module-level functions, which are covered in Creating and Accessing Files and Directories .
- A volume can be **closed (unmounted)**. During volume closing, any cached data is written to the underlying device and associated structures are freed.

Opening and Closing a Volume

A volume can be opened using FSVol_Open() and closed using FSVol_Close(), as shown in Listing - Opening a Volume in the *Volume Operations* page.

Listing - Opening a Volume

```
FS_VOL_HANDLE  vol_handle;
RTOS_ERR       err;

/* Assume that a block device has previously been opened with FSBlkDev_Open(). */

/* You may have to format the partition with FS_FAT_Fmt() before opening a volume. */
                                                    (1)

/* Open a volume named "vol0". */
vol_handle = FSVol_Open(blk_dev_handle,          /* Handle to block device where volume is opened.      */
                        1u,                      /* Partition #1.                                       */ (2)
                        "vol0",                  /* Unique volume name across all open volumes.         */
                        FS_VOL_OPT_DFLT,         /* Default options: write allowed, auto sync disabled.  */ (3)
                        &err);
if (err.Code != RTOS_ERR_NONE) {
   /* An error occurred. Error handling should be added here. */
}

/* You can now perform any file and directory operations in the opened volume. */

/* Close the volume. */
FSVol_Close(vol_handle, &err);
if (err.Code != RTOS_ERR_NONE) {
   /* An error occurred. Error handling should be added here. */
}
```

(1) Prior to opening the volume, you may need to high-level format the partition on which you want to open a volume. This is accomplished with the function FS_FAT_Fmt(), which is a file system-specific function. Refer to the section Formatting an Existing Partition for more details.

(2) The partition number uses a one-based index. Thus, the partition number #0 cannot be specified.

(3) The fourth argument allows you to indicate the volume's options. If the option FS_VOL_OPT_DFLT is used, the volume is opened with the following default options:

- Write allowed
- Auto-sync disabled

The other possible options are:

- FS_VOL_OPT_ACCESS_MODE_RD_ONLY: volume in read-only. No write access is allowed.
- FS_VOL_OPT_AUTO_SYNC: auto-sync is enabled. If this option is enabled, certain high-level operations will finish by explicitly flushing the cache instance. Refer to the section Cache Flush Points for more details about explicit cache flush points.

Syncing a Volume

All high-level operations that imply a write access to the volume go through the cache module of the File System core. The volume's data (both file data and metadata) may be cached but not yet synced on the physical media. Two methods can be used to sync the data onto the physical media using explicit cache flush points (refer to the section Cache Flush Points for more details about explicit cache flush points):

- Volume auto-sync option
- Explicit volume sync

The volume auto-sync is enabled via the option FS_VOL_OPT_AUTO_SYNC passed to the function FSVol_Open(). Refer to the section Opening and Closing a Volume for more details.

The explicit volume sync is done when you call the function FSVol_Sync() as illustrated in Listing - Explicit Volume Sync in the *Volume Operations* page.

Listing - Explicit Volume Sync

```
FS_VOL_HANDLE  vol_handle;

/\* A volume must have been previously been opened with FSVol_Open(). \*/

/\* Sync volume by flushing the cache buffers. \*/
FSVol_Sync(vol_handle, &err);
if (err.Code != RTOS_ERR_NONE) {
   /\* An error occurred. Error handling should be added here. \*/
}
```

Obtaining Information Associated to a Volume

The File System volume API offers some utility functions used to retrieve information associated to a given volume that may be useful for your application. There are five functions:

- FSVol_Query(): this function provides pieces of information about certain volume characteristics.
- FSVol_PartitionNbrGet(): this function provides the partition number on which the volume has been opened.
- FSVol_BlkDevGet(): this function provides the block device handle associated to a given volume handle.
- FSVol_NameGet(): this function returns the volume name associated to a given volume handle.
- FSVol_Get(): this function does the opposite of FSVol_NameGet() by returning the volume handle associated to a given volume name.

Listing - Volume Utility Functions in the *Volume Operations* page presents an example of usage of these functions.

Listing - Volume Utility Functions

```
FS_VOL_HANDLE      vol_handle;
CPU_CHAR           path_buf[50u];
FS_VOL_INFO        vol_info;
FS_FAT_VOL_CFG     fat_vol_info;
FS_PARTITION_NBR   partition_nbr;
FS_BLK_DEV_HANDLE  blk_dev_handle;
FS_LB_SIZE         dev_lb_size;
RTOS_ERR           err;

/* Assume that a volume named "test_vol" has previously been opened. */

                                                                    /* -------------- UTILITY 1 -------------- */
/* Get information about the volume. */
FSVol_Query(vol_handle,
            &vol_info,                                              /* Structure receiving info about volume.  */ (1)
            &fat_vol_info,                                          /* Optional info about volume file system. */ (2)
            &err)
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}

printf("Volume info:\r\n");
printf("- Number of sectors used: %u\r\n", vol_info.UsedSecCnt);
printf("- Number of sectors free: %u\r\n", vol_info.FreeSecCnt);
printf("- Total number of sectors: %u\r\n", vol_info.TotSecCnt);

                                                                    /* -------------- UTILITY 2 -------------- */
partition_nbr = FSVol_PartitionNbrGet(vol_handle, &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}

                                                                    /* -------------- UTILITY 3 -------------- */
FSVol_NameGet(vol_handle,
              path_buf,                                             /* Buffer receiving volume name            */
              50u,                                                  /* Buffer size in bytes.                   */
              &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
                                                                    /* Display retrieved information.          */
printf("Volume '%s' is associated to partition #%u.\r\n", path_buf, partition_nbr);

                                                                    /* -------------- UTILITY 4 -------------- */
                                                                    (3)
blk_dev_handle = FSVol_BlkDevGet(vol_handle);                       /* Get block device handle.                */
if (FS_BLK_DEV_HANDLE_IS_NULL(blk_dev_handle)) {                    (4)
    /* An error occurred. Error handling should be added here. */
}

dev_lb_size = FSBlkDev_LbSizeGet(blk_dev_handle, &err);             /* Get block device logical block size.    */
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}

printf("Volume '%s' is associated to a block device whose block size is %u.\r\n", path_buf, dev_lb_size);
```

(1) You must declare a volume information structure of type FS_VOL_INFO to retrieve the volume characteristics. All the fields of FS_VOL_INFO are described in FSVol_Query().

(2) You can optionally retrieve pieces of information about the file system formatted on the volume. In this example, information about FAT is obtained. Specify DEF_NULL if you don't need it.

(3) FSVol_BlkDevGet() returns the block handle associated to a given volume handle. With the block device handle, you can obtain block device characteristics such as the logical block size with function FSBlkDev_LbSizeGet(), the number of

blocks using function FSBlkDev_LbCntGet() and the block device alignment requirement with function FSBlkDev_AlignReqGet().

(4) An utility macro called FS_BLK_DEV_HANDLE_IS_NULL() allows you to verify if the block device handle is valid, that is a non-null handle.

Creating and Accessing Files and Directories

This page describes the high-level file system API that is used to manage files and directories.

- Opening and Closing a Volume
- Creating a File or a Directory
- Opening and Closing a File
- Opening and Closing a Directory
- Reading and Writing Files
- File and Directory Names and Paths
  - Absolute Path
  - Relative Path

Opening and Closing a Volume

Before files can be created and accessed, a volume needs to be opened (or equivalently, a partition needs to be mounted). A volume can be opened using FSVol_Open()and closed using FSVol_Close()as shown in Listing - Opening a Volume in the *Creating and Accessing Files and Directories* page.

The volume name provided to the function FSVol_Open() must be unique. The volume name string can be anything you want, and there is no restricted format to follow. For instance, the volume name could be:

- "vol0", "vol1", etc.: generic volume name with an incrementing number appended to the name
- "nand:0", "nand:1", etc.: here the volume name is composed of the media name on which the volume is opened, 'nand', and an incrementing number appended and separated by a single colon ':'

```
FS_BLK_DEV_HANDLE  blk_dev_handle;
FS_VOL_HANDLE      vol_handle;
RTOS_ERR           err;

/* Assumes that 'blk_dev_handle' points to a previously opened block device. */

/* Open a volume and name it "test_vol". */
vol_handle = FSVol_Open(blk_dev_handle,
                        1u,                        /* Open volume on formatted partition #1.          */
                        "test_vol",                /* Unique volume name across all open volumes.     */
                        FS_VOL_OPT_DFLT,           /* Default options: write allowed, auto sync disabled.  */
                        &err);
if (err.Code != RTOS_ERR_NONE) {
   /* An error occurred. Error handling should be added here. */
}

/* Perform operations on the volume here. */

/* Close the volume. */
FSVol_Close(vol_handle, &err);
if (err.Code != RTOS_ERR_NONE) {
   /* An error occurred. Error handling should be added here. */
}
```

Creating a File or a Directory

There are two approaches for creating files and directories. The first one, using FSEntry_Create(), is shown in Listing - Creating a File or Directory using FSEntry Create() in the *Creating and Accessing Files and Directories* page. The second one, using FSFile_Open() nand FSDir_Open(), is covered in the sections Opening and Closing a File and Opening and Closing a Directory . There are only two entry types: directory or file.

Listing - Creating a File or Directory using FSEntry Create()

```
RTOS_ERR  err;

/* Assumes that a volume named "test_vol" has previously been opened. */

/* Create directory 'test_dir' in 'test_vol' root directory. */
FSEntry_Create(FS_WRK_DIR_NULL,                    /* NULL working directory means absolute path used. */
               "test_vol/test_dir",                /* Absolute path to directory to create.            */
               FS_ENTRY_TYPE_DIR,                  /* Entry type is directory.                         */
               DEF_NO,                             /* Directory entry created even if it exists.       */
               &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}

/* Create file 'test_file.bin' in 'test_dir' directory. */
FSEntry_Create(FS_WRK_DIR_NULL,                    /* NULL working directory means absolute path used. */
               "test_vol/test_dir/test_file.bin",  /* Absolute path to file to create.                 */
               FS_ENTRY_TYPE_FILE,                 /* Entry type is file.                              */
               DEF_NO,                             /* File entry created even if it exists.            */
               &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

Opening and Closing a File

A file can be opened using FSFile_Open() and closed using FSFile_Close(), as shown in Listing - Opening a File in the *Creating and Accessing Files and Directories* page. The FSFile_Open() function accepts a certain number of flags, all prefixed with FS_FILE_ACCESS_MODE_XXXX, and which alter the function's behavior. These flags are explained in Table - File Access Mode Flags in the *Creating and Accessing Files and Directories* page. The function FSFile_Open() uses a working directory handle as a first argument. You can refer to section Using Working Directories for more details about working directory.

Listing - Opening a File

```
FS_FILE_HANDLE  file_handle;
RTOS_ERR        err;

/* Assumes that a volume named "test_vol" has previously been opened. */

/* Open 'test_vol/test_file.bin' file for reading (create it if needed). */
file_handle = FSFile_Open(FS_WRK_DIR_NULL,              /* NULL working directory means absolute path used. */
                          "test_vol/test_file.bin",     /* Absolute path to file to create.                 */
                          FS_FILE_ACCESS_MODE_CREATE |  /* File created if it does not exist.               */
                          FS_FILE_ACCESS_MODE_WR,       /* File can be read or written.                     */
                          &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}

/* The file handle can now be used for reading or writing data. */

/* Close the file. */
FSFile_Close(file_handle, &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

Table - File Access Mode Flags

Flags	Description
FS_FILE_ACCESS_MODE_CREATE	If present, this flag indicates that the file should be created before being opened, if it does not already exist. If the file already exists, the function's behavior is determined by the presence or absence of the FS_FILE_ACCESS_MODE_EXCL flag.
FS_FILE_ACCESS_MODE_EXCL	If present, this flag indicates that the file should not be opened if it already exists. If FS_FILE_ACCESS_MODE_CREATE is present, the file is created, otherwise the error is set to RTOS_ERR_NOT_FOUND.
FS_FILE_ACCESS_MODE_WR	If present, this flag indicates that the file should be opened for writing. That is, write operations on this file will be allowed or disallowed based on the presence or absence of this flag. If the flag is absent and a write operation is attempted on the opened file an RTOS_ERR_FILE_ACCESS_MODE_INVALID error is returned. FS_FILE_ACCESS_MODE_WR implies FS_FILE_ACCESS_MODE_RD.
FS_FILE_ACCESS_MODE_TRUNCATE	If present, this flag indicates that the file contents should be destroyed (i.e., file size zeroed) before the file is opened.
FS_FILE_ACCESS_MODE_APPEND	If present, this flag indicates that all written data should be appended to the end of the file. In other words, the current file position should automatically be set to the end of the file before any write operation takes place.
FS_FILE_ACCESS_MODE_RD	If present, this flag indicates that the file should be opened for reading (though it can also be opened for writing if FS_FILE_ACCESS_MODE_WR is also specified).

Opening and Closing a Directory

A directory can be opened using FSDir_Open()and closed using FSDir_Close() as shown in Listing - Opening a Directory in the *Creating and Accessing Files and Directories* page. Similarly to FSFile_Open(), FSDir_Open() accepts a certain number of flags to control the function's behavior: FS_DIR_ACCESS_MODE_CREATE and FS_DIR_ACCESS_MODE_EXCL. The meaning of these flags is analogous to the FSFile_Open() flags, and are explained in Table - Directory Access Mode Flags in the *Creating and Accessing Files and Directories* page. The function FSDir_Open() uses a working directory handle as a first argument. You can refer to section Using Working Directories for more details about working directory.

Listing - Opening a Directory

```
FS_DIR_HANDLE  dir_handle;
RTOS_ERR       err;

/* Assumes that a volume named "test_vol" has previously been opened. */

/* Open 'test_vol/test_dir' directory for reading (create it if needed). */
dir_handle = FSDir_Open(FS_WRK_DIR_NULL,                /* NULL working directory means absolute path used. */
                        "test_vol/test_dir",            /* Absolute path to directory to create.           */
                        FS_DIR_ACCESS_MODE_CREATE,      /* Directory created if it does not exist.         */
                        &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}

/* The directory handle can now be used for reading the directory's content. */

/* Close directory. */
FSDir_Close(dir_handle, &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

Table - Directory Access Mode Flags

Flags	Description
FS_DIR_ACCESS_MODE_NONE	If present, this flag indicates that the directory is not created at all and is opened only if the directory already exists. The main usage of FS_DIR_ACCESS_MODE_NONE is to get a directory handle of an existing directory. If the directory does not exist, the error RTOS_ERR_NOT_FOUND is returned. If FS_DIR_ACCESS_MODE_NONE is used with the other flags, it has no effect and only the other flags are considered.
FS_DIR_ACCESS_MODE_CREATE	If present, this flag indicates that the directory should be created before being opened, if it does not already exist. If the directory already exists, the function's behavior is determined by the presence or absence of the FS_DIR_ACCESS_MODE_EXCL flag.
FS_DIR_ACCESS_MODE_EXCL	If present, this flag indicates that the directory should not be opened if it already exists. If FS_DIR_ACCESS_MODE_CREATE is present, the directory is created; otherwise the error is set to RTOS_ERR_NOT_FOUND.

Reading and Writing Files

A file can be read from and written to using FSFile_Rd()and FSFile_Wr()respectively. This is shown in Listing - Reading and Writing Files in the *Creating and Accessing Files and Directories* page.

Note how the call to FSFile_PosSet()is used to bring the current file position back to the beginning of the file. This is necessary because the current file position is automatically incremented after each call to FSFile_Rd() and FSFile_Wr() by an amount equal to the number of bytes read or written. In other words, without the call to FSFile_PosSet(), the current file position would have been equal to sizeof(msg) - 1 and the following call to FSFile_Rd() would have immediately encountered the end of the file.

Also note that FSFile_Wr() returns the number of bytes written. Unless an error occurs, this value is always equal to the number of bytes to be written (the third parameter). This value is simply ignored here, as it is not needed. Similarly, FSFile_Rd() returns the number of bytes read from the file. However, unlike the value returned by FSFile_Wr(), this value may be different from the number of bytes given as a parameter, even if no error occurs. If the end of the file is reached before the specified number of bytes is read, the function returns the number of bytes read so far. This condition can be tested to determined whether the end of file has been reached, as shown below.

Listing - Reading and Writing Files

```
FS_FILE_HANDLE  file_handle;
CPU_CHAR        buf[50];
CPU_SIZE_T      rd_size;
CPU_CHAR        msg[] = "This is a test!";
RTOS_ERR        err;

/* Assumes that a volume named "test_vol" has previously been opened. */

/* Open a file (create or truncate if needed). */
file_handle = FSFile_Open(FS_WRK_DIR_NULL,                  /* NULL working dir. means absolute path used. */
                          "test_vol/test_file.bin",         /* Absolute path to file to create.           */
                          FS_FILE_ACCESS_MODE_CREATE   |    /* File created if it does not exist.         */
                          FS_FILE_ACCESS_MODE_TRUNCATE |    /* If file exists, size truncated to 0 bytes. */
                          FS_FILE_ACCESS_MODE_WR,           /* File can be read or written.               */
                          &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}

/* Write 'This is a test!' to 'test_vol/test_file.bin'. */
(void)FSFile_Wr(file_handle,                                /* File handle associated to open file.       */
                (void *)msg,                                /* Buffer containing data to write.           */
                sizeof(msg),                                /* Size of buffer in bytes.                   */
                &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

```
/* Set the current file position at the beginning of the file. */FSFile_PosSet(file_handle,/* File handle associated to open file.       */0u,/* Offset added to reference position.        */
            FS_FILE_ORIGIN_START,/* Ref. position for offset: start of file.    */&err);if(err.Code != RTOS_ERR_NONE){/* An error occurred. Error handling should be added here. */}/* Read the previously written message. */
rd_size =FSFile_Rd(file_handle,/* File handle associated to open file.       */(void *)&buf,/* Buffer receiving data from file.          */sizeof(buf),/* Size of buffer in bytes.                   */&err);if(err.Code != RTOS_ERR_NONE){/* An error occurred. Error handling should be added here. */}/* Verify whether the end of the file has been reached (this should be the case here). */if(rd_size <sizeof(buf)){printf("The end of file has been reached.\r\n");}/* Compare the read message with the previously written message. */if(Str_Cmp(buf, msg)!=0u){printf("Something went wrong!\r\n");while(1);}/* Close file. */FSFile_Close(file_handle,&err);if(err.Code != RTOS_ERR_NONE){/* An error occurred. Error handling should be added here. */}
```

File and Directory Names and Paths

Files and directories are identified by a path string. For example, a file can be opened with:

```
file_handle = FSFile_Open(FS_WRK_DIR_NULL,
                          "test_vol/file001.txt",
                          FS_FILE_ACCESS_MODE_WR,
                          &err);
```

In this case, test_vol/file001.txt is the path string.

An application specifies the path of a file or directory using either an **absolute** or a **relative** path.

> In the path string test_vol/file001.txt , the path separator is a slash '/'. The path separator can also be a backslash '\'. In that case, the example of path string becomes: test_vol\file001.txt . Note that there are two backslashes in the path. This is due to the fact that backslash is used as an escape character by compilers when generating special characters. In order to get the proper character value corresponding to the single backslash '', the path string must double the backslash. This precaution does not apply to the forward slash '/'.

Absolute Path

An absolute path is a character string which specifies a unique file, and follows the pattern:

<vol_name><... Path ...><File>

where

<vol_name>	is the name of the volume, identical to the string specified in FSVol_Open().
<... Path ...>	is the file path, which must always begin and end with a slash '/' or a backslash ''.
<File>	is the file (or leaf directory) name, including any extension.

Listing - Example of Absolute File and Directory Paths in the *Creating and Accessing Files and Directories* page shows some examples of absolute paths (error handling omitted for clarity):

Listing - Example of Absolute File and Directory Paths

```
/* Assume that a volume called "sd:vol0" has been previously open. */

dir_handle = FSDir_Open(FS_WRK_DIR_NULL,                    (1)
                        "sd:vol0",
                        FS_DIR_ACCESS_MODE_CREATE,
                        &err);

dir_handle = FSDir_Open(FS_WRK_DIR_NULL,                    (2)
                        "sd:vol0/dir01",
                        FS_DIR_ACCESS_MODE_CREATE,
                        &err);

file_handle = FSFile_Open(FS_WRK_DIR_NULL,                  (3)
                          "sd:vol0/file.txt",
```

```
&err);

file_handle =FSFile_Open(FS_WRK_DIR_NULL,(4)"sd:vol0/dir01/file01.txt",
                    FS_FILE_ACCESS_MODE_WR,&err);
```

Listing - Example of Absolute File and Directory Paths

(1) Opening the root directory of a specified volume.

(2) Opening a non-root directory.

(3) Opening a file in the root directory of a specified volume.

(4) Opening a file in a non-root directory.

In the code snippet above, all absolute paths containing the **slash '/'** separator could be replaced by a **backslash '|'** separator. Thus the different paths would become:

"sd:vol0\dir01"

"sd:vol0\file.txt"

"sd:vol0\dir01\file01.txt"

The code snippet would create the following tree structure:

sd:vol0 \dir01 \file01.txt \file.txt

Relative Path

Relative paths can be used if the working directory API (that is FSWrkDir_Xxxx()) is employed (cf. section Using Working Directories for more details about working directory). A relative path begins with neither a volume name nor a slash '/' or a backslash \ :

<... Relative Path ...><File>

where

<... Relative Path ...>	is the file path, which must not begin with a slash '/' or a backslash '' but must end with a '' or '/'.
<File>	is the file (or leaf directory) name, including any extension.

Two special path components can be used:

- Dot dot '..' moves the path to the parent directory.
- Dot '.' keeps the path in the current directory (basically, it does nothing).

A relative path is appended to the current working directory of the calling task to form the absolute path of the file or directory.

Listing - Example of Relative File and Directory Paths in the *Creating and Accessing Files and Directories* page shows some examples of relative paths (error handling omitted for clarity):

Listing - Example of Relative File and Directory Paths

```
FS_WRK_DIR_HANDLE  wrk_dir_handle;

/* Assume that a volume called "sd:vol0" has been previously open. */

/* Open a working directory mapped to an existing directory 'sd:vol0/test_dir'. */
wrk_dir_handle = FSWrkDir_Open(FS_WRK_DIR_NULL,
                               "sd:vol0/test_dir",
                               &err);

dir_handle = FSDir_Open(wrk_dir_handle,                    (1)
                        "../",
```

```
                FS_DIR_ACCESS_MODE_CREATE,&err);

dir_handle =FSDir_Open(wrk_dir_handle,(2)"./dir01",
                FS_DIR_ACCESS_MODE_CREATE,&err);

dir_handle =FSDir_Open(wrk_dir_handle,(3)
                DEF_NULL,
                FS_DIR_ACCESS_MODE_CREATE,&err);
dir_handle =FSDir_Open(wrk_dir_handle,(3a)".",
                FS_DIR_ACCESS_MODE_NONE,&err);

dir_handle =FSDir_Open(wrk_dir_handle,(3b)"./",
                FS_DIR_ACCESS_MODE_NONE,&err);

file_handle =FSFile_Open(wrk_dir_handle,(4)"../file.txt",
                 FS_FILE_ACCESS_MODE_WR,&err);

file_handle =FSFile_Open(wrk_dir_handle,(5)"./dir01/file01.txt",
                 FS_FILE_ACCESS_MODE_WR,&err);

file_handle =FSFile_Open(wrk_dir_handle,(6)"dir01/file02.txt",
                 FS_FILE_ACCESS_MODE_WR,&err);
```

(1) Opening the root directory relative to the directory "sd:vol0/test_dir" because two dots ".." points upwards in the hierarchy.

(2) Opening a non-root directory ("dir01") inside the directory "sd:vol0/test_dir" because one dot "." represents the current directory.

(3) Opening the current non-root directory ("test_dir"). The directory "sd:vol0/test_dir" associated to the working directory handle is considered the current directory to open because of the second argument set to DEF_NULL.

(3a) Opening the current non- root directory ("test_dir"). Specifying "." for the second argument is equivalent to DEF_NULL presented in (3). Here one dot "." represents the current directory.

(3b) Opening the current non- root directory ("test_dir"). Specifying "./" for the second argument is equivalent to "." presented in (3a) and DEF_NULL presented in (3).

(4) Opening a file (file.txt) in the root directory relative to the directory "sd:vol0/test_dir" because two dots ".." points upwards in the hierarchy.

(5) Opening a file (file01.txt) in a non-root directory ("dir01") located itself inside the directory "sd:vol0/test_dir" because one dot "." represents the current directory.

(6) Opening a file (file02.txt) in a non-root directory ("dir01") located itself inside the directory "sd:vol0/test_dir". Note in this example that the dot "." is not necessary.

In the code snippet above, all relative paths containing the **slash '/'** separator could be replaced by a **backslash '|'** separator. Thus the different paths would become:"..\"

".\dir01"

"..\file.txt"

".\dir01\file01.txt"

"dir01\file02.txt"

The code snippet would create the following tree structure:

sd:vol0 \test_dir working directory \dir01 \file01.txt \file02.txt \file.txt

Managing Removable Media

This section explains the possible methods to handle removable media and optionally fixed media.

- Handling Synchronous Connection and Disconnection
- Handling Asynchronous Connection and Disconnection
- SCSI Removable Media
  - Detection Regardless of Media Poll Task Presence
  - Automatic Media Naming
  - Micrium OS USB Host's Hub Task

Handling Synchronous Connection and Disconnection

The synchronous approach to media connection and disconnection handling is simple: presume that the media exists, is connected, and relies on returned errors (most likely an IO error) to detect disconnections. This approach is more naturally fitted for fixed media but can be used for removable media as well. Consider an application where a USB key is guaranteed to remain physically connected, for instance. In this context, it would make sense to use the synchronous approach even though the USB device is fundamentally a removable media.

Handling Asynchronous Connection and Disconnection

The storage sub-module provides an asynchronous notification mechanism that alleviates the burden of managing removable media connections and disconnections. See File System Basic Concepts for a detailed discussion on what constitutes removable media. The notification mechanism relies on the internal Media Poll task. This task will poll all media periodically to detect a connection or disconnection. A set of pre-init and post-init functions allows you to configure the Media Poll task: FSStorage_ConfigureMediaPollTaskStk(), FSStorage_PollTaskPeriodSet() and FSStorage_PollTaskPrioSet().The asynchronous notification is available when FS_STORAGE_CFG_MEDIA_POLL_TASK_EN is set to DEF_ENABLED in fs_storage_cfg.h. In order to receive the notifications, all you need to do is register two callbacks, one for connection and one for disconnection, using the FSStorage_ConfigureMediaConnCallback() API. Upon media connection or disconnection, the Media Poll task will call the corresponding application callback and pass the media handle as an argument. Listing - Media Connection / Disconnection Modification Mechanism Usage in the *Managing Removable Media* page shows a minimal example of the notification mechanism where the user-provided callbacks merely print a message on the standard output each time a connection or disconnection event occurs. In a real-world context, the callback would typically post another task responsible for the actual event handling.

Even though the asynchronous approach to media connection handling has been designed with removable media in mind, this approach could be used for fixed media as well. In this case, the connection callback would be called immediately after the module initialization and the disconnection callback would never be called. This can be convenient if you wish to write generic code that can handle all possible media, disregarding their removable or non-removable nature. Refer to section Media for more details about the media characterization.

Listing - Media Connection / Disconnection Motification Mechanism Usage

```
#define  APP_FS_CFG_MAX_MEDIA_NAME_LEN  20u

void  MyConnCb (FS_MEDIA_HANDLE  media_handle)
{
    RTOS_ERR  err;
    CPU_CHAR  name[APP_FS_CFG_MAX_MEDIA_NAME_LEN + 1u];      /* +1 for null character terminating string. */

    FSMedia_NameGet(media_handle, name, sizeof(name), &err);
    if (err.Code == RTOS_ERR_NONE) {
        printf("Media '%s' connected!\r\n", name);
    }
}

void  MyDisconnCb (FS_MEDIA_HANDLE  media_handle)       (1)
{
    RTOS_ERR  err;
    CPU_CHAR  name[APP_FS_CFG_MAX_MEDIA_NAME_LEN + 1u];      /* +1 for null character terminating string. */

    FSMedia_NameGet(media_handle, name, sizeof(name), &err);
    if (err.Code == RTOS_ERR_NONE) {
        printf("Media '%s' disconnected!\r\n", name);
    }
}

/* ... */

RTOS_ERR  err;

FSStorage_ConfigureMediaConnCallback(MyConnCb, MyDisconnCb);

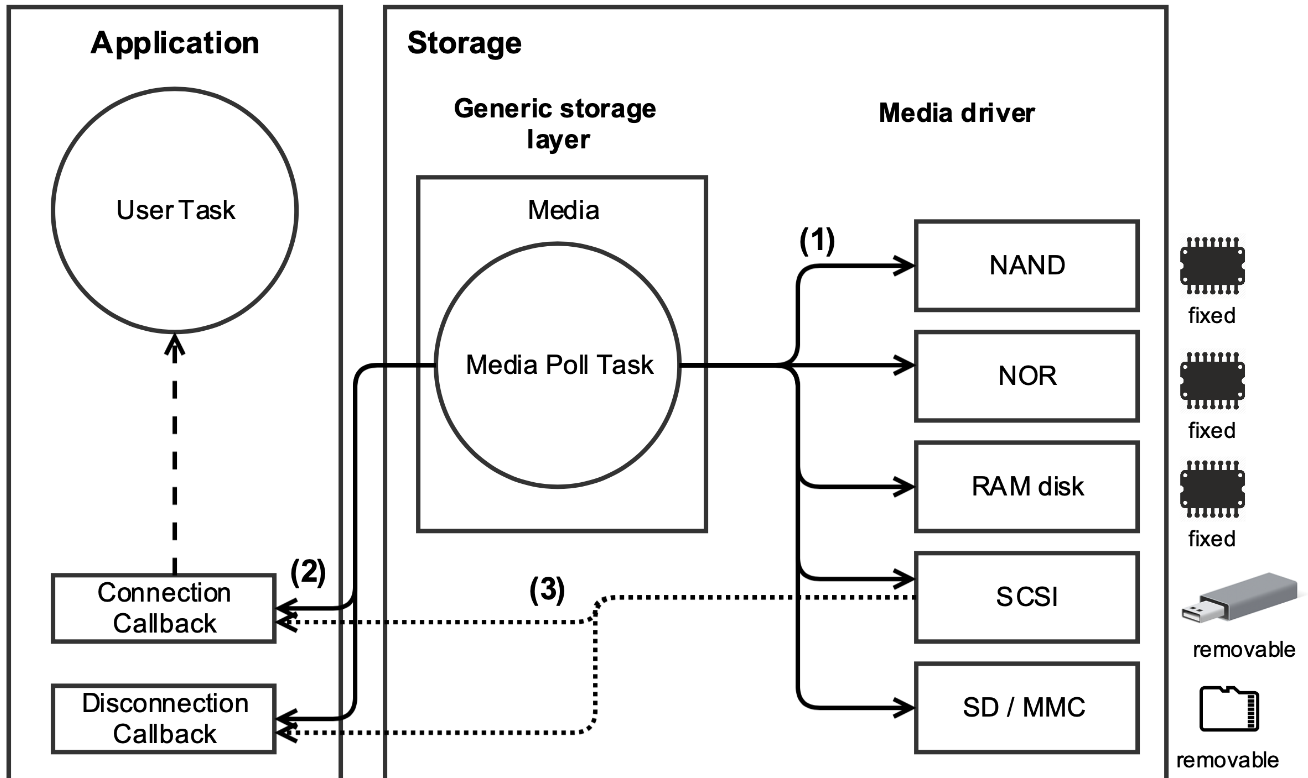
FSStorage_Init(&err)
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}

/* From this point on, if a removable media is inserted or removed,
   a corresponding message will be printed on the standard output. */
```

(1) In the disconnection callback, you could signal a task of your application that a removable media has been disconnected. If this task closes an opened volume with FSVol_Close(), you could get an error returned indicating an I/O error (RTOS_ERR_IO). In that case, the error should be considered as a normal situation. Indeed if the FAT journaling module is enabled via the configuration FS_FAT_CFG_JOURNAL_EN, when you close the volume, the File System stack deletes the journal file. Since the removable media (SCSI or SD) has been disconnected, the delete access on the physical media fails and an I/O error is returned.

Figure - Media Poll Task in the *Managing Removable Media* page summarizes the usage of the Media Poll task for the asynchronous media detection.

Figure - Media Poll Task



(1) The Media Poll task interrogates periodically each opened block device to know if a connection or disconnection has occurred.

(2) If a connection or disconnection has been detected, the Media Poll task will call the appropriate application callback. In general, the connection callback will notify a user task about the media connection. For fixed media, the user application is notified only once about the connection. The fixed media disconnection cannot happen of course.

(3) The SCSI media driver has the ability to notify the application about a device presence without using the Media Poll task's help. Refer to the section Detection Regardless of Media Poll Task Presence for more details.

SCSI Removable Media

Detection Regardless of Media Poll Task Presence

As explained in the section Media , which describes the media's physical nature, SD card and SCSI media types are considered to be removable media. Yet there is a difference between an SD card and a SCSI device: their persistence. From the File System perspective, an SD card is said to be persistent, as opposed to a SCSI device, which is non-persistent. This difference makes it possible for the SCSI storage driver to be independent from the Media Poll task presence. That is, regardless of the Media Poll task configuration (#define FS_STORAGE_CFG_MEDIA_POLL_TASK_EN in fs_storage_cfg.h), the user callbacks will always be called upon SCSI device connection or disconnection. If you want to perform file accesses on a SCSI device, you must rely on the application connection callbacks called by File System. On the other hand, if the Media Poll task is disabled, an SD card will not be detected asynchronously. You can still detect an SD card synchronously if the SD card is always inserted in the SD slot.

Automatic Media Naming

As described in the section Media , a media name is assigned by you to NAND, NOR, RAM disk and SD so that they are registered in the Platform Manager and known from Micrium OS File System for properly configuring them. SCSI devices are a bit different because the media name is internally assigned by the SCSI media driver upon the device's connection. While you have full control on the media name for NAND, NOR, RAM disk and SD, the SCSI media name follows this format:

"scsiXXYY" where XX is a unique ID assigned to the connected SCSI device and YY is the logical unit number (a SCSI device is composed of one to N logical units).

When your application connection callback is called upon the SCSI device connection's detection, you have access to the SCSI media handle as shown in Listing - Media Connection / Disconnection Modification Mechanism Usage in the *Managing Removable Media* page. If this media handle is passed to the function FSMedia_NameGet(), you will get for example the SCSI device with the name "scsi0001".

Micrium OS USB Host's Hub Task

The SCSI media driver works with Micrium OS USB Host to transport the SCSI commands over USB to memory sticks and memory card readers. Micrium OS USB Host has an internal task called the hub task responsible for all the hub-related events such as device connection, disconnection, etc. When a SCSI device connects to an embedded USB host, the USB Host hub task calls a connection callback provided by the SCSI media driver. Then this SCSI media driver's callback calls your application connection callback. This situation occurs only if the Media Poll task is disabled (#define FS_STORAGE_CFG_MEDIA_POLL_TASK_EN set to DEF_DISABLED in fs_storage_cfg.h). Within your application callback, you could call any functions that perform a file operation such as FSFile_Open(), FSFile_Rd(), etc. When calling these type of functions, several media read/write accesses can be done by the SCSI media driver passing through the different layers of Micrium OS USB Host stack. At this point, all the functions call stack is done within the USB Host hub task's context. For that specific case, the hub task must have a *stack size large enough* to accommodate worst case scenario of call stack. The default hub task's stack size is 768 elements as indicated in Table - Micrium OS Internal tasks in the *Appendix A - Internal Tasks* page. This stack size value should be sufficient for most typical cases. In general, your application connection callback would typically post another task responsible for the actual connection event handling. This other task would perform any file operations and would avoid any hub task's overflow issue.

Using File Buffers

Since Micrium OS File System is essentially I/O-bound (that is, it spends most of the time waiting for I/Os to complete), a rapid series of multiple small accesses can cause performance to degrade drastically. Micrium OS File System's internal cache helps to mitigate these performance penalties by combining multiple small accesses to a given logical block into a single access. However, the cache cannot combine accesses beyond the size of a logical block, which may create a bottleneck on media such as SD cards. Indeed, SD cards tend to have high access latency, which can be mitigated using dedicated multi-block read and write operations.

This is essentially what file buffers are designed for. In the case of write operations, chunks of data smaller than the file buffer are accumulated until the buffer is full, at which point the accumulated data is written all at once. For read operations, the whole file buffer is pre-loaded with file contents following the chunk of data that is being read, such that subsequent chunk of data are read directly from the internal buffer.

- Assigning a File Buffer
- Unassigning a File Buffer
- Flushing a File Buffer

Assigning a File Buffer

File buffers are assigned on a per-file descriptor basis (see File System Basic Concepts for details regarding file and directory descriptors) using FSFile_BufAssign(). The process of assigning a file buffer is illustrated by Listing - Assigning a File Buffer in the *Using File Buffers* page. Notice the presence of the FS_FILE_BUF_MODE_RD_WR flag that indicates that the file buffer should be used both for accumulating written data and pre-loading read data. The FS_FILE_BUF_MODE_RD and FS_FILE_BUF_MODE_WR flags may be used to indicate that the file buffer should be used only for read or only for write operations respectively.

Listing - Assigning a File Buffer

```
FS_FILE_HANDLE  file_handle;
CPU_INT08U      file_buf[4096];
RTOS_ERR        err;

/* Assumes that 'file_handle' points to a file previously opened with FSFile_Open(). */

/* Assign the file buffer for read and write operations. */
FSFile_BufAssign(file_handle,
                 (void *)file_buf,
                 FS_FILE_BUF_MODE_RD_WR,       /* Buffer access mode: data buffered for reads & writes.*/
                 4096u,                        /* Buffer size in bytes.                                 */
                 &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

Unassigning a File Buffer

A file buffer is tied to a given file descriptor for as long as the file descriptor remains opened. In other words, the only way of unassigning a file buffer is to close the corresponding file descriptor using FSFile_Close(). Once the corresponding file descriptor is closed, the buffer can be safely reused with another file descriptor. Sharing a buffer between multiple simultaneously opened descriptors may lead to severe corruption.

Flushing a File Buffer

Flushing a file buffer involves writing accumulated data to the media and/or discarding pre-loaded data. A file buffer is implicitly flushed whenever:

1. The buffer is full
2. The associated file descriptor is closed

You can also flush a file buffer explicitly using FSFile_BufFlush().

Using Working Directories

This section explains the high-level file system API that you can use to manage working directories on a specific volume.

Micrium OS File System supports the notion of working directories, which allows you to use relative paths as opposed to absolute paths. If you open a file or a directory, you can pass only a file or directory name instead of providing an absolute file or directory path. The relative path is consequently shorter than an absolute path, and can be as short as a simple file or directory name. Moreover, the File System also offers a working directory bound to a specific task. This functionality allows a given task to work in the same directory with the high-level file system API without having to constantly use absolute paths, which may require the task to know which volume it operates on. The task simply can use single file names. Refer to section File and Directory Names and Paths for more details about usage of paths in Micrium OS File System.

- Opening and Closing a Working Directory
- Functions Operating with a Working Directory
- Obtaining Information Associated to a Working Directory
- Task Working Directory

Opening and Closing a Working Directory

A working directory can be opened using FSWrkDir_Open() and closed using FSWrkDir_Close(), as shown in Listing - Opening a Working Directory in the *Using Working Directories* page.

Listing - Opening a Working Directory

```
FS_WRK_DIR_HANDLE  wrk_dir_handle;
RTOS_ERR           err;

/* Assume that a volume named "test_vol" has previously been opened. */

/* Open a working directory mapped to the directory 'test_vol/test_dir'. */
                             (1)
wrk_dir_handle = FSWrkDir_Open(FS_WRK_DIR_NULL,              /* NULL working dir means absolute path used.  */
                              "test_vol/test_dir",            /* Absolute path to directory to map.          */
                              &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}

/* The working directory handle can now be used for opening    */
/* a file or directory in the directory 'test_vol/test_dir'.   */

/* Close working directory. */
FSWrkDir_Close(wrk_dir_handle, &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

(1) The first argument of FSWrkDir_Open() is a handle to the current working directory. The principle is that a new working directory can be created relative to the current working directory. This argument can have two possible values:

- Null handle
- Non-null handle

If a null working directory handle is passed, then the path to the directory to map must be **absolute**. On the other hand, a non-null working directory handle implies a **relative** path to the directory to map. If you pass a null handle, we recommend that you use the macro FS_WRK_DIR_NULL.

This approach to using working directories offers many possibilities. For instance, creation of the working directory can be "cascaded." That is, a working directory handle can help create a new working directory relative to an existing working directory. Let's consider the following simple tree structure:

```
test_vol
  \—test_dir   ← working directory
     \—subdir
        \—subsubdir
           \—subsubsubdir ← working directory
```

Listing - Cascaded Working Directory in the *Using Working Directories* page shows different ways of creating working directories with one of them illustrating cascaded working directories. The code snippet below exhibits the creation of working directories mapped to the directories "test_dir" and "subsubsubdir". Note: error handling in this listing has been omitted for clarity.

Listing - Cascaded Working Directory

```
FS_WRK_DIR_HANDLE  wrk_dir_handle1;
FS_WRK_DIR_HANDLE  wrk_dir_handle2;
RTOS_ERR           err;

/\* Assume that a volume named "test_vol" has previously been opened. \*/

                                                    (1)
wrk_dir_handle1 = FSWrkDir_Open(FS_WRK_DIR_NULL,          /\* NULL working directory means absolute path used.  \*/
                        "test_vol/test_dir",      /\* Absolute path to directory to map.                \*/
                        &err);
                                                    /\* ------------------ VERSION 1 --------------------\*/
                                                    (2)
wrk_dir_handle2 = FSWrkDir_Open(FS_WRK_DIR_NULL,          /\* NULL working directory means absolute path used.  \*/
                                                    /\* Absolute path to directory to map.                \*/
                        "test_vol/test_dir/subdir/subsubdir/subsubsubdir",
                        &err);

/\* The working directory handles can now be used for opening          \*/
/\* file or directory in the directory 'test_dir' and 'subsubsubdir'. \*/

FSWrkDir_Close(wrk_dir_handle2, &err);

                                                      /\* ------------------ VERSION 2 ----------------- \*/
                                                      (3)
wrk_dir_handle2 = FSWrkDir_Open(wrk_dir_handle1,          /\* Non-NULL working dir means relative path used. \*/
                        "subdir/subsubdir/subsubsubdir",  /\* Relative path to directory to map.        \*/
                        &err);

/\* The working directory handles can now be used for opening          \*/
/\* file or directory in the directory 'test_dir' and 'subsubsubdir'. \*/

/\* Close working directory. \*/
FSWrkDir_Close(wrk_dir_handle2, &err);
FSWrkDir_Close(wrk_dir_handle1, &err);
```

(1) Open a working directory mapped to the directory 'test_dir'. The macro FS_WRK_DIR_NULL is used to pass a null working directory. In that case, the second argument of FSWrkDir_Open() must be an **absolute** path.

(2) Open a working directory mapped to the directory 'subsubsubdir'. Here again the macro FS_WRK_DIR_NULL is used. Consequently, the absolute path to 'subsubsubdir' must be provided. In that case, the working directory creation to 'subsubsubdir' is independent from 'test_dir' working directory handle.

(3) Open a working directory mapped to the directory 'subsubsubdir'. In this version, the 'test_dir' working directory handle is used instead of FS_WRK_DIR_NULL. The second argument must be **relative** to the working directory specified as first argument. In that case, the working directory creation to 'subsubsubdir' has been cascaded with the 'test_dir' working directory. It would be possible to create another working directory cascaded to the 'subsubsubdir' working directory. And so on.

Functions Operating with a Working Directory

The functions listed in Table - Functions Operating with a Working Directory in the *Using Working Directories* page all require one or two working handles as arguments. As soon as a working directory handle argument is present, the following argument will be a path to a file, a directory or an entry. As explained in the details of Listing - Opening a Working Directory in the *Using Working Directories* page , the working directory handle can be null or not. If null, the file/directory/entry path argument must be absolute. If non-null, the path is relative to the directory mapped to the working directory handle.

Table - Functions Operating with a Working Directory

Category	Functions	Number of Working Directory Handles as Argument	Example of Function Call
File	FSFile_Open()	1	See Creating and Accessing Files and Directories
File	FSFile_Copy()	2	-
Directory	FSDir_Open()	1	See Creating and Accessing Files and Directories
Entry	FSEntry_Create()	1	-
Entry	FSEntry_AttribSet()	1	-
Entry	FSEntry_Del()	1	-
Entry	FSEntry_Query()	1	-
Entry	FSEntry_Rename()	2	-
Entry	FSEntry_TimeSet()	1	-

This function FSEntry_TimeSet() is deprecated and will be replaced by sl_fs_entry_time_set() in a next release.

Obtaining Information Associated to a Working Directory

The File System working directory API offers some utility functions that may be handy for your application to retrieve information associated to a given working directory handle. There are two functions of that type:

- FSWrkDir_PathGet() : this function provides the absolute path associated to the given working directory handle
- FSWrkDir_VolGet(): this function provides the volume handle associated to the given working directory handle

Listing - Working Directory Utility Functions in the *Using Working Directories* page presents an example of usage of these functions.

Listing - Working Directory Utility Functions

```
FS_VOL_HANDLE      vol_handle;
FS_WRK_DIR_HANDLE  wrk_dir_handle;
CPU_CHAR           path_buf[50u];
CPU_SIZE_T         wrk_dir_path_len;
FS_VOL_INFO        vol_info;
RTOS_ERR           err;

/* Assume that a volume named "test_vol" has previously been opened. */

/* Open a working directory mapped to the directory 'test_dir'. */
wrk_dir_handle = FSWrkDir_Open(FS_WRK_DIR_NULL,              /* NULL working dir means absolute path used.  */
                               "test_vol/test_dir/subdir",   /* Absolute path to directory to map.          */
                               &err);
if (err.Code != RTOS_ERR_NONE) {
  /* An error occurred. Error handling should be added here. */
}
                                        /* ---------------- UTILITY 1 ---------------- */
wrk_dir_path_len = FSWrkDir_PathGet(wrk_dir_handle,          (1)
                                    path_buf,          /* Buffer receiving absolute path associated to handle. */
                                    50u,               /* Buffer size in bytes.                                */
                                    &err);
if (err.Code != RTOS_ERR_NONE) {
  /* An error occurred. Error handling should be added here. */
}
                                        /* Path displayed: "test_vol/test_dir/subdir". */
printf("Path associated to working directory handle: %s\r\n", path_buf);

                                        /* ---------------- UTILITY 2 ---------------- */
vol_handle = FSWrkDir_VolGet(wrk_dir_handle);                (2)
if (FS_VOL_HANDLE_IS_NULL(vol_handle)) {                     (3)
  /* An error occurred. Error handling should be added here. */
```

```
}/* Get information about the volume. */FSVol_Query(vol_handle,&vol_info,/* Structure receiving information about volume. */
          DEF_NULL,/* Optional information about volume file system.*/&err)if(err.Code != RTOS_ERR_NONE){/* An error occurred. Error handling should be added here. */}printf("Volume info:\r\n");printf("- Number of sectors used: %u\r\n", vol_info.UsedSecCnt);printf("- Number of sectors free: %u\r\n", vol_info.FreeSecCnt);printf("- Total number of sectors: %u\r\n", vol_info.TotSecCnt);/* Close working directory. */FSWrkDir_Close(wrk_dir_handle,&err);if(err.Code != RTOS_ERR_NONE){/* An error occurred. Error handling should be added here. */}
```

(1) Besides providing the path associated to a given working directory handle, the function FSWrkDir_PathGet() returns the number of characters composing the absolute path. If you need to allocate dynamically the buffer receiving the path, you can use FSWrkDir_PathGet() to obtain *only* the number of characters in the path in the following way:

```
wrk_dir_path_len = FSWrkDir_PathGet(wrk_dir_handle,
                                    DEF_NULL,
                                    0u,
                                    &err);
```

You just need to provide a null buffer pointer and specify 0 bytes for the buffer size.

(2) FSWrkDir_VolGet() returns the volume handle associated to a given working directory handle. With the volume handle, you can for instance obtain some volume's information with the function FSVol_Query().

(3) An utility macro called FS_VOL_HANDLE_IS_NULL() allows you to verify if the volume handle is valid, that is not a null handle.

Task Working Directory

When the configuration FS_CORE_CFG_TASK_WORKING_DIR_EN is enabled, an extended working directory API is available. This extended API is composed of three functions:

- FSWrkDir_Get(): obtain the current working directory handle associated to the current task.
- FSWrkDir_TaskBind(): bind a given directory to the current task. The directory becomes the current working directory.
- FSWrkDir_TaskUnbind(): unbind the current task from the current working directory.

The task's working directory allows the task to perform file operations inside a specific working directory. From this working directory, the task can create a new tree structure composed of files and directories without using absolute paths. For instance, absolute paths require the volume name, and the task does not always know the exact volume name in which it works. For example, you may have an application which creates a task as soon as a user logins to your file system product. Upon login, the user has a working space to perform file operations in a default directory. You can use FSWrkDir_TaskBind() to bind the user task to a default directory representing the current working directory. From this moment on, the user can do anything in this working directory using a working directory handle and relative paths to files and directories.

The current working directory binding with the current task is done by saving the working directory handle in the Micrium OS Kernel task registers (see the section Task Registers in the Task Management Kernel Services section).

Listing - Task Working Directory in the *Using Working Directories* page illustrates the usage of the task working directory. Let's assume that a user task has been created with the default directory being "test_vol/test_dir/subdir".

Listing - Task Working Directory

```
/* Assume that a volume named "test_vol" has previously been opened. */

static  void  App_TaskUser (void  *p_arg)
{
    CPU_CHAR          *p_default_dir_path;
    FS_WRK_DIR_HANDLE  cur_wrk_dir_handle;
    CPU_CHAR           path_buf[100u];
    RTOS_ERR           err;

                                          (1)
    cur_wrk_dir_handle = FSWrkDir_Get();            /* Obtain current working directory handle.         */

    (void)FSWrkDir_PathGet(cur_wrk_dir_handle,       (2)
                 path_buf,               /* Buffer receiving absolute path associated to handle. */
```

```
&err);if(err.Code != RTOS_ERR_NONE){/* An error occurred. Error handling should be added here. */}/* Display path: "test_vol" i.e. root directory.
*/printf("Task's current working directory: %s\r\n", path_buf);

  p_default_dir_path = (CPU_CHAR *)p_arg;/* Default directory path: "test_vol/test_dir/subdir".  */FSWrkDir_TaskBind(FS_WRK_DIR_NULL,/* NULL
working directory means absolute path used.    */
                p_default_dir_path,/* Absolute path to directory to bind to task.       */&err);if(err.Code != RTOS_ERR_NONE){/* An error occurred.
Error handling should be added here. */}while(1){/* Task performs any file operations in the working directory 'subdir'. */}}
```

(1) If the task calls FSWrkDir_Get() for the first time, the working directory handle will be a null handle. A null handle refers to the volume root directory.

(2) For your convenience, the macro FS_WRK_DIR_CUR can be used as the first argument of FSWrkDir_PathGet() in the example. FS_WRK_DIR_CUR represents a direct call to FSWrkDir_Get(). Thus the code at the beginning of the task becomes:

```
(void)FSWrkDir_PathGet(FS_WRK_DIR_CUR,
                 path_buf,
                 100u,
                 &err);
```

> The functions FSWrkDir_Get(), FSWrkDir_TaskBind() and FSWrkDir_TaskUnbind() apply to the task calling those functions. A given task cannot bind a working directory to another task.

Posix

The best-known API for accessing and managing files and directories is specified within the POSIX standard (IEEE Std 1003.1). The basis of some of this functionality – in particular buffered input/output – lies in the ISO C standard (ISO/IEC 9899), though many extensions provide new features and clarify existing behaviors. Functions and macros that are prototyped in four header files are of particular importance:

- stdio.h: standard buffered input/output (fopen(), fread(), etc), operating on FILE objects.
- dirent.h: directory accesses (opendir(), readdir(), etc), operating on DIR objects.
- unistd.h: miscellaneous functions, including working directory management (chdir(), getcwd()), ftruncate() and rmdir().
- sys/stat.h: file statistics functions and mkdir().

The Micrium OS File System provides a POSIX-like API based on a subset of the functions in these four header files. To avoid conflicts with the user compilation environment, files, functions and objects are renamed:

- All functions begin with 'fs_'. For example, fopen() is renamed fs_fopen(), opendir() is renamed fs_opendir(), getcwd() is renamed fs_getcwd(), etc.
- All objects begin with 'FS_'. So fs_fopen() returns a pointer to an FS_FILE and fs_opendir() returns a pointer to a FS_DIR.
- Some argument types are renamed. For example, the second and third parameters of fs_fread() are typed fs_size_t to avoid conflicting with other size_t definitions.

For your convenience, you may still use the POSIX native file system functions name thanks to preprocessor directives that map the Micrium OS File System POSIX-like functions name to the POSIX ones. For instance, the function fopen() can be used in your application. When compiling your code, the preprocessor will resolve the occurrence of fopen, which is a macro, by fs_open. The File System POSIX maps also some data types defined in the four standard header files previously listed to some File System data types. For instance, the data type FILE is resolved to FS_FILE. The File System POSIX-like API is available to your application by including the file fs_core_posix.h.

In your application file, the header file fs_core_posix.h can be included in addition to the standard header file stdio.h. For example, you may want to display some messages with the function printf() in addition to file operations. In order to avoid some compilation errors/warnings with certain toolchains, the proper inclusion of stdio.h is managed within fs_core_posix.h. In that case, you shall follow these two requirements:

- Include only fs_core_posix.h in your source file. stdio.h must be omitted
- Place fs_core_posix.h as the first file in your header files inclusion block to ensure that other header files do not accidentally include stdio.h which could cancel the stdio.h handling centralized in fs_core_posix.h. |

Supported Functions

The supported POSIX functions are listed in Table - POSIX API functions in the *Supported Functions* page. These are divided into five groups:

- The functions that operate on file objects (FS_FILE) are grouped under file access (or simply file) functions. An application stores information in a file system by creating a file or appending new information to an existing file. At a later time, this information may be retrieved by reading the file. Other functions support these capabilities; for example, the application can move to a specified location in the file or query the file system to get information about the file. Refer to the section Managing Files for more details about this group.
- The entries within a directory can be traversed using the directory access (or simply directory) functions, which operate on directory objects (FS_DIR). The name and properties of the entries are returned within a struct fs_dirent structure. Refer to the section Managing Directories for more details about this group.
- A separate set of file operations (or entry) functions manages the files and directories available on the system. Using these functions, the application can create, delete and rename files and directories. Refer to the section Managing Entries for more details about this group.
- A group of functions is the working directory functions. They allow to attach a working directory to a task and use relative paths. Refer to the subsection Working Directory for more details about this group.
- The final group is the time functions. They allow to manipulate the time that can be associated to files and directories.

Table - POSIX API functions

Function	POSIX Equivalent	File System / Sleeptimer Equivalent
File Access		
fs_fopen()	fopen()	FSFile_Open()
fs_fclose()	fclose()	FSFile_Close()
fs_fread()	fread()	FSFile_Rd()
fs_fwrite()	fwrite()	FSFile_Wr()
fs_ftruncate()	ftruncate()	FSFile_Truncate()
fs_feof()	feof()	FSFile_IsEOF()
fs_ferror()	ferror()	FSFile_IsErr()
fs_clearerr()	clearerr()	FSFile_ErrClr()
fs_fgetpos()	fgetpos()	FSFile_PosGet()
fs_fsetpos()	fsetpos()	FSFile_PosSet()
fs_fseek()	fseek()	FSFile_PosSet()
fs_ftell()	ftell()	FSFile_PosGet()
fs_rewind()	rewind()	FSFile_PosSet()
fs_fileno()	fileno()	-
fs_fstat()	fstat()	FSFile_Query()
fs_flockfile()	flockfile()	FSFile_Lock()
fs_ftrylockfile()	ftrylockfile()	FSFile_TryLock()
fs_funlockfile()	funlockfile()	FSFile_Unlock()
fs_setbuf()	setbuf()	FSFile_BufAssign()
fs_setvbuf()	setvbuf()	FSFile_BufAssign()
fs_fflush()	fflush()	FSFile_BufFlush()
Directory Access		
fs_opendir()	opendir()	FSDir_Open()
fs_closedir()	closedir()	FSDir_Close()
fs_readdir_r()	readdir_r()	FSDir_Rd()

Function	POSIX Equivalent	File System / Sleeptimer Equivalent
Entry		
fs_mkdir()	mkdir()	FSEntry_Create()
fs_remove()	remove()	FSEntry_Del()
fs_rename()	rename()	FSEntry_Rename()
fs_rmdir()	rmdir()	FSEntry_Del()
fs_stat()	stat()	FSEntry_Query()
Working Directory		
fs_chdir()	chdir()	FSWrkDir_TaskBind()
fs_getcwd()	getcwd()	FSWrkDir_PathGet()
Time		
fs_asctime_r()	asctime_r()	sl_sleeptimer_convert_date_to_str()*
fs_ctime_r()	ctime_r()	sl_sleeptimer_convert_time_to_date()*
fs_ctime_r()	ctime_r()	sl_sleeptimer_convert_date_to_str()*
fs_localtime_r()	localtime_r()	sl_sleeptimer_convert_time_to_date()*
fs_mktime()	mktime()	sl_sleeptimer_convert_date_to_time()*

*See Silicon Labs Documentation for more details.

Managing Files

- File State
- Opening, Reading and Writing Files
- Getting or Setting the File Position
- Configuring a File Buffer
- Diagnosing a File Error
- Atomic File Operations Using File Lock

File State

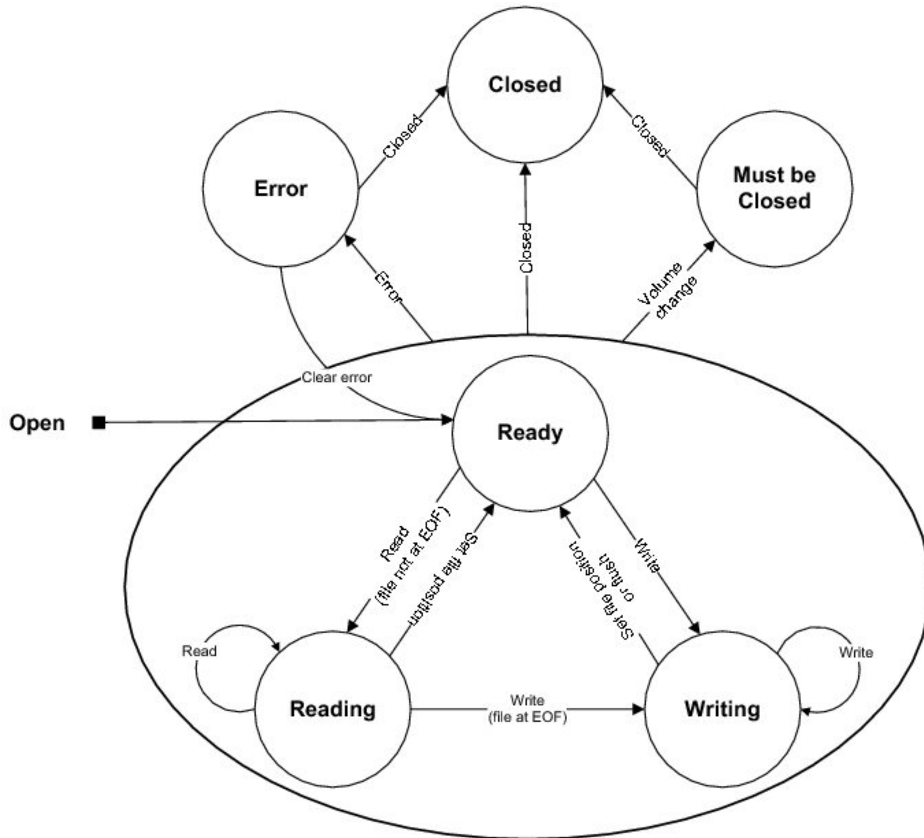
The file access functions provide an API for performing operations on any file stored in a volume's file system.

When a file is opened, a file object pointer called FS_FILE is returned. This object pointer can be passed as an argument for all file access functions, and the file object preserves information about the actual file and the state of the file access (see Figure - File State Transitions in the *Managing Files* page). The file access state includes:

- The file position (the next place data will be read/written)
- Error conditions
- The state of any file buffer, if file buffering is enabled

As data is read from or written to a file, the file position is incremented by the number of bytes transferred from/to the volume. The file position may also be manipulated directly by the application using the position set function (fs_fsetpos()). And the current absolute file position may be obtained with the position get function (fs_fgetpos()), which can be later used with the position set function.

Figure - File State Transitions



A file maintains flags that signal any errors encountered in the immediately-previous file access. Subsequent accesses will fail (under certain conditions outlined here) unless these flags are explicitly cleared (using fs_clearerr()).

There are actually two flags. One indicates whether the file encountered the end-of-file (EOF) during the previous access; if this flag is set, file reads (but not file writes) will fail. The other flag indicates device errors, and no subsequent file access will succeed (except file close) until this flag is cleared. The functions fs_ferror() and fs_feof() can be used to get the state of device error and EOF conditions, respectively.

If file buffering is enabled (FS_CORE_CFG_FILE_BUF_EN is DEF_ENABLED), then input/output buffering can be used to increase the efficiency of file reads and writes. A buffer can be assigned to a file using fs_setbuf() or fs_setvbuf(). The buffer's content can be flushed to the storage device using fs_fflush().

If a file is shared between several tasks in an application, a file lock can be employed to guarantee that a series of file operations are executed atomically. fs_flockfile() (or its non-blocking equivalent fs_ftrylockfile()) acquires the lock for a task (if it does not already own it). Accesses from other tasks will be blocked until fs_funlockfile() is called. This functionality is available if FS_CORE_CFG_FILE_LOCK_EN is DEF_ENABLED.

Opening, Reading and Writing Files

When an application needs to access a file, it must first open it using fs_fopen() as shown in Listing - Posix File Open in the *Managing Files* page.

Listing - Posix File Open

```
FS_FILE  *p_file;

p_file = fs_fopen("vol0/file.txt",                  /* Absolute path to file.                  */
                  "w+");                            /* File access mode: read/write/truncate/create. */
if (p_file == (FS_FILE *)0) {
    /* An error occurred. Error handling should be added here. */
}
```

The return value of this function should always be verified as non-NULL before the application proceeds to access the file. The first argument of this function is the path of the file. If working directories are disabled (FS_CORE_CFG_TASK_WORKING_DIR_EN set to DEF_DISABLED), this must be the absolute file path starting with a volume name (refer to section File and Directory Names and Paths for more details about absolute paths). The second argument of this function is a string indicating the file access mode. It must be one of the strings shown in Table - fs_fopen() File Access Mode Strings Interpretations in the *Managing Files* page. Note that in all instances, the 'b' (binary) option has no affect on the behavior of file accesses.

Table - fs_fopen() File Access Mode Strings Interpretations

fs_fopen() File Access Mode String	Read	Write	Truncate	Create	Append
"r" or "rb"	Yes	No	No	No	No
"w" or "wb"	No	Yes	Yes	Yes	No
"a" or "ab"	No	Yes	No	Yes	Yes
"r+" or "rb+" or "r+b"	Yes	Yes	No	No	No
"w+" or "wb+" or "w+b"	Yes	Yes	Yes	Yes	No
"a+" or "ab+" or "a+b"	Yes	Yes	No	Yes	Yes

After a file is opened, any of the file access functions that are valid for that mode can be called. The most commonly used functions are fs_fread() and fs_fwrite(), which reads or writes a certain number of 'items' from a file:

Listing - Posix File Read

```
int         cnt;
CPU_INT08U  *p_buf;
FS_FILE     *p_file;

cnt = fs_fread(p_buf,                   /* Pointer to buffer receiving data from file.    */
           1,                     /* Size, in bytes, of each item to be read.       */
           100,                   /* Number of items, each one with size of 1 byte. */
           p_file);               /* Pointer to file descriptor.                    */
if (cnt == 0) {
   /* An error occurred. Error handling should be added here. */
}

cnt = fs_fwrite(p_buf,                  /* Pointer to buffer containing data to file.     */
            1,                    /* Size, in bytes, of each item to be write.      */
            100,                  /* Number of items, each one with size of 1 byte. */
            p_file);              /* Pointer to file descriptor.                    */
if (cnt == 0) {
   /* An error occurred. Error handling should be added here. */
}
```

The return value, the number of items read (or written), should be less than or equal to the third argument. If the operation is a read, this value may be less than the third argument for one of two reasons. First, the file could have encountered the end-of-file (EOF), which means that there is no more data in the file. Second, the device could have been removed, or some other error could have prevented the operation. To diagnose the cause, the fs_feof() function should be used. This function returns a non-zero value if the file has encountered the EOF.

Once the file access is complete, the file *must* be closed. If your application fails to close files, then the File System could run out of resources, such as file objects.

An example of reading a file is shown below:

Listing - Posix File Read Example

```
void  App_Fnct (void)
{
   FS_FILE       *p_file;
```

```
  fs_size_t       cnt;
  unsigned  char   buf[50];
  int             eof;
  int             err;

  p_file =fs_fopen("vol0/file.txt","r");/* Open file in read only.                    */if(p_file !=(FS_FILE *)0){/* If file is opened, read from file.           */do{
        cnt =fs_fread(&buf[0],/* Pointer to buffer receiving data from file.   */1,/* Size, in bytes, of each item to be read.       */sizeof(buf),/*
Number of items, each one with size of 1 byte. */
                       p_file);if(cnt >0){printf("Read %d bytes.\r\n", cnt);}}while(cnt >=sizeof(buf));

      eof =fs_feof(p_file);/* Check for EOF.                          */if(eof !=0){(1)printf("Reached EOF.\r\n");}else{
          err =fs_ferror(p_file);/* Check for error.                          */if(err !=0){(2)/* An error occurred. Error handling should be added here.
*/}}fs_fclose(p_file);/* Close file.                          */}else{/* An error occurred. Error handling should be added here. */}...}
```

(1) To determine whether a file read terminates because of reaching the EOF or a device error/removal, the EOF condition should be checked using fs_feof().

(2) In most situations, either the EOF or the error indicator will be set on the file if the return value of fs_fread() is smaller than the buffer size. Consequently, this check is unnecessary.

Getting or Setting the File Position

Another common operation is getting or setting the file position. The functions fs_fgetpos() and fs_fsetpos() allow the application to 'store' a file location, continue reading or writing the file, and then go back to that position at a later time. An example of using file position get and set is given in Listing - File Position Set and Get Example in the *Managing Files* page

Listing - File Position Set and Get Example

```
void  App_Fnct (void)
{
    FS_FILE    *p_file;
    fs_fpos_t   pos;
    int         err;

    p_file = fs_fopen("vol0/file.txt", "r");                  /* Open file in read only.                */
    if (p_file == (FS_FILE *)0) {
        /* An error occurred. Error handling should be added here. */
    }

    /* Read some data from file. Each read will advance file position. */

    err = fs_fgetpos(p_file, &pos);                           /* Save current file position.            */
    if (err != 0) {
        /* An error occurred. Error handling should be added here. */
    }

    /* Read some data from file. Each read will advance file position. */

    err = fs_fsetpos(p_file, &pos);                           /* Set file to previously saved position.   */
    if (err != 0) {
        /* An error occurred. Error handling should be added here. */
    }

    /* Read data from saved file position. Each read will advance file position. */

    fs_fclose(p_file);                                        /* When finished, close file.             */
}
```

Configuring a File Buffer

In order to increase the efficiency of file reads and writes, input/output buffering capabilities are provided. Without an assigned buffer, reads and writes will be performed immediately within fs_fread() and fs_fwrite(). When a buffer is assigned,

data will always be read from or written to the buffer; device access will only occur once the file position moves beyond the window represented by the buffer.

fs_setbuf() or fs_setvbuf() assigns a buffer to a file. The buffer content can be flushed to the storage device with fs_fflush(). If a buffer is assigned to a file that was opened in update (read/write) mode, then a write may be followed by a read only if the buffer has been flushed (by calling fs_fflush() or a file positioning function). A read may be followed by a write only if the buffer has been flushed, except when the read encountered the end-of-file, in which case a write may happen immediately. The buffer is flushed automatically when the file is closed.

> File buffering is particularly important when data is written in small chunks to a medium with slow write time or limited endurance. An example is NOR flash, or even NAND flash, where write times are much slower than read times, and the lifetime of device is constrained by limits on the number of times each block can be erased and programmed.

Listing - File Buffer Usage Example

```
static  CPU_INT08U  App_FileBuf[512u];                    /* Define file buffer.                */

void  App_Fnct (void)
{
    FS_FILE  *p_file;
    int       err;

    p_file = fs_fopen("vol0/file.txt", "w");              /* Open file in read/write.           */
    if (p_file == (FS_FILE *)0) {
        /* An error occurred. Error handling should be added here. */
    }
                                                          (1)
    err = fs_setvbuf(p_file,
                     (void *)App_FileBuf,                 /* File buffer to assign to open file.   */
                     FS__IOFBF,                           /* Data buffered for reads & writes.     */
                     sizeof(App_FileBuf));                /* Size of buffer in bytes.              */
    if (err != 0) {
        /* An error occurred. Error handling should be added here. */
    }

    /* Write some data to file. Each write will add data to the file buffer. */
                                                          (2)
    fs_fflush(p_file);                                    /* Make sure data is written to file.    */
    if (err != 0) {
        /* An error occurred. Error handling should be added here. */
    }
                                                          (3)
    fs_fclose(p_file);                                    /* When finished, close file.            */
}
```

(1) The buffer *must* be assigned immediately after opening the file. An attempt to set the buffer after read or writing the file will fail.

(2) While it is not necessary to flush the buffer before closing the file, some applications may want to make sure at certain points that all previously written data is actually stored on the device before writing more.

(3) When closing the file, the file buffer will be emptied by writing its content to the disk.

Diagnosing a File Error

As explained in the section File State , the file maintains flags that indicate errors encountered in the immediately-previous file access. You may use the functions fs_ferror() and fs_feof() to identify a file error. Listing - File Error Diagnose Example in the *Managing Files* page illustrates a usage of those functions (Note: This code snippet is illustrative. No write should be done after opening a file in read-only, of course).

Listing - File Error Diagnose Example

```
void  App_Fnct (void)
{
    unsigned  char   data1[50];
    int              cnt;
    int              file_is_err;
    FS_FILE         *p_file;

    p_file = fs_fopen("vol0/file.txt", "r");                    /* Open file in read only.              */
    if (p_file == (FS_FILE *)0) {
        /* An error occurred. Error handling should be added here. */
    }
                                                  (1)/* This write will fail              */
                                                     /* because read-only file.           */
    cnt = fs_fwrite((void *)data1,
                    sizeof(unsigned char),
                    sizeof(data1),
                    p_file);
                                                  (2)/* Error indicator should be set      */
                                                     /* because of failed write.          */
    file_is_err = fs_ferror(p_file);
    APP_RTOS_ASSERT_CRITICAL((file_is_err != 0), DEF_FAIL);

    fs_clearerr(p_file);                                 (3) /* Clr err indicator.              */

    file_is_err = fs_ferror(p_file);                       (4)
    APP_RTOS_ASSERT_CRITICAL((file_is_err == 0), DEF_FAIL);
                                                  (5)/* This read should pass.            */
    cnt = fs_fread((void *)data1,
                   sizeof(unsigned char),
                   sizeof(data1),
                   p_file);
    if (cnt == 0) {
        /* An error occurred. Error handling should be added here. */
    }

    fs_fclose(p_file);                                  /* Close file.                      */
    ...
}
```

(1) The file write will fail because the file was opened in read-only mode.

(2) Calling fs_ferror() should indicate that the *error indicator* flag has been set because of the previous failed file write.

(3) The error indicator flag is reset by a call to the function fs_clearerr().

(4) The error indicator flag is checked again. The assertion should confirm that the flag has been reset.

(5) A file read is attempted and is expected to succeed, since the error indicator flag is reset. Note that if a file read had been tried before the call to fs_clearerr(), the read would have failed because the flag would have still been set.

Atomic File Operations Using File Lock

If a file is shared between several tasks in an application, a file lock can be employed to guarantee that a series of file operations are executed atomically. The function fs_flockfile() (or its non-blocking equivalent fs_ftrylockfile()) acquires the lock for a task (if it does not already own it). Accesses from other tasks will be blocked until fs_funlockfile() is called.

Each file has a lock count associated with it. This allows nested calls by a task to acquire a file lock; each of those calls must be matched with a call to fs_funlockfile(). Listing - File Lock Usage Example in the *Managing Files* page shows how the file lock functions can be used.

Listing - File Lock Usage Example

```
void  App_Fnct (void)
{
    unsigned  char   data1[50];
    unsigned  char   data2[10];
    int              cnt;
    FS_FILE         *p_file;

    /* A file has been open with fs_fopen(). */
                                                 (1)
    fs_flockfile(p_file);                        /* Lock file.                    */

                                                 /* Write data atomically.        */
    cnt = fs_fwrite(data1,
                    1,
                    sizeof(data1),
                    p_file);
    if (cnt == 0) {
        /* An error occurred. Error handling should be added here. */
    }
    cnt = fs_fwrite(data2,
                    1,
                    sizeof(data2),
                    p_file);
    if (cnt == 0) {
        /* An error occurred. Error handling should be added here. */
    }

    fs_funlockfile(p_file);                      /* Unlock file.                  */
    ...
}
```

(1) fs_flockfile() will block the calling task until the file is available. If the task must perform other operations without waiting for the file to be available, the non-blocking function fs_ftrylockfile() can be used.

Managing Directories

The directory access functions provide an API for reading the entries within a directory. The function fs_opendir() initiates this procedure, and each subsequent call to fs_readdir_r() returns information about a particular entry in a structure fs_dirent, until all entries have been examined. The fs_closedir() function releases any file system structures and locks.

Listing - Directory Content Listing Example in the *Managing Directories* page shows an example using the directory access functions to list the files and sub-directories in a directory.

Listing - Directory Content Listing Example

```
void  App_Fnct (void)
{
    FS_DIR           *p_dir;
    struct  fs_dirent   dirent;
    struct  fs_dirent  *p_dirent;
    int                 err;

    p_dir = fs_opendir("vol0/dir_test");               /* Open the directory.                          */
    if (p_dir == (FS_DIR *)0) {
        /* An error occurred. Error handling should be added here. */
    }
                                                  /* Read first directory entry. Usually 'dot' entry. */
    err = fs_readdir_r(p_dir,                         /* Directory handle.                         */
                       &dirent,                       /* Structure that will contain mainly entry name.   */
                       &p_dirent);                    /* Pointer to structure containing entry info.      */
    if (err != 0) {
        /* An error occurred. Error handling should be added here. */
    }

    if (p_dirent != (struct fs_dirent *)0) {                  /* If NULL, directory is empty.            */

        while (p_dirent != (struct fs_dirent *)0) {            /* While last entry not reached.           */
                                                               /* Display entry (file or directory) name.  */
            printf("entry found: %s\r\n", p_dirent->d_name);

                                                               /* Read next entry int the directory table. */
            err = fs_readdir_r(p_dir,
                               &dirent,                        (1)
                               &p_dirent);
            if (err != 0) {
                /* An error occurred. Error handling should be added here. */
            }
                                                               (2)
        }
    }

    fs_closedir(p_dir);                                /* Close the directory.                  */
    ...
}
```

(1) The second argument fs_readdir_r(), is a pointer to a struct fs_dirent, which has two members. The field d_name[] is the most useful and holds the name of the entry. For more information about the structure fs_dirent, refer to the notes section of fs_readdir_r().

(2) If you need more information about an entry such as file or directory, read and/or write attributes, time attributes, etc, you may call fs_stat().

Working Directory

All file and directory paths often use an **absolute** path by specifying an explicit volume as illustrated below:

Listing - Absolute paths

```
p_file = fs_fopen("sdcard:vol0\\file.txt", "r");            /* File on explicitly-specified volume.      */

p_dir = fs_opendir("sdcard:vol0\\dir10");                   /* Directory on explicitly-specified volume. */
```

> All absolute path variations described in the subsection File and Directory Names and Paths for the File System native API can be used also for the POSIX API.

The file and directory path can also be **relative**. Paths are then specified relative to the working directory of the current task. In order to use relative paths, you must enable the configuration FS_CORE_CFG_TASK_WORKING_DIR_EN in fs_core_cfg.h. In that case, you will be able to open a file or directory relatively to the working directory, as shown below:

Listing - Special paths

```
p_file1 = fs_fopen("file.txt", "r");            /* File in working directory.           */
p_file2 = fs_fopen("..\\file.txt", "r");        /* File in parent of working directory.    */

p_dir1 = fs_opendir("dir11");                   /* Directory in working directory.       */
p_dir2 = fs_opendir("..\\dir20");               /* Directory in parent of working directory. */
```

The two standard special path components are supported:

- The path component ".." moves to the parent directory of the current working directory.
- The path component "." makes no change; essentially it means the current working directory.

Prior to using a relative path in fs_fopen() or fs_opendir(), you must first use the function fs_chdir() to set the working directory. You can retrieve the current working directory attached to a task by calling the function fs_getcwd().

All relative path variations described in the subsection File and Directory Names and Paths for the File System native API can be used also for the POSIX API.

Managing Entries

The entry access functions provide an API for performing single operations on file system entries (files and directories), such as renaming or deleting a file. Each of these operations is atomic. Consequently, in the absence of device access errors, either the operation will have completed or no change to the storage device will have been made upon function return.

- Create a new directory: fs_mkdir()
- Delete an existing file or directory: fs_remove()
- Rename an existing file or directory: fs_rename()

Optional FAT Journaling

The File System's FAT journaling module is an optional feature that provides protection against unexpected power failures during file system operations.

In the FAT file system, cluster allocation information is stored separately from file data and metadata. This means that file operations that make even simple changes to the content of a file are *non-atomic*. Atomic operations either complete immediately or (in the event of an error) do not happen at all; they never have a state halfway in between. The repercussions of this can be innocuous (for example, wasted disk space) or very serious (corrupted directories, corrupted files, and data loss). In order to prevent this kind of data corruption, you can use the File System FAT journaling module.

- What Journaling Guarantees
- How Journaling Works
- How To Use Journaling
  - Usage Advice
- Limitations of Journaling
  - Removable Media
  - Journaling and API-Level Atomicity
  - Journaling and Device Drivers

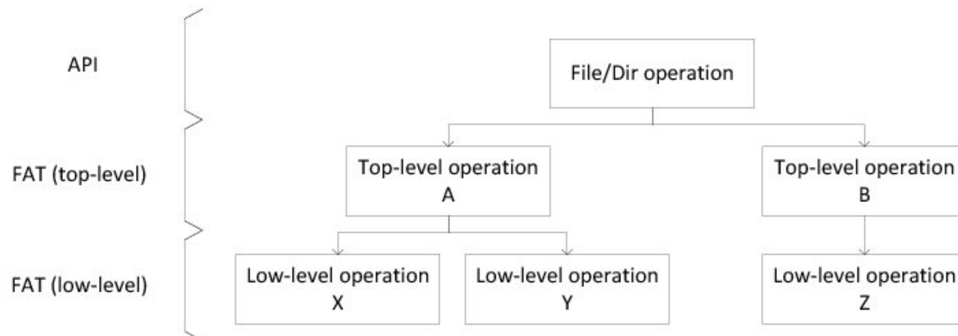
What Journaling Guarantees

In short, journaling guarantees that the file system remains consistent on disk. Journaling prevents the directory hierarchy, file names, file metadata and cluster allocation information from becoming corrupted in case of an untimely interruption (such as a power failure or application crash). However, while journaling protects the integrity of the file system, it does not necessarily protect your data integrity (that is the content of your files). For example, if the application crashes while a write operation is being performed, the data could end up only partially written to the media (see Journaling API Level Atomicity).

How Journaling Works

In order to understand how the journaling module works, you should first understand how API-level operations relate to the underlying FAT layer operations. As seen in Figure - Relation Between API and FAT Layer Operations in the *Optional FAT Journaling* page, an API-level operation is made of one or more top-level FAT operations, which, in turn, are made of one or more low-level FAT operations.

Figure - Relation Between API and FAT Layer Operations



Let's take as an example a file rename operation. The API-level rename operation involves one top-level FAT rename operation and the following low-level FAT sub-operations:

1. Create a directory entry that uses the new file name.
2. Update the newly created directory entry so that it is identical to the original one.
3. Remove the original directory entry.

Without journaling, a failure occurring during the rename operation could leave the file system in any of the following corrupted states:

1. The original directory entry is intact, but orphaned LFN entries remain due to a partially-created directory entry.
2. The new directory entry now exists (creation has been completed), but orphaned LFN entries remain because the original directory entry was only partially deleted.
3. Two directory entries now exist, both pointing to the same data: one containing the original name and another one containing the new name.

Using the journaling module, any of the previous corrupted states would be either rolled back or completed when the volume is remounted. This is possible because, prior to performing any low-level FAT operation, the journaling system logs recovery information in a special file called the journal file. By reverting or completing successive underlying low-level FAT operations, the journaling module also allows top-level FAT operations to be reverted or completed, thus making them atomic. In our previous example, the journaled rename operation could have only one of the two following outcomes:

1. The original directory entry is intact and everything appears as if nothing had happened.
2. The new directory entry has been created and the original one has been completely deleted, so that the file has been cleanly renamed.

How To Use Journaling

The FAT journaling module is active when the configuration constant FS_FAT_CFG_JOURNAL_EN is set to DEF_ENABLED.

The journaling system can be started on a per-volume basis. There are no specific functions required to start and stop the journal operations. Starting journaling is done automatically when the volume is opened with FSVol_Open(). It ensures that the FAT journal is ready when the first file operations are performed. In the same manner, stopping journaling is done when the volume is closed with FSVol_Close().

When the volume is mounted, a special hidden file called "ucfs.jnl" is created in the root directory. Each time a top-level FAT operation is journaled, this file will be written. When the volume closes, the file "ucfs.jnl" is deleted. If a power failure occurs, the "ucfs.jnl" file will still exist when the volume is re-opened with FSVol_Open(), and the file will be consulted to replay some journaled operations to maintain the file system's integrity on disk.

Journal operations (that is logging top-level FAT operations in the file "ucfs.jnl") take place automatically when calling certain high-level file system functions. The list below shows the high-level functions that involves journal operations:

- FSVol_Open()
- FSVol_Close()
- FSDir_Open()
- FSFile_Open()
- FSFile_Close()
- FSFile_Rd()
- FSFile_Wr()
- FSFile_PosSet()
- FSFile_Copy()
- FSFile_Truncate()
- FSFile_BufFlush()
- FSEntry_Create()
- FSEntry_Rename()
- FSEntry_Del()
- FSEntry_AttribSet()
- FSEntry_TimeSet() (this function is deprecated and will be replaced by sl_fs_entry_time_set() in a next release)

The file system operations that are logged by the journaling module are only those that perform write operations in the FAT table or in a directory table. Read operations will never be logged, as they do not modify the FAT table or a directory table. You may notice that the list above includes the functions FSFile_Rd () and FSFile_PosSet () . The journaled operations involved in these functions relate to file buffer management, which may require flushing the file buffer, and thus possibly updating the FAT table or a directory table. They do not relate to read operations.

Usage Advice

There are a few aspects that you should be aware of to benefit from FAT journaling protection.

1. In addition to enabling FS_FAT_CFG_JOURNAL_EN , we highly recommend that you also enable FS_CORE_CFG_ORDERED_WR_EN. This constant enables ordered write operations within the File System cache module. This is particularly important, as it ensures that the cache blocks containing journaled data are flushed to disk before the non-journaled data. Otherwise in the event of a power failure or application crash, the journal module may not properly replay the journaled data when the volume is re-mounted, leaving the file system possibly corrupted. Refer to section Core Configuration for more details about FS_CORE_CFG_ORDERED_WR_EN.
2. You may want to enable the volume auto-sync functionality to ensure that the journal file content is always kept up-to-date, in case of a power failure. The journaled data are cached within the File System cache module during write operations. The cache buffers are flushed to the physical media when a flush point is encountered. Enabling the volume auto-sync functionality activates automatic flush points at the end of certain file operations (for instance, file write and file truncate). This is an additional precaution to ensure that the journaled data are synced regularly by the File System cache module. It relieves your application from having to flush the cache manually using FSVol_Sync() after certain write operations. If volume auto-sync is not used, the journal may not be able to replay all journaled entries because some of them had still been cached when a power failure occurred. The volume auto-sync feature is enabled as an option flag of FSVol_Open(). Refer to the section Syncing a Volume for more details about the auto-sync.
3. The number of blocks associated with a cache instance can affect the reliability of journal operations. As mentioned, the journaled data are cached, and the cache module is flushed when encountering a flush point. There are explicit flush points enabled by the volume auto-sync, and there are also implicit flush points. Implicit flush points are where the File System cache module run out of cache blocks and needs to evict a block. As the number of cache blocks increases, the implicit flush points come at longer intervals, and consequently so do the synchronizations of the journal data to the physical media. Thus you may want to reduce the number of cache blocks for a cache instance so that the journal file on-disk is updated frequently. Refer to the page Creating and Assigning a Cache for more details about how to set the number of cache blocks with FSCache_DfltAssign() or FSCache_Create().

> Point #1 above has no impact on write performance. For points #2 and #3, activating the volume auto-sync feature and/or reducing the number of cache blocks may have an negative impact on write performance, since the flush points can occur more frequently. Applying points #2 and #3 may result in a tradeoff between performance and metadata integrity. In general, activating the FAT journal module still remains an useful option to ensure the file system metadata integrity while maintaining acceptable performance.

Limitations of Journaling

When properly used, the journaling system provides reliable protection for your file system metadata. But to ensure proper operation, you should be aware of certain limitations, and follow the corresponding recommendations. Failure to observe these recommendations could reduce the benefits of using the journaling system, and potentially lead to file system corruption.

Journaling and FAT16/32 Removable Media

The journaling module recovery process is based on the assumption that the file system has not been modified or altered since the failure occurred. Therefore, mounting a journaled volume on a host such as Windows, Linux or Mac OS should be avoided as much as possible. If it must be done, you must first make sure that the volume has been cleanly unmounted from your embedded application by using FSVol_Close() so that the journal file has been deleted.

Journaling and API-Level Atomicity

While the journaling system provides atomicity for top-level FAT-layer operations, it does not provide atomicity for all API-level operations. Most of the time, an API-level file system operation will result in a single top-level FAT operation being performed (see How Journaling Works). In that case, the API-level operation is guaranteed to be atomic. For example, a call to FSEntry_Rename() will result in a single FAT rename operation being performed (assuming that renaming is not cross-volume), which is guaranteed to be atomic.

By comparison, a call to FSFile_Truncate() will likely result in many successive top-level FAT operations being performed. Therefore, the API-level truncate operation is not guaranteed to be atomic. The following API-level functions are atomic:

- FSDir_Open()
- FSFile_Open()
- FSFile_Close()
- FSFile_Rd()
- FSFile_PosSet()
- FSFile_BufFlush()
- FSEntry_Create()
- FSEntry_Rename() (atomic if the source and the destination are on the same volume)
- FSEntry_Del()
- FSEntry_AttribSet()
- FSEntry_TimeSet() (this function is deprecated and will be replaced by sl_fs_entry_time_set() in a next release)

Non-atomic API level operations are listed below and Table - Non-Atomic API Level Operations in the *Optional FAT Journaling* page describes the possible interruption side effects for each non-atomic function.

- FSEntry_Rename()
- FSFile_Copy()
- FSFile_Wr()
- FSFile_Truncate()

Table - Non-Atomic API Level Operations

API-Level Operation	API-Level Function	Possible Interruption Side Effects
Entry rename	FSEntry_Rename() with the destination being on a different volume than the source.	The destination file size could end up being less than the source file size.
Entry copy	FSFile_Copy() regardless of source and destination volumes location.	If the destination file exists and the flag to overwrite the existing file is set, the existing file could be deleted, but the copy not completed. The destination file size could end up being smaller than the source file size.

API-Level Operation	API-Level Function	Possible Interruption Side Effects
File write (data appending)	FSFile_Wr() with file buffers enabled.	The file size could be changed to any value between the original file size and the new file size.
File write (data overwriting)	FSFile_Wr() with or without file buffers.	If existing data contained in a file is overwritten with new data, data at overwritten locations could end up corrupted.
File truncate	FSFile_Truncate() with file buffers enabled.	The file size could be changed to any value between the original file size and the new file size.

Journaling and Device Drivers

Data can be lost in case of unexpected reset or power-failure in either the File System Core layer or in the Storage layer. Your entire system is fail-safe only if *both* layers are fail-safe. The FAT journaling add-on makes the File System Core layer fail-safe. Some of the File System storage drivers are guaranteed to provide fail-safe sector operations; this is the case with the NOR and NAND flash drivers. For other drivers (SD and SCSI), the fail-safety of the sector operations depends on the underlying hardware.

Media-Specific Operations

This section describes the media-specific functions which can be used to get additional controls and information about a media.

- Opening and Closing Media
- NAND
- NOR
- RAM Disk
- SD
- SCSI

Opening and Closing Media

When using some media-specific functions, you must first open media with the function FS*<MEDIA>_Open() (where MEDIA can be NAND, NOR, SCSI or SD). You then get a media-specific handle that you can pass to other media-specific functions. When you have finished with the media operations, you can close it by calling the function FS*<MEDIA>_Close(). Listing - Media-Specific Open/Execute/Close Generic Sequence in the *Media-Specific Operations* page illustrates, in a generic way, the media-specific open/execute/close sequence.

Listing - Media-Specific Open/Execute/Close Generic Sequence

```
FS_MEDIA_HANDLE          media_handle;
FS_<MEDIA-NAME>_HANDLE   <media-name>_handle;
RTOS_ERR                 err;

/* Assume 'media_handle' has been obtained from FSMedia_Get(). */
                                                   (1)

/* Open a media device. */
<media-name>_handle = FS_<MEDIA>_Open(media_handle, &err);      (2)
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}

/* Perform any media-specific operations. */

/* Open the media device. */
FS_<MEDIA>_Close(<media-name>_handle, &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

(1) FSMedia_Get() allows you to retrieve a media handle associated with a given media name.

(2) A media device (NAND, NOR, SCSI or SD) is opened. You get a specific media handle allowing you to execute an operation on the opened media.RAM Disk does not define open and close functions.

NAND

The NAND media driver can be tailored by the following specific configurations defined in fs_storage_cfg.h (refer to the section NAND-Specific Options for a detailed description of each configuration):

- FS_NAND_CFG_AUTO_SYNC_EN
- FS_NAND_CFG_UB_META_CACHE_EN
- FS_NAND_CFG_DIRTY_MAP_CACHE_EN
- FS_NAND_CFG_UB_TBL_SUBSET_SIZE
- FS_NAND_CFG_RSVD_AVAIL_BLK_CNT
- FS_NAND_CFG_MAX_RD_RETRIES
- FS_NAND_CFG_MAX_SUB_PCT
- FS_NAND_CFG_DUMP_SUPPORT_EN

The NAND media driver offers the specific functions listed in Table - NAND-Specific Functions in the *Media-Specific Operations* page.

Table - NAND-Specific Functions

Function	Handle Type to Provide	Description	Notes
FS_NAND_Open()	FS_MEDIA_HANDLE	Opens NAND media	
FS_NAND_Close()	FS_NAND_HANDLE	Closes NAND media	
FS_NAND_BlkErase()	FS_NAND_HANDLE	Erases a physical block	A NAND flash block is composed of a number of pages. A page is equivalent to a sector, and so a page is the smallest programmable unit within a NAND. Thus, erasing a block is equivalent to erasing several sectors.
FS_NAND_ChipErase()	FS_NAND_HANDLE	Erases an entire NAND chip	Erase counts for each block will also be erased, affecting wear-leveling mechanism.
FS_NAND_Dump()	FS_NAND_HANDLE	Dumps raw NAND contents in multiple data chunks through a user-supplied callback function	Used mainly for debugging purposes.
FS_NAND_FTL_ConfigureLowParams()	FS_MEDIA_HANDLE	Configures NAND Flash Transaction Layer (FTL) parameters	

NOR

The NOR media driver provides one specific configuration defined in fs_storage_cfg.h (refer to the section NOR-Specific Options for a detailed description of this configuration):

- FS_NOR_CFG_WR_CHK_EN

The NOR media driver offers the specific functions listed in Table - NOR-Specific Functions in the *Media-Specific Operations* page.

Table - NOR-Specific Functions

Function	Handle Type to Provide	Description	Notes
FS_NOR_Open()	FS_MEDIA_HANDLE	Opens NOR media	
FS_NOR_Close()	FS_NOR_HANDLE	Closes NOR media	
FS_NOR_Rd()	FS_NOR_HANDLE	Reads data from NOR media	
FS_NOR_Wr()	FS_NOR_HANDLE	Writes data to NOR media	
FS_NOR_BlkErase()	FS_NOR_HANDLE	Erases a physical block	A NOR flash block is composed of a number of sectors. Thus erasing a block is equivalent to erasing several sectors.
FS_NOR_ChipErase()	FS_NOR_HANDLE	Erases an entire NOR chip	
FS_NOR_XIP_Cfg()	FS_NOR_HANDLE	Configures NOR flash and (Quad) SPI controller in XIP (eXecute-In-Place) mode	More details about XIP can be found on the page NOR eXecute In Place .
FS_NOR_Get()	FS_NOR_HANDLE	Gets a NOR handle from a given media handle	
FS_NOR_BlkCntGet()	FS_NOR_HANDLE	Gets the number of blocks on a NOR	
FS_NOR_BlkSizeLog2Get()	FS_NOR_HANDLE	Gets the base-2 logarithm of a NOR block size	
FS_NOR_FTL_ConfigureLowParams()	FS_MEDIA_HANDLE	Configures NOR Flash Transaction Layer (FTL) parameters	
FS_NOR_FTL_LowCompact()	FS_BLK_DEV_HANDLE	Low-level compact a NOR device	A NOR block device must be opened with FSBlkDev_Open() prior to using this function.
FS_NOR_FTL_LowDefrag()	FS_BLK_DEV_HANDLE	Low-level defragment a NOR device	A NOR block device must be opened with FSBlkDev_Open() prior to using this function.

RAM Disk

In contrast to NAND, NOR, SCSI and SD, the RAM Disk media driver does not offer open/close functions or many other specific functions. The only function is FS_RAM_Disk_Add() which creates a RAM disk region and adds it to the Platform Manager . Listing - Example of RAM Disk Creation in the *Media-Specific Operations* page shows an example of usage of this function.

Listing - Example of RAM Disk Creation

```
/* RAM Disk zone definition.                          */
#define  EX_FS_RAM_RESERVED_SEC_NBR        37u    /* RAM Disk number of sectors reserved to file system.  */
```

```
/* RAM Disk total number of sectors.                    */(1)
#define  EX_FS_RAM_SEC_NBR              (EX_FS_RAM_RESERVED_SEC_NBR +64u)

static  CPU_INT08U  Ex_FS_RAM_Disk[EX_FS_RAM_SEC_SIZE * EX_FS_RAM_SEC_NBR];

FS_RAM_DISK_CFG  disk_info;
RTOS_ERR         err;/* Initialize a RAM disk information structure.        */
disk_info.DiskPtr = (void *)Ex_FS_RAM_Disk;/* Pointer to table simulating a RAM disk zone.        */(2)
disk_info.LbCnt   = EX_FS_RAM_SEC_NBR;/* Number of sectors.                            */
disk_info.LbSize  = EX_FS_RAM_SEC_SIZE;/* Sector size in bytes.                          *//* Add RAM Disk instance to file system.
*/FS_RAM_Disk_Add("ram0",/* String identifying this RAM Disk instance.          */&Ex_FS_RAM_Disk_Cfg,/* Configuration describing RAM area.
*/&err);if(err.Code != RTOS_ERR_NONE){/* An error occurred. Error handling should be added here. */}(3)/* After the RAM disk creation, you must
high-level format it with FS_FAT_Fmt(). */
```

(1) The RAM disk zone will contain sectors reserved to the file system metadata. These reserved sectors are additional to the file data sectors. Thus be aware that the total RAM disk size, that is (FS_RAM_DISK_CFG.LbCnt * FS_RAM_DISK_CFG.LbSize), is not exclusively dedicated to your file data. Refer to the question "When writing a few files of a few KB in size in a RAM_disk, I get the error RTOS_ERR_VOL_FULL " in the File System Troubleshooting page for more details.

(2) DEF_NULL can be passed to the field FS_RAM_DISK_CFG.DiskPtr. In this case, the RAM disk zone is created from the LIB module's heap region. Set the configuration LIB_MEM_CFG_HEAP_SIZE accordingly to accommodate your disk size specified by (FS_RAM_DISK_CFG .LbCnt * FS_RAM_DISK_CFG.LbSize).

(3) A RAM disk zone is a non-volatile memory that must be formatted. You shall use the function FS_FAT_Fmt() to format it as a FAT partition. Refer to the section Formatting an Existing Partition for more details about the high-level format.

SD

The SD media driver provides one specific configuration defined in fs_storage_cfg.h (refer to the section SD-Specific Options for a detailed description of this configuration):

- FS_SD_SPI_CFG_CRC_EN

The SD media driver offers the specific functions listed in Table - SD-Specific Functions in the *Media-Specific Operations* page.

Table - SD-Specific Functions

Function	Handle Type to Provide	Description
FS_SD_Open()	FS_MEDIA_HANDLE	Opens an SD Card or SPI media
FS_SD_Close()	FS_SD_HANDLE	Closes an SD Card or SPI media
FS_SD_CID_Rd()	FS_SD_HANDLE	Read SD CID (Card IDentification number) register
FS_SD_CSD_Rd()	FS_SD_HANDLE	Read SD CSD (Card Specific Data) register
FS_SD_InfoGet()	FS_SD_HANDLE	Get SD information

Listing - Example of SD Information in the *Media-Specific Operations* page shows an example of SD-specific information retrieved with the function FS_SD_InfoGet(). If you are using the Media Poll task (refer to the section Handling Asynchronous Connection and Disconnection for additional information) to detect the insertion of an SD card, you may want to use FS_SD_InfoGet() as shown in this code listing to display information about the connected SD card.

Listing - Example of SD Information

```
FS_MEDIA_HANDLE  media_handle
FS_SD_HANDLE     sd_handle;
FS_SD_INFO       sd_info;
RTOS_ERR         err;

/* Open an SD device. */
sd_handle = FS_SD_Open(media_handle, &err);
if (err.Code != RTOS_ERR_NONE) {
  /* An error occurred. Error handling should be added here. */
```

```
}/* Get information about the SD device. */FS_SD_InfoGet(sd_handle,&sd_info,/* Structure to receive device information.          */&err);if(err.Code != RTOS_ERR_NONE){/* An error occurred. Error handling should be added here. */}/* Display specific information about SD device. */printf("SD info: \r\n");printf("- Sector size       : %d bytes\r\n",  sd_info.BlkSize);printf("- Nbr of sectors    : %d\r\n",        sd_info.NbrBlks);printf("- Max clk freq      : %d Hz\r\n",     sd_info.ClkFreq);printf("- Comm timeout      : %d cycles\r\n", sd_info.Timeout);printf("- Card Type         : %d\r\n",        sd_info.CardType);printf("- High Capacity     : %d\r\n",        sd_info.HighCapacity);printf("- ManufID           : 0x%X\r\n",      sd_info.ManufID);printf("- OEM_ID            : 0x%X\r\n",      sd_info.OEM_ID);printf("- Product serial nbr: %d\r\n",        sd_info.ProdSN);printf("- Product name      : %s\r\n",        sd_info.ProdName);/* Close an SD device. */FS_SD_Close(sd_handle,&err);if(err.Code != RTOS_ERR_NONE){/* An error occurred. Error handling should be added here. */}
```

SCSI

The SCSI media driver offers the specific functions listed in Table - SCSI-Specific Functions in the *Media-Specific Operations* page.

Table - SCSI-Specific Functions

Function	Handle Type to Provide	Description
FS_SCSI_Open()	FS_MEDIA_HANDLE	Opens a SCSI media
FS_SCSI_Close()	FS_SCSI_HANDLE	Closes a SCSI media
FS_SCSI_LU_InfoGet()	FS_SCSI_HANDLE	Get SCSI logical unit information

Listing - Example of SCSI Information in the *Media-Specific Operations* page shows an example of SCSI-specific information retrieved with the function FS_SCSI_LU_InfoGet(). If you are using the Media Poll task (refer to the section Handling Asynchronous Connection and Disconnection for additional information) to detect the insertion of a SCSI device, you may want to use FS_SCSI_LU_InfoGet() as shown in this code listing to display information about the connected SCSI.

Listing - Example of SCSI Information

```
FS_MEDIA_HANDLE  media_handle
FS_SCSI_HANDLE   scsi_handle;
FS_SCSI_LU_INFO  lu_info;
RTOS_ERR         err;

/* Open a SCSI device. */
scsi_handle = FS_SCSI_Open(media_handle, &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}

/* Get information about the SCSI device. */
FS_SCSI_LU_InfoGet(scsi_handle,
                   &lu_info,                    /* Structure to receive device information.   */
                   &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}

/* Display specific information about SCSI device. */
printf("LU info: \r\n");
printf("- Sector size    : %d bytes\r\n", lu_info.SecDfltSize);
printf("- Nbr of sectors: %d\r\n",        lu_info.SecCnt);
printf("- Vendor         : %s\r\n",       lu_info.VendorID_StrTbl);
printf("- Product        : %s\r\n",       lu_info.ProductID_StrTbl);

/* Close a SCSI device. */
FS_SCSI_Close(scsi_handle, &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

NOR eXecute In Place

This page describes how to use the NOR media driver to execute code directly from serial NOR flash memories. The method is called XIP – for eXecute In Place. You may want to read the page NOR Flash XIP to get familiar with the different notions associated with XIP.

The NOR media driver provides a low-level API that allows you to manage an external NOR flash. This API is presented in Table - NOR-Specific Functions in the *Media-Specific Operations* page. You must use the following functions to perform XIP:

- FS_NOR_Rd()
- FS_NOR_Wr()
- FS_NOR_BlkErase()
- FS_NOR_ChipErase()
- FS_NOR_XIP_Cfg()

The above functions should be used in a specific sequence, as described below. In this example, a Quad SPI controller is considered.

1. Write the binary image using the indirect mode of the Quad SPI controller: FS_NOR_ChipErase(), FS_NOR_BlkErase() and FS_NOR_Wr().
2. Read back the binary image using the indirect mode of the Quad SPI controller to verify image integrity: FS_NOR_Rd().
3. Enter XIP mode by configuring the Quad SPI controller in memory-mapped mode, and optionally by configuring the NOR flash device and Quad SPI controller in XIP mode: FS_NOR_XIP_Cfg().
4. Make the processor jump to the QSPI memory location where the image begins. Here, there are two options:
    1. Configure the processor's Program Counter to jump to the QSPI memory start address.
    2. Direct the processor to restart and to boot directly from the QSPI external flash device if the system supports this feature.

At this point, the processor fetches its instructions from the NOR flash memory and executes *in place* the stored image.

If the XIP program subsequently returns processor execution to a program stored in the microcontroller's internal flash, the internal program may exit the XIP mode by calling FS_NOR_XIP_Cfg() again.

Listing - NOR Media Driver API - XIP Enable/Disable in the *NOR eXecute In Place* page shows how to enable the XIP mode, jump to the flash memory for execution, and disable the XIP mode.

Listing - NOR Media Driver API - XIP Enable/Disable

```
#define  NOR_XIP_FLASH_MEM_MAP_START_ADDR           0x04000000

CPU_ADDR    flash_src;
CPU_INT08U  blk_size_log2;
CPU_INT32U  blk_ix_start = 0u;                  /* Image starts at block index #0.                  */
RTOS_ERR    err;

/* A nor handle has been obtained with FS_NOR_Open(). */
/* A binary image has been previously written to the NOR memory. */

blk_size_log2 = FS_NOR_BlkSizeLog2Get(nor_handle, &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}

                                        /* ------------------ ENTER XIP MODE ------------------ */
FS_NOR_XIP_Cfg(nor_handle,
               DEF_YES,                          (1)
               &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
                                        /* ------------------ EXECUTE IMAGE ------------------- */
flash_src = NOR_XIP_FLASH_MEM_MAP_START_ADDR + (blk_ix_start << blk_size_log2);
App_NOR_XIP_ImageExec(flash_src);                (2)

[...]

                                        /* ------------------ EXIT XIP MODE ------------------- */
FS_NOR_XIP_Cfg(nor_handle,
               DEF_NO,                           (3)
               &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

(1) When entering XIP mode with FS_NOR_XIP_Cfg(), two situations can occur depending on whether XIP is supported by the NOR flash memory. If the NOR chip supports XIP, FS_NOR_XIP_Cfg() will activate XIP mode in the NOR chip and in the Quad SPI controller. Also, the Quad SPI controller will use direct mode (that is, memory-mapped mode) so that the flash device is seen internally by the processor. This configuration corresponds to the XIP hardware. If the NOR chip does not support XIP, FS_NOR_XIP_Cfg() will configure only the Quad SPI controller in direct mode. It is the XIP software configuration.

(2) Make the processor jump to a specified flash address where the image starts. The flash address must be in the address range dedicated to the external flash which is memory-mapped in the CPU address space. In this example, there are two directions for the processor jump depending again on whether the NOR flash supports XIP.

- If the NOR does support XIP: the internal flash application may execute a branch instruction with the specified flash address so that the processor continues directly its execution flow from the external NOR flash device. Or the internal flash application code can provoke a processor reset, and during its boot sequence, the processor executes the newly written image from the NOR flash memory.

Some processors are able to execute an image from the external NOR memory during their power-up and boot sequence. In that case, there are two conditions that must be met:

- The processor must be able to boot from an external NOR flash device using a Quad SPI controller.
- The NOR flash device must support XIP mode during the memory power-up.
- If the NOR does not support XIP, there is only one option. The internal flash application executes a branch instruction with the specified flash address so that the processor continues directly its execution flow from the external NOR memory.

The function App_NOR_XIP_ImageExec() is a dummy function created for this code snippet to illustrate a common action that you perform do to process the image. Micrium OS File System does not provide this function.

(3) If the XIP program subsequently returns control to the application code stored in the MCU internal flash, you may want to disable the XIP mode. In that case, FS_NOR_XIP_Cfg() is called again with a flag set to DEF_NO for instance. The function will perform the opposite actions described in note (6).

Prior to enabling the XIP mode, a binary image must be written to the flash device. Listing - NOR Media Driver API - XIP Image Write/Verify in the *NOR eXecute In Place* page shows an example of functions that allow you to write a binary image into the flash memory and verify it.

Listing - NOR Media Driver API - XIP Image Write/Verify

```
#define  NOR_PHY_SUBSEC_SIZE          (4u * 1024u)

CPU_INT08U   *p_buf;
CPU_INT08U   *p_image_start;
CPU_INT08U   *p_image_src;
CPU_INT08U    blk_size_log2;
CPU_INT32U    blk_ix_start;
CPU_INT32U    blk_ix_stop;
CPU_INT32U    blk_ix;
CPU_INT32U    start_pos;
CPU_INT32U    rem_size;
CPU_INT32U    image_blk_cnt;
CPU_INT32U    wr_size;
CPU_INT32U    rd_size;
CPU_INT32U    image_size;
CPU_BOOLEAN   ok = DEF_OK;
RTOS_ERR      err;

/* A nor handle has been obtained with FS_NOR_Open(). */

blk_size_log2 = FS_NOR_BlkSizeLog2Get(nor_handle, &err);
if (err.Code != RTOS_ERR_NONE) {
   /* An error occurred. Error handling should be added here. */
}

                                          /* Alloc test resources.                         */
p_buf = (CPU_INT08U *)Mem_SegAlloc(DEF_NULL,
                                   p_seg,
                                   NOR_PHY_SUBSEC_SIZE,
                                   &err);
if (err.Code != RTOS_ERR_NONE) {
   /* An error occurred. Error handling should be added here. */
}

                                          /* --------- WRITE IMAGE WITH INDIRECT WRITE ---------- */
p_image_start = App_NOR_XIP_ImageGet(&image_size);     (1)
if (p_image_start == DEF_NULL) {
   /* An error occurred. Error handling should be added here. */
}
                                          /* Compute how many blocks needed to hold entire image. */
image_blk_cnt = (image_size + ((1 << blk_size_log) - 1u) / (1 << blk_size_log));
blk_ix_start  = 0u;                       /* Start at the first block of NOR flash.           */
blk_ix_stop   = blk_ix_start + image_blk_cnt;
                                          /* Erase block(s) for proper image writing.         */
for (blk_ix = blk_ix_start; blk_ix < blk_ix_stop; blk_ix++) {
   FS_NOR_BlkErase(nor_handle, blk_ix, &err);        (2)
   if (err.Code != RTOS_ERR_NONE) {
      /* An error occurred. Error handling should be added here. */
   }
}

rem_size    = image_size;
start_pos   = blk_ix_start << blk_size_log2;        /* Get physical start addr relative to flash addr range.*/
p_image_src = p_image_start;

while (rem_size > 0u) {                   /* Write entire image.                           */

   wr_size = DEF_MIN(NOR_PHY_SUBSEC_SIZE, rem_size);
                                          /* Write N bytes to flash device.                */
   FS_NOR_Wr(nor_handle,                          (3)
             p_image_src,
             start_pos,
             wr_size,
             &err);
   if (err.Code != RTOS_ERR_NONE) {
      /* An error occurred. Error handling should be added here. */
   }
```

```
    rem_size    -= wr_size;
    start_pos   += wr_size;
    p_image_src += wr_size;}/* ---- VERIFY IMAGE INTEGRITY WITH INDIRECT READ ----- */
rem_size     = image_size;
start_pos    = blk_ix_start << blk_size_log2;/* Get physical start addr relative to flash addr range.*/
p_image_src = p_image_start;while(rem_size >0u){/* Read entire image.                    */

    rd_size =DEF_MIN(NOR_PHY_SUBSEC_SIZE, rem_size);Mem_Clr(p_buf, rd_size);/* Read N bytes from flash device.
*/FS_NOR_Rd(nor_handle,(4)
           p_buf,
           start_pos,
           rd_size,&err);if(err.Code != RTOS_ERR_NONE){/* An error occurred. Error handling should be added here. */}/* Verify image integrity.
*/
    ok =App_NOR_XIP_ImageVerify(p_buf,
                                p_image_src,
                                rd_size);if(err.Code != RTOS_ERR_NONE){/* An error occurred. Error handling should be added here. */}

    rem_size    -= rd_size;
    start_pos   += rd_size;
    p_image_src += rd_size;}
```

(1) For the illustrative purpose of the NOR media driver API example, a pointer to a buffer holding the entire binary image is retrieved. The source binary image can, of course, be written by chunks in the external NOR memory. You could have for instance:

```
// While entire image not processed
// Get image chunk
// Write image chunk with FS_NOR_Wr()
// Read back this chunk with FS_NOR_Rd() to ensure image chunk is correct
```

The following functions App*NOR_XIP_ImageGet(), App* NOR_XIP_ImageVerify() are dummy functions created for this code snippet to illustrate some common actions that you may do to process the image. Micrium OS File System does not provide these functions.

(2) To ensure the binary image is written properly, erase the required blocks with FS_NOR_BlkErase(). Indeed, in a NOR memory, a write cannot program the bits from 0 to 1. Thus a block erase should be done prior to writing. The erase operation will program the bits from 0 to 1. Moreover, instead of erasing selectively some blocks, you can erase the entire chip by using FS_NOR_ChipErase().

(3) Writing an image chunk with FS_NOR_Wr() will use the indirect mode of the Quad SPI controller. The indirect mode can be seen as a FIFO mode in which the software controls the transfers using several registers. It allows high-throughput data transfer with as little system overhead as possible.

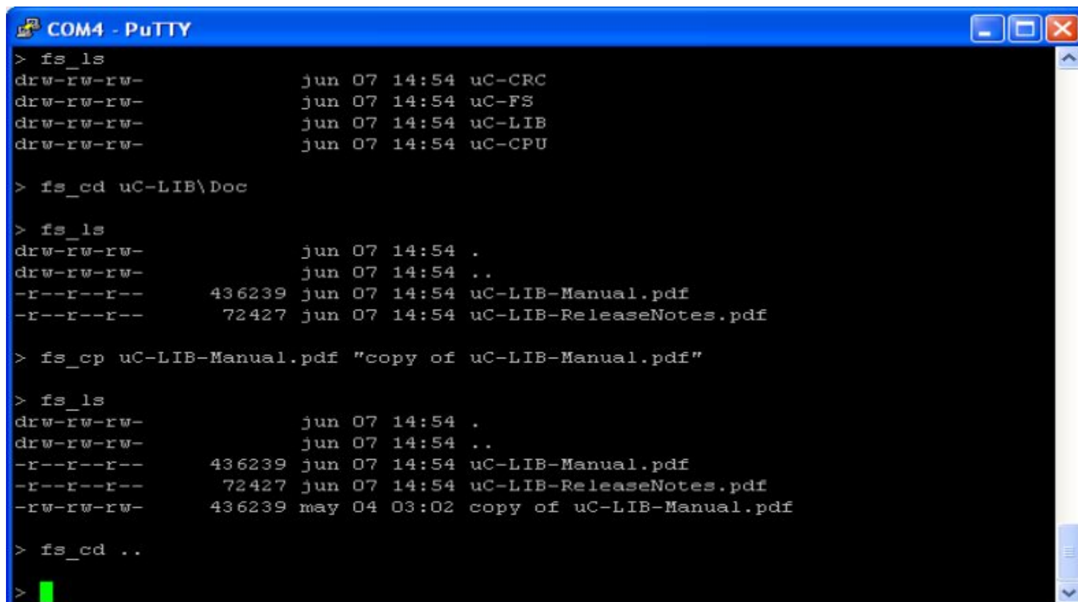
(4) Reading back the image chunk will also use the indirect mode of the Quad SPI controller.

Shell Commands

- Using the Shell Commands
- Commands

The command line interface is a traditional method for accessing the file system on a remote system, or in a device with a serial port (for instance, RS-232 or USB). A group of shell commands, derived from standard UNIX equivalents, are available for Micrium OS File System. These may be used to simply evaluate the File System suite, or to gather debugging information, or even to become part of the primary method of access in your final product.

Figure - File System Shell Command Usage



Using the Shell Commands

In order to use the File System shell commands, you must first initialize:

- The Shell module by calling Shell_Init()
- The File System shell sub-module by calling FSCore_Init()

Usually, the next step would be to add a table containing all the File System shell commands to the Shell module by using the function Shell_CmdTblAdd(). This step is done automatically when you call FSCore_Init(). All the File System shell commands presented in Table - File System Shell Commands in the *Shell Commands* page are available in the Shell module. In order to execute a specific command received from the host via a USB or serial link, your application must call the function Shell_Exec(). When you provide a string describing the command and its arguments (for instance "fs_ls"), the Shell module will parse the command's string and call an associated callback which will decode the command's arguments, if any, and perform the command's action. If the command is required to output information back to the host, an output callback, which was passed as an argument of Shell_Exec(), is called by the Shell module. This output callback, provided by you, will forward the response of the command's execution to the host via the serial or USB link. Refer to Shell Module Programming Guide for more details about each Shell module steps.

> **The File System shell sub-module can manage only a single shell terminal of the host at a time. For instance, if you open two independent shell terminals on the host, and each of the terminals works in the same volume but in a different working directory, a file command such as fs_cat done in the first terminal may refer to the working directory of the second terminal and not the current terminal. You should always run any file commands from the same terminal.**

Commands

The supported commands, listed in Table - File System Shell Commands in the *Shell Commands* page, are equivalent to the standard UNIX commands of the same names, although the functionality is usually simpler, with fewer or no special options. The only exception is fs_lsblk, which is specific to Micrium OS File System. The File System shell commands can be divided in five groups:

- **Block device**: the sole command in this group allows you to obtain information about the block devices opened on the target with FSBlkDev_Open().
- **Volume**: the commands in this group allow you to perform operations on a volume such mount, format, unmount, etc.
- **File**: the commands in this group allow you to perform file operations such as display file content, obtain file statistics, copy a file, etc.
- **Directory**: the commands in this group allow you to manage directory operations such as display a directory's content, change the working directory, etc.
- **Entry**: the sole command in this group allows you to delete a file or directory.

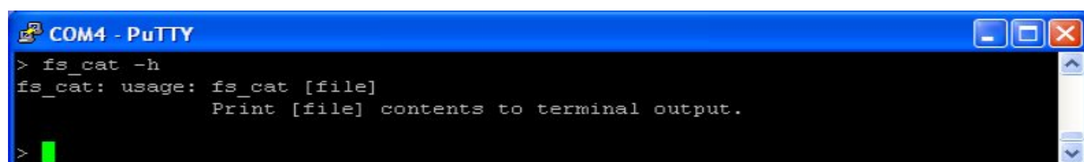
> All the commands in the groups File, Directory and Entry must be executed on a formatted volume. That is, you must always execute the command fs_mount prior to any commands in those groups. You may have to format the mounted volume with fs_mskfs if the volume has not been yet formatted. Note that the command fs_mskfs formats the volume as a FAT file system.

Table - File System Shell Commands

Command	Argument Needed	Description
Block Device		
fs_lsblk	✗	List all open block devices and display information about each partition composing the block device
Volume		
fs_df	✓	Report disk free space
fs_mount	✓	Mount volume
fs_mkfs	✓	Format a volume
fs_umount	✓	Unmount volume
File		
fs_cat	✓	Print a file to the terminal output
fs_cp	✓	Copy a file
fs_date	✓	Write the date and time to terminal output, or set the system date and time
fs_mv	✓	Move files
fs_od	✓	Dump file contents to terminal output
fs_touch	✓	Change file modification time
fs_wc	✓	Determine the number of newlines, words and bytes in a file
Directory		
fs_cd	✓	Change the working directory
fs_ls	✗	List directory contents
fs_mkdir	✓	Make a directory
fs_pwd	✗	Write the pathname of current working directory to the terminal output
fs_rmdir	✓	Remove a directory
Entry		
fs_rm	✓	Remove a directory entry, that is a file or a directory

Information about each command can be obtained using the help (-h) option:

Figure - Help Option Output



File System Hardware Porting Guide

File System Hardware Porting Guide

The Micrium OS File System module includes a certain number of drivers that support a range of memory controllers.

Each memory controller used by your project requires a Board Support Package (BSP). The BSP has two purposes:

- It initializes and configures any resources needed by the memory controller, but which are provided by external modules. These include the clock, GPIO, interrupts, etc.
- It provides hardware information to the memory controller driver.

Micrium provides example BSPs for several popular platforms. If one is available for your platform, we recommend that you use it as a starting point. However, if no example BSP is available for your platform, the information in this section will help you understand how to create your own BSP and port the File System module to your platform.

- File System Memory Driver Selection Guide
- File System Memory BSP Functions Guide
- File System Memory Hardware Information
- File System Memory Controller Registration to the Platform Manager

File System Memory Driver Selection Guide

- Memory Controller Types
  - NAND
  - NOR
  - SD

Memory Controller Types

The File System module supports three types of memory interfaces: NAND, NOR and SD.

Note that RAM Disk and SCSI do not have dedicated memory interfaces.

Indeed, RAM Disk emulates a volatile memory region aimed to quickly show the File System module capabilities. RAM Disk is created using the RAM of your program. The program RAM can be located in the internal (for instance SRAM) or external (for example DDR SDRAM) memory of the microcontroller. The File system does not provide any driver for such memory.

SCSI devices exist using a transport medium such as USB. The memory controller equivalent for SCSI is, in fact, a USB host controller. Consequently, supporting SCSI devices requires that you have a USB host controller on your platform.

NAND

Microcontroller units (MCU) usually offer a few different types of peripherals to access external NAND flash devices. The peripheral depends on the NAND chip interface to the MCU: parallel or serial. For parallel NAND, the MCU can offer a dedicated NAND controller or a more flexible memory controller able to support both parallel NAND and NOR flash devices. For serial NAND, the prevailing MCU peripheral is an SPI controller.

The Micrium OS File System offers a NAND media driver that:

- Supports parallel NAND. Drivers are offered for some memory controllers.
- Does not support serial NAND.

NOR

Microcontrollers usually offer a few different types of peripherals to access external NOR flash devices. The peripheral depends on the NOR chip interface to the MCU: parallel or serial. For parallel NOR, the MCU can offer a dedicated NOR

controller or a more flexible memory controller able to support both parallel NAND and NOR flash devices. For serial NOR, two MCU peripherals are common, either a Quad SPI controller or an SPI controller.

The Micrium OS File System offers a NOR media driver that:

- Supports parallel NOR. However, no memory controller drivers are available for the moment.
- Supports serial NOR. Drivers are available for some Quad SPI controllers. SPI controller drivers are implemented by the Micrium OS IO-SPI module. Moreover, the File System offers drivers for some NOR serial flash devices (refer to the File System Drivers page to view the supported NOR).

SD

An MCU can support SD cards using two types of peripherals: SPI or SDHC controller. An SD card can operate either in SD mode, managed by an SDHC controller, or in SPI mode, managed by an SPI controller.

The Micrium OS File System offers an SD media driver that:

- Supports SD cards in SD mode. Drivers are available for some SDHC controllers.
- Support SD cards in SPI mode. SPI controller drivers are implemented by the Micrium OS IO-SPI module.

File System Memory BSP Functions Guide

A Board Support Package contains a set of functions that support your hardware platform on Micrium OS. These functions are called by the memory controller driver, and they perform hardware configuration and initialization of the clock, I/O pins, interrupts, and so on. It is your responsibility to either create these functions from scratch, or to modify example code to fit the needs and specifics of your platform.

A BSP provides a layer of abstraction between the memory controller driver and the context of the memory controller in your platform. This is useful because although the driver knows the details of a specific controller IP (such as the registers that it contains, and how to properly read and write them), the driver does not know the context in which the controller exists in your hardware solution: what are the I/O and interrupt pins, clock source, and so on. The functions of a BSP can be (and often are) customized based on how the controller is wired to your board.

Each memory controller you intend to use will need its own BSP, and each BSP defines the set of functions called by the drivers. Each of these functions is described below according to the media and memory controller type.

- NOR
  - Quad SPI Controller
    - Initialize
    - Clock Configuration
    - I/O Configuration
    - Chip Select Enable
    - Chip Select Disable
    - Clock Frequency Get
    - ISR Handling
  - SPI Controller
- SCSI
  - USB Host Controller
- SD
  - SD Host Controller (SDHC)
    - Add
    - Open
    - Close
    - Lock
    - Unlock
    - Command Start
    - Command End Wait
    - Command Data Read
    - Command Data Write
    - Block Count Maximum Get
    - Bus Width Maximum Get

    - Bus Width Set
    - Clock Frequency Set
    - Timeout Data Set
    - Timeout Response Set
    - ISR Handling
  - SPI Controller

NOR

Quad SPI Controller

Initialize

The Initialize function is called by the Quad SPI driver when a NOR media is added to the Storage layer. Listing - BSP Init Function Signature in the *File System Memory BSP Functions Guide* page shows the signature of this function.

Listing - BSP Init Function Signature

```
CPU_BOOLEAN  BSP_FS_NOR_QuadSPI_Init (FS_NOR_QUAD_SPI_ISR_HANDLE_FNCT   isr_fnct,
                                       void                             *p_drv_data);
```

Its purpose is to initialize and allocate resources that will be needed by the Quad SPI controller BSP.

The initialization function receives two arguments: isr_fnct and p_drv_data, which are used when handling interrupts from the Quad SPI controller. The argument isr_fnct is a pointer to a driver function that must be called when a Quad SPI interrupt occurs, and this driver function takes p_drv_data as an argument. So it is important to save these as global variables in your BSP. If the memory controller driver does not implement interrupt-based communication, these two arguments can be ignored by using the macro PP_UNUSED_PARAM().

You must return DEF_OK from this function if it executes successfully, or DEF_FAIL, otherwise.

Clock Configuration

The Clock Configuration function is called by the Quad SPI driver when a NOR media is added to the Storage layer. Listing - BSP Clock Configure Function Signature in the *File System Memory BSP Functions Guide* page shows the signature of the function.

Listing - BSP Clock Configure Function Signature

```
CPU_BOOLEAN  BSP_FS_NOR_QuadSPI_ClkCfg (void);
```

Its purpose is to configure the clock of the Quad SPI controller by accessing certain registers of the microcontroller's clock management peripheral.

You must return DEF_OK from this function if it executed successfully, or DEF_FAIL, otherwise.

I/O Configuration

The I/O Configuration function is called by the Quad SPI driver when a NOR media is added to the Storage layer. Listing - BSP IO Configure Function Signature in the *File System Memory BSP Functions Guide* page shows the signature of the function.

Listing - BSP IO Configure Function Signature

```
CPU_BOOLEAN  BSP_FS_NOR_QuadSPI_IO_Cfg (void);
```

Its purpose is to configure the I/O pins for the Quad SPI controller.

You must return DEF_OK from this function if it executed successfully, or DEF_FAIL, otherwise.

Chip Select Enable

The Chip Select Enable function is called by the Quad SPI driver each time communication is started with the NOR chip. Listing - BSP Chip Select Enable Function Signature in the *File System Memory BSP Functions Guide* page shows the signature of this function.

Listing - BSP Chip Select Enable Function Signature

```
CPU_BOOLEAN  BSP_FS_NOR_QuadSPI_ChipSelEn (CPU_INT16U  part_slave_id);
```

Its purpose is to select the NOR chip in order to allow communication with the NOR chip, including access to NOR registers and data memory regions. Selecting the NOR flash device is performed by driving the Chip Enable signal, and usually, an I/O peripheral register performs this job. The argument part_slave_id represents the slave ID of the NOR chip, which is used when a serial bus is shared with other external devices. If the NOR flash chip is the only device connected to the serial bus, the slave ID can be ignored.

You may not have to implement this function if the Chip Select signal is controlled directly by a Quad SPI controller's register.

You must return DEF_OK from this function if it executed successfully, or DEF_FAIL, otherwise.

Chip Select Disable

The Chip Select Disable function is called by the Quad SPI driver to end communication with the NOR chip. Listing - Chip Select Disable Function Signature in the *File System Memory BSP Functions Guide* page shows the signature of this function.

Listing - Chip Select Disable Function Signature

```
CPU_BOOLEAN  BSP_FS_NOR_QuadSPI_ChipSelDis (CPU_INT16U  part_slave_id);
```

Its purpose is to deselect the NOR chip in order to prevent further communication with the NOR chip, including access to NOR registers and data memory regions. Selecting the NOR flash device is performed by driving the Chip Enable signal, and usually, an I/O peripheral register performs this job. The argument part_slave_id represents the slave ID of the NOR chip, which is used when a serial bus is shared with other external devices. If the NOR flash chip is the only device connected to the serial bus, the slave ID can be ignored.

You may not have to implement this function if the Chip Select signal is controlled directly by a Quad SPI controller's register.

You must return DEF_OK from this function if it executed successfully, or DEF_FAIL, otherwise.

Clock Frequency Get

The Clock Frequency Get function is called by the driver when the controller clock is configured. Listing - BSP Clock Frequency Get Function Signature in the *File System Memory BSP Functions Guide* page shows the signature of this function.

Listing - BSP Clock Frequency Get Function Signature

```
CPU_INT32U  BSP_FS_NOR_QuadSPI_ClkFreqGet (void)
```

It is used to determine the clock divider that must be used to achieve the desired serial clock. The function must always return the clock frequency, in Hertz, of the clock that feeds the Quad SPI controller.

ISR Handling

The CMSIS standard function QSPIx_IRQHandler() is called each time a QSPI interrupt is triggered. This function is defined as *weak* in the Silicon Labs Gecko SDK. If the memory controller driver implements interrupt-based communication, this function must be redefined in the BSP to properly handle the IRQ. Otherwise, it is not required.

SPI Controller

Writing a BSP for NOR SPI means writing a BSP that uses the SPI driver offered by the Micrium OS IO-SPI module. Refer to SPI BSP Functions Guide for more details about the BSP functions to implement.

SCSI

USB Host Controller

Writing a BSP for SCSI means writing a BSP that uses the USB host driver offered by the Micrium OS USB Host module. Refer to USB Host BSP Functions Guide for more details about the BSP functions to implement.

SD

SD Host Controller (SDHC)

The SDHC (also called SD Card) BSP has a particularity compared to the BSPs for NOR QSPI/SPI and SD SPI: the BSP is also the memory controller's driver.

Usually, the BSP is responsible for setting the clock, interrupts, and GPIO needed by the memory controller. Thus, the BSP configures the other peripherals to allow the memory controller driver to work. The BSP is not normally a driver that implements the memory controller functions. However, in the case of SD Card, the BSP is also the memory controller's driver. This particularity will be removed in a future release of Micrium OS File System in order to simplify the SD Card BSP API.

Add

The Add function is called by the driver when the application adds an SD media. Listing - Add Function Signature in the *File System Memory BSP Functions Guide* page shows the signature of this function.

Listing - Add Function Signature

```
void  *BSP_FS_SD_Card_Add (RTOS_ERR  *p_err);
```

Its purpose is to allocate and configure any software resources needed by the controller driver.

Open

The Open function is called by the driver when the application opens an SD block device. Listing - Open Function Signature in the *File System Memory BSP Functions Guide* page shows the signature of this function.

Listing - Open Function Signature

```
CPU_BOOLEAN  BSP_FS_SD_Card_Open (void);
```

Its purpose is to initialize and allocate resources that will be needed by the memory controller driver.

This function, for example, can configure:

- The input clocks required by the controller
- Any needed GPIO pins
- The interrupt (in the interrupt controller) if the controller is used in interrupt mode
- Any registers required to start the controller's operations

Close

The Close function is called by the driver when the application closes an SD block device. Listing - Close Function Signature in the *File System Memory BSP Functions Guide* page shows the signature of this function.

Listing - Close Function Signature

```
void  BSP_FS_SD_Card_Close (void);
```

Its purpose is to terminate and free resources that were used by the memory controller driver. For instance, it may be required to disable the controller's operation and to reset some of the controller's registers to a known state.

Lock

The Lock function is called by the driver when a sequence of SD operations must be protected. Listing - Lock Function Signature in the *File System Memory BSP Functions Guide* page shows the signature of this function.

Listing - Lock Function Signature

```
void  BSP_FS_SD_Card_Lock (void);
```

Its purpose is to protect the SD card bus from multiple concurrent accesses. Indeed, more than one SD card can share the card bus.

Unlock

The Unlock function is called by the driver when the sequence of SD operations to be protected is finished. Listing - Unlock Function Signature in the *File System Memory BSP Functions Guide* page shows the signature of this function.

Listing - Unlock Function Signature

```
void  BSP_FS_SD_Card_Unlock (void);
```

Its purpose is to unlock the SD card bus.

Command Start

The Command Start function is called by the driver when the command phase of an SD card mode transfer starts. Listing - Command Start Function Signature in the *File System Memory BSP Functions Guide* page shows the signature of this function.

Listing - Command Start Function Signature

```
void  BSP_FS_SD_Card_CmdStart (FS_DEV_SD_CARD_CMD  *p_cmd,
                               void                *p_data,
                               FS_DEV_SD_CARD_ERR  *p_err);
```

Its purpose is to send a command to the SD card to start an operation such as a register or a data memory access.

Command End Wait

The Command End Wait function is called by the driver while it waits for the response of the command phase of an SD card mode transfer. Listing - Command End Wait Function Signature in the *File System Memory BSP Functions Guide* page shows the signature of this function.

Listing - Command End Wait Function Signature

```
void  BSP_FS_SD_Card_CmdEndWait (FS_DEV_SD_CARD_CMD  *p_cmd,
                                 CPU_INT32U          *p_resp,
                                 FS_DEV_SD_CARD_ERR  *p_err);
```

Its purpose is to receive a response from the SD card following the reception of a command. The response gives information about the command's execution status.

Command Data Read

The Command Data Read function is called by the driver if the command previously sent requires a data read phase. Listing - Command Data Read Function Signature in the *File System Memory BSP Functions Guide* page shows the signature of this function.

Listing - Command Data Read Function Signature

```
void  BSP_FS_SD_Card_CmdDataRd (FS_DEV_SD_CARD_CMD  *p_cmd,
                                void                *p_dest,
                                FS_DEV_SD_CARD_ERR  *p_err);
```

Its purpose is to receive data from the SD card memory region. A data read phase transports only data from the memory and never register content. A register's content is retrieved as part of the command response phase.

Command Data Write

The Command Data Write function is called by the driver if the command previously sent requires a data write phase. Listing - Command Data Write Function Signature in the *File System Memory BSP Functions Guide* page shows the signature of this function.

Listing - Command Data Write Function Signature

```
void  BSP_FS_SD_Card_CmdDataWr (FS_DEV_SD_CARD_CMD  *p_cmd,
                                void                *p_src,
                                FS_DEV_SD_CARD_ERR  *p_err);
```

Its purpose is to send data to the SD card memory region. A data write phase transports only data aimed to be stored in the memory and will never carry register content. A register's content is carried over the argument field of the command phase.

Block Count Maximum Get

The Block Count Maximum Get function is called by the driver when the application opens an SD block device. Listing - Block Count Maximum Get Function Signature in the *File System Memory BSP Functions Guide* page shows the signature of this function.

Listing - Block Count Maximum Get Function Signature

```
CPU_INT32U  BSP_FS_SD_Card_BlkCntMaxGet (CPU_INT32U  blk_size);
```

Its purpose is to retrieve the maximum number of blocks that can be transferred with a multiple read or multiple write command.

Bus Width Maximum Get

The Bus Width Maximum Get function is called by the driver when the application opens an SD block device. Listing - Bus Width Maximum Get Function Signature in the *File System Memory BSP Functions Guide* page shows the signature of this function.

Listing - Bus Width Maximum Get Function Signature

```
CPU_INT08U  BSP_FS_SD_Card_BusWidthMaxGet (void);
```

Its purpose is to retrieve the bus width of the SD card bus. Typically the bus width is 1 or 4 bit.

Bus Width Set

The Bus Width Set function is called by the driver when the application opens an SD block device. Listing - Bus Width Set Function Signature in the *File System Memory BSP Functions Guide* page shows the signature of this function.

Listing - Bus Width Set Function Signature

```
void  BSP_FS_SD_Card_BusWidthSet (CPU_INT08U  width);
```

Its purpose is to set the data transfer width to 1-bit or 4-bit mode.

Clock Frequency Set

The Clock Frequency Set function is called by the driver when the application opens an SD block device. Listing - Clock Frequency Set Function Signature in the *File System Memory BSP Functions Guide* page shows the signature of this function.

Listing - Clock Frequency Set Function Signature

```
void  BSP_FS_SD_Card_ClkFreqSet (CPU_INT32U  freq);
```

Its purpose is to set the SD card bus operating frequency. Usually, during the identification phase, the SD bus operates at between 100 and 400 kHz. Once initialized, the SD card can operate up to 25 MHz for default speed mode and up to 50 MHz for high-speed mode.

Timeout Data Set

The Timeout Data Set function is called by the driver when the application opens an SD block device. Listing - Timeout Data Set Function Signature in the *File System Memory BSP Functions Guide* page shows the signature of this function.

Listing - Timeout Data Set Function Signature

```
void  BSP_FS_SD_Card_TimeoutDataSet (CPU_INT32U  to_clks);
```

Its purpose is to configure the timeout used during the data phase of an SD transfer. The unit is expressed in card bus clock periods.

Timeout Response Set

The Timeout Response Set function is called by the driver when the application opens an SD block device. Listing - Timeout Response Set Function Signature in the *File System Memory BSP Functions Guide* page shows the signature of this function.

Listing - Timeout Response Set Function Signature

```
void  BSP_FS_SD_Card_TimeoutRespSet (CPU_INT32U  to_ms);
```

Its purpose is to configure the timeout used during the response phase of an SD transfer. It helps to limit the waiting time of the response to a command. The unit is expressed in milliseconds. Sometimes nothing needs to be configured in the SDHC controller as this one uses a fixed hardware timeout of N periods of the SD clock.

ISR Handling

The CMSIS standard function SDIO_IRQHandler() is called each time an SD interrupt is triggered. This function is defined as *weak* in the Silicon Labs Gecko SDK. It must be redefined in the BSP to properly handle the IRQ.

SPI Controller

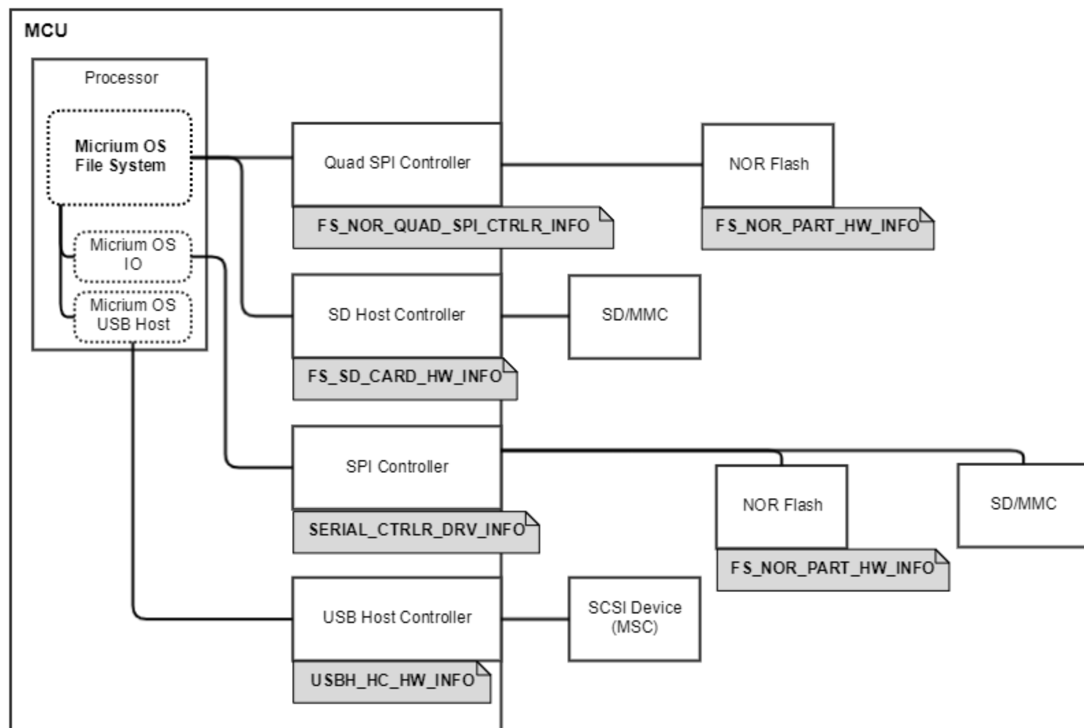
Writing a BSP for SD SPI means writing a BSP that uses the SPI driver offered by the Micrium OS IO-SPI module. Refer to SPI BSP Functions Guide for more details about the BSP functions to implement.

File System Memory Hardware Information

This section presents the various memory hardware information structures used to configure the different memories (NOR, SCSI and SD). Figure - Memories and their Hardware Information in the *File System Memory Hardware Information* page gives at a glance the different memories and their associated hardware information structure (in rectangles with a grey background).

- NOR
  - Quad SPI
    - Hardware Driver Information
      - Quad SPI Controller Information
      - NOR Flash Device Information
    - BSP API Structure
    - Device Hardware Information
- SCSI
- SD
  - Card
    - Hardware Driver Information
    - BSP API Structure
    - Device Hardware Information
  - SPI

Figure - Memories and their Hardware Information



NOR

Quad SPI

The NOR Quad SPI (QSPI) hardware information is a structure of type FS_NOR_QUAD_SPI_HW_INFO. This structure is used by the macro FS_NOR_QUAD_SPI_HW_INFO_REG() to register the NOR QSPI into the Platform Manager . For more information about using the macro FS_NOR_QUAD_SPI_HW_INFO_REG(), refer to the NOR Quad SPI Registration section. The NOR QSPI hardware information structure contain other structures which provide information about the QSPI controller and the NOR flash device. All structures composing the NOR QSPI hardware information structure are explained in the sections below.

Hardware Driver Information

You need to define the two following structures to fully describe the NOR Quad SPI hardware information:

- FS_NOR_QUAD_SPI_CTRLR_INFO (for the QSPI controller)
- FS_NOR_PART_INFO (for the NOR flash device)

The following sub-sections give you more details about each structures' field. Listing - Typical NOR Hardware Initialization in the *File System Memory Hardware Information* page illustrate a typical initialization of the two structures in the file bsp_fs_nor_quad_spi.c.

Listing - Typical NOR Hardware Initialization

```
/* Quad SPI controller hardware info.                  */
const  FS_NOR_QUAD_SPI_CTRLR_INFO  BSP_FS_NOR_QuadSPICtrl_HwInfo = {
    .DrvApiPtr           = (FS_NOR_QUAD_SPI_DRV_API *)&FS_NOR_QuadSpiDrvAPI_Silabs_EFM32GG11,
    .BspApiPtr           = (FS_NOR_QUAD_SPI_BSP_API *)&QuadSPI_Ctrlr_BSP_API,
    .BaseAddr            = QSPI0_BASE,
    .AlignReq            = sizeof(CPU_ALIGN),
    .FlashMemMapStartAddr = QSPI0_MEM_BASE,
    .BusWidth            = FS_NOR_SERIAL_BUS_WIDTH_SINGLE_IO
};

                                /* NOR info composed of QSPI controller & NOR memory chip.*/
const  FS_NOR_QUAD_SPI_HW_INFO  BSP_FS_NOR_QuadSPI_HwInfo = {
    .PartHwInfo.PhyApiPtr = (CPU_INT32U)(&FS_NOR_PHY_MX25R_API),
    .PartHwInfo.ChipSelID = 0u,
    .CtrlrHwInfoPtr       = &BSP_FS_NOR_QuadSPICtrl_HwInfo
};
```

Quad SPI Controller Information

The Quad SPI driver requires information about the Quad SPI controller on your MCU, which you provide using a structure of type FS_NOR_QUAD_SPI_CTRLR_INFO. Some pieces of information can be found in the MCU's reference manual. Listing - Typical NOR Hardware Initialization in the *File System Memory Hardware Information* page shows an example of this structure's initialization.

Table - FS_NOR_QUAD_SPI_CTRLR_INFO structure in the *File System Memory Hardware Information* page describes each configuration field of this structure.

Table - FS_NOR_QUAD_SPI_CTRLR_INFO structure

Field	Description
.DrvApiPtr	Pointer to QSPI driver API.
.BspApiPtr	Pointer to QSPI BSP API. Refer to the next section BSP API Structure for more information.
.BaseAddr	Base address of the QSPI controller register set. This corresponds to the address of the first register.
.AlignReq	Buffer alignment requirement if the QSPI controller uses DMA transfers. If no special buffer alignment is required, you may initialize this field to sizeof(CPU_ALIGN).

Field	Description
.FlashMemMapStartAddr	Start address of the NOR flash range within the microcontroller's internal memory. This field is reserved for future considerations. Must be always DEF_NULL. The Quad SPI controller supports a functional mode called memory-mapped or direct mode. In memory-mapped mode, the external flash memory is mapped in the microcontroller's address space and is seen by the system as if it was internal memory. This mode allows to both access data and execute code from the external flash memory. The memory-mapped mode works with the NOR flash device configured in XIP (eXecute In Place) mode. When the external flash device is memory-mapped, an address range is dedicated to it besides the QSPI registers address range.
.BusWidth	Width of the serial bus connected between the microcontroller and the NOR flash. Possible values:
	FS_NOR_SERIAL_BUS_WIDTH_SINGLE_IO 1-bit SPI bus
	FS_NOR_SERIAL_BUS_WIDTH_DUAL_IO 2-bit SPI bus
	FS_NOR_SERIAL_BUS_WIDTH_QUAD_IO 4-bit SPI bus

NOR Flash Device Information

The NOR flash device driver requires information about the NOR memory chip interfaced to your MCU, which you provide using a structure of type FS_NOR_PART_INFO. Listing - Typical NOR Hardware Initialization in the *File System Memory Hardware Information* page shows an example of this structure's initialization.

Table - FS_NOR_PART_INFO Structure in the *File System Memory Hardware Information* page describes each configuration field available in this structure.

Table - FS_NOR_PART_INFO Structure

Field	Description
.PhyApiPtr	Pointer to NOR PHY driver API. Consult the NOR flash drivers list for more information about the API structure name to use.
.ChipSelID	NOR chip select ID used by the Quad SOI controller driver to select or unselect the NOR flash device for communication. If the NOR chip is the only device connected to a shared bus, you can specify slave ID #0.

BSP API Structure

In order to provide a pointer to the BSP functions needed by the Quad SPI controller driver, you need to create a structure of type FS_NOR_QUAD_SPI_BSP_API. You must define this structure in the NOR QSPI's BSP file (bsp_fs_nor_quad_spi.c).

Table - FS_NOR_QUAD_SPI_BSP_API structure in the *File System Memory Hardware Information* page describes each field of this structure.

Listing - Example of NOR QSPI BSP API Structure in the *File System Memory Hardware Information* page shows an example of a BSP API structure.

Listing - Example of NOR QSPI BSP API Structure

```
static const ST_QUAD_SPI_BSP_API  QuadSPI_Ctrlr_BSP_API = {
    .Init       = BSP_FS_NOR_QuadSPI_Init,
    .ClkCfg     = BSP_FS_NOR_QuadSPI_ClkCfg,
    .IO_Cfg     = BSP_FS_NOR_QuadSPI_IO_Cfg,
    .IntCfg     = BSP_FS_NOR_QuadSPI_IntCfg,
    .ChipSelEn  = BSP_FS_NOR_QuadSPI_ChipSelEn,
    .ChipSelDis = BSP_FS_NOR_QuadSPI_ChipSelDis,
    .ClkFreqGet = BSP_FS_NOR_QuadSPI_ClkFreqGet
};
```

Device Hardware Information

The last step is to create the main device hardware information structure. This structure encompasses other structures. Listing - Typical NOR Hardware Initialization in the *File System Memory Hardware Information* page shows an example of this structure's initialization.

Table - FS_NOR_QUAD_SPI_HW_INFO structure in the *File System Memory Hardware Information* page describes each configuration field of this structure.

Table - FS_NOR_QUAD_SPI_HW_INFO structure

Field	Description
.PartHwInfo	Structure describing the NOR flash device characteristics. Refer to section Hardware Driver Information for more details about its fields.
.CtrlrHwInfoPtr	Pointer to the structure describing the Quad SPI controller characteristics. Refer to section Hardware Driver Information for more details about its fields.

SCSI

As mentioned in the note at the beginning of the Memory Controller Types section, an SCSI device is bound to the existence of a USB host. Consequently, SCSI does not have a dedicated hardware information. Instead you need to provide the hardware information for a USB host in order to use the SCSI media driver. Refer to USB Host HCD Hardware Information for more details about providing the USB host hardware information.

SD

Card

The SD Card hardware information is composed of a structure of type FS_SD_CARD_HW_INFO. This structure is used by the macro FS_SD_CARD_HW_INFO_REG() to register the SD Card into the Platform Manager . Refer to section SD Card Registration for more details about using the FS_SD_CARD_HW_INFO_REG() macro. The SD Card hardware information structure can contain other structures which provide information about the BSP API, the buffer alignment requirement, etc. All structures composing the SD Card hardware information structure are explained in the sections below.

Hardware Driver Information

The SD Card media driver requires information about the SD Card controller on your MCU. A structure is not necessary as there is only one piece of information needed. This unique information is:

- The buffer alignment requirement. You may need to specify that if the SD Card controller uses DMA transfers. If no special buffer alignment is required, you may initialize this field to sizeof(CPU_ALIGN) .

BSP API Structure

In order to provide a pointer to the BSP functions for the SD Card memory controller (SDHC) driver, you have to create a structure of type FS_SD_CARD_BSP_API. You must define this structure in the SD Card's BSP file (bsp_fs_sd_card.c).

Table - FS_SD_CARD_BSP_API structure in the *File System Memory Hardware Information* page describes each field of this structure.

Table - FS_SD_CARD_BSP_API structure

Field	Description
.Add	Pointer to the BSP add function.
.Open	Pointer to the BSP open function.
.Close	Pointer to the BSP close function.
.Lock	Pointer to the BSP lock function.
.Unlock	Pointer to the BSP unlock function.
.CmdStart	Pointer to the BSP command start function.

Field	Description
.CmdEndWait	Pointer to the BSP command end wait function.
.CmdDataRd	Pointer to the BSP command data read function.
.CmdDataWr	Pointer to the BSP command data write function.
.BlkCntMaxGet	Pointer to the BSP block count maximum get function.
.BusWidthMaxGet	Pointer to the BSP bus width maximum get function.
.BusWidthSet	Pointer to the BSP bus width set function.
.ClkFreqSet	Pointer to the BSP clock frequency set function.
.TimeoutDataSet	Pointer to the BSP timeout data set function.
.TimeoutRespSet	Pointer to the BSP timeout response set function.

Listing - Example of SD Card BSP API Structure in the *File System Memory Hardware Information* page shows an example of a BSP API structure.

Listing - Example of SD Card BSP API Structure

```
static const FS_SD_CARD_BSP_API BSP_FS_SD_Card_STM32F746G_DISCO_BSP_API = {
    .Add            = BSP_FS_SD_Card_Add,
    .Open           = BSP_FS_SD_Card_Open,
    .Close          = BSP_FS_SD_Card_Close,
    .Lock           = BSP_FS_SD_Card_Lock,
    .Unlock         = BSP_FS_SD_Card_Unlock,
    .CmdStart       = BSP_FS_SD_Card_CmdStart,
    .CmdEndWait     = BSP_FS_SD_Card_CmdEndWait,
    .CmdDataRd      = BSP_FS_SD_Card_CmdDataRd,
    .CmdDataWr      = BSP_FS_SD_Card_CmdDataWr,
    .BlkCntMaxGet   = BSP_FS_SD_Card_BlkCntMaxGet,
    .BusWidthMaxGet = BSP_FS_SD_Card_BusWidthMaxGet,
    .BusWidthSet    = BSP_FS_SD_Card_BusWidthSet,
    .ClkFreqSet     = BSP_FS_SD_Card_ClkFreqSet,
    .TimeoutDataSet = BSP_FS_SD_Card_TimeoutDataSet,
    .TimeoutRespSet = BSP_FS_SD_Card_TimeoutRespSet
};
```

Device Hardware Information

The last step is to create the main device hardware information structure. This structure contains only pointers to other structures.

Table - FS_SD_CARD_HW_INFO structure in the *File System Memory Hardware Information* page describes each configuration field of this structure.

Table - FS_SD_CARD_HW_INFO structure

Field	Description
.BspApiPtr	Pointer to the BSP API structure you created in the BSP API Structure step.
.AlignReq	Buffer alignment requirement if SD Card controller uses DMA transfers. Refer to section Hardware Driver Information for more details about this field.

SPI

SD SPI does not need a specific hardware information structure. SD SPI relies on the proper hardware information settings for SPI. Refer to SPI Hardware Information for more details about providing the SPI hardware information.

File System Memory Controller Registration to the Platform Manager

-

- NOR Registration
  - Quad SPI
- SCSI Registration
- SD Registration
  - SD Card
  - SD SPI

Once the hardware information structure for your memory controller is ready, it must be registered with the Platform Manager . This should normally be done with the BSP_OS_Init() function that is located in the file bsp_os.c.

There are four different macros that you can call to register a memory controller. Table - Memory Controller Register Macros in the *File System Memory Controller Registration to the Platform Manager* page describes the different macros.

Table - Memory Controller Register Macros

Macro	Description	Definition Location
FS_NOR_QUAD_SPI_HW_INFO_REG()	Registers a NOR Quad SPI controller communicating with an external NOR flash device.	fs_nor.h
FS_SD_CARD_HW_INFO_REG()	Registers an SD Card controller (also called SDHC) communicating with an external SD Card in card mode.	fs_sd_card.h
FS_SD_SPI_HW_INFO_REG()	Registers an external SD communicating with the MCU in SPI mode.	fs_sd_spi.h

NOR Registration

Quad SPI

Listing - Example of NOR Quad SPI Controller Registration in the *File System Memory Controller Registration to the Platform Manager* page shows an example of how to register a NOR Quad SPI controller.

Listing - Example of NOR Quad SPI Controller Registration

```
#include  <rtos_description.h>

#if defined(RTOS_MODULE_FS_STORAGE_NOR_AVAIL)                    (1)
BSP_HW_INFO_EXT(const  FS_NOR_QUAD_SPI_CTRLR_INFO, BSP_FS_NOR_QuadSPI_HwInfo);
#endif

void  BSP_OS_Init (void)
{

   /* ... */
                                         /* ------- REGISTER MEMORY CONTROLLERS ------- */
#if defined(RTOS_MODULE_FS_STORAGE_NOR_AVAIL)                    (2)
   FS_NOR_QUAD_SPI_HW_INFO_REG("nor0", &BSP_FS_NOR_QuadSPI_HwInfo);
#endif

}
```

(1) Since the global variable for NOR hardware information is defined in another file, you must declare it as external in your bsp_os.c file. Always use the BSP_HW_INFO_EXT() macro.

(2) Registering a NOR Quad SPI controller with the tag "nor0". The tag will be used internally by the File System stack later on when calling the function FSStorage_Init(). The NOR hardware information structure is described in the NOR Quad SPI section of the Hardware Information page.

SCSI Registration

As mentioned in the note at the beginning of the Memory Controller Types section, an SCSI device is bound to the existence of a USB host. Consequently, SCSI does not have a dedicated register macro. Instead you need to use the USB host register macro described in USB Host Controller Registration to the Platform Manager .

SD Registration

SD Card

Listing - Example of SD Card Controller Registration in the *File System Memory Controller Registration to the Platform Manager* page shows an example of how to register an SD Card (SDHC) controller.

Listing - Example of SD Card Controller Registration

```
#include <rtos_description.h>

#if defined(RTOS_MODULE_FS_STORAGE_SD_CARD_AVAIL)            (1)
BSP_HW_INFO_EXT(const FS_SD_CARD_HW_INFO, BSP_FS_SD_Card_HwInfo);
#endif

void BSP_OS_Init (void)
{

    /* ... */
                                                /* ------- REGISTER MEMORY CONTROLLERS ------- */
#if defined(RTOS_MODULE_FS_STORAGE_SD_CARD_AVAIL)
    FS_SD_CARD_HW_INFO_REG("sd0", &BSP_FS_SD_Card_HwInfo);     (2)
#endif

}
```

(1) Since the global variable for SD Card hardware information is defined in another file, you must declare it as external in your bsp_os.c file. Always use the BSP_HW_INFO_EXT() macro.

(2) Registering an SD Card (SDHC) controller with the tag "sd0". The tag will be used internally by the File System stack later on when calling the function FSStorage_Init(). The SD Card hardware information structure is described in the SD Card section of the Hardware Information page .

SD SPI

Listing - Example of SD SPI Controller Registration in the *File System Memory Controller Registration to the Platform Manager* page shows an example of how to register an SD Card controller and an SD using an SPI controller.

Listing - Example of SD SPI Controller Registration

```
#include <rtos_description.h>

#ifdef RTOS_MODULE_IO_SERIAL_AVAIL                    (1)
BSP_HW_INFO_EXT(const SERIAL_CTRLR_DRV_INFO, BSP_Serial_SiliconLabs_Bus0_DrvInfo);
#endif

void BSP_OS_Init (void)
{

    /* ... */
                                                /* ------- REGISTER MEMORY CONTROLLERS ------- */
#if defined(RTOS_MODULE_IO_SERIAL_AVAIL)
    IO_SERIAL_CTRLR_REG("spi0", &BSP_Serial_SiliconLabs_Bus0_DrvInfo); (2)
#endif
#if defined(RTOS_MODULE_FS_STORAGE_SD_SPI_AVAIL)
    FS_SD_SPI_HW_INFO_REG("sd0", "spi0", 0u);             (3)
#endif

}
```

(1) Since the global variable for SPI hardware information is defined in another file, you must declare it as external in your bsp_os.c file. Always use the BSP_HW_INFO_EXT() macro.

(2) Registering an SPI controller with the tag "spi0". The tag will be used later on when calling the function SPI_BusAdd() . The SPI hardware information structure is described in SPI Hardware Information .

(3) Registering an SD in SPI mode with the tag "sd0". The tag will be used internally by the File System stack later on when calling the function FSStorage_Init(). You need to indicate the SPI controller tag, "spi0" in this case, to which the SD is interfaced to. The third macro parameter specifies a slave ID in case there are other slaves connected to the same SPI bus besides SD.

File System Drivers

File System Drivers

This page presents the File System drivers that are currently supported with Micrium OS. Browse any of the driver's page for more technical details.

NOR Flash Device Drivers

Driver	IP Provider	Interface to MCU
Q	Winbond	Serial (SPI)
R	Macronix	Serial (Quad SPI)

Specifications

IP Provider	Macronix
Interface to MCU	Serial (Quad SPI)
Data Mode	Single IO
XIP (eXecute In Place) Mode	**not supported by chip** *
DTR (Double Transfer Rate) Mode	**not implemented**
NOR Flash Part API Name	FS_NOR_PHY_MX25R_API
Tested with	- EFM32G11 Quad SPI controller + NOR Serial Macronix MX25R3235F (32 Mbit)
	- NOR Serial Macronix MX25R8035F (8 Mbit)

* A NOR flash device configured in XIP (eXecute In Place) mode requires only an address (no instruction) to output data, which improves random access time and eliminates the need to shadow code onto RAM for faster execution. If XIP mode is activated, the Quad SPI controller on the microcontroller must support a functional mode called memory-mapped or direct mode. In memory-mapped mode, the external flash memory is mapped in the microcontroller's address space and is seen by the processor as if it was internal memory. This mode allows you to both access data and execute code from the external flash memory. If the NOR chip does not offer XIP mode, the software can still perform a XIP-like mode by using the Quad SPI controller in memory-mapped mode. More details about XIP can be found on the page NOR eXecute In Place .

Specifications

IP Provider	Winbond
Interface to MCU	Serial (SPI)
Data Mode	Single IO
XIP (eXecute In Place) Mode	**not supported by chip** *
DTR (Double Transfer Rate) Mode	**not implemented**
NOR Flash Part API Name	FS_NOR_PHY_W25Q_API
Tested with	- NOR Serial Winbond W25Q80BL (1 MB)

File System Troubleshooting

File System Troubleshooting

- When calling FSBlkDev Open(), I get an error.
- When calling FSVol Open(), I get an error.
- When calling FSFile Open() / FSDir Open() / FSWrkDir Open() / FSFile Write() / FSDir Truncate(), I get an error.
- I want to partition my media, which options do I have?
- I am creating a cache instance, which parameters values should I choose for the cache configuration?
  - One Cache Per Block Device Type
  - One Cache for Multiple Block Devices
- When writing a few files of a few KB in size in a RAM disk, I get the error RTOS ERR VOL FULL.
- I want to open a file or directory on a default volume, can I still do it?

When calling FSBlkDev_Open(), I get an error.

When opening a block device, you may encounter these typical errors:

- RTOS_ERR_SEG_OVF: some internal resources could not be allocated on a given memory segment. Increase the memory segment size. By default the File System Core and Storage layers use the heap region as a default memory segment. In that case, the heap size is increased using LIB_MEM_CFG_HEAP_SIZE. If the heap size is too low, this error is more likely to happen with a NAND or NOR media. Both of these media need a certain quantity of RAM to allocate internal resources for the proper operation of the Flash Translation Layer (FTL).
- RTOS_ERR_BLK_DEV_FMT_INVALID and RTOS_ERR_BLK_DEV_CORRUPTED: these errors indicate that the low-level format is invalid and the block device metadata is corrupted, respectively. They are only returned by NAND and NOR media when a low-level mounting is performed. These errors mean that the FTL was not able to use the metadata retrieved from the physical media. You need to low-level format the media in order to recover from these errors by using FSMedia_LowFmt(). Refer to section Low-Level Formatting a Media for more details.

> If your NAND or NOR metadata becomes corrupted for any reason, please note that a low-level format will prevent you from accessing your file data if the media was formatted with a FAT file system. Basically, your file data will be lost. The low-level format operation builds a new metadata structure on the physical NAND or NOR flash device that will be used by the FTL. The file data are still present but cannot be retrieved anymore as a new file system information must also be written. In the case of NAND media, you may use the debug utility function called FS_NAND_Dump() to take a NAND image snapshot for further analysis of the low-level corruption root cause before low-level formatting.

- RTOS_ERR_BLK_DEV_FMT_INCOMPATIBLE: the device low-level format is incompatible with the user configuration. This error applies only to NAND media. It occurs because the NAND FTL configuration read from the flash block metadata does not match the user configuration. This error may arise in two situations regarding how the NAND FTL configuration is specified. The first situation is that you have manually specified the NAND FTL configuration using the function FS_NAND_FTL_ConfigureLowParams(). In that case, verify your FS_NAND_FTL_CFG structure parameters. The second situation is that the automatic NAND FTL configuration has not worked, and you did not specify a valid NAND FTL configuration. In that case it might be a low-level corruption issue. You need to investigate further to confirm the possible corruption and the solution may imply to low-level format the NAND chip with FSMedia_LowFmt().
- RTOS_ERR_IO: a communication error has occurred with the media. This error mostly applies to removable media such as SD and does not really apply to fixed media such as NAND and NOR. The main reason is that the SD card was not inserted when calling FSBlkDev_Open(). Ensure that the SD card is inserted or use the Media Poll task to detect the SD card insertion and open the device with FSBlkDev_Open() after the detection.

When calling FSVol_Open(), I get an error.

When opening a volume, you may encounter these typical errors:

-

RTOS_ERR_VOL_FMT_INVALID: the volume format information of the file system type is invalid. For a FAT volume, the boot sector signature is invalid, some parameters of the BIOS Parameter Block are invalid, etc. This error usually indicates a volume format corruption and you have to high-level format the volume to recover from it. The high-level format is done with FS_FAT_Fmt(). Refer to section Formatting an Existing Partition for more details.
- RTOS_ERR_PARTITION_INVALID: the partition is invalid (for instance, the first sector signature is incorrect or the partition type is not recognized). The partition on which the volume is open is certainly corrupted. You need to high-level format with FS_FAT_Fmt().
- RTOS_ERR_VOL_CORRUPTED: the volume is corrupted because the file system metadata cannot be understood correctly or is invalid. For a FAT volume, the only situation this error occurs is when the FAT journal module (cf. Optional FAT Journaling) is not able to replay after a power-failure in order to restore the FAT metadata in a known state. In that case, you need to high-level format with FS_FAT_Fmt(). Refer to section Formatting an Existing Partition for more details.

Please note that you will loose your files and directories when performing a high-level format. You may take some extra precautions to try recovering your data before proceeding with a high-level format:

- If the media is removable such as SD and SCSI devices: you can connect the removable media to a computer and try to use any data recovery tools. Indeed, a corrupted volume does not necessarily mean that your data has been completely wiped out or lost. Some or all of your data may still be on the media, but invisible to the Micrium OS File System.
- If the media is fixed such as NAND or NOR: you may use the debug utility function called FS_NAND_Dump() to take a NAND image snapshot for further analysis of the high-level corruption root cause before high-level formatting. NOR does not offer for the moment an image dump utility function.

When calling FSFile_Open() / FSDir_Open() / FSWrkDir_Open() / FSFile_Write() / FSDir_Truncate(), I get an error.

When calling one of these functions, you may encounter these typical errors:

- RTOS_ERR_VOL_CORRUPTED: the volume is corrupted because the file system metadata cannot be understood correctly or is invalid. For a FAT volume, the FAT is corrupted and some cluster chains cannot be followed or allocated for a file or directory; a directory table is corrupted, and the essential "dot dot" entry cannot be found. In that case, you can consider the file or directory as corrupted. You may delete the file or directory entry using FSEntry_Del(). This file and its content will be deleted. For a directory all files and sub-directories, are also deleted.
- RTOS_ERR_IO: a communication error has occurred with the media (for instance, NAND, SD, etc.). In that case, you may retry the file or directory operation if the communication error is temporary.

I want to partition my media, which options do I have?

The MBR is usually the first sector found on partitioned medium. It holds the information on how the logical partitions, containing file systems, are organized (that is presence of a partitions table) on that medium. If you need to partition your media, you have three options:

1. My medium has only one unique partition and no Master Boot Record (MBR) sector is present. In that case, use FSPartition_Init() to create your unique partition and the MBR sector.
2. My medium has only one unique partition and a Master Boot Record (MBR) sector is present. In that case, use FS_FAT_Fmt() to create and format your unique partition containing a FAT file system without an MBR sector.
3. My medium has more than one partition and a Master Boot Record (MBR) sector is necessarily present. In that case, use a combination of FSPartition_Init() and FSPartition_Add() to create the first partition and add any subsequent partitions (usually up to a maximum of four partitions) to the partitions table of the MBR sector.

For all these options, refer to Creating and Formatting Partitions for more details.

I am creating a cache instance, which parameters values should I choose for the cache configuration?

When creating a cache instance, you need to provide a cache configuration (structure FS_CACHE_CFG) composed of:

- a memory segment (.BlkMemSegPtr)
- a cache buffer alignment (.Align)

- a maximum logical block size (.MinLbSize)
- a minimum logical block size (.MaxLbSize)
- a number of cache blocks (.MinBlkCnt)

You need to consider two use cases in order to choose the configuration values:

1. There is one cache instance per block device type. In that situation, the values of .MinLbSize, .MaxLbSize and .MinBlkCnt can be tightly coupled to the block device capacities.
2. There is one unique cache instance shared by multiple block devices. In that situation, the values for .MaxLbSize and .MinBlkCnt shall take into account all block device capacities and the number of block devices.

The tables below present some typical values according to the two situations listed above.

One Cache Per Block Device Type

FSCache_DfltAssign() can be used to ease the configuration in this case.

Parameter	.BlkMemSegPtr	.Align	.MinLbSize	.MaxLbSize	.MinBlkCnt
NAND	DEF_NULL (use LIB Heap)	sizeof(CPU_ALIGN)	2048 or 4096	2048 or 4096	2
NOR	DEF_NULL (use LIB Heap)	sizeof(CPU_ALIGN)	2048	2048	2
RAM	DEF_NULL (use LIB Heap)	sizeof(CPU_ALIGN)	512	512	2
SCSI	DEF_NULL (use LIB Heap)	sizeof(CPU_ALIGN)	512	512	2
SD	DEF_NULL (use LIB Heap)	sizeof(CPU_ALIGN)	512	512	2
Reference Value	-	-	-	-	-
Note			FSBlkDev_LbSizeGet() can be used.	FSBlkDev_LbSizeGet() can be used.	FSBlkDev_AlignReqGet() can be used.

One Cache for Multiple Block Devices

Parameter	.BlkMemSegPtr	.Align	.MinLbSize	.MaxLbSize	.MinBlkCnt
NAND	-	-	2048 or 4096	2048 or 4096	-
NOR	-	-	2048	2048	-
RAM	-	-	512	512	-
SCSI	-	-	512	512	-
SD	-	-	512	512	-
Reference Value	DEF_NULL (use LIB Heap)	sizeof(CPU_ALIGN)	512	4096	4
Note			Minimum among all block devices.	Maximum among all block devices.	Enough blocks to keep a good performance in case of concurrent accesses to different block devices.

Refer to Creating and Assigning a Cache for more details.

When writing a few files of a few KB in size in a RAM disk, I get the error RTOS_ERR_VOL_FULL.

This error may occur because your RAM disk zone is too small. In that case, increase the number of sectors composing your RAM disk to fit the files' size you need to write. A RAM disk is created using the function FS_RAM_Disk_Add(). This function takes a parameter of type FS_RAM_DISK_CFG to describe the RAM disk size in terms of number of sectors and sector size. The main reason why a RAM disk is too small is because the RAM disk has a certain number of sectors reserved for the FAT file system formatting information (FAT metadata). This number of reserved sectors for FAT is subtracted from the total size of the RAM disk specified by FS_RAM_DISK_CFG.LbCnt. Consequently, the number of useful sectors to write your file data is limited. Let's assume a RAM disk volume formatted in FAT12. FAT12 allows you to define a disk with a maximum size of 32 MB, which fits most of the time on the internal or external RAM of an embedded system using the File System module's RAM disk media driver. In that case, the number of sectors reserved for FAT metadata is about 36 sectors. Thus you need at least 18432 bytes (that is 36 sectors * 512 bytes per sector) just for the FAT metadata. In this example, your RAM disk size must be greater than 36 sectors to allow for a few files of a few KB.

Refer to the section RAM Disk for more details about the creation of a RAM disk zone.

I want to open a file or directory on a default volume, can I still do it?

In µC/FS, it was possible to open a file or directory on a default volume by using the back slash "\" or forward slash "/" at the beginning of the absolute path as presented below:

Listing - Open a file or directory on a default volume

```
p_file = fs_fopen("\\file.txt", "w");
p_file = fs_opendir("\\")
```

The default volume is not supported anymore with Micrium OS File System. You must specify the volume name on which the file or directory is opened. Refer to the section File and Directory Names and Paths for more details about the absolute path with Micrium OS File System.

Network

Network

The Network module is a complete TCP/IP stack designed specifically for embedded systems. Built from the ground up with Micrium quality, scalability, and reliability, it enables the rapid configuration of required network options to minimize time-to-market.

The Network module allows for adjustment of the memory footprint based on design requirements. It can be configured to include only those network components necessary to the system. When a component is not used, it is not included in the build, saving valuable memory space.

The Network module support IPv4 and IPv6, DHCP and DNS client; and provides multiple network applications such as HTTP, MQTT, SMTP, SNTP, FTP, Telnet and more.

This manual describes how to initialize, start, and use the Network module. It provides details about the various configuration values and their uses and a porting guide for your hardware. It also provides information such as overview, configuration possibilities, implementation details and examples of typical usage.

More information on Micrium OS Network can be found in the following pages:

- Network Overview
- Integrating Network Into Your Project
- Network Core Example Applications
  - Network Socket Example Applications
- Network Core Configuration
  - Network Core Compile-Time Configurations
  - Network Core Run-Time Configurations
  - Network Interface Controller Configuration
  - Network Interface Start Setup
- Network Core Start-Up
  - Initializing the Network Core Module
  - Adding a Network Interface
  - Starting a Network Interface
- Network Core Programming Guide
  - DNS Client Programming
  - DHCP Client Programming
  - IP Address Programming
  - Interface Programming
  - Socket Programming
  - Multicast Programmimg
  - Wireless Programming
  - Network Tasks Programming
- Network Core Hardware Porting Guide
  - Network Driver Selection Guide
  - Network BSP Functions Guide
  - Network Hardware Information
  - Network Controller Registration to the Platform Manager
- Network Core Troubleshooting
- Application Modules
  - HTTP Client Module
  - HTTP Server Module
  - MQTT Client Module
  - SNTP Client Module
  - SMTP Client Module

- FTP Client Module
- IPerf Module
- Telnet Server Module
- TFTP Client Module
- TFTP Server Module

Network Overview

Network Overview

- Specifications
  - Layer Model
  - Network Protocols
  - Network Interface
  - Socket API
- Features
  - Sockets
  - TCP
  - IPv4
  - IPv6
  - Network Interface
- Limitations
  - Socket
  - IPv4
  - IPv6
- Network Applications

Specifications

Layer Model

A TCP/IP stack is divided into layers of same functionality. Each communication protocol belongs to one of the layers. Table - Network Layer Model in the *Network Overview* page shows the different layers of the TCP/IP model and also the OSI model. At the right, a list of the protocols associated with each layer that the Micrium OS Network module supports.

Table - Network Layer Model

<table>
<tr><th>OSI Model</th><th>TCP/IP Model (Dod)</th><th colspan="4">Micrium OS Network Module</th></tr>
<tr><td rowspan="2">Application</td><td rowspan="4">Application</td><td colspan="4">HTTP server/client, MQTT client, SNTP client, Telnet server, SMTP client, TFTP Server/client, FTP client</td></tr>
<tr><td colspan="4">DHCP client, DNS client</td></tr>
<tr><td>Presentation</td><td colspan="4">SSL, TLS</td></tr>
<tr><td>Session</td><td colspan="4">Socket</td></tr>
<tr><td>Transport</td><td>Transport</td><td colspan="4">TCP, UDP</td></tr>
<tr><td>Network</td><td>Internet</td><td colspan="2">IPv4, IGMP, ICMPv4, ARP</td><td colspan="2">IPv6, MLDP, ICMPv6, NDP</td></tr>
<tr><td rowspan="2">DataLink</td><td rowspan="3">Network Access</td><td colspan="4">IF / 802x</td></tr>
<tr><td colspan="2">Ethernet</td><td colspan="2">WiFi</td></tr>
<tr><td>Physical</td><td>Ethernet Driver</td><td>PHY Driver</td><td>WiFi Driver</td><td>WiFi Manager</td></tr>
</table>

Network Protocols

Here is a list of the network protocols and their corresponding RFC that the Network module supports.

Layer	Protocol		RFC	RFC #
Application	HTTP		Hypertext Transfer Protocol -- HTTP/1.1	2616
			The WebSocket Protocol	6455
	MQTT		NA	NA
	SNTP		Simple Network Time Protocol	4330
	Telnet		Telnet Protocol Specification	854
			Telnet Echo Option	857
			Telnet Suppress Go Ahead Option	858
	DNS		Domain Name	1034, 1035
	DHCP		Dynamic Host Configuration Protocol	2131
Transport	UDP		User Datagram Protocol	768
	TCP		Transmission Control Protocol	793, 813, 879, 896, 2001, 2584, 2988
			TCP Congestion Control	5681
			Computing TCP's Retransmission Timer	6298
Internet	IPv4	IPv4	Internet Protocol	791, 950, 1071 , 3927
			IP Datagram Reassembly Algorithms	4294
			Host Extensions for IP Multicasting	1112
		ARP	Ethernet Address Resolution Protocol	826
		ICMP	Internet Control Message Protocol Specification	792
		IGMPv2	Internet Group Management Protocol, Version 2	2113
	IPv6	IPv6	IPv6 Specification	2460, 2464, 4291
			IPv6 Node Requirements	4294
			Default Address Selection for Internet Protocol version 6	3484
			Basic Socket Interface Extensions for IPv6	3493
		NDP	Neighbor Discovery for IP version 6	4861
		ICMPv6	Internet Control Message Protocol Version 6	4443
		MLD	Multicast Listener Discovery (MLD) for IPv6	2710
Link	Ethernet			894, 1042

Network Interface

Currently, the Network module supports Ethernet interface that uses Ethernet frame and/or IEEE 802.4.

Also, WiFi interfaces can be used as long as the device implements by itself IEEE 802.11. Basically, the network stack only sends some commands to the module to scan for wireless networks, join or leave a specific network. The WiFi module must be able to encrypt and decrypt by itself all the network data. The network data between the host and the wireless module is transferred using IEEE 802.4. Currently, only SPI is supported as a communication bus between the host and the wireless module.

Socket API

The user application interfaces to the Micrium's network stack via a well known API called BSD sockets (or our native socket interface). The application can send and receive data to/from other hosts on the network via this interface. Many books and tutorials exist about BSD sockets programming, mostly the concepts explained in those can be applied to Micrium OS Network module socket programming.

Features

Sockets

- BSD 4.x API
- Many socket options (Blocking mode, keep Alive, windows size, timeouts, etc.)
- Possible to secure a connection (SSL/TLS) by simply using socket options API.

TCP

- Congestion control
- Delayed ACKs
- Nagle Algorithm (Slow Start)
- Keepalive option
- MSS and Windows options
- MSL option
- Listen queue size and child limitation option

IPv4

- Multiple IP addresses per interface
- Automatic routing
- IP fragmentation re-assembly support
- Multicast transmission and reception (IGMPv2)
- ICMP (Echo request/response)

IPv6

- IPv6 Node
- Multiple IP addresses per interface
- IPv6 Multicast (MLD)
- Neighbor Discovery (NDP)
- ICMPv6 (Echo request/response)
- IPv6 Stateless Address Auto configuration
- Duplicate Address Detection
- IP addresses configuration's function with an option to non-blocking.

Network Interface

- Loopback
- Ethernet (802.3 & Ethernet)
- WiFi Module*
- Custom interface

*WiFi module must embed 802.11 protocol (Authentication, Encryption, etc.). It must be possible to bypass the module's TCP/IP stack.

Limitations

Socket

- Socket errno not supported.

IPv4

1. ONLY supports a single default gateway per interface.
2. IPv4 forwarding/routing NOT currently supported.

3. Transmit fragmentation NOT currently supported.
4. IPv4 Security options NOT supported.

IPv6

- Transmit fragmentation NOT currently supported.
- IPv6 Extension Headers is NOT currently supported.
- IPv6 Path MTU not supported.

Network Applications

For more information on the specifications and limitations of the available network applications, refer to the following sections:

- HTTP Client Overview
- HTTP Server Overview
- MQTT Client Overview
- SNTP Client Overview
- Telnet Server Overview
- IPerf Module
- SMTP Client Module
- FTP Client Module
- TFTP Server Module
- TFTP Client Module

Integrating Network Into Your Project

Micrium OS Network module is composed of several components, each of which is a set of files that implement specific functions. The Network module consists of one mandatory component named "Network" and several optional components for the various network applications and drivers. To use Network, you must add these files to your project and populate your RTOS Description File.

Starting the Network Module Quickly

Micrium offers a set of example applications that demonstrate some of the features of the Network module and help you start the development of your application. We recommend that you start from one of these examples.

The following sections describe the more complex example applications available.

- FTP Client Example Application
- HTTP Client Example Applications
- HTTP Server Example Applications
- IPerf Example Applications
- MQTT Client Example Applications
- Network Core Example Applications
- SMTP Client Example Applications
- SNTP Client Example Applications
- Telnet Server Example Applications
- TFTP Client Example Application

Configuring Network

Micrium OS Network module can be configured to optimize memory usage and features.

- See Network Core Compile-Time Configurations for more information about configuring the network core module at compile-time.
- See Network Core Run-Time Configurations how the network core module can be configured at run-time.

Each of the network's applications (HTTP, MQTT, etc.) have also their own compile-time and run-time configurations. Refer to their specific sections for more details.

Network Core Example Applications

Network Core Example Applications

This section describes the examples that are related to the TCP/IP module of Micrium OS.

- Network Module(s) Initialization Example
  - Description
  - Configuration
  - Location
  - API
  - Notes
- Network Core Initialization Example
  - Description
  - Configuration
    - Mandatory
    - Optional
  - Location
  - API
  - Notes
- Start Network Interface(s) Example
  - Description
  - Configuration
    - Optional
  - Location
  - API
  - Notes

Network Module(s) Initialization Example

Description

This example shows how to initialize network module(s). It basically calls the self-contained module examples when the module is available and the example is enabled. See Example Applications section for further information about how to enable an example. It accomplishes the following tasks:

- Initialize the network core using:
  - Network Core Initialization Example
- Initialize network application module(s), when it is part of your project and the example activated using the following example:
  - HTTP Client Initialization Example
  - HTTP Server Initialization Example
  - MQTT Client Initialization Example
  - SNTP Client Initialization Example
  - SMTP Client Initialization Example
  - IPerf Initialization Example
  - Telnet Server Initialization Example

Configuration

No specific configuration applies to this example. See sub-module example(s) for further information about the configuration.

Location

> /examples/net/ex_network_init.c
>
> /examples/net/ex_network_init.h

API

API	Description
Ex_NetworkInit()	This function performs the different example steps mentioned in the section Description . The function must be called by your application task prior to calling any other file system Network examples.

Notes

This example doesn't start the network core. Once the module are initialized the network interface must be started as described by Network Core Start Interfaces Example

Network Core Initialization Example

Description

This is a generic example for the TCP/IP core module initialization. It accomplishes the following tasks:

1. Change default task's stacks size, if specified by the example configuration
2. Initialize the TCP/IP core module
3. Add one Ethernet controller, if enabled.
4. Add one WiFi module, if enabled.
5. Change default the task's priority, if specified by the example configuration

> By default, this example will assume the presence of an Ethernet controller and a WiFi module when available in Micrium OS. The example assumes that the default interfaces are named "eth0" for the Ethernet controller and "wifi0" for the WiFi module.

Configuration

Mandatory

The following #define must be added in ex_description.h to allow the default main example (ex_main.c) to initialize the network module correctly:

#define	Description
EX_NETWORK_INIT_AVAIL	Lets the upper example layer know that the Network Initialization example is present and must be called by other examples.

Optional

The following #define can be added to ex_description.h, as described in Example Applications section, to change default configuration value used by the example:

#define	Default value	Description
EX_NET_CORE_TASK_STK_SIZE	Not defined, used default set by the core (1024)	When defined, the default stack size assigned to the TCP/IP core task stack is changed at run time to this value.

#define	Default value	Description
EX_NET_CORE_SRV_TASK_STK_SIZE	Not defined, used default set by the core (512)	When defined, the default stack size assigned to the TCP/IP core services task stack is changed at run time to this value.
EX_NET_CORE_TASK_PRIO	Not defined, used default set by the core (16)	When defined, the TCP/IP core task priority is changed at run time to this value.
EX_NET_CORE_SRV_TASK_PRIO	Not defined, used default set by the core (28)	When defined, the TCP/IP core services task priority is changed at run time to this value.

Location

```
/examples/net/core_init/ex_net_core_init.c

/examples/net/core_init/ex_net_core_init.h
```

API

API	Description
Ex_Net_CoreInit()	Initialize the Network TCP/IP core and add an Ethernet Interface and/or a WiFi Interface.

Notes

This example doesn't start the network core. Once the module is initialized, the network interface must be started as described by Network Core Start Interfaces Example

Start Network Interface(s) Example

Description

This is a generic example that shows how to start network interface(s). It accomplishes the following tasks:

- If Ethernet is available:
  - Configure IP addresses
  - Start the Ethernet Interface
  - Wait setup completed (DHCP, IPv6 Autoconf and more)
- If WiFi is available:
  - Configure IP addresses
  - Start the Ethernet Interface
  - Wait setup completed (DHCP, IPv6 Autoconf and more)

Configuration

Optional

The following #define can be added to ex_description.h, as described in Example Applications section, to change default configuration value used by the example:

#define	Default value	Description
EX_NET_CORE_IF_ETHER_MAC_ADDR	DEF_NULLUse default provided by the BSP	Specify the MAC address to assign to the Ethernet interface
EX_NET_CORE_IF_ETHER_IPv4_DHCP_EN	DEF_ENABLED	Specify if DHCP-IPv4 must be performed on the Ethernet interface

#define	Default value	Description
EX_NET_CORE_IF_ETHER_IPv4_STATIC_ADDR	"10.10.10.111"	Change the static IPv4 address to assign to the Ethernet interface.If DHCP fails, you can always access the target using this address.If defined to DEF_NULL, no static address will be configured.
EX_NET_CORE_IF_ETHER_IPv4_STATIC_MASK	"255.255.255.0"	Specify the static IPv4 address mask to assign to the Ethernet interface.
EX_NET_CORE_IF_ETHER_IPv4_STATIC_GATEWAY	"10.10.10.1"	Specify the static IPv4 address gateway to assign to the Ethernet interface.
EX_NET_CORE_IF_ETHER_IPv6_DAD_EN	DEF_DISABLED	Specify if IPv6 Duplicate Address detection must be performed on IPv6 address on the Ethernet interface
EX_NET_CORE_IF_ETHER_IPv6_AUTOCONF_EN	DEF_ENABLED	Specify if IPv6 Autoconf must be performed on the Ethernet interface
EX_NET_CORE_IF_ETHER_IPv6_AUTOCONF_DAD_EN	DEF_ENABLED	Specify if IPv6 Duplicate Address detection must be performed on IPv6 Autoconf address on the Ethernet interface
EX_NET_CORE_IF_ETHER_IPv6_STATIC_LOCAL_ADDR	"fe80::1111:1111"	Change the static IPv6 address to assign to the Ethernet interface.If Autoconf fails, you can always access the target using this address.If defined to DEF_NULL, no static address will be configured.
EX_NET_CORE_IF_ETHER_IPv6_STATIC_LOCAL_PREFIX_LEN	64u	IPv6 static address prefix len
EX_NET_CORE_IF_WIFI_MAC_ADDR	DEF_NULL	Specify the MAC address to assign to the WiFi interface
EX_NET_CORE_IF_WIFI_IPv4_DHCP_EN	DEF_ENABLED	Specify if DHCP-IPv4 must be performed on the WiFi interface
EX_NET_CORE_IF_WIFI_SSID	"Wifi_AP_SSID"	Specify the Wi-Fi Access point to connect on
EX_NET_CORE_IF_WIFI_PASSWORD	"password"	Specify the Wi-Fi Access point password
EX_NET_CORE_IF_WIFI_IPv4_STATIC_ADDR	"192.168.1.222"	Change the static IPv4 address to assign to the WiFi interface.If DHCP fails, you can always access the target using this address.If defined to DEF_NULL, no static address will be configured.
EX_NET_CORE_IF_WIFI_IPv4_STATIC_MASK	"255.255.255.0"	Specify the static IPv4 address mask to assign to the WiFi interface.
EX_NET_CORE_IF_WIFI_IPv4_STATIC_GATEWAY	"192.168.1.1"	Specify the static IPv4 address gateway to assign to the WiFi interface.
EX_NET_CORE_IF_WIFI_IPv6_DAD_EN	DEF_DISABLED	Specify if IPv6 Duplicate Address detection must be performed on IPv6 address on the WiFi interface
EX_NET_CORE_IF_WIFI_IPv6_AUTOCONF_EN	DEF_ENABLED	Specify if IPv6 Autoconf must be performed on the WiFi interface
EX_NET_CORE_IF_WIFI_IPv6_AUTOCONF_DAD_EN	DEF_ENABLED	Specify if IPv6 Duplicate Address detection must be performed on IPv6 Autoconf address on the WiFi interface

#define	Default value	Description
EX_NET_CORE_IF_WIFI_IPv6_STATIC_LOCAL_ADDR	"fe80::1111:2222"	Change the static IPv6 address to assign to the WiFi interface.If Autoconf fails, you can always access the target using this address.If defined to DEF_NULL, no static address will be configured.
EX_NET_CORE_IF_WIFI_IPv6_STATIC_LOCAL_PREFIX_LEN	64 u	IPv6 static address prefix len

Location

/examples/net/core_init/ex_net_core_init.c

/examples/net/core_init/ex_net_core_init.h

API

API	Description
Ex_Net_CoreStartIF()	Configure and starts network interface(s) and wait setup complete.

Notes

The following API can be called instead of Ex_Net_CoreStartIF() , if you want to control the start flow and you don't want to wait for the setup completion on all interfaces.

API	Description
Ex_Net_CoreStartEther()	Start the Ethernet Interface.
Ex_Net_CoreStartWiFi()	Start the WiFi Interface.

Network Socket Example Applications

This section describes the examples that are related to the TCP/IP module of Micrium OS.

- TCP Echo Client
  - Description
  - Configuration
  - Location
  - API
  - Notes
- TCP Echo Server
  - Description
  - Configuration
  - Location
  - API
  - Notes
- UDP Echo Client
  - Description
  - Configuration
  - Location
  - API
  - Notes
- UDP Echo Server
  - Description
  - Configuration
  - Location
  - API
  - Notes
- Multicast Echo Server

- Description
- Configuration
- Location
- API
- Notes
- TCP SSL/TLS Client Connection
  - Description
  - Configuration
  - Location
  - API
  - Notes
- TCP SSL/TLS Server
  - Description
  - Configuration
  - Location
  - API
  - Notes

TCP Echo Client

Description

This is a generic example that shows how to create a simple TCP echo client. It accomplishes the following tasks:

- Open a socket
- Configure socket's address
- Connect to the server
- Transmit data to the server
- Receive echo response from the server
- Close socket

Configuration

None.

Location

```
/examples/net/socket/ex_net_sock_tcp_client.c

/examples/net/socket/ex_net_sock.h

/examples/net/socket/ex_net_sock_tcp_server.py
```

API

API	Description
Ex_Net_SockTCP_Client()	Create a TCP Echo client

Notes

The file ex_net_sock_tcp_server.py, can be used on any PC to create the associate TCP echo server.

TCP Echo Server

Description

This is a generic example that shows how to create a simple TCP echo IPv4 or IPv6 server. It accomplishes the following tasks:

Open a socket
- Configure socket's address
- Bind the socket.
- Receive data on the socket
- Transmit to source the data received.
- Close socket on fatal fault error.

Configuration

None.

Location

```
/examples/net/socket/ex_net_sock_tcp_server.c

/examples/net/socket/ex_net_sock.h
```

API

API	Description
Ex_Net_SockTCP_ServerIPv4()	Create an Echo server which accepts IPv4 connection
Ex_Net_SockTCP_ServerIPv6()	Create an Echo server which accepts IPv6 connection

Notes

None

UDP Echo Client

Description

This is a generic example that shows how to create a simple UDP echo client. It accomplishes the following tasks:

- Open a socket
- Configure socket's address
- Transmit data to the server
- Receive echo response from the server
- Close socket.

Configuration

None.

Location

```
/examples/net/socket/ex_net_sock_udp_client.c

/examples/net/socket/ex_net_sock.h
```

API

API	Description
Ex_Net_SockUDP_Client()	Create an UDP Echo client

Notes

None

UDP Echo Server

Description

This is a generic example that shows how to create a simple UDP echo IPv4 or IPv6 server. It accomplishes the following tasks:

- Open a socket
- Configure socket's address
- Bind the socket.
- Receive data on the socket
- Transmit to source the data received.
- Close socket on fatal fault error.

Configuration

None.

Location

```
/examples/net/socket/ex_net_sock_tcp_server.c

/examples/net/socket/ex_net_sock.h
```

API

API	Description
Ex_Net_SockUDP_ServerIPv4()	Create an Echo server which accepts IPv4 connection
Ex_Net_SockUDP_ServerIPv6()	Create an Echo server which accepts IPv6 connection

Notes

None

Multicast Echo Server

Description

This is a generic example that shows how to create an UDP Multicast echo server using IPv4 address. It accomplishes the following tasks:

- Join the multicast group
- Open a socket
- Configure socket's address
- Bind the socket.
- Receive data on the socket
- Transmit to source the data received.
- Close socket on fatal fault error.

Configuration

None.

Location

```
/examples/net/socket/ex_net_sock_mcast_echo_server.c

/examples/net/socket/ex_net_sock.h
```

API

API	Description
Ex_Net_SockMCastEchoServer()	Create a Multicast Echo server which accept IPv4 connection

Notes

The file ex_net_sock_mcast_echo_client.py, can be used on any PC to create the associate TCP echo client.

TCP SSL/TLS Client Connection

Description

This is a generic example that shows how to connect a client to a server over SSL/TLS. It accomplishes the following tasks:

- Install CA certificate
- Open a TCP socket
- Configure socket's option to be secure
- Configure socket's address
- Connect; establish a secure connection with the server.

Configuration

None.

Location

```
/examples/net/socket/ex_net_sock_secure_client.c

/examples/net/socket/ex_net_sock.h

/examples/net/socket/ex_go_daddy.cer
```

API

API	Description
Ex_Net_SockSecureClientConnect()	Initialize a client secure socket.

Notes

None.

TCP SSL/TLS Server

Description

This is a generic example that shows how to create a server that accept secure connection using SSL/TLS. It accomplishes the following tasks:

- Open a TCP socket
- Configure socket's option to be secure
- Bind the socket
- Listen

Configuration

None.

Location

```
/examples/net/socket/ex_net_sock_secure_serverc

/examples/net/socket/ex_net_sock.h
```

API

API	Description
Ex_Net_SockSecureServerInit()	Initialize server's listen socket to accept connection over TLS/SSL.

Notes

None

Network Core Configuration

Network Core Configuration

Micrium OS Network core includes the TCP/IP stack and the service modules DHCP and DNS. Before using these components, you must properly configure the network core. There are two groups of configuration parameters for the network core:

- Network Core Compile-Time Configurations
- Network Core Run-Time Configurations

Furthermore, to successfully configure a network interface, there are two additional configuration layers:

- Network Interface Controller Configuration
- Network Interface Start Setup

Network Core Compile-Time Configurations

To configure the Micrium OS Network core module, you make use of approximately 50 #defines located in your application's net_cfg.h file. The core module uses #defines because they allow code and data sizes to be scaled based on the features you choose to enable and the number of network objects you configure. This allows the ROM and RAM footprints of the Network module to be adjusted based on your application's requirements.

Most of the #defines should be configured with the default configuration values. A handful of these values will likely never be changeable because there is currently only one configuration choice available. This leaves approximately a dozen #defines that you might need to configure.

We recommend that you begin the configuration process with the default values, which are shown in **bold**.

The topics in this section are organized following the order in the template configuration file, net_cfg.h

- Mandatory Configurations
  - Debug Features Configuration
  - Counters Configuration
  - Statistic Pool Configuration
  - Network Interfaces Configuration
  - Address Resolution Protocol (ARP) Configuration
  - Neighbor Discovery Protocol (NDP) Configuration
  - IPv4 Layer Configuration
  - IPv6 Layer Configuration
  - Multicast Configuration (IGMP and MLDP)
  - Socket Layer Configuration
  - TCP Layer Configuration
  - UDP Layer Configuration
  - Transport Layer Security Configuration
  - Core Modules Configuration
- Advanced Configurations

Mandatory Configurations

Debug Features Configuration

Table - Debug Feature Constants

Constant	Description	Possible Values
NET_DBG_CFG_MEM_CLR_EN	Clears the internal network data structures when allocated or de-allocated. By clearing, all bytes in internal data structures are set to ‘0’ or to their default initialization values. This configuration is typically set to DEF_DISABLED unless you need to examine the contents of the internal network data structures for debugging purposes. Having the internal network data structures cleared generally helps to differentiate between “proper” data and “pollution”. Note that the network performance can be affected when enabled.	DEF_ENABLED or DEF_DISABLED

Counters Configuration

The Network core module contains code that increments counters to keep track of statistics such as the number of packets received, the number of packets transmitted, etc. It also contains counters that are incremented when error conditions are detected.

Table - Counter Configuration Constants

Constant	Description	Possible Values
NET_CTR_CFG_STAT_EN	Allocates code and data space used to keep track of statistics.	DEF_ENABLED or DEF_DISABLED
NET_CTR_CFG_ERR_EN	Allocates code and data space used to keep track of errors.	DEF_ENABLED or DEF_DISABLED

Statistic Pool Configuration

The Network core module uses pools to allocate many of its internal objects. Using the macro configurations below, the statistics can be enabled for each of those pools individually.

Table - Statistic Pool Configuration Constants

Constant	Description	Possible Values
NET_STAT_POOL_BUF_EN	Allocates code and data space for the Buffers statistic pool.	DEF_ENABLED or DEF_DISABLED
NET_STAT_POOL_ARP_EN	Allocates code and data space for the ARP cache statistic pool.	DEF_ENABLED or DEF_DISABLED
NET_STAT_POOL_NDP_EN	Allocates code and data space for the NDP cache statistic pool.	DEF_ENABLED or DEF_DISABLED
NET_STAT_POOL_IGMP_EN	Allocates code and data space for the IGMP host group statistic pool.	DEF_ENABLED or DEF_DISABLED
NET_STAT_POOL_MLDP_EN	Allocates code and data space for the MLDP host group statistic pool.	DEF_ENABLED or DEF_DISABLED
NET_STAT_POOL_TMR_EN	Allocates code and data space for the Network Timer statistic pool.	DEF_ENABLED or DEF_DISABLED
NET_STAT_POOL_SOCK_EN	Allocates code and data space for the Socket statistic pool.	DEF_ENABLED or DEF_DISABLED
NET_STAT_POOL_CONN_EN	Allocates code and data space for the Connection statistic pool.	DEF_ENABLED or DEF_DISABLED
NET_STAT_POOL_TCP_CONN_EN	Allocates code and data space for the TCP Connection statistic pool.	DEF_ENABLED or DEF_DISABLED

Network Interfaces Configuration

Table - Interface Configuration Constants

Constant	Description	Possible Values
NET_IF_CFG_MAX_NBR_IF	Specifies the maximum number of network interfaces that the network module can create.	**1 u** if a single network interface is present.
NET_IF_CFG_LOOPBACK_EN	Allocates code and data space used to support the loopback interface for internal-only communication.	DEF_ENABLED or DEF_DISABLED
NET_IF_CFG_TX_SUSPEND_TIMEOUT_MS	Specifies the network interface transmit suspend timeout value in milliseconds.	**1 u**
NET_IF_CFG_WAIT_SETUP_READY_EN	Specifies whether the wait feature for the interface setup is enabled. When enabled, your application can wait for the network interface setup to complete by using the function NetIF_WaitSetupReady() . The network interface setup consists mainly of IP address configurations. The NetIF_WaitSetupReady() function will wait for all configuration processes that were specified in the start function to complete before returning.	DEF_ENABLED or DEF_DISABLED

Address Resolution Protocol (ARP) Configuration

Table - ARP Configuration Constants

Constant	Description	Possible Values
NET_ARP_CFG_CACHE_NBR	Specifies the number of ARP cache entries.	3 u

ARP caches the mapping of IPv4 addresses to physical (i.e., MAC) addresses. NET_ARP_CFG_NBR_CACHE specifies the number of ARP cache entries. Each cache entry requires approximately 18 bytes of RAM, plus seven pointers, plus a hardware address and protocol address (10 bytes assuming Ethernet interfaces and IPv4 addresses).

The number of ARP caches required by your application depends on how many different hosts your application is expected to communicate with. If your application communicates *only* with hosts on remote networks via the local network's default gateway (i.e., router), then you need to configure only a single ARP cache.

To test the network module with a small network, the default number of three ARP caches should be sufficient.

Neighbor Discovery Protocol (NDP) Configuration

Table - NDP Configuration Constants

Constant	Description	Possible Values
NET_NDP_CFG_CACHE_NBR	Configures the number of NDP Neighbor cache entries.	6 u
NET_NDP_CFG_DEST_NBR	Configures the number of NDP Destination cache entries.	5 u
NET_NDP_CFG_PREFIX_NBR	Configures the number of NDP Prefix entries.	5 u
NET_NDP_CFG_ROUTER_NBR	Configures the number of NDP Router entries.	1 u

NDP caches the mapping of IPv6 addresses to physical (i.e., MAC) addresses. NET_NDP_CFG_NBR_CACHE specifies the number of NDP Neighbor cache entries. Each cache entry requires approximately 18 bytes of RAM, plus seven pointers, plus a hardware address and protocol address (22 bytes assuming Ethernet interfaces and IPv6 addresses).

NDP also caches recent IPv6 destination addresses by mapping next-hop address to the final destination address, thereby relieving the network stack from having to re-calculate the next-hop for each packet to send. NET_NDP_CFG_DEST_NBR

specifies the number of NDP destination caches available for the TCP/IP stack.

In IPv6, routers send router advertisement messages to inform hosts about different values such as the on-link IPv6 prefix. Those on-link prefixes are stored in an NDP prefix list. NET_NDP_CFG_PREFIX_NBR specifies the number of prefix entries available in the list.

IPv6 defines an algorithm to chose the optimal router on the network to use to transmit packets in case more than one IPv6 router is available. NET_NDP_CFG_ROUTER_NBR specifies the number of router entries that can be stored by the TCP/IP stack.

IPv4 Layer Configuration

Table - IPv4 Configuration Constants

Constant	Description	Possible Values
NET_IPv4_CFG_EN	Enables the IPv4 module.	DEF_ENABLED or DEF_DISABLED
NET_IPv4_CFG_LINK_LOCAL_EN	Enables the IPv4 Link-Local address module.	DEF_ENABLED or DEF_DISABLED
NET_IPv4_CFG_IF_MAX_NBR_ADDR	Specifies the maximum number of IPv4 addresses that may be configured for all network interfaces at run-time.	At least **1**

IPv6 Layer Configuration

Constant	Description	Possible Values
NET_IPv6_CFG_EN	Enables the IPv6 module.	DEF_ENABLED or DEF_DISABLED
NET_IPv6_CFG_ADDR_AUTO_CFG_EN	Enables the IPv6 Stateless Address Auto-Configuration module.	DEF_ENABLED or DEF_DISABLED
NET_IPv6_CFG_DAD_EN	Enables the Duplication Address Detection (DAD) module.	DEF_ENABLED or DEF_DISABLED
NET_IPv6_CFG_IF_MAX_NBR_ADDR	Specifies the maximum number of IPv6 addresses that may be configured for all network interfaces at run-time.	At least **2**

Table - IPv6 Configuration Constants

Multicast Configuration (IGMP and MLDP)

Table - Multicast Configuration Constants

Constant	Description	Possible Values
NET_MCAST_CFG_IPv4_RX_EN	Enables multicast support in reception for IPv4.	DEF_ENABLED or DEF_DISABLED
NET_MCAST_CFG_IPv4_TX_EN	Enables multicast support in transmission for IPv4.	DEF_ENABLED or DEF_DISABLED
NET_MCAST_CFG_HOST_GRP_NBR_MAX	Specifies the maximum number of IGMP and MLDP host groups that may be joined at any time.	4 u

NET_MCAST_CFG_MAX_NBR_HOST_GRP configures the maximum number of IGMP and MLDP host groups that may be joined at any time. Each group entry requires approximately 12 bytes of RAM, plus three pointers, plus a protocol address (4 bytes assuming IPv4 address).

The number of host groups required by your application depends on how many host groups you expect your application to join at a given time. Since each configured multicast address requires its own host group, we recommend that you configure at least one host group per multicast address used by the application, plus one additional host group. Thus for a single multicast address, we recommend that you set NET_MCAST_CFG_MAX_NBR_HOST_GRP to an initial value of 2.

If IPv6 is enabled, MLDP multicast is enabled automatically and NET_MCAST_CFG_MAX_NBR_HOST_GRP must be adjusted because an interface with IPv6 support will need at least two multicast address.

Socket Layer Configuration

The Network module supports BSD 4.*x* sockets and basic socket API for the TCP/UDP/IP protocols.

Table - Socket Configuration Constants

Constant	Description	Possible Values
NET_SOCK_CFG_SOCK_NBR_TCP	Specifies the total number of TCP connections.	5
NET_SOCK_CFG_SOCK_NBR_UDP	Specifies the total number of UDP connections.	2
NET_SOCK_CFG_SEL_EN	Enables socket select functionality.	DEF_ENABLED or DEF_DISABLED
NET_SOCK_CFG_CONN_ACCEPT_Q_SIZE_MAX	Specifies the maximum size of the stream-type sockets' accept queue.	2
NET_SOCK_CFG_RX_Q_SIZE_OCTET	Specifies the buffer size of the socket receive queue.	4096
NET_SOCK_CFG_TX_Q_SIZE_OCTET	Specifies the buffer size of the socket transmit queue.	4096

TCP Layer Configuration

Table - TCP Configuration Constants

Constant	Description	Possible Values
NET_TCP_CFG_EN	Enables the TCP module.	DEF_ENABLED or DEF_DISABLED

UDP Layer Configuration

Table - UDP Configuration Constants

Constant	Description	Possible Values
NET_UDP_CFG_RX_CHK_SUM_DISCARD_EN	Used to determine whether received UDP packets without a valid checksum are discarded or are handled and processed. Before a UDP datagram checksum is validated, it is necessary to check whether the UDP datagram was transmitted with or without a computed checksum.	DEF_ENABLED or DEF_DISABLED
NET_UDP_CFG_TX_CHK_SUM_EN	Used to determine whether UDP checksums are computed for transmission to other hosts.	DEF_ENABLED or DEF_DISABLED

Transport Layer Security Configuration

Table - Security Management Constants

Constant	Description	Possible Values
NET_SECURE_CFG_MAX_NBR_SOCK_SERVER	Specifies the total number of server secure sockets.	5

Constant	Description	Possible Values
NET_SECURE_CFG_MAX_NBR_SOCK_CLIENT	Specifies the total number of client secure sockets.	5
NET_SECURE_CFG_MAX_CERT_LEN	Specifies the maximum length (in octets) of server certificates.	2048
NET_SECURE_CFG_MAX_KEY_LEN	Specifies the maximum length (in octets) of server keys.	2048
NET_SECURE_CFG_MAX_NBR_CA	Specifies the maximum number of certificate authorities that can be installed.	1
NET_SECURE_CFG_MAX_CA_CERT_LEN	Specifies the maximum length (in octets) of certificate authority certificates.	2048

Core Modules Configuration

The network core module includes basic network services such as DHCP and DNS client, which can be disabled if not required.

Table - Core Modules Configuration

Constant	Description	Possible Values
NET_DHCP_CLIENT_CFG_MODULE_EN	Enables the DHCP client module.	DEF_ENABLED or DEF_DISABLED
NET_DNS_CLIENT_CFG_MODULE_EN	Enables the DNS client module.	DEF_ENABLED or DEF_DISABLED

Advanced Configurations

The Network module has many compile-time configurations that are already defined by the module with default values. Those values should work for most cases, but your application can still override them if necessary.

Table - Network Core Default Configuration

{@Manual intervention complex table}

Name	Description	Default Value
NET_TMR_CFG_NBR_TMR	By default, the network timer pool size has no limits.To limit the maximum number of network timers, you can adjust this configuration.	LIB_MEM_BLK_QTY_UNLIMITED
Socket		
NET_SOCK_DFLT_BSD_EN	Enable or Disable BSD API and data type declaration. By default, the API and declaration are included.Note that your application doesn't require BSD API and you experimenting issue with duplication declaration with your compiler you can disable the BSD compatibility. If you BSD compatibility is required and you experimenting duplication declaration you have to change your toolchain setting to remove the BSD and POSIX declaration.	DEF_ENABLED

Name	Description	Default Value
NET_SOCK_DFLT_NO_BLOCK_EN	Sets all sockets as non-blocking. By default, sockets are set to block.Note that it's possible to change the socket's blocking mode at run-time using the Socket Option API.	DEF_DISABLED
NET_SOCK_DFLT_PORT_NBR_RANDOM_BASE	Specifies the initial random port number. By default, the random port number start at 49152.	49152
NET_SOCK_DFLT_TIMEOUT_RX_Q_MS	Specifies the timeout for the socket reception queue in milliseconds.When a socket is set as blocking, the specified timeout value is used.Timeout values may also be configured to never time out with the network time constant NET_TMR_TIME_INFINITE.Note that it's possible to change at run-time any timeout values using the Socket Option API.	10000
NET_SOCK_DFLT_TIMEOUT_CONN_REQ_MS	Specifies the default timeout for the socket connection connect request in milliseconds.When a socket is set as blocking the specified timeout value is used.Timeout values may also be configured to never time out with the network time constant NET_TMR_TIME_INFINITE.Note that it's possible to change at run-time any timeout values using the Socket Option API.	10000
NET_SOCK_DFLT_TIMEOUT_CONN_ACCEPT_MS	Specifies the timeout for the socket connection accept in milliseconds.When a socket is set as blocking the following specified timeout value is used.Timeout values may also be configured to never time out with the network time constant NET_TMR_TIME_INFINITE.Note that it's possible to change at run-time any timeout values using the Socket Option API.	10000

Name	Description	Default Value
NET_SOCK_DFLT_TIMEOUT_CONN_CLOSE_MS	Specifies the timeout for the socket connection close in milliseconds.When a socket is set as blocking the following specified timeout value is used.Timeout values may also be configured to never time out with the network time constant NET_TMR_TIME_INFINITE.Note that it's possible to change at run-time any timeout values using Socket Option API.	10000
TCP		
NET_TCP_DFLT_RX_WIN_SIZE_OCTET	Specifies the TCP RX window size.By default, the TCP RX window is set to equal the socket RX queue size. TCP windows must be properly configured to optimize performance (see section Configuring Window Sizes).Note that it's possible to decrease window size at run time using Socket Option API.	NET_SOCK_CFG_RX_Q_SIZE_OCTET

Name	Description	Default Value
NET_TCP_DFLT_TX_WIN_SIZE_OCTET	Specifies the TCP TX window size.By default, the TCP TX window is set to equal the socket TX queue size. TCP windows must be properly configured to optimize performance (see section Configuring Window Sizes).Note that it's possible to decrease window size at run time using Socket Option API.	NET_SOCK_CFG_TX_Q_SIZE_OCTET
NET_TCP_DFLT_TIMEOUT_CONN_MAX_SEG_SEC	Specifies the TIME_WAIT connection timeout.As shown in the TCP state diagram (see RFC #793), before moving to the 'CLOSED' state, the connection waits for a short period (typically 2 minutes), which is called the TIME_WAIT state. This timeout prevents the TCP connection from becoming immediately available for subsequent TCP connections. But this delay can be a problem for embedded systems that cannot handle many simultaneous connections, especially when many TCP connections are made in a small period of time. Therefore this timeout is set to 0 by default to avoid this kind of problem, and the connection is made available as soon as the TIME_WAIT state is reached.Note that it is possible to change the MSL timeout for a specific TCP connection using Socket Option API.	0

Name	Description	Default Value
NET_TCP_DFLT_TIMEOUT_CONN_FIN_WAIT_2_SEC	Specifies the FIN_WAIT_2 connection timeout.In the FIN_WAIT_2 state, the local endpoint is waiting for a connection termination request from the remote endpoint. To prevent a connection from being left open indefinitely if the termination request never arrives, the TCP connection's timer is set to 15 seconds by default, and when it expires the connection is dropped.	15
NET_TCP_CFG_NBR_CONN	Specifies the total number of TCP connections.The number of TCP connections is based on the number of TCP sockets and the accept queue size when TIME_WAIT is set to 0 ms. However, since the default TIME_WAIT value can be modified, you might need to increase the number of TCP connections in order to establish new connections when waiting for the TIME_WAIT expiration.	0
NET_TCP_DFLT_TIMEOUT_CONN_ACK_DLY_MS	Specifies the ACK delay in milliseconds.By default, an ACK is generated within 500 ms of the arrival of the first unacknowledged packet, as specified in RFC #2581, Section 4.2.	500
NET_TCP_DFLT_TIMEOUT_CONN_RX_Q_MS	Specifies the default timeout for the TCP Reception Queue in milliseconds.This value determines how long a socket is allowed to remain blocking when receiving.Timeout values may also be configured to never time out with the network time constant NET_TMR_TIME_INFINITE.Note that it's possible to change at run-time any timeout values using the Socket Option API.	1000
NET_TCP_DFLT_TIMEOUT_CONN_TX_Q_MS	Specifies the default timeout for the TCP Transmission Queue, in milliseconds.This value determines how long a socket is allowed to remain blocking when transmitting.Timeout values may also be configured to never time out with network time constant, NET_TMR_TIME_INFINITE.Note that it's possible to change at run-time any timeout values using the Socket Option API.	1000
Checksum Offload		
By default, all checksum are validated by the stack. However, it is possible to enable or disable validation for specific checksums. Enable any of the following defines in your net_cfg.h file to change the default behavior. Turning off checksum validation can also provide a performance boost.		
NET_IPV4_CFG_CHK_SUM_OFFLOAD_RX_EN	Turn off validation in reception for IPv4 checksums.	DEF_DISABLED
NET_IPV4_CFG_CHK_SUM_OFFLOAD_TX_EN	Turn off calculation in transmission for IPv4 checksums.	DEF_DISABLED
NET_ICMP_CFG_CHK_SUM_OFFLOAD_RX_EN	Turn off validation in reception for ICMP checksums.	DEF_DISABLED
NET_ICMP_CFG_CHK_SUM_OFFLOAD_TX_EN	Turn off calculation in transmission for ICMP checksums.	DEF_DISABLED

Name	Description	Default Value
NET_UDP_CFG_CHK_SUM_OFFLOAD_RX_EN	Turn off validation in reception for UDP checksums.	DEF_DISABLED
NET_UDP_CFG_CHK_SUM_OFFLOAD_TX_EN	Turn off calculation in transmission for UDP checksums.	DEF_DISABLED
NET_TCP_CFG_CHK_SUM_OFFLOAD_RX_EN	Turn off validation in reception for TCP checksums.	DEF_DISABLED
NET_TCP_CFG_CHK_SUM_OFFLOAD_TX_EN	Turn off calculation in transmission for TCP checksums.	DEF_DISABLED

Network Core Run-Time Configurations

- Network Core Configuration
- Optional Configurations
  - Specific Function Calls
    - Memory Segments
    - Network Core Task's Stack Size
    - Network Core Services Task's Stack Size
    - DNS Configuration
  - Externalized Structure
- Post-Init Configurations

This section describes the application-specific configurations of the Micrium OS Network module, that are specified at run-time.

- For more information on how default run-time configuration values can be changed or used, see Initialization of the Common Module .
- For more information on how to initialize any Micrium OS modules, see Stacks Initialization Methods .

Network Core Configuration

To initialize the Micrium OS Network Core module, you must call the function Net_Init(). This function does not have any mandatory configuration argument.

Optional Configurations

The configurations described below are optional; if you do not set them in your application, the default values will apply. The default values can be retrieved via the structure Net_InitCfgDflt.

There are two different ways of overriding the default values:

- The first method is to via the four functions listed in the table below.
- The second method is for your application to define the configuration structure over the one declared by Micrium OS (as an *extern* variable).

Note that in either case, these configurations must be set *before* calling the Net_Init() function.

Method 1 - Specific Function Calls

Memory Segments

Specifies a pointer to a memory segment to use when allocating control data for the network core.

Configuration Type	Function to Call	Default	Field From Default Configuration Structure
MEM_SEG*	Net_ConfigureMemSeg()	General-purpose heap .	.MemSegPtr

Network Core Task's Stack Size

Specifies the size (in CPU_STK elements) of the stack for the network core task. The core stack is used for the main TCP/IP core operations: receive packets, demux packets up to the socket layer, network timer handling, free network buffers, etc.

Configuration Type	Function to Call	Default	Field From Default Configuration Structure
CPU_STK_SIZECPU_STK*	Net_ConfigureCoreTaskStk()	A stack of 512 elements allocated on network core's memory segment.	.CoreStkSizeElements.CoreStkPtr

Network Core Services Task's Stack Size

Specifies the size (in CPU_STK elements) of the stack for the network core services task. The core services task is used for the DHCP client and DNS client module.

Configuration Type	Function to Call	Default	Field From Default Configuration Structure
CPU_STK_SIZECPU_STK*	Net_ConfigureCoreSvcTaskStk()	A stack of 512 elements allocated on network core's memory segment.	.CoreSvcStkSizeElements.CoreSvcStkPtr

DNS Configuration

Specifies the configuration structure for the DNS client core module. The fields in this structure are as follows:

Field	Description	Possible Values
.HostNameLenMax	Maximum host name length that can be requested.	Any 16-bit unsigned integer, 255 u by default.
.CacheEntriesMaxNbr	Number of entries the cache can contain.	Any unsigned integer, LIB_MEM_BLK_QTY_UNLIMITED by default.
.AddrIPv4MaxPerHost	The maximum number of IPv4 addresses that can be stored by a host in the cache.Only valid if IPv4 is enabled.	Any unsigned integer, LIB_MEM_BLK_QTY_UNLIMITED by default.
.AddrIPv6MaxPerHost	The maximum number of IPv6 addresses that can be stored by a host in the cache.Only valid if IPv6 is enabled.	Any unsigned integer, LIB_MEM_BLK_QTY_UNLIMITED by default.
.OpDly_ms	Configures the operations delay in integer milliseconds.	Any 8-bit unsigned integer, 50 u by default.
.ReqRetryNbrMax	Configures maximum of request resolution retry.	Any 8-bit unsigned integer, 3 u by default.
.ReqRetryTimeout_ms	Configures timeout before a request resolution retry.	Any 16-bit unsigned integer, 1000 u by default.
.DfltServerAddrFallbackEn	Enables the DNS process to fallback to the default DNS server address if no address is available. By default, the fallback DNS server address is **8.8.8.8**	DEF_ENABLED or DEF_DISABLED

Configuration Type	Function to Call	Default	Field From Default Configuration Structure
DNSc_CFG	Net_ConfigureDNS_Client()	See structure's description.	.DNSc_Cfg

Method 2 - Externalized Structure

If you prefer to use the advanced configuration method, you must set the flag RTOS_CFG_EXTERNALIZE_OPTIONAL_CFG to DEF_ENABLED. In addition, the Network module will assume that your application will define a NET_INIT_CFG structure named Net_InitCfg. Optionally, you may assign Net_InitCfg the value of NET_INIT_CFG_DFLT to begin, and then modify some fields (see the table from Method 1) to suit your application's needs. See Initializing the Network Core Module for more information.

Post-Init Configurations

This section describes the configurations that can be set at any time during execution *after* you call the function Net_Init().

These configurations are optional. If you do not set them in your application, the default configurations will apply.

Configurations	Description	Type	Function to call	Default
Network core task priority	Net_Init() creates a task for handling the TCP/IP core. You can change the priority of this task at any time.	RTOS_TASK_PRIO	Net_CoreTaskPrioSet()	See Appendix A - Internal Tasks .
Network core services task priority	Net_Init() creates a task for handling core services such as the DHCP client and the DNS client. You can change the priority of this task at any time.	RTOS_TASK_PRIO	Net_CoreSvcTaskPrioSet()	See Appendix A - Internal Tasks .

Network Interface Controller Configuration

The approach to configuring your network interface controller is a bit different depending on the interface type. The sections below discuss the different strategies for the two supported network interfaces: Ethernet and Wi-Fi.

- Ethernet Controller
- Wi-Fi Controller

Ethernet Controller

For an Ethernet interface controller, all the application and hardware parameters are grouped into one global configuration structure named NET_DEV_CFG_ETHER. Each Ethernet interface present on your board must have its own NET_DEV_CFG_ETHER object. This object must also be defined in the BSP files of your project because it will be passed as an argument when registering the network interface with the BSP_OS_Init() function. Refer to the section Network Core Hardware Porting Guide for more details.

The NET_DEV_CFG_ETHER object is passed to the network stack during the registration process inside the BSP. So, when your application adds an Ethernet interface (using the NetIF_Ether_Add() function), there is no need to pass arguments describing the interface configuration.

Table - NET_DEV_CFG_ETHER Configuration Structure in the *Network Interface Controller Configuration* page describes each field available in this configuration structure.

Table - NET_DEV_CFG_ETHER Configuration Structure

Field	Description
.RxBufPoolType	Specifies the memory location for the receive data buffers. Buffers may be located either in main memory or in a dedicated memory region. This setting is used by the Interface layer to initialize the Rx memory pool. This field must be set to one of two macros: NET_IF_MEM_TYPE_MAIN or NET_IF_MEM_TYPE_DEDICATED. You may want to set this field when using DMA with dedicated memory. It is possible that you might have to store descriptors within the dedicated memory if your device requires it.

Field	Description
.RxBufLargeSize	Specifies the size of all receive buffers. Specifying a value is required. The default buffer length is set to 1518 bytes, which corresponds to the Maximum Transmission Unit (MTU) size for an Ethernet network. For DMA-based Ethernet controllers, you must set the receive data buffer size to be greater or equal to the size of the largest receivable frame. If the size of the total buffer allocation is greater than the amount of available memory in the chosen memory region, a run-time error will be generated when the device is initialized.
.RxBufLargeNbr	Specifies the number of receive buffers that will be allocated to the device. There should be at least one receive buffer allocated, and it is recommended to have at least ten receive buffers. The optimal number of receive buffers depends on your application.
.RxBufAlignOctets	Specifies the required alignment of the receive buffers, in bytes. Some devices require that the receive buffers be aligned to a specific byte boundary. Additionally, some processor architectures do not allow multi-byte reads and writes across word boundaries and therefore may require buffer alignment. In general, it is probably a best practice to align buffers to the data bus width of the processor, which may improve performance. For example, a 32-bit processor may benefit from having buffers aligned on a four-byte boundary.
.RxBufIxOffset	Specifies the receive buffer offset in bytes. Most devices receive packets starting at base index zero in the network buffer data areas. However, some devices may buffer additional bytes prior to the actual received Ethernet packet. This setting configures an offset to ignore these additional bytes. If a device does not buffer any additional bytes ahead of the received Ethernet packet, then an offset of 0 must be specified. However, if a device does buffer additional bytes ahead of the received Ethernet packet, then you should configure this offset with the number of additional bytes.
.TxBufPoolType	Specifies the memory placement of the transmit data buffers. Buffers may be placed either in main memory or in a dedicated memory region. This field is used by the Interface layer, and it should be set to one of two macros: NET_IF_MEM_TYPE_MAIN or NET_IF_MEM_TYPE_DEDICATED. When DMA descriptors are used, they may be stored in dedicated memory.
.TxBufLargeSize	Specifies the size of the large transmit buffers in bytes. This field has no effect if the number of large transmit buffers is configured to zero. Setting the size of the large transmit buffers below 1594 bytes may hinder the TCP/IP module's ability to transmit full sized IP datagrams since IP transmit fragmentation is not yet supported. We recommend setting this field between 1594 and 1614 bytes in order to accommodate all of the maximum transmit packet sizes for TCP-IP's protocols.You can also optimize the transmit buffer if you know in advance what the maximum size of the packets the user will want to transmit through the device are.
.TxBufLargeNbr	Specifies the number of large transmit buffers allocated to the device. You may set this field to zero to make room for additional small transmit buffers, however, the size of the maximum transmittable packet will then depend on the size of the small transmit buffers.
.TxBufSmallSize	Specifies the small transmit buffer size. For devices with a minimal amount of RAM, it is possible to allocate small transmit buffers as well as large transmit buffers. In general, we recommend a small transmit buffer size of 64 bytes, however, you may adjust this value according to the application requirements. This field has no effect if the number of small transmit buffers is configured to zero.
.TxBufSmallNbr	Specifies the numbers of small transmit buffers. This field controls the number of small transmit buffers allocated to the device. You may set this field to zero to make room for additional large transmit buffers if required.
.TxBufAlignOctets	Specifies the transmit buffer alignment in bytes. Some devices require that the transmit buffers be aligned to a specific byte boundary. Additionally, some processor architectures do not allow multi-byte reads and writes across word boundaries, and therefore may require buffer alignment. In general, it's probably a best practice to align buffers to the data bus width of the processor which may improve performance. For example, a 32-bit processor may benefit from having buffers aligned on a four-byte boundary.

Field	Description
.TxBufIxOffset	Specifies the transmit buffer offset in bytes. Most devices only need to transmit the actual Ethernet packets as prepared by the higher network layers. However, some devices may need to transmit additional bytes prior to the actual Ethernet packet. This setting configures an offset to prepare space for these additional bytes. If a device does not transmit any additional bytes ahead of the Ethernet packet, the default offset of zero should be configured. However, if a device does transmit additional bytes ahead of the Ethernet packet then configure this offset with the number of additional bytes.
.MemAddr	Specifies the starting address of the dedicated memory region for devices with this memory type. For devices with non-dedicated memory, you can initialize this field to zero. You may use this setting to put DMA descriptors into the dedicated memory.
.MemSize	Specifies the size of the dedicated memory region in bytes for devices with this memory type. For devices with non-dedicated memory, you can initialize this field to zero. You may use this setting to put DMA descriptors into the dedicated memory.
.Flags	Specify the optional configuration flags. Configure (optional) device features by logically OR'ing bit-field flags:NET_DEV_CFG_FLAG_NONE — No device configuration flags selected.NET_DEV_CFG_FLAG_SWAP_OCTETS — Swap data bytes (i.e., swap data words' high-order bytes with data words' low-order bytes, and vice-versa) if required by device-to-CPU data bus wiring and/or CPU endian word order.
.RxDescNbr	Specifies the number of receive descriptors. For DMA-based devices, this value is used by the device driver during initialization in order to allocate a fixed-size pool of receive descriptors to be used by the device. The number of descriptors must be less than the number of configured receive buffers. We recommend setting this value to something within 40% and 70% of the number of receive buffers. Non-DMA based devices may configure this value to zero. You must use this setting with DMA based devices and at least two descriptors must be set.
.TxDescNbr	Specifies the number of transmit descriptors. For DMA based devices, this value is used by the device driver during initialization to allocate a fixed size pool of transmit descriptors to be used by the device. For best performance, it's recommended to set the number of transmit descriptors equal to the number of small, plus the number of large transmit buffers configured for the device. Non-DMA based devices may configure this value to zero. You must use this setting with DMA based devices and set at least two descriptors.
.BaseAddr	Specifies the base address of the device's hardware/registers.
.DataBusSizeNbrBits	Specifies the size of device's data bus (in bits), if available.
.HW_AddrStr	Specifies the desired device hardware address; may be NULL address or string if the device hardware address is configured or set at run-time.Depending on the driver, if this value is kept NULL or invalid, most of the device driver will automatically try to load and use the hardware address located in the memory of the device.
.CfgExtPtr	Specifies the pointer to an extension configuration structure if needed, else set to DEF_NULL.

There is an additional configuration structure for PHY controllers: NET_PHY_CFG_ETHER. You might need to adjust some of the configuration fields in this structure depending on your specific needs (for example, the link speed). See PHY Configuration and Information for more details.

Wi-Fi Controller

In contrast to the Ethernet interface, the Wi-Fi interface separates the application-specific configuration parameters from the hardware parameters for the controller. So, when adding a Wi-Fi interface with NetIF_WiFi_Add(), you must pass a NET_IF_BUF_CFG argument. The NET_IF_BUF_CFG structure contains all the fields related to the network buffer configuration of a Wi-Fi interface.

Table - NET_IF_BUF_CFG Configuration Structure

Filed	Description
.RxBufLargeNbr	Specifies the number of receive buffers that will be allocated to the device. There should be at least one receive buffer allocated, and we recommend at least ten receive buffers. The optimal number of receive buffers depends on your application.
.TxBufLargeNbr	Specifies the number of large transmit buffers allocated to the device. You may set this field to zero to make room for additional small transmit buffers; however, the size of the maximum transmittable packet will then depend on the size of the small transmit buffers.
.TxBufSmallNbr	Specifies the numbers of small transmit buffers. This field controls the number of small transmit buffers allocated to the device. You may set this field to zero to make room for additional large transmit buffers if required.
.BufAlignOctets	Specifies the network buffer alignment in bytes. Some devices require that the buffers be aligned to a specific byte boundary. Additionally, some processor architectures do not allow multi-byte reads and writes across word boundaries and therefore may require buffer alignment. In general, it's probably a best practice to align buffers to the data bus width of the processor, which may improve performance. For example, a 32-bit processor may benefit from having buffers aligned on a four-byte boundary.
.BufPoolType	Specifies the memory location for the network buffers. Buffers may be located either in main memory or in a dedicated memory region. This field must be set to one of two macros: NET_IF_MEM_TYPE_MAIN or NET_IF_MEM_TYPE_DEDICATED.
.MemAddr	Specifies the starting address of the dedicated memory region for devices with this memory type. For devices with non-dedicated memory, you can initialize this field to zero.
.MemSize	Specifies the size of the dedicated memory region in bytes for devices with this memory type. For devices with non-dedicated memory, you can initialize this field to zero.

Network Interface Start Setup

To start a network interface, you need to make only one API call. This is done by providing a configuration structure as an argument to the NetIF_Ether_Start() or NetIF_WiFi_Start() functions. Each interface type has its own configuration structure since some parameters are unique. These structures include parameters for the local IP address and DHCP setup, among others.

Using this configuration structure is optional; both NetIF_Ether_Start() and NetIF_WiFi_Start() can be called with a null pointer as an argument instead. If you start the network interface without a configuration, the interface will not be active on the network until you provide the necessary configuration information using additional API calls. This would include, for example, setting the local IP address, network mask, and gateway. See the IP Address Programming section of the Network Core Programming Guide for more details.

The following sections describe the parameters for the Network Interface Start configuration structure.

Ethernet Interface

Table - NET_IF_ETHER_CFG Setup Structure in the *Network Interface Start Setup* page shows the setup structure for the Ethernet interface. To start and configure the Ethernet interface, call NetIF_Ether_Start() using this structure as the second argument in the function.

Table - NET_IF_ETHER_CFG Setup Structure

<table>
<tr><th>Field</th><th colspan="2">Description</th></tr>
<tr><td>.HW_AddrStr</td><td colspan="2">Species the hardware address (MAC) for the network interface. If null, the device driver will try to use the hardware address located in the memory of the device.</td></tr>
<tr><td rowspan="4">.IPv4</td><td colspan="2">Specifies all the IPv4 setup parameters.</td></tr>
<tr><td>.Static</td><td>Specifies all the parameters for a static IPv4 address.
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>.Addr</td><td>String withdesiredIPv4 address. If null, no static IPv4 address will be configured.</td></tr>
<tr><td>.Mask</td><td>String with the IPv4 mask of your network.</td></tr>
<tr><td>.Gateway</td><td>String with the IPv4 address of the network gateway to use. Set to Null, if no gateway on the network.</td></tr>
</table></td></tr>
<tr><td>.DHCPc</td><td>Specifies all the parameters for the DHCP client.
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>.En</td><td>Enables the DHCP client on the network interface.</td></tr>
<tr><td>.Cfg</td><td>Specifies a configuration structure for the DHCP client on the network interface. Use the macro DHCPc_CFG_DFLT to set the fields to the default values.
<table>
<tr><th>Field</th><th>Description</th><th>Default Value</th></tr>
<tr><td>.ServerPortNbr</td><td>UDP port number for the DHCP server. The network stack will send DHCP messages to destinations with this port number.</td><td>67</td></tr>
<tr><td>.ClientPortNbr</td><td>UDP port number for the DHCP client. The network stack will open a UDP socket on this port to receive incoming DHCP messages.</td><td>68</td></tr>
<tr><td>.TxRetryNbr</td><td>Configures the maximum number of transmission retries.</td><td>2</td></tr>
<tr><td>.TxTimeout_ms</td><td>Configures the value for the socket timeout on each transmission attempt, in milliseconds.</td><td>3000</td></tr>
<tr><td>.ValidateAddr</td><td>Enables the address validation. Once the DHCP server has assigned the client an address, the later may perform a final check prior to use this address in order to make sure it is not being used by another host on the network.</td><td>DEF_NO</td></tr>
</table></td></tr>
<tr><td>.OnCompleteHook</td><td>Specifies a pointer to callback function that notifies the application when the DHCP process is done.</td></tr>
</table></td></tr>
<tr><td>.LinkLocal</td><td>Specifies all the setup parameters for the IPv4 link-local address.
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>.En</td><td>Enables the IPv4 link-local process on the network interface.</td></tr>
<tr><td>.OnCompleteHook</td><td>Specifies a pointer to a callback function that notifies the application when the IPv4 link-local process is done.</td></tr>
</table></td></tr>
<tr><td rowspan="4">.IPv6</td><td colspan="2">Specifies all the IPv6 setup parameters.</td></tr>
<tr><td>.Static</td><td>Specifies all the parameters for a static IPv6 address.
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>.Addr</td><td>String withdesiredIPv6 address. If null, no static IPv6 address will be configured.</td></tr>
<tr><td>.PrefixLen</td><td>Length of the IPv6 address prefix.</td></tr>
<tr><td>.DAD_En</td><td>Enables Duplication Address Detection (DAD) during the configuration of the static IPv6 address.</td></tr>
</table></td></tr>
<tr><td>.AutoCfg</td><td>Specifies the parameters to autoconfigure an IPv6 address.
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>.En</td><td>Enables the SLAAC process on the network interface.</td></tr>
<tr><td>.DAD_En</td><td>Enables the Duplication Address Detection (DAD) for the SLAAC process.</td></tr>
</table></td></tr>
<tr><td>.Hook</td><td>Pointer to a callback function that notifies the application when the IPv6 address configuration is done.</td></tr>
</table>

Wi-Fi Interface

Table - NET_IF_WIFI_CFG Setup Structure in the *Network Interface Start Setup* page shows the setup structure for the Wi-Fi interface. To start and configure the Wi-Fi interface, call NetIF_WiFi_Start() using this structure as the second argument in the function. As you can see, it is almost the same structure as for the Ethernet interface, plus additional Wi-Fi-specific parameters.

Table - NET_IF_WIFI_CFG Setup Structure

Field	Description
.HW_AddrStr	Same as for the Ethernet interface. See Table - NET_IF_ETHER_CFG Setup Structure in the *Network Interface Start Setup* page
.IPv4	Same as for the Ethernet interface. See Table - NET_IF_ETHER_CFG Setup Structure in the *Network Interface Start Setup* page
.IPv6	Same as for the Ethernet interface. See Table - NET_IF_ETHER_CFG Setup Structure in the *Network Interface Start Setup* page
.Band	Configures the wireless band used by the Wi-Fi device: NET_DEV_2_4_GHZ, NET_DEV_5_0_GHZ, NET_DEV_DUAL

Network Core Start Up

Network Core Start-Up

This page provides a summary of the steps necessary to initialize the network core, and to start a network interface. Theses steps are discussed in more detail on pages in this section.

Steps 1 and 3 each provide two different options. One method uses **predefined configurations** and gets the interface started easily; the other offers more **flexibility** but requires additional API calls.

Initialize the Network Core Module

There are two ways you can initialize the network core module :

- Call Net_Init() directly without modifying the default configuration.
- Modify the configuration before calling Net_Init().

Add a Network Interface

To add a network interface , you must specify a string identifier such as "eth0" or "wifi0" using the appropriate Add Network function. This identifier must be registered with the Platform Manager using BSP_OS_Init().

- For Ethernet devices: The device configuration structure needed for adding the interface is defined in the device's BSP file (e.g., bsp_net_ether_xx.c).
- For Wi-Fi devices: The device configuration structure needed for adding the interface is pre-configured in the device driver. The only configuration you need to pass to the NetIF_WiFi_Add() function is the NET_IF_BUF_CFG buffer configuration structure.

Start the Network Interface

The network interface can be started in one of two ways:

- Start the network interface with a configuration. To do so, call NetIF_xxx_Start() with a correctly-populated NET_IF_XXX_CFG structure in its second parameter. This starts the interface with a properly-configured local IP address, and/or with a DHCP Client process (the IP address will, therefore, be assigned shortly after interface start), and optionally the Link-local address assignment feature.
- Start the network interface *without* a configuration. To do so, call NetIF_xxx_Start() with DEF_NULL in its second parameter. Doing so will start the network interface immediately, but you will have to set up the interface manually (the local IP address, DHCP Client, IPv6 Auto Config, and Link-local address enabling) by calling the appropriate API functions.

The following sub-sections cover each of these steps in more detail.

Initializing the Network Core Module

This section describes the basic steps required to initialize the Network core module.

- Initializing the Network Core Module Using the Default Configuration
- Initializing the Network Core Module Using Optional Configuration
  - Override Using a Specific Function Call
  - Override Using Advanced Configuration

To initialize the Network core module, you call the function Net_Init(). This function initializes and allocates memory for the different layers of the TCP/IP core, as well as for the **DHCP client** module and the **DNS client** module, which are also part of the Network Core.

The initialization can be performed in one of two ways:

- Using the default configuration
- Using optional configuration

Initializing the Network Core Module Using the Default Configuration

This initialization method uses the configuration values from the structure Net_InitCfgDflt. Using this method will be appropriate unless you have a specific need to override any of the default configuration values.

Listing - Example of call to Net_Init()

```
RTOS_ERR  err;

Net_Init(&err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

Initializing the Network Core Module Using Optional Configuration

This initialization method overrides the default configuration values. There are two options:

Option 1 - Override Using a Specific Function Call

Suppose you needed the network core task to have a different size than the default value. In this case, you would use the Net_ConfigureCoreTaskStk() function to alter the configuration prior to calling Net_Init().

Here is an example of how this is done.

Listing - Example of call to Net_Init()

```
CPU_STK_SIZE   stk_size    = Net_InitCfgDflt.CoreStkSizeElements;
void          *p_stk_base  = Net_InitCfgDflt.CoreStkPtr;
RTOS_ERR       err;

stk_size = 1024u;                          /* Modify only what is needed          */
Net_ConfigureCoreTaskStk(stk_size, p_stk_base);  /* Set the new Core Task configuration values */

Net_Init(&err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

Option 2 - Override Using Advanced Configuration

To use this approach, the advanced configuration flag must be enabled in the OS.

Listing - rtos_cfg.h

```
#define RTOS_CFG_EXTERNALIZE_OPTIONAL_CFG DEF_ENABLED
```

When this is enabled, the structure Net_InitCfg must be defined somewhere in the application. The example below shows Net_InitCfg defined in full using the default values.

Listing - Example of advanced configuration

```
RTOS_ERR err;

const NET_INIT_CFG Net_InitCfg = {
      .DNSc_Cfg = {
            .HostNameLenMax      = 255u,
            .CacheEntriesMaxNbr = 6u,
            .AddrIPv4MaxPerHost = 3u,
            .AddrIPv6MaxPerHost = 3u,
            .TaskDly_ms          = 50u,
            .ReqRetryNbrMax      = 2u,
            .ReqRetryTimeout_ms = 1000u
      },
      .CoreStkSizeElements = 512u,
      .CoreStkPtr          = DEF_NULL,
      .SrvStkSizeElements  = 512u,
      .SrvStkPtr           = DEF_NULL,
      .MemSegPtr           = DEF_NULL
};

Net_Init(&err);
if (err.Code != RTOS_ERR_NONE) {
   /* An error occurred. Error handling should be added here. */
}
```

Adding a Network Interface

The Network module offers two types of network interfaces: Ethernet and Wi-Fi. Each interface type has its own *Add* function.

Once the network core module has been initialized successfully, you can start adding your network interface(s) with the NetIF_Ether_Add() function (for Ethernet interfaces) and the NetIF_WiFi_Add() function (for Wi-Fi interfaces). You must add each network interface you want to use in your application using one of those two functions.

The *Add* functions require an ID string argument: for example, "eth0". This ID string is the same string that was used in the BSP_OS_Init() function earlier when the network interface was registered. See section Network Core Hardware Porting Guide for more details.

Ethernet Interface

Listing - Example of call to NetIF_Ether_Add() in the *Adding a Network Interface* page shows an example of call to NetIF_Ether_Add() using the default arguments.

Listing - Example of call to NetIF_Ether_Add()

```
NET_IF_NBR      if_nbr;
RTOS_ERR        err;

#ifdef  NET_IF_ETHER_MODULE_EN
                                        /* -------------- ADD ETHERNET INTERFACE -------------- */
if_nbr = NetIF_Ether_Add("eth0", DEF_NULL, DEF_NULL, &err);
if (err.Code != RTOS_ERR_NONE) {
   /* An error occured. Error handling should be added here. */
}
#endif
```

The structure NET_DEV_CFG_ETHER includes configuration parameters such as the number of network buffers and memory locations. You can modify these parameters as you see fit, according to your application's needs. This structure is defined in the BSP of the network controller and is passed when the network interface is registered via the BSP_OS_Init() function. For more details on the NET_DEV_CFG_ETHER structure, refer to Network Interface Controller Configuration , and for more details on the registration process of the network interface(s), refer to Network Controller Registration to the Platform Manager .

Wi-Fi Interface

Listing - Example of call to NetIF_WiFi_Add() in the *Adding a Network Interface* page shows an example of a call to NetIF_WiFi_Add() using the default arguments.

In contrast to the Ethernet interface, the Wi-Fi interface has an additional configuration argument of type NET_IF_BUF_CFG. This configuration structure contains network buffer parameters. For more details on configuring arguments to pass to NetIF_WiFi_Add(), see Network Interface Controller Configuration .

Listing - Example of call to NetIF_WiFi_Add()

```
static  const  NET_IF_BUF_CFG Ex_NetWiFi_IF_BufCfg = {
       .RxBufLargeNbr  = 10u,
       .TxBufLargeNbr  = 8u,
       .TxBufSmallNbr  = 4u,
       .BufAlignOctets = LIB_MEM_BUF_ALIGN_AUTO,
       .BufPoolType    = NET_IF_MEM_TYPE_MAIN,
       .MemAddr        = 0x0,
       .MemSize        = 0x0
};

NET_IF_NBR     if_nbr;
RTOS_ERR       err;

#ifdef  NET_IF_WIFI_MODULE_EN
                                            /* -------------- ADD WIRELESS INTERFACE -------------- */
if_nbr = NetIF_WiFi_Add("wifi0",
                  &Ex_NetWiFi_IF_BufCfg,
                   DEF_NULL,
                   DEF_NULL,
                  &err);
if (err.Code != RTOS_ERR_NONE) {
   /* An error occurred. Error handling should be added here. */
}
#endif
```

Lookback Interface

Also, a virtual Loopback network interface can be added automatically if the #define configuration NET_IF_CFG_LOOPBACK_EN is enabled (see Network Core Compile-Time Configurations).

Starting a Network Interface

- Starting the Network Interface
  - Ethernet Interface
  - Wi-Fi Interface
  - Network Interface Ready
- Starting the Network Interface Without a Configuration

The last step in making your network interface active on the network is to start it. Similar to the *Add* function, each type of network interface has its own *Start* function.

Each *Start* function has an optional argument for a setup structure that includes parameters for IPv4, IPv6, DHCP and other interface-specific parameters. This allows your application to define and populate a setup structure with parameters that match your network's characteristics and your application's needs. Including the argument causes the Network Core Stack to perform the minimal setup required for the interface to be visible on the network. For more details on the network interface setup structure, see Network Interface Start Setup .

The sections below show how to start a network interface according to its type; either Ethernet of Wi-Fi.

Starting the Network Interface

Ethernet Interface

Listing - Example of call to NetIF_Ether_Start() in the *Starting a Network Interface* page shows an example of call to NetIF_Ether_Start().

For more information on configuring arguments to pass to NetIF_Ether_Start(), see Network Interface Start Setup .

For a complete example on how to start an Ethernet interface, see the example files.

Listing - Example of call to NetIF_Ether_Start()

```
static const NET_IF_ETHER_CFG Ex_NetIF_CfgDflt = {
    .HW_AddrStr                  = "00:17:4A:B0:00:01",
    .IPv4.Static.Addr            = DEF_NULL,
    .IPv4.Static.Mask            = DEF_NULL,
    .IPv4.Static.Gateway         = DEF_NULL,
    .IPv4.DHCPc.En               = DEF_YES,
    .IPv4.DHCPc.Cfg              = DHCPc_CFG_DFLT,
    .IPv4.DHCPc.OnCompleteHook   = Ex_DHCPc_SetupResult,
    .IPv4.LinkLocal.En           = DEF_NO,
    .IPv4.LinkLocal.OnCompleteHook = DEF_NULL,
    .IPv6.Static.Addr            = DEF_NULL,
    .IPv6.Static.PrefixLen       = 0,
    .IPv6.Static.DAD_En          = DEF_NO,
    .IPv6.AutoCfg.En             = DEF_YES,
    .IPv6.AutoCfg.DAD_En         = DEF_YES,
    .IPv6.Hook                   = Ex_IPv6_AddrCfgResult
};

NET_IF_ETHER_CFG  if_cfg;
NET_IF_NBR        if_nbr;
RTOS_ERR          err;

if_cfg = Ex_NetIF_CfgDflt;

                                        /* ------------- START ETHERNET INTERFACE ------------- */
NetIF_Ether_Start(if_nbr, &if_cfg, &err);
if (err.Code != RTOS_ERR_NONE) {
   /* An error occured. Error handling should be added here. */
}
```

Wi-Fi Interface

Listing - Example of call to NetIF_WiFi_Start() in the *Starting a Network Interface* page shows an example of call to NetIF_WiFi_Start().

For more information on configuring arguments to pass to NetIF_WiFi_Start(), see Network Interface Start Setup .

For a complete example on how to start a Wi-Fi Interface, see the example files.

Listing - Example of call to NetIF_WiFi_Start()

```
static const NET_IF_WiFi_CFG Ex_NetIF_CfgDflt = {
    .HW_AddrStr              = "00:17:4A:B0:00:01",
    .IPv4.Static.Addr        = DEF_NULL,
    .IPv4.Static.Mask        = DEF_NULL,
    .IPv4.Static.Gateway     = DEF_NULL,
    .IPv4.DHCPc.En           = DEF_YES,
    .IPv4.DHCPc.Cfg          = DHCPc_CFG_DFLT,
    .IPv4.DHCPc.OnCompleteHook    = Ex_DHCPc_SetupResult,
    .IPv4.LinkLocal.En       = DEF_NO,
    .IPv4.LinkLocal.OnCompleteHook = DEF_NULL,
    .IPv6.Static.Addr        = DEF_NULL,
    .IPv6.Static.PrefixLen   = 0,
    .IPv6.Static.DAD_En      = DEF_NO,
    .IPv6.AutoCfg.En         = DEF_YES,
    .IPv6.AutoCfg.DAD_En     = DEF_YES,
    .IPv6.Hook               = Ex_IPv6_AddrCfgResult,
    .Band                    = NET_DEV_BAND_2_4_GHZ
};

NET_IF_WIFI_CFG  if_cfg;
NET_IF_NBR       if_nbr;
RTOS_ERR         err;

if_cfg = Ex_NetIF_CfgDflt;

                                   /* ------------- START WIRELESS INTERFACE ------------- */
NetIF_WiFi_Start(if_nbr, &if_cfg, &err);
if (err.Code != RTOS_ERR_NONE) {
   /* An error occurred. Error handling should be added here. */
}
```

Network Interface Ready

Because the network interface start function doesn't block, other mechanisms exist to let your application know when the setup of the interface is complete. You can use the different callback functions (DHCP, IPv6, IPv4. Link-Local) inside the NET_IF_xxx_CFG structure you are passing to the interface start function. You can use those callbacks to notify your application when a specific setup process has finished. They will also give you information on the process that just completed: for example, the status of the process, the IP address that was just configured, etc.

Micrium OS also includes a module to that allows your application to wait for the network interface setup to be complete. To enable this module, you must enable the NET_IF_CFG_WAIT_SETUP_READY_EN configuration in your net_cfg.h file (see section Network Core Compile-Time Configurations). So once the network interface is started, your application can begin waiting for the setup to be complete by using the API function NetIF_WaitSetupReady(). This function will wait for the various processes that you enabled in the NET_IF_xxx_CFG argument (DHCP, IPv6, etc.), and it will exit when every process has finished, if no error occurred. You can also pass a reference to a NET_IF_APP_INFO type variable. The function will populate structure with the information collected during the setup.

Starting the Network Interface Without a Configuration

Alternatively, the network interface can be started immediately without a configuration.

Listing - Starting the interface without a configuration

```
NET_IF_NBR       if_nbr;
RTOS_ERR         err;

                                   /* ------------- START ETHERNET INTERFACE ------------- */
NetIF_Ether_Start(if_nbr, DEF_NULL, &err);
if (err.Code != RTOS_ERR_NONE) {
   /* An error occurred. Error handling should be added here. */
}
```

Note that in this case, the interface setup must be done by calling the appropriate API functions after the interface has been started:

Function	Description	Section in the Network Core Programming Guide
NetIPv4_CfgAddrAdd()	Set a static IPv4 address on the network interface.	IPv4 - Assign an Address to a Network Interface
DHCPc_IF_Add()	Enable the DHCP client (IPv4) on the network interface.	DHCP Client Programming
NetIPv4_AddrLinkLocalCfg()	Enable the IPv4 Link-Local address setup on the network interface.	IPv4 - Link-Local Address Setup
NetIPv6_AddrAutoCfgEn()	Enable the IPv6 SLAAC setup on the network interface.	IPv6 - Stateless Address Auto configuration (SLAAC)
NetIPv6_CfgAddrAdd()	Set a static IPv6 address on the network interface.	IPv6 - Assign an Address to a Network Interface

Network Core Programming Guide

Network Core Programming Guide

Once you have the proper configuration in place and your network interfaces are up , the next step is to start programming. This is where you delve into the programming of your application.

To help you get started, this guide covers:

- Socket Programming
- Multicast Programming
- Wireless Programming

It also touches on the topic of lower-level programming functions, including:

- DNS Client Programming
- DHCP Client Programming
- IP Address Programming
- Interface Programming

> High-level application programmingDon't forget to take a look at our Application Modules for more information about programming with high-level applications such as the HTTP Server.

DNS Client Programming

This page shows how to use the DNS Client of the Network module. The DNS client module is part of the network core module of Micrium OS.

- DNS Client Configuration
  - Compile-Time Configuration
  - Configuration Structure
- DNS Server Address Setup
- Domain Name Translation

DNS Client Configuration

Compile-Time Configuration

To use the DNS Client module, it must be enabled at compile-time in the Network Core configuration file (net_cfg.h) through NET_DNS_CLIENT_CFG_MODULE_EN. See section Core Modules Configuration for more details.

Configuration Structure

You must set the DNS Client module configuration before calling Net_Init(). To do so, use the API function Net_ConfigureDNS_Client(). Table - DNS Client Configuration Structure in the *DNS Client Programming* page shows the configuration fields. See DNS Client Run-Time Configurations for further details.

Below is an example of how the DNS client configuration can be overridden prior to network initialization.

Listing - Example of call to Net_Init()

```
DNSc_CFG  dns_cfg;
RTOS_ERR  err;

dns_cfg = Net_InitCfgDflt.DNSc_Cfg;          /* Get the default DNS configuration values */
```

```
dns_cfg = Net_InitCfgDflt.DNSc_Cfg;/* Get the default DNS configuration values */
dns_cfg.DfltServerAddrFallbackEn = DEF_DISABLED;/* Modify only what is needed           */Net_ConfigureDNS_Client(&dns_cfg);/* Set the new DNS configuration value      */Net_Init(&err);if(err.Code != RTOS_ERR_NONE){/* An error occurred. Error handling should be added here. */}
```

DNS Server Address Setup

The DNS server address can be configured automatically by using the DNS server option information received during the DHCP (IPv4) client process or during the IPv6 SLAAC.

The DNS server address can also be configured manually by using one of the two available API functions: DNSc_CfgServerByStr() or DNSc_CfgServerByAddr(). A manually-configured DNS server address will always have precedence over an auto-configured DNS server address.

Furthermore, the DNS client module offers the possibility to fall back on a default DNS server address (8.8.8.8) if no other DNS server address is available. This last option can be disabled in the DNS client run-time configuration structure.

Domain Name Translation

To retrieve the IP address of a hostname, you can use tThe API function DNSc_GetHost().

The network module also offers a high-level API functions to open TCP or UDP client sockets by passing the destination hostname instead of the IP address: NetApp_ClientStreamOpenByHostname() and NetApp_ClientDatagramOpenByHostname(). Those functions perform the DNS translation.

DHCP Client Programming

This page describes how to use the DHCP Client of the Network module. The DHCP client module is part of the network core module of Micrium OS.

The DHCP Client module is for only IPv4 addressing. Furthermore, the DHCP Client process operates independently of the IPv4 Static Addressing on a network interface. Therefore the two can be present on a network interface at the same time.

Each network interface on which you want to use DHCP will have its own DHCP process.

- DHCP Client Configuration
  - Compile-Time Configuration
  - Configuration Structure
  - Callback Function
- Start DHCP Client Process
- Reboot DHCP Client Process
- Stop DHCP Client Process

DHCP Client Configuration

Compile-Time Configuration

First, to use the DHCP Client module, it must be enabled at compile-time in the net_cfg.h configuration file. See section Core Modules Configuration for more details.

Configuration Structure

Because each network interface can have its own DHCP process, each interface has its own configuration structure. Table - DHCPc_CFG Configuration Structure in the *DHCP Client Programming* page shows the configuration fields.

Your application will need to declare and define a variable of this structure type, and pass it to the DHCPc_IF_Add() function. This function call must be performed for each network interface you wish to use DHCP on. See DHCP Client Run-Time Configuration for further information about run-time configuration.

Callback Function

An optional callback function can also be passed to the DHCPc_IF_Add() function. This callback must be defined by the application and will be called by the network module upon the end of the DHCP process. Thus, the application can use this callback to receive a notification when the DHCP process is completed, to validate that the process was successful, and to recover the IPv4 address that was configured. Listing - DHCPc_ON_COMPLETE_HOOK Signature in the *DHCP Client Programming* page below shows the callback function prototype and provides details on the different arguments that will be passed to this callback.

Listing - DHCPc_ON_COMPLETE_HOOK Signature

```
void  Ex_DHCPc_Callback (NET_IF_NBR     if_nbr,
                         DHCPc_STATUS   status,
                         NET_IPv4_ADDR  addr,
                         NET_IPv4_ADDR  mask,
                         NET_IPv4_ADDR  gateway,
                         RTOS_ERR       err);
```

Argument	Description
if_nbr	Network interface number on which the DHCP process has complete.
status	Status of the DHCP process:DHCPc_STATUS_SUCCESSDHCPc_STATUS_FAIL_ADDR_USEDDHCPc_STATUS_FAIL_OFFER_DECLINEDHCPc_STA
addr	IPv4 address that has been configured via the DHCP process. NET_IPv4_ADDR_NONE if the process was not a success.
mask	IPv4 address mask of the configured address. NET_IPv4_ADDR_NONE if the process was not a success.
gateway	IPv4 address of the network gateway. NET_IPv4_ADDR_NONE if the process was not a success.
err	Error value returned by the DHCP process if a fault error occurred.

Start DHCP Client Process

There are two approaches to starting a DHCP process on a network interface:

- The DHCP process can be started when calling the NetIF_xx_Start() function. This function will start the network interface and also start the DHCP process automatically when the interface link goes up. This approach is covered in the section Starting a Network Interface ,.
- The process can be started manually with the function DHCPc_IF_Add().

Once a network interface is added by the application, it is then possible to add a DHCP process to it by calling the function DHCPc_IF_Add(). This will ensure that once the network interface goes up after being started , a DHCP process will start to retrieve an IPv4 address from a DHCP server present on the network.

Listing - example of call to DHCPc_IF_Add() in the *DHCP Client Programming* page shows a simple example.

DHCPc_IF_Add() takes as the first argument the network interface number on which to start the DHCP process.

The second argument is a pointer to the configuration structure for the DHCP process. See Table - DHCPc_CFG Configuration Structure in the *DHCP Client Programming* page for more details on the configuration fields. A null pointer can be passed instead and in this case, the DHCP process will use the default configuration values.

The third argument is the callback function to notify the application upon the end of the DHCP process. See Listing - DHCPc_ON_COMPLETE_HOOK Signature in the *DHCP Client Programming* page for more details on the callback function. This parameter can also be a null pointer and in this case, the application will not be notified.

Listing - Example of call to DHCPc_IF_Add()

```
NET_IF_NBR  if_nbr;
DHCPc_CFG   cfg;
RTOS_ERR    err;

cfg = {
  .ServerPortNbr = DHCPc_CFG_PORT_SERVER_DFLT,
```

```
.TxRetryNbr   = DHCPc_CFG_TX_RETRY_NBR_DFLT,.TxTimeout_ms  = DHCPc_CFG_TX_WAIT_TIMEOUT_MS_DFLT,.ValidateAddr  = DEF_NO
};

if_nbr =Net_Ether_Add(DEF_NULL,&err);if(err.Code != RTOS_ERR_NONE){/* An error occurred. Error handling should be added here.
*/}DHCPc_IF_Add(if_nbr,&cfg, Ex_DHCPc_Callback,&err);if(err.Code != RTOS_ERR_NONE){/* An error occurred. Error handling should be added
here. */}
```

Reboot DHCP Client Process

The rebooting of the DHCP Client process occurs automatically under two circumstances:

- The network link state goes from DOWN to UP.
- The lease has expired or is about to expire.

In some cases, however, you may want to force a reboot to renew an address lease earlier. To do this, simply call the DHCPc_IF_Reboot() function, specifying the interface number for which to reboot the DHCP Client process.

Stop DHCP Client Process

The DHCP Client process can be stopped with DHCPc_IF_Remove(), specifying the interface number for which to stop the DHCP Client process.

IP Address Programming

- Include Files
- Configuration
- API Reference
- IP Address Conversion
- Static IP Address Configuration On An Interface
- Dynamic IP Address Configuration On An Interface (IPv4 only))
- Stateless Address Auto-Configuration (IPv6 only)
- IP Addressing with Socket Programming

The following sections provide sample code describing how to manipulate IP address (IPv4 and IPv6).

Include Files

Wherever you want to manipulate IP address, you should include one or many of these files:

Include file	Description
rtos/net/include/net_ipv4.h	Functions used for IPv4 API.
rtos/net/include/net_ipv6.h	Functions used for IPv6 API.
rtos/net/include/net_ascii.h	Functions and Macro used for conversion utilities API.

Configuration

Some parameters should be configured and/or optimized for your project requirements. See the section IPv4 Layer Configuration and IPv6 Layer Configuration under Network Core Compile-Time Configurations for further details.

API Reference

IP Address Configuration Functions

Those functions are related to the configuration and removal of IP address on an Interface.

Function name	Description
IPv4	
NetIPv4_CfgAddrAdd()	Add a statically-configured IPv4 host address, subnet mask, and default gateway to an interface.
NetIPv4_CfgAddrRemove()	Remove a configured IPv4 host address from an interface.
NetIPv4_CfgAddrRemoveAll()	Remove all configured IPv4 host address(es) from an interface.
IPv6	
NetIPv6_AddrAutoCfgEn()	Enable the IPv6 Stateless Address Auto-Configuration procedure.
NetIPv6_AddrAutoCfgDis()	Disable the IPv6 Stateless Address Auto-Configuration procedure.
NetIPv6_AddrSubscribe()	Set the hook function to receive the IPv6 Address Configuration process result.
NetIPv6_CfgAddrAdd()	Add a statically-configured IPv6 host address to an interface.
NetIPv6_CfgAddrRemove()	Remove a configured IPv6 host address & multicast solicited mode address from an interface.
NetIPv6_CfgAddrRemoveAll()	Remove all configured IPv6 host address(es) from an interface.

Interface Configured IP Address Functions

These functions are associated to information on interface configured IP address.

Function Name	Description
IPv4	
NetIPv4_GetAddrHost()	Get an interface's configured IPv4 host address(es).
NetIPv4_GetAddrSrc()	Get corresponding configured IPv4 host address to use as source address for a remote IPv4 address.
NetIPv4_GetAddrDfltGateway()	Get the default gateway IPv4 address for a host's configured IPv4 address.
NetIPv4_GetAddrSubnetMask()	Get the IPv4 address subnet mask for a host's configured IPv4 address.
NetIPv4_IsAddrHost()	Validate an IPv4 address as one the host's IPv4 address(es).
NetIPv4_IsAddrHostCfgd()	Validate an IPv4 address as one the host's configured IPv4 address(es).
NetIPv4_IsAddrsHostCfgdOnIF()	Check if any IPv4 address(es) are configured on an interface.
IPv6	
NetIPv6_GetAddrHost()	Get an interface's configured IPv6 host address(es).
NetIPv6_GetAddrSrc()	Find the best matched source address in the IPv6 configured host addresses for the specified destination address.
NetIPv6_IsAddrHostCfgd()	Validate an IPv6 address as a configured IPv6 host address on an enabled interface.
NetIPv6_IsAddrsCfgdOnIF()	Validate if any IPv6 host addresses are configured on a specific interface.

Generic IP Address Information Functions

Those functions gives you generic information on a specific IP address.

Function Name	Description
	IPv4
NetIPv4_IsAddrBroadcast()	Validate an IPv4 address as the limited broadcast IPv4 address.
NetIPv4_IsAddrClassA()	Validate an IPv4 address as a Class-A IPv4 address.
NetIPv4_IsAddrClassB()	Validate an IPv4 address as a Class-B IPv4 address.
NetIPv4_IsAddrClassC()	Validate an IPv4 address as a Class-C IPv4 address.
NetIPv4_IsAddrLocalHost()	Validate an IPv4 address as a Localhost IPv4 address.
NetIPv4_IsAddrLocalLink()	Validate an IPv4 address as a link-local IPv4 address.
NetIPv4_IsAddrThisHost()	Validate an IPv4 address as the 'This Host' initialization IPv4 address.
NetIPv4_IsValidAddrHost()	Validate an IPv4 address as a valid IPv4 host address.
NetIPv4_IsValidAddrHostCfgd()	Validate an IPv4 address as a valid, configurable IPv4 host address.
NetIPv4_IsValidAddrSubnetMask()	Validate an IPv4 address subnet mask.
	IPv6
NetIPv6_GetAddrMatchingLen()	Compute the number of identical most significant bits of two IPv6 addresses.
NetIPv6_GetAddrScope()	Get the scope of a specific IPv6 address.
NetIPv6_IsValidAddrHost()	Validate an IPv6 host address.
NetIPv6_IsAddrLinkLocal()	Validate an IPv6 address as a link-local IPv6 address.
NetIPv6_IsAddrSiteLocal()	Validate an IPv6 address as a site-local address.
NetIPv6_IsAddrMcast()	Validate an IPv6 address as a multicast address.
NetIPv6_IsAddrMcastSolNode()	Validate an IPv6 address as a solicited node multicast address.
NetIPv6_IsAddrMcastAllNodes()	Validate an IPv6 address as the all nodes multicast address.
NetIPv6_IsAddrMcastAllRouters()	Validate an IPv6 address as the all routers multicast address.
NetIPv6_IsAddrMcastRsvd()	Validate the IPv6 address as a a reserved multicast IPv6 address.
NetIPv6_IsAddrUnspecified()	Validate an IPv6 address as the unspecified IPv6 address.
NetIPv6_IsAddrLoopback()	Validate an IPv6 address as the IPv6 loopback address.

IP Address Conversion Utilities

Those functions allows conversion from a string representation of an IP address to the TCP/IP stack address representation and vice-versa.

Function name	Description
NetASCII_Str_to_IP()	Convert a string of an IPv4 or IPv6 address in their respective decimal notation to an IPv4 or IPv6 address.
NetASCII_Str_to_IPv4()	Convert a string of an IPv4 address in dotted-decimal notation to an IPv4 address in host-order.

Function name	Description
NetASCII_Str_to_IPv6()	Convert a string of an IPv6 address in common-decimal notation to an IPv6 address.
NetASCII_IPv4_to_Str()	Convert an IPv4 address in host-order into an IPv4 dotted-decimal notation ASCII string.
NetASCII_IPv6_to_Str()	Convert an IPv6 address into an IPv6 colon-decimal notation ASCII string.

IP Address Conversion

The Network module contains functions to perform various string operations on IP addresses.

The following example shows how to use the NetASCII module in order to convert IP addresses to and from their dotted-decimal representations:

Listing - IP address string conversion

```
NET_IPv4_ADDR  ipv4_addr;
NET_IPv6_ADDR  ipv6_addr;
CPU_INT08U     ipv4_str[16];
CPU_INT08U     ipv6_str[40];
RTOS_ERR       err;

/* IPv4 */
NetASCII_Str_to_IP("192.168.1.65", &ipv4_addr, NET_IPv4_ADDR_SIZE, &err);
NetASCII_IPv4_to_Str(ipv4_addr, &ipv4_str[0], DEF_NO, &err);

/* IPv6 */
NetASCII_Str_to_IP("fe80::1111:1111", &ipv6_addr, NET_IPv6_ADDR_SIZE, &err);
NetASCII_IPv6_to_Str(&ipv6_addr, &ipv6_str[0], DEF_NO, &err);
```

Static IP Address Configuration On An Interface

See section IPv4 - Assign an Address to a Network Interface and IPv6 - Assign an Address to a Network Interface for examples on how to configure a static IPv4 or IPv6 address on a specific Interface.

Dynamic IP Address Configuration On An Interface (IPv4 only)

IPv4 address can be configured dynamically on an interface by using the DHCP Client module.

If you want to develop your own network application to setup dynamic addressing, the IPv4 functions NetIPv4_CfgAddrAddDynamic(), NetIPv4_CfgAddrAddDynamicStart(), NetIPv4_CfgAddrAddDynamicStop() should be used.

Dynamic address configuration is not yet available for IPv6.

Stateless Address Auto-Configuration (IPv6 only)

The IPv6 protocol defines an Auto-Configuration procedure allowing a network interface to set itself an IPv6 *Link-Local address* based on its Interface ID and also a IPv6 *global address* if an IPv6 router is present on the local network.

Refer to section IPv6 - Stateless Address Auto configuration (SLAAC) for more details.

IP Addressing with Socket Programming

When developing a network application that sends and/or receives data, IP address handling will be required.

When you setup a server application, you MUST bind to an already configured IP address. This address will be used as the destination address for all the clients communicating with the server and will be used as the source address for the packets sent by the server.

You can use the functionNetIPv4_GetAddrHost() or NetIPv6_GetAddrHost() to recover all the IPv4 or IPv6 addresses configured on a specific Interface.

You also have the option to bind to the wildcard address (0.0.0.0 for IPv4 and :: for IPv6). In that case, the server socket is not bound to any IP address in particular, therefore any packets send to the corresponding server port number will be received by the server regardless of the destination address. When sending packets, the Network module TCP/IP stack will take care of setting the best source address for the destination.

When you setup a client application, you MAY want to bind to a specific configured IP address. Knowing the destination IP address, you can use the functionNetIPv4_GetAddrSrc() orNetIPv6_GetAddrSrc() to find the best suited configured IP address for your destination.

If you don't bind to an IP address, the Network module TCP/IP stack will take care of setting the source address of packets to send.

Refer to section Socket Programming for more information on using network sockets.

IPv4 - Assign an Address to a Network Interface

The following sections provide sample code describing how to configure an IPv4 address.

For IPv4, the address configuration may be performed using Micrium OS DHCP Client or manually during run-time.

If the manual configuration is chosen, the following functions may be used to set the IPv4, network mask, and gateway addresses for a specific interface.

NetASCII_Str_to_IP() NetIPv4_CfgAddrAdd()

More than one set of IPv4 addresses may be configured for a specific network interface by calling the functions above. The constant NET_IPv4_CFG_IF_MAX_NBR_ADDR specified in net_cfg.h determines the maximum number of IPv4 addresses that may be assigned to an interface.

Note that on the default interface, the first IPv4 address added will be the default address used for all default IPv4 communication.

The first function aids the developer by converting a string format IPv4 address such as “192.168.1.2” to its hexadecimal equivalent. The second function is used to configure an interface with the specified IPv4, network mask and gateway addresses. An example is shown in listing Address Configuration Example .

Listing - IPv4 Address Configuration Example

```
CPU_BOOLEAN    cfg_success;
NET_IPv4_ADDR  ipv4_addr;
NET_IPv4_ADDR  ipv4_msk;
NET_IPv4_ADDR  ipv4_gateway;
RTOS_ERR       err;

NetASCII_Str_to_IP((CPU_CHAR*)"192.168.1.2",   &ipv4_addr,    NET_IPv4_ADDR_SIZE, &err);     /* See Note #1 */
NetASCII_Str_to_IP((CPU_CHAR*)"255.255.255.0", &ipv4_msk,     NET_IPv4_ADDR_SIZE, &err);
NetASCII_Str_to_IP((CPU_CHAR*)"192.168.1.1",   &ipv4_gateway, NET_IPv4_ADDR_SIZE, &err);

cfg_success = NetIPv4_CfgAddrAdd(if_nbr,                         /* See Note #2 */
                                 ipv4_addr,                      /* See Note #3 */
                                 ipv4_msk,                       /* See Note #4 */
                                 ipv4_gateway,                   /* See Note #5 */
                                &err);                           /* See Note #6 */
```

- NetASCII_Str_to_IP() requires four arguments. The first function argument is a string representing a valid IP address. The second argument is a pointer to the IP address object that will received the conversion result. The third argument is the size of the address object and the last argument is a pointer to a RTOS_ERR to contain the return error code. Upon successful conversion, the return error will contain the value RTOS_ERR_NONE and the function will return a variable of type NET_IP_ADDR_FAMILY containing the family type (IPv4 or IPv6) of the address converted.
- The first argument is the number representing the network interface that is to be configured. This value is obtained as the result of a successful call to NetIF_Ether_Add() or NetIF_WiFi_Add().
- The second argument is the NET_IPv4_ADDR value representing the IPv4 address to be configured.
- The third argument is the NET_IPv4_ADDR value representing the subnet mask address that is to be configured.
- The fourth argument is the NET_IPv4_ADDR value representing the default gateway IPv4 address that is to be configured.
- The fifth argument is a pointer to a RTOS_ERR variable containing the return error code for the function. If the interface address information is configured successfully, then the return error code will contain the value RTOS_ERR_NONE. Additionally, function returns a Boolean value of DEF_OK or DEF_FAIL depending on the result. Either the return value or the

RTOS_ERR variable may be checked for return status; however, the RTOS_ERR contains more detailed information and should therefore be the preferred check.

IPv4 - Link-Local Address Setup

The Link-Local process assigns an IPv4 local address to an interface when no other IPv4 addresses are configured on the interface. This would be the case under the following circumstances:

1. No static address was configured on the interface, **and**
2. No DHCP client process is running on the interface, or a DHCP client process is running but has failed to assign an address.

The Link-Local process is started automatically if configured accordingly when starting the network interface .

Here are some use cases for Link-Local:

- If the network start configuration specifies (a) no static IP address, (b) DHCP enabled and (c) Link-Local enabled, then a Link-Local IP address will automatically be assigned if the DHCP process fails.
- If the network start configuration specifies (a) no static IP address, (b) DHCP disabled and (c) Link-Local enabled, then a Link-Local IP address will automatically be assigned.

If Link-Local is not enabled at the time the interface is started, it can be done manually, as explained in the following section.

Starting the Link-Local Process Manually

When the Link-Local process is not started through the network interface start, it must be launched manually when your application detects the absence of any local IP address on an interface. The Link-Local process is not a task that runs in the background, so it will not be invoked automatically. It is a good practice to check for the presence of an IP address on an interface when (a) the DHCP process fails and (b) a static IP address is removed from the interface. In either case, if the interface is left with no local IP address, the Link-Local process should be launched if required by your application.

To start the Link-Local process, you must call NetIPv4_AddrLinkLocalCfg() and provide the interface number on which to run the process, as well a callback function to be called when the process completes.

Your application's callback function must have the following signature, as defined by NET_IPv4_LINK_LOCAL_COMPLETE_HOOK:

Listing - NET_IPv4_LINK_LOCAL_COMPLETE_HOOK signature

```
static  void  NetIF_IPv4_LinkLocalCompleteHook (NET_IF_NBR                  if_nbr,
                                                NET_IPv4_ADDR               link_local_addr,
                                                NET_IPv4_LINK_LOCAL_STATUS  status,
                                                RTOS_ERR                    err)
```

When the callback function is called, the local IP address assigned to the interface will be available in link_local_addr, provided that the status is NET_IPv4_LINK_LOCAL_STATUS_SUCCEEDED.

Here is an example that shows starting a Link-Local process when a DHCP Client process has failed. It takes place inside the hook function of the DHCP Client process (Ex_DHCPc_Hook(), in this example).

Listing - Example of manually starting the Link Local process

```
static  void  Ex_IPv4_LinkLocalAddrCfgResult (NET_IF_NBR                if_nbr,
                                              NET_IPv4_ADDR             link_local_addr,
                                              NET_IPv4_LINK_LOCAL_STATUS  status,
                                              RTOS_ERR                  err)
{
    CPU_CHAR  addr_str[NET_ASCII_LEN_MAX_ADDR_IPv4];
    RTOS_ERR  local_err;

    switch (status) {
        case NET_IPv4_LINK_LOCAL_STATUS_SUCCEEDED:
             if (link_local_addr != NET_IPv4_ADDR_NONE) {
                 NetASCII_IPv4_to_Str(link_local_addr, addr_str, DEF_YES, &local_err);
             }
             EX_TRACE("IPv4 link local address: %s, was configured successfully!\n", addr_str);
             break;

        case NET_IPv4_LINK_LOCAL_STATUS_FAILED:
      /* Handle Link Local address assignation failure */

        default:
             break;
    }
}

static  void  Ex_DHCPc_Hook (NET_IF_NBR     if_nbr,
                             DHCPc_STATUS   status,
                             NET_IPv4_ADDR  addr,
                             NET_IPv4_ADDR  mask,
                             NET_IPv4_ADDR  gateway,
                             RTOS_ERR       err)
{

    /* Process the DHCP callback */

    if (status != DHCPc_STATUS_SUCCESS) {
        NetIPv4_AddrLinkLocalCfg(if_nbr, Ex_IPv4_LinkLocalAddrCfgResult, &local_err);
    }
}
```

Stopping the Link-Local Process

You can stop the Link-Local process by calling NetIPv4_AddrLinkLocalCfgRemove() and specifying the interface number on which to stop the process.This will also remove the configured Link-Local address on the network interface.

IPv6 - Assign an Address to a Network Interface

The following functions may be utilized to set the IPv6 address for a specific interface:

NetASCII_Str_to_IP() NetIPv6_CfgAddrAdd() NetIPv6_AddrSubscribe()

More than one set of IPv6 addresses may be configured for a specific network interface by calling the functions above. The constant NET_IPv6_CFG_IF_MAX_NBR_ADDR specified in net_cfg.h determines the maximum number of IPv6 addresses that may be assigned to an interface.

Note that on the default interface, the first IPv6 address added will be the default address used for all default IPv6 communication.

The first function aids the developer by converting a string format IPv6 address such as “fe80::1111:1111” to its network equivalent. The second function is used to configure an interface with the specified IPv6 address. An example is shown in listing Address Configuration Example .

Listing - IPv6 Address Configuration Example

```
CPU_BOOLEAN    cfg_success;
NET_IPv6_ADDR  ipv6_addr;
NET_FLAGS      ipv6_flags;
RTOS_ERR       err;

NetASCII_Str_to_IP((CPU_CHAR *)"fe80::1111:1111",   /* See Note #1 */
                   &ipv6_addr,
                    NET_IPv6_ADDR_SIZE,
                   &err);

ipv6_flags = 0;
DEF_BIT_SET(ipv6_flags, NET_IPv6_FLAG_BLOCK_EN);          /* See Note #2 */
DEF_BIT_SET(ipv6_flags, NET_IPv6_FLAG_DAD_EN);            /* See Note #3 */

cfg_success = NetIPv6_CfgAddrAdd(if_nbr,           /* See Note #4 */
                                 &ipv6_addr,        /* See Note #5 */
                                  64,               /* See Note #6 */
                                  ipv6_flags,       /* See Note #7 */
                                 &err);             /* See Note #8 */
```

1. See NetASCII_Str_to_IP for more details.
2. Set Address Configuration as blocking.
3. Enable DAD with Address Configuration.
4. The first argument is the number representing the network interface that is to be configured. This value is obtained as the result of a successful call to NetIF_Ether_Add() or NetIF_WiFi_Add().
5. The second argument is the pointer to the NET_IPv6_ADDR value representing the IPv6 address to be configured.
6. The third argument is the IPv6 prefix length of the address to configured.
7. The fourth argument is a set of network flags holding options specific to the address configuration process.
8. The fifth argument is a pointer to a RTOS_ERR variable containing the return error code for the function. If the interface address information is configured successfully, then the return error code will contain the value RTOS_ERR_NONE. Additionally, function returns a Boolean value of DEF_OK or DEF_FAIL depending on the result. Either the return value or the RTOS_ERR variable may be checked for return status; however, the RTOS_ERR contains more detailed information and should therefore be the preferred check.

As shown in Address Configuration Example , the NetIPv6_CfgAddrAdd() function can take as an argument a set of network flags. The following options are available:

Flags	Description
NET_IPv6_FLAG_BLOCK_EN	Enables blocking mode.
NET_IPv6_FLAG_DAD_EN	Enables Duplication Address Detection (DAD) with the address configuration process.

It is therefore possible to make the function blocking or not, or to enable Duplication Address Detection with the address configuration.

If the function is made non-blocking, it is possible to set a hook function to notify the application when the address configuration process has finished. The API function NetIPv6_AddrSubscribe() can be used to set the hook function. Refer to section IPv6 Address Configuration Hook Function for all the details on the hook function format and usage.

Address Configuration Example in the *IPv6 - Assign an Address to a Network Interface* page shows an example of a non-blocking IPv6 static address configuration.

Listing - Non-Blocking IPv6 Address Configuration Example

```
CPU_BOOLEAN    cfg_success;
NET_IPv6_ADDR  ipv6_addr;
NET_FLAGS      ipv6_flags;
RTOS_ERR       err;

NetASCII_Str_to_IP((CPU_CHAR *)"fe80::1111:1111",     /* Convert IPv6 string address to 128 bit address.     */
                   &ipv6_addr,
                    NET_IPv6_ADDR_SIZE,
                   &err);
```

```
/* Set hook function to received addr cfg result.      */NetIPv6_AddrSubscribe(&App_AddrCfgResult,/* TODO update pointer to hook fnct implemented in App. */&err);

ipv6_flags = 0;DEF_BIT_CLR(ipv6_flags, NET_IPv6_FLAG_BLOCK_EN);/* Set Address Configuration as non-blocking.        */DEF_BIT_SET(ipv6_flags, NET_IPv6_FLAG_DAD_EN);/* Enable DAD with Address Configuration.          */

cfg_success = NetIPv6_CfgAddrAdd(if_nbr,/* Add a statically-configured IPv6 host address to ... */&ipv6_addr,/* ... the interface. */64,
                     ipv6_flags,&err);
```

IPv6 - Stateless Address Autoconfiguration (SLAAC)

The IPv6 protocol defines an address Auto-Configuration procedure allowing a network interface to set itself an IPv6 Link-Local address based on its Interface ID. The Auto-Configuration process will also query the local network to find an IPv6 router that could send prefix information to set an IPv6 global address.

The TCP/IP stack supports only the EUI-64 format for interface ID. This format creates a 64 bits ID based on the 48 bits MAC address of the interface. Those 64 bits will become the 64 least significant bits of the IPv6 addresses configured with the Stateless Auto-Configuration process.

There are two methods for configuring the IPv6 Stateless Auto-Configuration process:

1. Using the configuration structure NET_IF_ETHER_CFG or NET_IF_WIFI_CFG, as explained in Network Interface Start Setup and Starting a Network Interface , or
2. Using the following functions manually or after the interface has been started: NetIPv6_AddrSubscribe() NetIPv6_AddrAutoCfgEn()

Below is an example of manually starting the Auto-configuration process for an IPv6 network interface.

Listing - Starting the Auto-configuration process

```
NetIPv6_AddrAutoCfgEn(if_nbr,
            DEF_YES,
            &err);
```

The IPv6 Auto-Configuration procedure in the Network module is a non-blocking process. To retrieve the result of the Auto-Configuration, a hook function can be configured that will be called by the TCP/IP stack each time an address has been configured on an interface. The API function used to set the hook function is NetIPv6_AddrSubscribe() .

> Note: the hook function is called each time an address has been configured, for any reason, and not just when Auto-configuration is complete. The response must be analyzed within the hook function to determine the type of configured address. More detail on how to handle the hook response may be found in IPv6 - Address Configuration Hook Function .

IPv6 - Address Configuration Hook Function

When doing any sort of address assignment to an IPv6 interface, a hook function will be called at the end of the assignment process, provided that the NetIPv6_AddrSubscribe() function was used.

Listing - Setting an IPv6 Hook function

```
RTOS_ERR    err;

NetIPv6_AddrSubscribe(&Ex_IPv6_AddrCfgResult,
             &err);
```

The hook function will be called at the end of each of three possible address configuration processes:

1. Static address assignment
2. Auto-configuration of a Link Local address
3. Auto-configuration of a global address

The hook function will be called upon either success or failure of the process. It is the application's responsibility to analyze the status of the response and the IPv6 address type that has been configured.

Below is an example of handling an IPv6 address configuration hook function call.

Listing - handling an IPv6 address configuration hook function call

```
static  void  Ex_IPv6_AddrCfgResult (       NET_IF_NBR                  if_nbr,
                                     NET_IPv6_CFG_ADDR_TYPE      addr_type,
                                const  NET_IPv6_ADDR            *p_addr_cfgd,
                                     NET_IPv6_ADDR_CFG_STATUS   addr_cfg_status)
{
    CPU_CHAR   ip_string[NET_ASCII_LEN_MAX_ADDR_IPv6];
    RTOS_ERR   err;

    PP_UNUSED_PARAM(if_nbr);

    if (p_addr_cfgd != DEF_NULL) {
        NetASCII_IPv6_to_Str((NET_IPv6_ADDR *)p_addr_cfgd, ip_string, DEF_NO, DEF_YES, &err);
        EX_TRACE("IPv6 Address Link Local: %s\n", ip_string);
    }

    switch (addr_type) {
        case NET_IPv6_CFG_ADDR_TYPE_STATIC:
             switch (addr_cfg_status) {
                case NET_IPv6_ADDR_CFG_STATUS_SUCCEED:
                     EX_TRACE("IPv6 Address Static: %s, configured successfully\n", ip_string);
                     break;

                case NET_IPv6_ADDR_CFG_STATUS_DUPLICATE:
                     EX_TRACE("IPv6 Address Static already exists on the network\n");
                     break;

                default:
                     EX_TRACE("IPv6 Address Static configuration failed.\n");
                     break;
             }
             break;

        case NET_IPv6_CFG_ADDR_TYPE_AUTO_CFG_LINK_LOCAL:
             switch (addr_cfg_status) {
                case NET_IPv6_ADDR_CFG_STATUS_SUCCEED:
                     EX_TRACE("IPv6 Address Link Local: %s, configured successfully\n", ip_string);
                     break;

                case NET_IPv6_ADDR_CFG_STATUS_DUPLICATE:
                     EX_TRACE("IPv6 Address Link Local already exists on the network\n");
                     break;

                default:
                    printf("IPv6 Address Link Local configuration failed.\n");
                    break;
             }
             break;

        case NET_IPv6_CFG_ADDR_TYPE_AUTO_CFG_GLOBAL:
             switch (addr_cfg_status) {
                case NET_IPv6_ADDR_CFG_STATUS_SUCCEED:
                     EX_TRACE("IPv6 Address Global: %s, configured successfully\n", ip_string);
                     break;

                case NET_IPv6_ADDR_CFG_STATUS_DUPLICATE:
                     EX_TRACE("IPv6 Address Global already exists on the network\n");
                     break;

                default:
                     EX_TRACE("IPv6 Address Global configuration failed.\n");
                     break;
             }
             break;

        default:
             break;
    }
```

```
}
```

Interface Programming

This section describe how to use and control Interface's options such:

- Network Interface Hardware Address (MAC)
- Network Interface Link State
- Network Interface Maximum Transmit Unit (MTU)

Configuration

Some parameters should be configured and/or optimized for your project requirements. See the section Network Interface Start Setup for further details.

Using Network Interface Programming API

Wherever you want to configure or access an Interface option, you should include one or many of these files:

Include file	Description
rtos/net/include/net_if.h	Functions used for Interface API
rtos/net/include/net_if_ether.h	Ethernet Interface's API reference (used with NetIF_Ether_Add()).
rtos/net/include/net_if_wifi.h	Wireless Interface's API reference (used with NetIF_WiFi_Add()).

API Reference

All Interface APIs are presented in the section Network Interface Functions .

Function Name	Description
NetIF_Add()	Add a network device and hardware as a network interface.
NetIF_AddrHW_Get()	Get network interface's hardware address.
NetIF_AddrHW_IsValid()	Validate a network interface hardware address.
NetIF_AddrHW_Set()	Set network interface's hardware address.
NetIF_CfgPerfMonPeriod()	Configure the network interface Performance Monitor Handler timeout.
NetIF_CfgPhyLinkPeriod()	Configure network interface Physical Link State Handler timeout.
NetIF_GetRxDataAlignPtr()	Get an aligned pointer into a receive application data buffer.
NetIF_GetTxDataAlignPtr()	Get an aligned pointer into a transmit application data buffer.
NetIF_IO_Ctrl()	Handle network interface and/or device-specific (I/O) control(s).
NetIF_IsEn()	Validate network interface as enabled.
NetIF_IsEnCfgd()	Validate configured network interface as enabled.
NetIF_ISR_Handler()	Handle a network interface's device interrupts.
NetIF_IsValid()	Validate network interface number.
NetIF_IsValidCfgd()	Validate configured network interface number.
NetIF_LinkStateGet()	Get network interface's last known physical link state.
NetIF_LinkStateSubscribe()	Subscribe to get notified when an interface link state changes.
NetIF_LinkStateUnsubscribe()	Unsubscribe to get notified when interface link state changes.
NetIF_LinkStateWaitUntilUp()	Wait for a network interface's physical link state to be UP.
NetIF_MTU_Get()	Get network interface's MTU.
NetIF_MTU_Set()	Set network interface's MTU.
NetIF_Start()	Start a network interface.
NetIF_Stop()	Stop a network interface.

Network Interface Hardware Address (MAC)

Default MAC Address

When starting an Interface, the Network module can automatically use a MAC address either defined in the configuration or already loaded in MAC controller. As specified in Network Interface Start Setup each device configuration contains a variable which specifies whether the MAC address must be:

- loaded from the MAC controller (indicated by a null or empty string in the configuration structure)
- loaded from a custom MAC address obtained at run-time
- loaded as is from a string in the configuration structure

Getting the MAC Address

Many types of network interface hardware require the use of a link layer protocol address. In the case of Ethernet, this address is sometimes known as the hardware address or MAC address. In some applications, it may be desirable to get the current configured hardware address for a specific interface. This may be performed by calling NetIF_AddrHW_Get() with the appropriate arguments.

Listing - Calling NetIF_AddrHW_Get()

```
NetIF_AddrHW_Get( if_nbr,               (1)
             &addr_hw_sender[0],      (2)
             &addr_hw_len,           (3)
             &err);                 (4)
```

1. The first argument specifies the interface number from which to get the hardware address. The interface number is acquired upon the successful addition of the interface.
2. The second argument is a pointer to a CPU_INT08U array used to provide storage for the returned hardware address. This array *must* be sized large enough to hold the returned number of bytes for the given interface's hardware address. The lowest index number in the hardware address array represents the most significant byte of the hardware address.
3. The third function is a pointer to a CPU_INT08U variable that the function returns the length of the specified interface's hardware address.
4. The fourth argument is a pointer to a RTOS_ERR variable containing the return error code for the function. If the hardware address is successfully obtained, then the return error code will contain the value RTOS_ERR_NONE.

Setting MAC Address

Some applications prefer to configure the hardware device's hardware address via software during run-time as opposed to a run-time auto-loading EEPROM as is common for many Ethernet devices. If the application is to set or change the hardware address during run-time, this may be performed by calling NetIF_AddrHW_Set() with the appropriate arguments. Alternatively, the hardware address may be statically configured via the device configuration structure and later changed during run-time.

Listing - Calling NetIF_AddrHW_Set()

```
NetIF_AddrHW_Set( if_nbr,          (1)
             &addr_hw[0],       (2)
             &addr_hw_len,      (3)
             &err);            (4)
```

1. The first argument specifies the interface number to set the hardware address. The interface number is acquired upon the successful addition of the interface.
2. The second argument is a pointer to a CPU_INT08U array which contains the desired hardware address to set. The lowest index number in the hardware address array represents the most significant byte of the hardware address.
3. The third function is a pointer to a CPU_INT08U variable that specifies the length of the hardware address being set. In most cases, this can be specified as sizeof(addr_hw) assuming addr_hw is declared as an array of CPU_INT08U.

4. The fourth argument is a pointer to a RTOS_ERR variable containing the return error code for the function. If the hardware address is successfully obtained, then the return error code will contain the value RTOS_ERR_NONE.

Note: In order to set the hardware address for a particular interface, it *must* first be stopped. The hardware address may then be set, and the interface re-started.

Getting a Host MAC Address on the Network

In order to determine the MAC address of a host on the network, the Network Protocol Stack must have an ARP cache entry for the specified host protocol address. An application may check to see if an ARP cache entry is present by calling NetARP_CacheGetAddrHW() .

If an ARP cache entry is not found, the application may call NetARP_CacheProbeAddrOnNet() to send an ARP request to all hosts on the network. If the target host is present, an ARP reply will be received shortly; the application should wait and then call NetARP_CacheGetAddrHW() to determine if the ARP reply has been entered into the ARP cache.

The following example shows how to obtain the Ethernet MAC address of a host on the local area network:

Listing - Obtaining the Ethernet MAC address of a host

```
void AppGetRemoteHW_Addr (NET_IF_NBR  if_nbr)
{
  NET_IPv4_ADDR   addr_ipv4_local;
  NET_IPv4_ADDR   addr_ipv4_remote;
  CPU_CHAR       *p_addr_ipv4_local;
  CPU_CHAR       *p_addr_ipv4_remote;
  CPU_CHAR        addr_hw_str[NET_IF_ETHER_ADDR_SIZE_STR];
  CPU_INT08U      addr_hw[NET_IF_ETHER_ADDR_SIZE];
  RTOS_ERR        err;

                                    /* ------------- PREPARE IPv4 ADDRs ------------- */
  p_addr_ipv4_local = "10.10.1.10";   /* MUST be one of host's configured IPv4 addrs.   */
  addr_ipv4_local   = NetASCII_Str_to_IPv4(p_addr_ipv4_local, &err);
  if (err.Code != RTOS_ERR_NONE) {
    printf(" Error #%d converting IPv4 address %s", err.Code, p_addr_ipv4_local);
    return;
  }

  p_addr_ipv4_remote = "10.10.1.50";  /* Remote host's IPv4 addr to get hardware addr.  */
  addr_ipv4_remote = NetASCII_Str_to_IPv4(p_addr_ipv4_remote, &err);
  if (err.Code != RTOS_ERR_NONE) {
    printf(" Error #%d converting IPv4 address %s", err.Code, p_addr_ipv4_remote);
    return;
  }

  addr_ipv4_local  = NET_UTIL_HOST_TO_NET_32(addr_ipv4_local);
  addr_ipv4_remote = NET_UTIL_HOST_TO_NET_32(addr_ipv4_remote);

                                    /* ------------- PROBE ADDR ON NET -------------- */
  NetARP_CacheProbeAddrOnNet(NET_PROTOCOL_TYPE_IP_V4,
            (CPU_INT08U *)&addr_ipv4_local,
            (CPU_INT08U *)&addr_ipv4_remote,
                          sizeof(addr_ipv4_remote),
                         &err);
  if (err.Code != RTOS_ERR_NONE) {
    printf(" Error #%d probing address %s on network", err.Code, addr_ipv4_remote);
    return;
  }

  OSTimeDly(2);                       /* Delay short time for ARP to probe network.     */

                                    /* ----- QUERY ARP CACHE FOR REMOTE HW ADDR ----- */
(void)NetARP_CacheGetAddrHW(if_nbr,
                         &addr_hw[0],
                          sizeof(addr_hw_str),
          (CPU_INT08U *)&addr_ipv4_remote,
                          sizeof(addr_ipv4_remote),
                         &err);
  switch (err.Code) {
    case RTOS_ERR_NONE:
        NetASCII_MAC_to_Str(&addr_hw[0],
                            &addr_hw_str[0],
                             DEF_NO,
                             DEF_YES,
                            &err);
        if (err.Code != RTOS_ERR_NONE) {
            printf(" Error #%d converting hardware address", err.Code);
            return;
        }

        printf(" Remote IPv4 Addr %s @ HW Addr %s\n\r", p_addr_ipv4_remote, &addr_hw_str[0]);
        break;

    case RTOS_ERR_NOT_FOUND:
        printf("  Remote IPv4 Addr %s NOT found on network\n\r", p_addr_ipv4_remote);
        break;
```

```
case RTOS_ERR_NET_ADDR_UNRESOLVED:printf(" Remote IPv4 Addr %s NOT YET found on network\n\r", p_addr_ipv4_remote);break;

default:printf(" Error #%d querying ARP cache", err.Code);break;}}
```

Since ARP module is only used in affiliation with IPv4, getting the MAC address associated with an IP address is only supported for IPv4 address. IPv6 layer uses NDP (Neighbor Discovery Protocol) to associate IPv6 address with Link-Layer address. Unfortunately, NDP does not yet provides API function to get the MAC address associated with an IPv6 address.

Network Interface Link State

Getting the Current Link State for an Interface

The Network module provides two mechanisms for obtaining interface link state.

1. A function which reads a global variable that is periodically updated.
2. A function which reads the current link state from the hardware.

Method 1 provides the fastest mechanism to obtain link state since it does not require communication with the physical layer device. For most applications, this mechanism is suitable and if necessary, the polling rate can be increased by calling NetIF_CfgPhyLinkPeriod() . In order to utilize Method 1, the application may call NetIF_LinkStateGet() which returns NET_IF_LINK_UP or NET_IF_LINK_DOWN.

The accuracy of Method 1 can be improved by using a physical layer device and driver combination that supports link state change interrupts. In this circumstance, the value of the global variable containing the link state is updated immediately following a link state change. Therefore, the polling rate can be reduced further if desired and a call to NetIF_LinkStateGet() will return the actual link state.

Method 2 requires the application to call NetIF_IO_Ctrl() with the option parameter set to either NET_IF_IO_CTRL_LINK_STATE_GET or NET_IF_IO_CTRL_LINK_STATE_GET_INFO.

- If the application specifies NET_IF_IO_CTRL_LINK_STATE_GET, then NET_IF_LINK_UP or NET_IF_LINK_DOWN will be returned.
- Alternatively, if the application specifies NET_IF_IO_CTRL_LINK_STATE_GET_INFO, the link state details such as speed and duplex will be returned.

The advantage to Method 2 is that the link state returned is the actual link state as reported by the hardware at the time of the function call. However, the overhead of communicating with the physical layer device may be high and therefore some cycles may be wasted waiting for the result since the connection bus between the CPU and the physical layer device is often only a couple of MHz.

Calling NetIF_IO_Ctrl() will poll the hardware for the current link state. Alternatively, NetIF_LinkStateGet() gets the approximate link state by reading the interface link state flag. Polling the Ethernet hardware for link state takes significantly longer due to the speed and latency of the MII bus. Consequently, it may not be desirable to poll the hardware in a tight loop. Reading the interface flag is fast, but the flag is only periodically updated by the Net IF every 250ms (default) when using the generic Ethernet PHY driver. PHY drivers that implement link state change interrupts may change the value of the interface flag immediately upon link state change detection. In this scenario, calling NetIF_LinkStateGet() is ideal for these interfaces.

Listing - Calling NetIF_IO_Ctrl()

```
NetIF_IO_Ctrl( if_nbr,                              (1)
             NET_IF_IO_CTRL_LINK_STATE_GET_INFO,     (2)
             &link_state,                          (3)
             &err);
```

1. The first argument specifies the interface number from which to get the physical link state.
2. The second argument specifies the desired function that NetIF_IO_Ctrl() will perform. In order to get the current interfaces' link state, the application should specify this argument as either:

NET_IF_IO_CTRL_LINK_STATE_GET NET_IF_IO_CTRL_LINK_STATE_GET_INFO

1. The third argument is a pointer to a link state variable that must be declared by the application and passed to NetIF_IO_Ctrl() .

Wait Until the Cable is Connected (Link Up)

Sometime the application might want to wait until the link is up before starting doing things on the network. The function NetIF_LinkStateWaitUntilUp() can be use to do such functionality.

Subscribe / Unsubscribe to Link State Changes

The Network stack offers two API functions: NetIF_LinkStateSubscribe() , NetIF_LinkStateUnsubscribe() , allowing the customer application to implement a callback function that will be called when a link state change occurs.

Increasing the Rate of Link State Polling

The application may increase the link state polling rate by calling NetIF_CfgPhyLinkPeriod() . The default value is 250ms.

Forcing an Ethernet PHY to a Specific Link State

The generic PHY driver that comes with the Network module does **not** provide a mechanism for disabling auto-negotiation and specifying a desired link state. This restriction is required in order to remain MII register block-compliant with all (R)MII compliant physical layer devices.

However, the Network module does provide a mechanism for coaching the physical layer device into advertising only the desired auto-negotiation states. This may be achieved by adjusting the physical layer device configuration structure, NET_PHY_CFG_ETHER, with alternative link speed and duplex values. This configuration structure is generally located in the BSP file for the device (bsp_net_ether_xxx.c)

The following is an example physical layer device configuration structure.

```
NET_PHY_CFG_ETHER NetPhy_Cfg_Generic_0 = {
  0,
  NET_PHY_BUS_MODE_MII,
  NET_PHY_TYPE_EXT,
  NET_PHY_SPD_AUTO,
  NET_PHY_DUPLEX_AUTO
};
```

The parameters NET_PHY_SPD_AUTO and NET_PHY_DUPLEX_AUTO may be changed to match any of the following settings:

```
NET_PHY_SPD_10
NET_PHY_SPD_100
NET_PHY_SPD_1000
NET_PHY_SPD_AUTO
NET_PHY_DUPLEX_HALF
NET_PHY_DUPLEX_FULL
NET_PHY_DUPLEX_AUTO
```

This mechanism is only effective when both the physical layer device attached to the target and the remote link state partner support auto-negotiation.

Refer to section PHY Configuration and Information for detailed information on the PHY device configuration.

Network Interface Maximum Transmit Unit (MTU)

Getting the MTU

On occasion, it may be desirable to have the application aware of an interface's Maximum Transmission Unit. The MTU for a particular interface may be acquired by calling NetIF_MTU_Get() with the appropriate arguments.

Listing - Calling NetIF_MTU_Get()

```
mtu = NetIF_MTU_Get(if_nbr, &err);   (1)

1. NetIF_MTU_Get() requires two arguments. The first function argument is the interface number to get the current configured MTU, and the second argument is a pointer to a NET_ERR to contain the return error code. The interface number is acquired upon the successful addition of the interface, and upon the successful return of the function, the return error variable will contain the value NET_IF_ERR_NONE. The result is returned into a local variable of type NET_MTU.

#### Setting the MTU

 Some networks prefer to operate with a non-standard MTU. If this is the case, the application may specify the MTU for a particular interface by calling  [NetIF_MTU_Set()](../network-api/01-network-core-api.md#net-if-mtu-set) with the appropriate arguments.

###### Listing - Calling NetIF_MTU_Set()


```C
NetIF_MTU_Set(if_nbr, mtu, &err);   (1)
```

1. NetIF_MTU_Set() requires three arguments. The first function argument is the interface number of the interface to set the specified MTU. The second argument is the desired MTU to set, and the third argument is a pointer to a NET_ERR variable that will contain the return error code. The interface number is acquired upon the successful addition of the interface, and upon the successful return of the function, the return error variable will contain the value NET_IF_ERR_NONE and the specified MTU will be set.

Note: The configured MTU cannot be greater than the largest configured transmit buffer size associated with the specified interfaces' device minus overhead. Transmit buffer sizes are specified in the device configuration structure for the specified interface. For more information about configuring device buffer sizes, refer to section Network Interface Controller Configuration .

Socket Programming

The following sections provide sample code describing how sockets work.

For those interested in BSD socket programming, there are plenty of books, online references, and articles dedicated to this subject.

The sockets API provides many configuration options, and it's very difficult to cover all variations of its use. The examples have been designed to be as simple as possible. Hence, only basic error checking is performed. When it comes to building real applications, those checks should be extended to deliver a product that is as robust as possible.

Socket Description

A network socket is an inter-process communication implementation using an IP stack on a network. Sockets allow one application process to communicate with another whether it is local on the same system or remote over the network. You use and control a socket using an application programming interface (API) provided by the Network stack and which is usually based on Berkeley sockets standard.

Two socket types are identified: Datagram sockets and Stream sockets and using the following standard protocol:

Protocol	Description
IP	Internet Protocol provides network routing using IPv4 or IPv6 addressing.
UDP	User Datagram Protocol - Datagram sockets (Connectionless sockets)
TCP	Transmission Control Protocol - Stream sockets (Connection-oriented sockets)

Configuration
```

Some parameters should be configured and/or optimized for your project requirements. See the section Socket Layer Configuration for further details.

Using Network Socket Programming API

Wherever you want to configure, access or use a socket you should include one or many of these files:

Include file	Description
rtos/net/include/net_app.h	Functions used for NetApp API
rtos/net/include/net_sock.h	Functions used for NetSock API
rtos/net/include/net_bsd.h	Functions used for BSD API
rtos/net/include/net_tcp.h	Functions used for Stream Socket Option API
/rtos/net/include/net_util.h	Functions and Macro used for conversion utilities API
/rtos/net/include/net_ascii.h	Functions and Macro used for conversion utilities API
/rtos/net/include/net_ipv4.h	Functions used for IPv4 options API
/rtos/net/include/net_ipv6.h	Functions used for IPv6 options API

API Reference

In addition to the BSD 4.x sockets application interface (API), the Network stack gives the developer the opportunity to use Micrium's own socket functions with which to interact.

Although there is a great deal of similarity between the two APIs, the parameters of the two sets of functions differ slightly. The purpose of the following sections is o give developers a first look at Micrium's functions by providing concrete examples of how to use the API.

The Network module sockets can be accessed and controlled using 3 different APIs set:

- BSD APIs are following BSD 4.x functions prototype and there are presented in the section BSD Functions .
- Net Sock APIs are the low-level socket API. They are based on BSD sockets but with more arguments and it can return the error cause. Net Sock APIs are presented in the section Network Socket Functions .
- Net APP APIs include more parameters of the Net Sock API, to accomplish some operation that should be always performed when calling Socket API, such as setting timeout, retry, apply delay and such. Basically, there a wrapper of Net Sock APIs. Net APP APIs are presented in the section Network Application Interface Functions .

Functions

The following functions must be used to open, connect, receive, transmit, close and so on:

BSD	Net Sock	Net App	Description
socket()	NetSock_Open()	NetApp_SockOpen()	Create a datagram (i.e., UDP) or stream (i.e., TCP) type socket.
bind()	NetSock_Bind()	NetApp_SockBind()	Assign network addresses to sockets.
connect()	NetSock_Conn()	NetApp_SockConn()	Connect a local socket to a remote socket address.
listen() (TCP only)	NetSock_Listen()	NetApp_SockListen()	Set a socket to accept incoming connections.
accept() (TCP only)	NetSock_Accept()	NetApp_SockAccept()	Wait for new socket connections on a listening server socket.
recv() / recvfrom()	NetSock_RxData() / NetSock_RxDataFrom()	NetApp_SockRx()	Copy up to a specified number of bytes received from a remote socket into an application memory buffer.
send() / sendto()	NetSock_TxData() / NetSock_TxDataTo()	NetApp_SockTx()	Copy bytes from an application memory buffer into a socket to send to a remote socket.
close()	NetSock_Close()	NetApp_SockConn()	Terminate communication and free a socket.

BSD	Net Sock	Net App	Description
select()	NetSock_Sel()	N/A	Check if any sockets are ready for available read or write operations or error conditions.
N/A	NetSock_SelAbort()	N/A	Abort a select (i.e., unblock task(s) waiting on socket(s) using the select)

Conversion Utilities

The following functions should be used when converting 16 or 32-bit values from or to network order or even when converting IP address.

BSD	Micrium Network Functions	Description
htonl()	NET_UTIL_HOST_TO_NET_32()	Convert 32-bit integer values from network-order to CPU host-order.
htons()	NET_UTIL_HOST_TO_NET_16()	Convert 16-bit integer values from network-order to CPU host-order.
ntohl()	NET_UTIL_NET_TO_HOST_32()	Convert 32-bit integer values from network-order to CPU host-order.
ntohs()	NET_UTIL_NET_TO_HOST_16()	Convert 16-bit integer values from network-order to CPU host-order.
inet_addr() (IPv4 only)	NetASCII_Str_to_IPv4()	Convert a string of an IPv4 address in dotted-decimal notation to an IPv4 address in host-order.
inet_aton() (IPv4 only)	NetASCII_Str_to_IPv4()	Convert an IPv4 address in ASCII dotted-decimal notation to a network protocol IPv4 address in network-order.
inet_ntoa() (IPv4 only)	NetASCII_IPv4_to_Str()	Convert an IPv4 address in host-order into an IPv4 dotted-decimal notation ASCII string.

Socket option function

It is possible to configure many options on a socket, with the followings function you could configure majority of socket's option but some other option can be accessed using some specific API, see Socket Options section.

BSD	Net Sock	Description
getsockopt()	NetSock_OptGet()	Get a specific option value on a specific socket.
setsockopt()	NetSock_OptSet()	Set a specific option value on a specific socket.

Select() Macro

The following functions should be used when select() is used. It helps to configure and read descriptors to a select() calls.

BSD	Net Sock	Description
FD_CLR()	NET_SOCK_DESC_CLR()	Remove a socket file descriptor ID as a member of a file descriptor set.
FD_ISSET()	NET_SOCK_DESC_IS_SET()	Check if a socket file descriptor ID is a member of a file descriptor set.
FD_SET()	NET_SOCK_DESC_SET()	Add a socket file descriptor ID as a member of a file descriptor set.
FD_ZERO()	NET_SOCK_DESC_INIT()	Initialize/zero-clear a file descriptor set.

Miscellaneous

Here is a list of other APIs functions that can help in certain conditions:

Function	Description
NetApp_SetSockAddr()	Setup a socket address from an IPv4 or an IPv6 address.
NetApp_ClientStreamOpen()	Open a stream socket for a client using an IPv4 or IPv6 address.

Function	Description
NetApp_ClientStreamOpenByHostname()	Open a stream socket for a client using a hostname (resolve the hostname using DNS + Autoselect remote address)
NetApp_TimeDly_ms()	Delay for a specified time, in milliseconds.
NetSock_GetConnTransportID()	Gets a socket's transport layer connection handle ID (e.g., TCP connection ID) if available.
NetSock_IsConn()	Check if a socket is connected to a remote socket.
NetSock_PoolStatResetMaxUsed()	Reset Network Sockets' statistics pool's maximum number of entries used.

Socket Error Codes

When socket functions return error codes, the error codes should be inspected to determine if the error is a temporary, non-fault condition (such as no data to receive) or fatal (such as the socket has been closed).

The following errors are not fatal and if they occur, we recommend trying again the operation after a delay if the socket is in a none-blocking mode.

Error Code	Description
RTOS_ERR_IF_LINK_DOWN	The network interface link is down.
RTOS_ERR_POOL_EMPTY	One of the network pool from which the stack is trying to get an element is empty. Probably a network buffer pool.
RTOS_ERR_TIMEOUT	The operation timed out.
RTOS_ERR_WOULD_BLOCK	The operation would have timed out if the socket was in blocking mode.

For any other error code returned, we recommend closing the socket.

Network Sockets Concepts

Byte-Ordering

TCP/IP has an universal standard to allow communication between any platforms. Thus, it's necessary to have method arranging information so that big-endian and little-endian machines can understand each other. On platforms where data is already correctly ordered, the functions do nothing and are frequently macro'd into empty statements. Byte-ordering functions should always be used as they do not impact performance on systems that are already correctly ordered and they promote code portability.

BSD	Micrium Native Functions	Description
htonl()	NET_UTIL_HOST_TO_NET_32	Convert 32-bit integer values from network-order to CPU host-order.
htons()	NET_UTIL_HOST_TO_NET_16()	Convert 16-bit integer values from network-order to CPU host-order.
ntohl()	NET_UTIL_NET_TO_HOST_32()	Convert 32-bit integer values from network-order to CPU host-order.
ntohs()	NET_UTIL_NET_TO_HOST_16()	Convert 16-bit integer values from network-order to CPU host-order.

The four byte-ordering functions stand for host to network short, host to network long, network to host short, and network to host long. htons() translates a short integer from host byte-order to network byte-order. htonl is similar, translating a long integer. The other two functions do the reverse, translating from network byte-order to host byte-order.

IP Addresses and Data Structures

Communication using sockets requires configuring or reading network addresses from network socket address structures.

IP Address Versions

There are two version of IP. The most known version is the IPv4 address which is 4 bytes long and commonly written in dots and numbers format, such as

```
192.168.1.1
```

The next generation of IP is IPv6 which is also supported by the Network module. The IPv6 address is 128 bits long and it is represented using numbers, letters and semi-colons such as:

```
fe80:14f:c9d2:12::51
```

IPv4 Addresses

The BSD socket API defines a generic socket address structure as a blank template with no address-specific configuration.

Listing - Generic (non-address-specific) address structures

```
struct  sockaddr {                    /* Generic BSD   socket address structure    */
    CPU_INT16U  sa_family;               /*  Socket address family                   */
    CPU_CHAR    sa_data[14];             /*  Protocol-specific address information    */
};

typedef  struct  net_sock_addr {          /* Generic Micrium socket address structure  */
    NET_SOCK_ADDR_FAMILY   AddrFamily;
    CPU_INT08U             Addr[14];
} NET_SOCK_ADDR;
```

...as well as specific socket address structures to configure each specific protocol address family's network address configuration (e.g., IPv4 socket addresses):

Listing - Internet (IPv4) address structures

```
struct  in_addr {
    NET_IP_ADDR  s_addr;                    /* IPv4 address (32 bits)                 */
};

struct  sockaddr_in {                       /* BSD    IPv4 socket address structure     */
    CPU_INT16U       sin_family;              /*  Internet address family (e.g., AF_INET) */
    CPU_INT16U       sin_port;                /*  Socket  address port number (16 bits)  */
    struct in_addr   sin_addr;                /*  IPv4   address       (32 bits)          */
    CPU_CHAR         sin_zero[8];             /*  Not used (all zeros)                   */
};

typedef  struct  net_sock_addr_ipv4 {           /* Micrium socket address structure        */
    NET_SOCK_ADDR_FAMILY   AddrFamily;
    NET_PORT_NBR           Port;
    NET_IPv4_ADDR          Addr;
    CPU_INT08U             Unused[8];
} NET_SOCK_ADDR_IPv4;
```

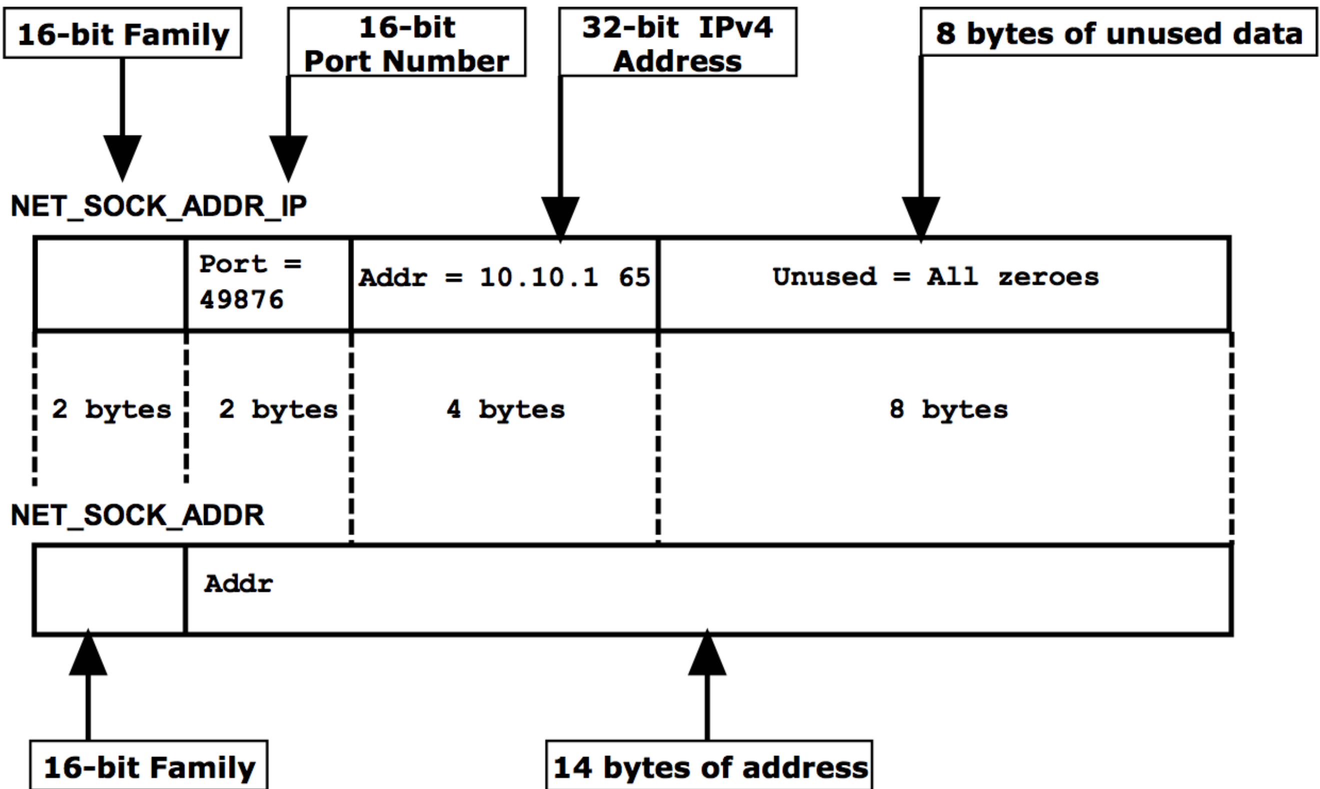
A socket address structure's AddrFamily/sa_family/sin_family value *must* be read/written in host CPU byte order, while all Addr/sa_data values *must* be read/written in network byte order (big endian).

Even though socket functions – both Micrium and BSD – pass pointers to the generic socket address structure, applications *must* declare and pass an instance of the specific protocol's socket address structure (e.g., an IPv4 address structure). For microprocessors, that requires data access to be aligned to appropriate word boundaries. This forces compilers to declare an appropriately-aligned socket address structure so that all socket address members are correctly aligned to their appropriate word boundaries.

Caution: Applications should avoid, or be cautious when, declaring and configuring a generic byte array as a socket address structure, since the compiler may not correctly align the array or the socket address structure's members to appropriate word boundaries.

The figure below shows an example IPv4 instance of the Network module NET_SOCK_ADDR_IPv4 (sockaddr_in) structure overlaid on top of NET_SOCK_ADDR (sockaddr) the structure.

Figure - NET_SOCK_ADDR_IP is the IPv4 specific instance of the generic NET_SOCK_ADDR data structure



A socket could configure the example socket address structure in the figure below to bind on IPv4 address 10.10.1.65 and port number 49876 with the following code:

Listing - Bind on 10.10.1.65

```
NET_SOCK_ADDR_IPv4   addr_local;
NET_IPv4_ADDR        addr_ip;
NET_PORT_NBR         addr_port;
NET_SOCK_RTN_CODE    rtn_code;
NET_ERR              err;

addr_ip   = NetASCII_Str_to_IPv4("10.10.1.65", &err);
addr_port = 49876;
Mem_Clr(&addr_local,
        sizeof(addr_local));
addr_local.AddrFamily = NET_SOCK_ADDR_FAMILY_IP_V4;              /\* = AF_INET†† Figure 9-1 \*/
addr_local.Addr       = NET_UTIL_HOST_TO_NET_32(addr_ip);
addr_local.Port       = NET_UTIL_HOST_TO_NET_16(addr_port);
rtn_code              = NetSock_Bind(                 sock_id,
                                    (NET_SOCK_ADDR   \*)&addr_local,  /\* Cast to generic addr† \*/
                                                  sizeof(addr_local),
                                                  &err);
```

† The address of the specific IPv4 socket address structure is cast to a pointer to the generic socket address structure.

IPv6 Addresses

Socket addresses structures for IPv6 are similar to IPv4.

Listing - Internet (IPv4) address structures

```
struct  in6_addr {
    in6_addr_t          s_addr[16];
};

struct  sock_addr_in6 {
        sa_family_t   sin6_family;       /* Internet address family (AF_INET6)          */
        in_port_t     sin6_port;         /* Socket  address port number (16 bits)       */
        CPU_INT32U    sin6_flowinfo;     /* Flow info.                                  */
    struct  in6_addr     sin6_addr;      /* IPv6   address.  (128 bits)                 */
        CPU_INT32U    sin6_scope_id;     /* Scope zone ix.                              */
};

typedef  struct  net_sock_addr_ipv6 {
    NET_SOCK_ADDR_FAMILY    AddrFamily;
    NET_PORT_NBR            Port;
    CPU_INT32U              FlowInfo;
    NET_IPv6_ADDR           Addr;
    CPU_INT32U              ScopeID;
} NET_SOCK_ADDR_IPv6;
```

A socket could configure the example socket address structure in the figure below to bind on IPv6 address FE80::1111:AAAA and port number 49876 with the following code:

Listing - Bind on 10.10.1.65

```
NET_SOCK_ADDR_IPv6   addr_local;
NET_IPv6_ADDR        addr_ip;
NET_PORT_NBR         addr_port;
NET_SOCK_RTN_CODE    rtn_code;
NET_ERR              err;

addr_ip   = NetASCII_Str_to_IPv6("FE80::1111:AAAA", &err);
addr_port = 49876;
Mem_Clr(&addr_local,
        sizeof(addr_local));
addr_local.AddrFamily = NET_SOCK_ADDR_FAMILY_IP_V6;                /\* = AF_INET6 \*/
Mem_Copy((void    \*)&addr_local-\>Addr,
        (void    \*)&addr_ip,
        (CPU_SIZE_T) sizeof(addr_local));
addr_local.Port       = NET_UTIL_HOST_TO_NET_16(addr_port);
rtn_code              = NetSock_Bind(                  sock_id,
                          (NET_SOCK_ADDR   \*)&addr_local,  /\* Cast to generic addr† \*/
                                         sizeof(addr_local),
                                        &err);
```

† The address of the specific IPv6 socket address structure is cast to a pointer to the generic socket address structure.

Complete send() Operation

send() returns the number of bytes actually sent out. This might be less than the number that are available to send. The function will send as much of the data as it can. The developer must make sure that the rest of the packet is sent later.

Listing - Completing a send()

```
{
    int  total     = 0;       /\* how many bytes we've sent     \*/
    int  bytesleft = \*len;   /\* how many we have left to send  \*/
    int  n;

    while (total \< \*len) {
        n = send(s, buf + total, bytesleft, 0);                    (1)
        if (n == -1) {
            break;
        }
        total += n;                                        (2)
        bytesleft -= n;                                    (3)
    }
}
```

(1) Send as many bytes as there are transmit network buffers available.

(2) Increase the number of bytes sent.

(3) Calculate how many bytes are left to send.

This is another example that, for a TCP/IP stack to operate smoothly, sufficient memory to define enough buffers for transmission and reception is a design decision that requires attention if optimum performance for the given hardware is desired.

IPv4 to IPv6

For a client migrating from IPv4 to IPv6 is basically just using the correct structure following the IP address version to use. For a server, if both IP address must be supported at same time, the application might require some modification, depending it's design since two listening socket will be required one for IPv4 and another for IPv6.

Loopback for Inter-Process Communication

It is possible for tasks to communicate with sockets via the localhost interface which must be enabled. See Network Core Configuration .

Micrium Native Socket Error Codes

When socket functions return error codes, the error codes should be inspected to determine if the error is a temporary, non-fault condition (such as no data to receive) or fatal (such as the socket has been closed).

Handling Socket Error Codes

Whenever any of the following *transitory error* codes are returned by any Network module's socket function,

```
RTOS_ERR_POOL_EMPTY
RTOS_ERR_NET_IF_LINK_DOWN
RTOS_ERR_TIMEOUT
RTOS_ERR_WOULD_BLOCK
```

The following action must be taken:

- If the socket is a blocking socket, the operation must be retried.
- If the socket is a non-blocking socket, the application must delay and then the operation must be retried.

Otherwise, if the error code is something other than

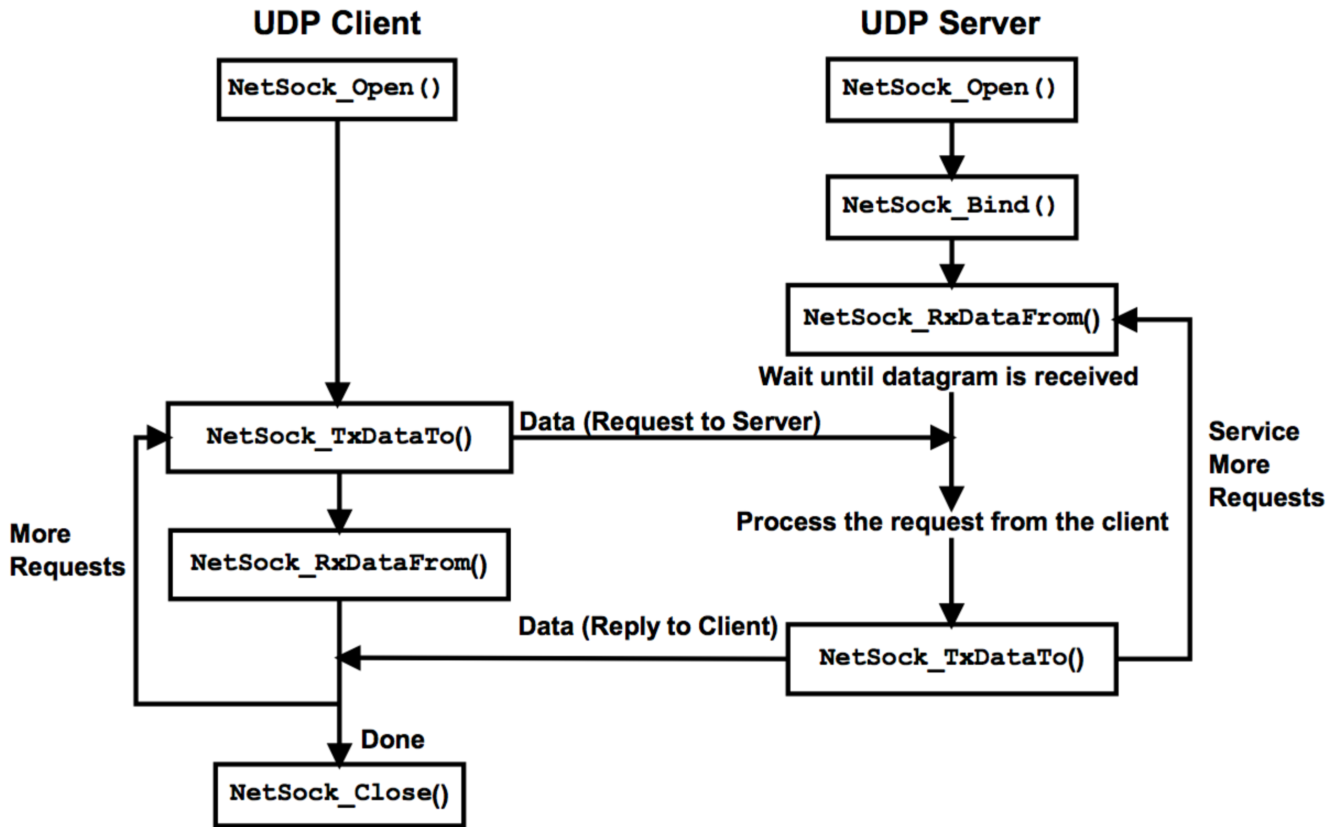
RTOS_ERR_NONE

Then that socket *must* be immediately closed() 'd without further access by any other socket functions:

Datagram Socket (UDP)

The figure below reproduces a diagram that introduces sample code using the typical socket functions for a UDP client-server application. The example uses the Micrium proprietary socket API function calls. A similar example could be written using the BSD socket API.

Figure - Socket calls used in a typical UDP client-server application



The code in the listing below implements a UDP server. It opens a socket and binds an IP address, listens and waits for a packet to arrive at the specified port.

Datagram Server (UDP Server)

Listing - Datagram Server

```
#include  <rtos/net/include/net_cfg_net.h>
#include  <rtos/net/include/net_type.h>
#include  <rtos/net/include/net_sock.h>
#include  <rtos/net/include/net_app.h>
#include  <rtos/net/include/net_util.h>

#ifdef  NET_IPv4_MODULE_EN
#include  <rtos/net/include/net_ipv4.h>
#endif

#ifdef  NET_IPv6_MODULE_EN
#include  <rtos/net/include/net_ipv6.h>
#endif

#define  UDP_SERVER_PORT  10001
#define  RX_BUF_SIZE        15

/*
*********************************************************************************************************
*                                   Ex_Net_SockUDP_ServerIPv4()
*
* Description : UDP Echo server:
*
*               (a) Open a socket.
*               (b) Configure socket's address.
*               (c) Bind that socket.
*               (d) Receive data on the socket.
*               (e) Transmit to source the data received.
*               (f) Close socket on fatal fault error.
*
* Argument(s) : None.
*
* Return(s)   : None.
*
* Note(s)     : None.
*********************************************************************************************************
*/
#ifdef  NET_IPv4_MODULE_EN
void  Ex_Net_SockUDP_ServerIPv4 (void)
{
    NET_SOCK_ID          sock;
    NET_SOCK_ADDR_IPv4   server_sock_addr_ip;
    NET_SOCK_ADDR_IPv4   client_sock_addr_ip;
    NET_SOCK_ADDR_LEN    client_sock_addr_ip_size;
    NET_SOCK_RTN_CODE    rx_size;
    NET_SOCK_RTN_CODE    tx_size;
    NET_SOCK_DATA_SIZE   tx_rem;
    CPU_CHAR             rx_buf[RX_BUF_SIZE];
    CPU_INT32U           addr_any = NET_IPv4_ADDR_ANY;
    CPU_INT08U          *p_buf;
    CPU_BOOLEAN          fault_err;
    RTOS_ERR             err;

                                                /* ---------------- OPEN IPV4 SOCKET ----------------- */
    sock = NetSock_Open(NET_SOCK_PROTOCOL_FAMILY_IP_V4,
                        NET_SOCK_TYPE_DATAGRAM,
                        NET_SOCK_PROTOCOL_UDP,
                       &err);
    if (err.Code != RTOS_ERR_NONE) {
        return;
    }

                                                /* ----------- CONFIGURE SOCKET'S ADDRESS ------------ */
    NetApp_SetSockAddr((NET_SOCK_ADDR *)&server_sock_addr_ip,
                                        NET_SOCK_ADDR_FAMILY_IP_V4,
                                        UDP_SERVER_PORT,
                       (CPU_INT08U    *)&addr_any,
```

```
                    NET_IPv4_ADDR_SIZE,&err);if(err.Code != RTOS_ERR_NONE){NetSock_Close(sock,&err);return;}/* ------------------ BIND
SOCKET -------------------- */NetSock_Bind(            sock,(NET_SOCK_ADDR *)&server_sock_addr_ip,
                    NET_SOCK_ADDR_SIZE,&err);if(err.Code != RTOS_ERR_NONE){NetSock_Close(sock,&err);return;}

  fault_err = DEF_NO;do{/* ----- WAIT UNTIL RECEIVING DATA FROM A CLIENT ----- */
    client_sock_addr_ip_size = sizeof(client_sock_addr_ip);

    rx_size = NetSock_RxDataFrom(              sock,
                                rx_buf,
                                RX_BUF_SIZE,
                                NET_SOCK_FLAG_NONE,(NET_SOCK_ADDR *)&client_sock_addr_ip,&client_sock_addr_ip_size,
                                DEF_NULL,
                                DEF_NULL,
                                DEF_NULL,&err);
    switch (err.Code){
      case RTOS_ERR_NONE:
        tx_rem = rx_size;
        p_buf  = (CPU_INT08U *)rx_buf;/* ---- TRANSMIT THE DATA RECEIVED TO THE CLIENT ----- */do{
          tx_size = NetSock_TxDataTo(              sock,
                                    p_buf,
                                    tx_rem,
                                    NET_SOCK_FLAG_NONE,(NET_SOCK_ADDR *)&client_sock_addr_ip,
                                    client_sock_addr_ip_size,&err);
          switch (err.Code){
            case RTOS_ERR_NONE:
              tx_rem -=  tx_size;
              p_buf   = (CPU_INT08U *)(p_buf + tx_size);break;

            case RTOS_ERR_POOL_EMPTY:
            case RTOS_ERR_NET_IF_LINK_DOWN:
            case RTOS_ERR_TIMEOUT:
            case RTOS_ERR_WOULD_BLOCK:OSTimeDlyHMSM(0,0,0,5, OS_OPT_TIME_DLY,&err);break;

            default:
              fault_err = DEF_YES;break;}}while(tx_rem >0);break;

      case RTOS_ERR_WOULD_BLOCK:
      case RTOS_ERR_TIMEOUT:break;

      default:
        fault_err = DEF_YES;break;}}while(fault_err == DEF_NO);/* ------------- FATAL FAULT SOCKET ERROR ------------
*/NetSock_Close(sock,&err);/* This function should be reached only when a fatal   *//* fault error has occurred.                    */}
#endif

/*
*********************************************************************************************************
*                              Ex_Net_SockUDP_ServerIPv6()
*
* Description : UDP Echo server:
*
*              (a) Open a socket.
*              (b) Configure socket's address.
*              (c) Bind that socket.
*              (d) Receive data on the socket.
*              (e) Transmit to source the data received.
*              (f) Close socket on fatal fault error.
*
* Argument(s) : None.
*
* Return(s)   : None.
*
* Note(s)     : None.
*********************************************************************************************************
*/
#ifdef  NET_IPv6_MODULE_EN
void  Ex_Net_SockUDP_ServerIPv6 (void){
  NET_SOCK_ID        sock;
```

```
    NET_SOCK_ADDR_IPv6   server_sock_addr_ip;
    NET_SOCK_ADDR_IPv6   client_sock_addr_ip;
    NET_SOCK_ADDR_LEN    client_sock_addr_ip_size;
    NET_SOCK_RTN_CODE    rx_size;
    NET_SOCK_RTN_CODE    tx_size;
    NET_SOCK_DATA_SIZE   tx_rem;
    CPU_CHAR             rx_buf[RX_BUF_SIZE];
    CPU_INT08U          *p_buf;
    CPU_BOOLEAN          fault_err;
    RTOS_ERR             err;/* ------------- OPEN IPV6 SOCKET ------------ */
    sock =NetSock_Open(NET_SOCK_PROTOCOL_FAMILY_IP_V6,/* IPv6 Socket family.                */
                       NET_SOCK_TYPE_DATAGRAM,/* Datagram socket.                     */
                       NET_SOCK_PROTOCOL_UDP,/* UDP protocol.                         */&err);if(err.Code != RTOS_ERR_NONE){return;}/* --------
CONFIGURE SOCKET'S ADDRESS ------- *//* Populate the NET_SOCK_ADDR_IP structure for *//* the server address and port, and convert    *//* it
to network order.                  */NetApp_SetSockAddr((NET_SOCK_ADDR *)&server_sock_addr_ip,
                              NET_SOCK_ADDR_FAMILY_IP_V6,
                              UDP_SERVER_PORT,(CPU_INT08U    *)&NET_IPv6_ADDR_ANY,
                              NET_IPv6_ADDR_SIZE,&err);if(err.Code != RTOS_ERR_NONE){NetSock_Close(sock,&err);return;}/* ---------------- BIND
SOCKET -------------- */NetSock_Bind(              sock,/* Bind the newly-created socket to the       */(NET_SOCK_ADDR *)&server_sock_addr_ip,/*
address and port specified by             */
                           NET_SOCK_ADDR_SIZE,/* server_sock_addr_ip.                 */&err);if(err.Code != RTOS_ERR_NONE)
{NetSock_Close(sock,&err);return;}

    fault_err = DEF_NO;do{/* - WAIT UNTIL RECEIVING DATA FROM A CLIENT - */
      client_sock_addr_ip_size =sizeof(client_sock_addr_ip);
      rx_size =NetSock_RxDataFrom(              sock,/* Receive data on port UDP_SERVER_PORT.      */(void         *)rx_buf,
                                  RX_BUF_SIZE,
                                  NET_SOCK_FLAG_NONE,(NET_SOCK_ADDR *)&client_sock_addr_ip,&client_sock_addr_ip_size,
                                  DEF_NULL,
                                  DEF_NULL,
                                  DEF_NULL,&err);
      switch (err.Code){
        case RTOS_ERR_NONE:
          tx_rem =  rx_size;
          p_buf  =(CPU_INT08U *)rx_buf;do{/* ------ TRANSMIT RECEIVED DATA TO THE CLIENT ------- *//* Transmit data to IP address and port of
the client. */
            tx_size =NetSock_TxDataTo(              sock,(void         *)p_buf,
                                      tx_rem,
                                      NET_SOCK_FLAG_NONE,(NET_SOCK_ADDR *)&client_sock_addr_ip,
                                      client_sock_addr_ip_size,&err);
            switch (err.Code){
              case RTOS_ERR_NONE:
                tx_rem -=  tx_size;
                p_buf   =(CPU_INT08U *)(p_buf + tx_size);break;

              case RTOS_ERR_POOL_EMPTY:
              case RTOS_ERR_NET_IF_LINK_DOWN:
              case RTOS_ERR_TIMEOUT:
              case RTOS_ERR_WOULD_BLOCK:OSTimeDlyHMSM(0,0,0,5, OS_OPT_TIME_DLY,&err);break;

              default:
                fault_err = DEF_YES;break;}}while(tx_rem >0);break;

        case RTOS_ERR_WOULD_BLOCK:
        case RTOS_ERR_TIMEOUT:break;

        default:
          fault_err = DEF_YES;break;}}while(fault_err == DEF_NO);/* ------------- FATAL FAULT SOCKET ERROR ------------
*/NetSock_Close(sock,&err);/* This function should be reached only when a fatal   *//* fault error has occurred.                    */}
#endif
```

Datagram Client (UDP Client)

The code in the listing below implements a UDP client. It sends a 'Hello World!' message to a server that listens on the UDP_SERVER_PORT.

Listing - Datagram Client

```
#define  UDP_SERVER_IP_ADDR  "192.168.1.100"
#define  UDP_SERVER_PORT            10001
#define  UDP_SERVER_TX_STR     "Hello World!"

CPU_BOOLEAN  TestUDPClient (void)
{
    NET_SOCK_ID         sock;
    NET_IP_ADDR         server_ip_addr;
    NET_SOCK_ADDR_IPv4  server_sock_addr_ip;
    NET_SOCK_ADDR_LEN   server_sock_addr_ip_size;
    CPU_CHAR           *pbuf;
    CPU_INT16S          buf_len;
    NET_SOCK_RTN_CODE   tx_size;
    RTOS_ERR            err;

    pbuf    = UDP_SERVER_TX_STR;
    buf_len = Str_Len(UDP_SERVER_TX_STR);

    sock = NetSock_Open( NET_SOCK_ADDR_FAMILY_IP_V4,                              (1)
                         NET_SOCK_TYPE_DATAGRAM,
                         NET_SOCK_PROTOCOL_UDP,
                        &err);
    if (err.Code != RTOS_ERR_NONE) {
        return (DEF_FAIL);
    }

    server_ip_addr = NetASCII_Str_to_IP(UDP_SERVER_IP_ADDR, &err);                (2)
    if (err.Code != RTOS_ERR_NONE) {
        NetSock_Close(sock, &err);
        return (DEF_FAIL);
    }

    server_sock_addr_ip_size = sizeof(server_sock_addr_ip);                       (3)
    Mem_Clr(&server_sock_addr_ip,
             server_sock_addr_ip_size);
    server_sock_addr_ip.AddrFamily = NET_SOCK_ADDR_FAMILY_IP_V4;
    server_sock_addr_ip.Addr       = NET_UTIL_HOST_TO_NET_32(server_ip_addr);
    server_sock_addr_ip.Port       = NET_UTIL_HOST_TO_NET_16(UDP_SERVER_PORT);

    tx_size = NetSock_TxDataTo(                 sock,                            (4)
                                                pbuf,
                                                buf_len,
                                                NET_SOCK_FLAG_NONE,
                               (NET_SOCK_ADDR *)&server_sock_addr_ip,
                                                sizeof(server_sock_addr_ip),
                                               &err);

    NetSock_Close(sock, &err);                                                    (5)
    if (err.Code != RTOS_ERR_NONE) {
        return (DEF_FAIL);
    }
    return (DEF_TRUE);
}
```

1. Open a datagram socket (UDP protocol).
2. Convert an IPv4 address from ASCII dotted-decimal notation to a network protocol IPv4 address in host-order.
3. Populate the NET_SOCK_ADDR_IP structure for the server address and port, and convert it to network order.
4. Transmit data to host DATAGRAM_SERVER_IP_ADDR on port DATAGRAM_SERVER_PORT.

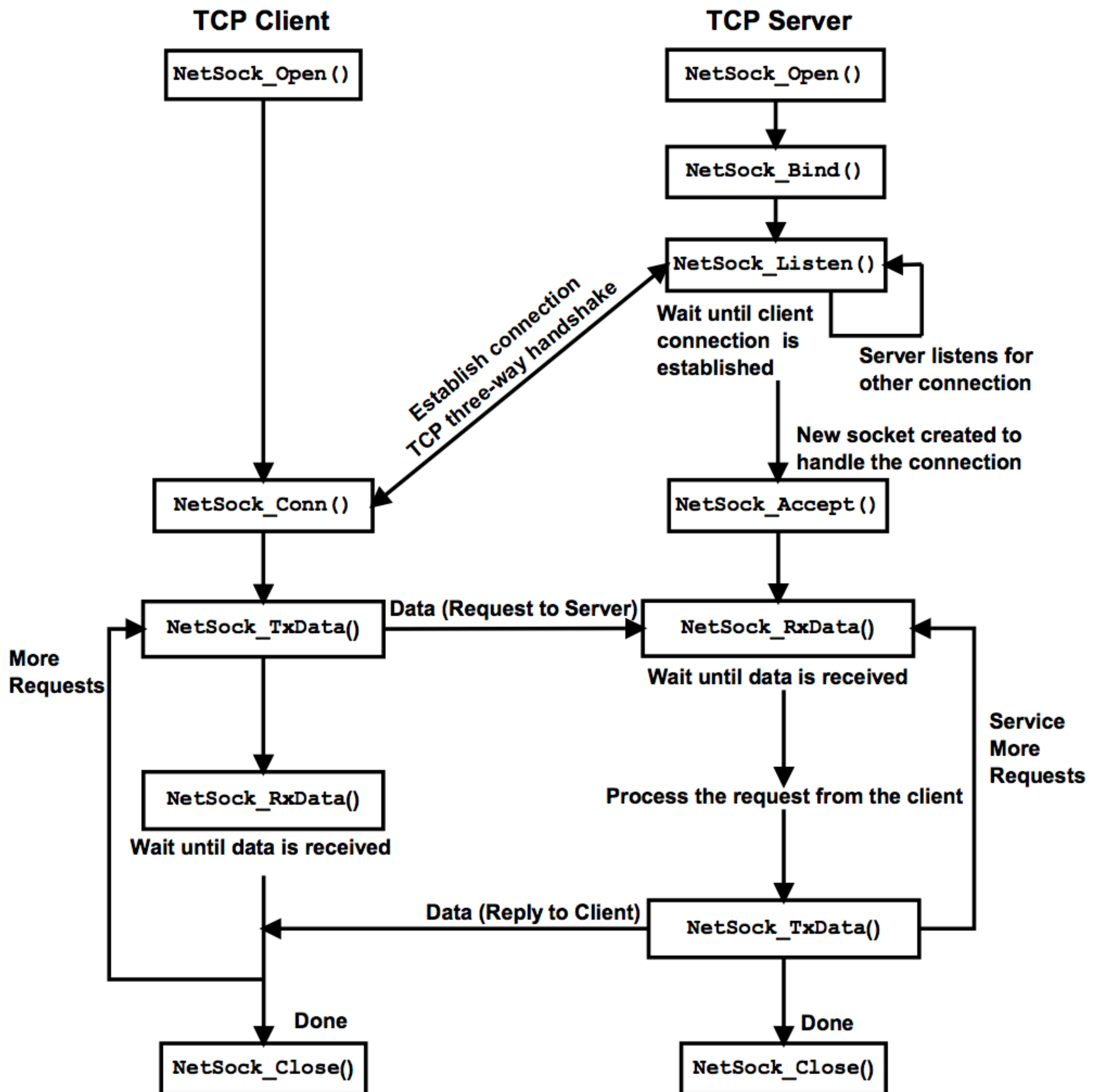
5. Close the socket.

Stream Socket (TCP)

The figure below uses the Micrium proprietary socket API function calls. A similar example could be written using the BSD socket API.

Typically, after a TCP server starts, TCP clients can connect and send requests to the server. A TCP server waits until client connections arrive and then creates a dedicated TCP socket connection to process the client's requests and reply back to the client (if necessary). This continues until either the client or the server closes the dedicated client-server connection. Also while handling multiple, simultaneous client-server connections, the TCP server can wait for new client-server connections

Figure - Socket calls used in a typical TCP client-server application



Stream Server (TCP Server)

This example presents a very basic client-server application over a TCP connection. The server presented is simply waits for a connection and send the string 'Hello World!'. See section Socket Functions API for a list of all socket API functions.

Listing - Stream Server

```
#define  TCP_SERVER_PORT                10000
#define  TCP_SERVER_CONN_Q_SIZE             1
#define  TCP_SERVER_TX_STR          "Hello World!"

CPU_BOOLEAN  TestTCPServer (void)
{
    NET_SOCK_ID          sock_listen;
    NET_SOCK_ID          sock_req;
    NET_SOCK_ADDR_IPv4  server_sock_addr_ip;
    NET_SOCK_ADDR_LEN   server_sock_addr_ip_size;
    NET_SOCK_ADDR_IPv4  client_sock_addr_ip;
    NET_SOCK_ADDR_LEN   client_sock_addr_ip_size;
    CPU_BOOLEAN          attempt_conn;
    CPU_CHAR            *pbuf;
    CPU_INT16S           buf_len;
    NET_SOCK_RTN_CODE   tx_size;
    RTOS_ERR             err;

    pbuf    = TCP_SERVER_TX_STR;
    buf_len = Str_Len(TCP_SERVER_TX_STR);

    sock_listen = NetSock_Open( NET_SOCK_ADDR_FAMILY_IP_V4,                    (1)
                                NET_SOCK_TYPE_STREAM,
                                NET_SOCK_PROTOCOL_TCP,
                               &err);
    if (err.Code != RTOS_ERR_NONE) {
        return (DEF_FALSE);
    }

    server_sock_addr_ip_size = sizeof(server_sock_addr_ip);                   (2)
    Mem_Clr(&server_sock_addr_ip,
             server_sock_addr_ip_size);
    server_sock_addr_ip.AddrFamily = NET_SOCK_ADDR_FAMILY_IP_V4;
    server_sock_addr_ip.Addr       = NET_UTIL_HOST_TO_NET_32(NET_SOCK_ADDR_IP_WILD_CARD);
    server_sock_addr_ip.Port       = NET_UTIL_HOST_TO_NET_16(TCP_SERVER_PORT);
    NetSock_Bind(                   sock_listen,                                (3)
                 (NET_SOCK_ADDR *)&server_sock_addr_ip,
                                  NET_SOCK_ADDR_SIZE,
                                 &err);
    if (err.Code != RTOS_ERR_NONE) {
        NetSock_Close(sock_listen, &err);
        return (DEF_FALSE);
    }

    NetSock_Listen( sock_listen,                                                (4)
                    TCP_SERVER_CONN_Q_SIZE,
                   &err);
    if (err.Code != RTOS_ERR_NONE) {
        NetSock_Close(sock_listen, &err);
        return (DEF_FALSE);
    }

    do {
        client_sock_addr_ip_size = sizeof(client_sock_addr_ip);

        sock_req = NetSock_Accept(                  sock_listen,                (5)
                                  (NET_SOCK_ADDR *)&client_sock_addr_ip,
                                                   &client_sock_addr_ip_size,
                                                   &err);
        switch (err.Code) {
            case RTOS_ERR_NONE:
                attempt_conn = DEF_NO;
                break;
                                                 /* Transitory Errors     */
        case RTOS_ERR_POOL_EMPTY:
            case RTOS_ERR_NET_IF_LINK_DOWN:
            case RTOS_ERR_TIMEOUT:
```

```
        case RTOS_ERR_WOULD_BLOCK:OSTimeDlyHMSM(0,0,0,5, OS_OPT_TIME_DLY,&err);break;
        default:
           attempt_conn = DEF_NO;break;}}while(attempt_conn == DEF_YES);if(err.Code != RTOS_ERR_NONE)
{NetSock_Close(sock_req,&err);return(DEF_FALSE);}

   tx_size =NetSock_TxData( sock_req,(6)
                    pbuf,
                    buf_len,
                    NET_SOCK_FLAG_NONE,&err);NetSock_Close(sock_req,&err);(7)NetSock_Close(sock_listen,&err);return(DEF_TRUE);}
```

1. Open a stream socket (TCP protocol).
2. Populate the NET_SOCK_ADDR_IP structure for the server address and port, and convert it to network order.
3. Bind the newly created socket to the address and port specified by server_sock_addr_ip.
4. Set the socket to listen for a connection request coming on the specified port.
5. Accept the incoming connection request, and return a new socket for this particular connection. Note that this function call is being called from inside a loop because it might timeout (no client attempts to connect to the server).
6. One the connection has been established between the server and a client, transmit the message. Note that the return value of this function is not used here, but a real application should make sure all the message has been sent by comparing that value with the length of the message.
7. Close both listen and request sockets. When the server need to stay active, the listen socket stays open so that I can accept additional connection requests. Usually, the server will wait for a connection, accept() it, and OSTaskCreate() a task to handle it.

Stream Client (TCP Client)

The client of the listing below connects to the specified server and receives the string the server sends.

Listing - Stream Client

```
#define  TCP_SERVER_IP_ADDR  "192.168.1.101"
#define  TCP_SERVER_PORT           10000
#define  RX_BUF_SIZE                  15

CPU_BOOLEAN  TestTCPClient (void)
{
    NET_SOCK_ID        sock;
    NET_IP_ADDR        server_ip_addr;
    NET_SOCK_ADDR_IPv4 server_sock_addr_ip;
    NET_SOCK_ADDR_LEN  server_sock_addr_ip_size;
    NET_SOCK_RTN_CODE  conn_rtn_code;
    NET_SOCK_RTN_CODE  rx_size;
    CPU_CHAR           rx_buf[RX_BUF_SIZE];
    RTOS_ERR           err;

    sock = NetSock_Open( NET_SOCK_ADDR_FAMILY_IP_V4,                        (1)
                         NET_SOCK_TYPE_STREAM,
                         NET_SOCK_PROTOCOL_TCP,
                        &err);
    if (err.Code != RTOS_ERR_NONE) {
        return (DEF_FAIL);
    }

    server_ip_addr = NetASCII_Str_to_IP(TCP_SERVER_IP_ADDR, &err);          (2)
    if (err.Code != RTOS_ERR_NONE) {
        NetSock_Close(sock, &err);
        return (DEF_FAIL);
    }

    server_sock_addr_ip_size = sizeof(server_sock_addr_ip);                 (3)
    Mem_Clr(&server_sock_addr_ip,
             server_sock_addr_ip_size);
    server_sock_addr_ip.AddrFamily = NET_SOCK_ADDR_FAMILY_IP_V4;
    server_sock_addr_ip.Addr       = NET_UTIL_HOST_TO_NET_32(server_ip_addr);
    server_sock_addr_ip.Port       = NET_UTIL_HOST_TO_NET_16(TCP_SERVER_PORT);
    conn_rtn_code = NetSock_Conn(                  sock,                    (4)
                                 (NET_SOCK_ADDR *)&server_sock_addr_ip,
                                  sizeof(server_sock_addr_ip),
                                 &err);
    if (err.Code != RTOS_ERR_NONE) {
        NetSock_Close(sock, &err);
        return (DEF_FAIL);
    }

    rx_size = NetSock_RxData( sock,                                         (5)
                              rx_buf,
                              RX_BUF_SIZE,
                              NET_SOCK_FLAG_NONE,
                             &err);
    if (err.Code != RTOS_ERR_NONE) {
        NetSock_Close(sock, &err);
        return (DEF_FAIL);
    }

    NetSock_Close(sock, &err);                                              (6)
    return (DEF_TRUE);
}
```

1. Open a stream socket (TCP protocol).
2. Convert an IPv4 address from ASCII dotted-decimal notation to a network protocol IPv4 address in host-order.
3. Populate the NET_SOCK_ADDR_IPv4 structure for the server address and port, and convert it to network order.
4. Connect the socket to a remote host.

5. Receive data from the connected socket. Note that the return value for this function is not used here.However, a real application should make sure everything has been received.
6. Close the socket.

TCP Connection Configuration

The Network module provides a set of APIs to configure TCP connections on an individual basis.

These APIs are listed below and detailed in TCP Functions :

- NetTCP_ConnCfgMaxSegSizeLocal()
- NetTCP_ConnCfgReTxMaxTh()
- NetTCP_ConnCfgReTxMaxTimeout()
- NetTCP_ConnCfgRxWinSize()
- NetTCP_ConnCfgTxWinSize()
- NetTCP_ConnCfgTxAckImmedRxdPushEn()
- NetTCP_ConnCfgTxNagleEn()
- NetTCP_ConnCfgTxKeepAliveEn()
- NetTCP_ConnCfgTxKeepAliveTh()
- NetTCP_ConnCfgTxAckDlyTimeout()

Socket Options

Specific Socket Option API

The Network module provides a set of APIs to configure sockets on an individual basis. These APIs are listed below and detailed in Network Socket Functions :

NetSock_CfgBlock()

NetSock_CfgSecure()

NetSock_CfgRxQ_Size()

NetSock_CfgTxQ_Size()

NetSock_CfgTxIP_TOS() (IPv4 only)

NetSock_CfgTxIP_TTL() (IPv4 only)

NetSock_CfgTxIP_TTL_Multicast() (IPv4 only)

NetSock_CfgTimeoutConnAcceptDflt()(TCP)

NetSock_CfgTimeoutConnAcceptGet_ms()

NetSock_CfgTimeoutConnAcceptSet()

NetSock_CfgTimeoutConnCloseDflt()

NetSock_CfgTimeoutConnCloseGet_ms()

NetSock_CfgTimeoutConnCloseSet()

NetSock_CfgTimeoutConnReqDflt()

NetSock_CfgTimeoutConnReqGet_ms()

NetSock_CfgTimeoutConnReqSet()

NetSock_CfgTimeoutRxQ_Dflt()

NetSock_CfgTimeoutRxQ_Get_ms()

NetSock_CfgTimeoutRxQ_Set()

NetSock_CfgTimeoutTxQ_Dflt()

NetSock_CfgTimeoutTxQ_Get_ms()

NetSock_CfgTimeoutTxQ_Set()

Generic Socket Option API

The Network module provides two APIs to read and configure socket option values. These APIs are listed below and detailed in Network Socket Functions :

> NetSock_OptGet()
>
> NetSock_OptSet()

Their BSD equivalent are listed below. See also BSD Functions .

> getsockopt() (TCP/UDP)
>
> setsockopt() (TCP/UDP)

Controlling Socket Blocking Options

By default all sockets are configured to block. It is possible to change that behavior by using the socket option API NetSock_CfgBlock() or when by setting a flag before calling Socket API. It is also possible to change the default configuration, see in net_cfg.h for further details.

MSL

Maximum Segment Lifetime (MSL) is the time a TCP segment can exist in the network, and is defined as two minutes. 2MSL is twice this lifetime. It is the maximum lifetime of a TCP segment on the network because it supposes segment transmission and acknowledgment.

Currently, Micrium does not support multiple sockets with identical connection information. This prevents new sockets from binding to the same local addresses as other sockets. Thus, for TCP sockets, each close() incurs the TCP 2MSL timeout and prevents the next bind() from the same client from occurring until after the timeout expires. This is why the 2MSL value is used. This can lead to a long delay before the socket resource is released and reused. The Network module configures the TCP connection's default maximum segment lifetime (MSL) timeout value, specified in integer seconds. A starting value of 3 seconds is recommended.

If TCP connections are established and closed rapidly, it is possible that this timeout may further delay new TCP connections from becoming available. Thus, an even lower timeout value may be desirable to free TCP connections and make them available for new connections as rapidly as possible. However, a 0 second timeout prevents the Network module from performing the complete TCP connection close sequence and will instead send TCP reset (RST) segments.

For UDP sockets, the sockets close() without delay. Thus, the next bind() is not blocked.

TCP Keep-Alives

The Network module support TCP Keep-Alives. RFC #1122 stipulate that "Keep-Alive mechanism periodically probes the other end of a connection when connection is otherwise idle, even when there is not data to be sent."

It is possible to enable the Keep-Alive option for a specific socket by using the function NetSock_OptSet() with the option NET_SOCK_OPT_SOCK_KEEP_ALIVE.

Knowing your TCP connection ID, you can also used the API function NetTCP_ConnCfgTxKeepAliveEn().

Each time the connection Idle timeout occurs, the stack will send a keep-alive message to probe the other end of the connection. The timeout value for a given TCP connection can be set with the NetSock_OptSet() function and the

NET_SOCK_OPT_TCP_KEEP_IDLE option. Else, knowing your TCP connection ID, the API function NetTCP_ConnCfgIdleTimeout() can also be use.

Secure Sockets (TLS or SSL)

- Server Sample
- Client Sample

If a network security module (such as Mocana - NanoSSL) is available, the Network module socket security option APIs can be used to secure sockets. The port layer developed for the network security layer is responsible for securing the sockets and applying the security strategy over typical socket programming functions. From an application point of view, the usage of the Network security manager is very simple. It requires a few simple steps depending on whether the application is a server or a client. Basically, it provides APIs to install the required keying material and to set the secure flag on a specific socket:

> NetSock_CfgSecure()
>
> NetSock_CfgSecureServerCertKeyInstall()
>
> NetSock_CfgSecureClientCertKeyInstall()
>
> NetSock_CfgSecureClientCommonName()
>
> NetSock_CfgSecureClientTrustCallBack()

The stack must have been configured to support Transport layer security in net_cfg.h, see Transport Layer Security Configuration . Obviously, TLS or SSL can be used only with a TCP connection. Once the socket is configured as secure and the connection is established, all the data transferred using standard socket API are automatically encrypted between the client and the server.

Server Sample

In order to achieve secure handshake connections, some keying material must be installed before performing any secure socket operation. The server needs to install a public key certificate / private key pair to send to the clients that want to connect. Certificate and key can be delivered by a Certificate Authority or can be generated using a tool such as OpenSSL.

The following example demonstrates how to secure a server using a PEM certificate from a constant buffer.

Listing - Server Sample

```
#include  <rtos/net/include/net_sock.h>
#include  <rtos/net/include/net_app.h>
#include  <rtos/net/include/net_util.h>
#include  <rtos/net/include/net_bsd.h>

#define APP_CFG_SECURE_CERT                                   \
"MIIEEjCCAvqgAwIBAgIBBzANBgkqhkiG9w0BAQUFADAaMRgwFgYDVQQDEw9WYWxp\
Y29yZS1EQzEtQ0EwHhcNMTEwMzE4MTcwMTQyWhcNMjEwMzE1MTcwMTQyWjCBkDEL\
MAkGA1UEBhMCVVMxCzAJBgNVBAgTAkNBMQ8wDQYDVQQHEwZJcnZpbmUxHjAcBgNV\
BAoTFVZhbGljb3JlIFRlY2hub2xvZ2llczEhMB8GA1UEAxMYbGFuLWZ3LTAxLnZh\
bGljb3JlLmxvY2FsMSAwHgYJKoZIhvcNAQkBFhFhZG1pbkBsb2NhbGRvbWFpbjCC\
ASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBALwGOahytiwshzz1s/ngxy1+\
+VrXZYjKSEzMYbJCUhK9xA5fz8pGtOZIXI+CasZPSbXv+ZDLGpSpeFnOL49pIYRs\
vmTxg2n3AlZbP6pD9OPU8rmufsTvXAmQGxxIkdmWiXYJk0pbj+U698me6DKMV/sy\
3ekQaQC2I2nr8uQw8RhuNhhlkWyjBWdXnS2mLNLSan2Jnt8rumtAi3B+vF5Vf0Fa\
kLJNt45R0f5jjuab+qw4PKMZEQbqe0XTNzkxdD0XNRBdKlajffoZPBJ7xkfuKUA3\
cMjXKzetABoKvsv+ElfvqIrl9RXvTXy52EaQmVhiOyBHrScq4RbwtDQsd59Qmk0C\
AwEAAaOB6zCB6DAJBgNVHRMEAjAAMBEGCWCGSAGG+EIBAQQEAwIGQDA0BglghkgB\
hvhCAQ0EJxYlRWFzeS1SU0EgR2VuZXJhdGVkIFNlcnZlciBDZXJ0aWZpY2F0ZTAd\
BgNVHQ4EFgQUrq5KF11M9rpKm75nAs+MaiK0niYwUQYDVR0jBEowSIAU2Q9eGjzS\
LZhvIRRKO6c4Q5ATtuChHqQcMBoxGDAWBgNVBAMTD1ZhbGljb3JlLURDMS1DQYIQ\
T9aBcT0uXoxJmC0ohp7oSTATBgNVHSUEDDAKBggrBgEFBQcDATALBgNVHQ8EBAMC\
BaAwDQYJKoZIhvcNAQEFBQADggEBAAUMm/9G+mhxVlYK4anc34FMqu88NQy8lrh0\
loNfHhlEKnerzMz+nQGidf+KBg5K5U2Jo8e9gVnrzz1gh2RtUFvDjgosGIrgYZMN\
yreNUD2I7sWtuWFQyEuewbs8h2MECs2xVktkqp5KPmJGCYGhXbi+zuqi/19cIsly\
yS01kmexwcFMXyX4YOVbG+JFHy1b4zFvWgSDULj14AuKfc8RiZNvMRMWR/Jqlpr5\
xWQRSmkjuzQMFavs7soZ+kHp9vnFtY2D6gF2cailk0sdG0uuyPBVxEJ2meifG6eb\
o3FQzdtIrB6oMFHEU00P38SJq+mrDItPDRXNLa2Nrtc1EJtmjws="

#define APP_CFG_SECURE_KEY                                   \
"MIIEogIBAAKCAQEAvAY5qHK2LCyHPPWz+eDHLX75WtdIiMpITMxhskJSEr3EDI/P\
yka05khcj4Jqxk9Jte/5kMsalKI4Wc4vj2mVhGy+ZPGDafcCVIs/qkP049Tyua5+\
xO9cCZAbHEiR2ZaJdgmTSluP5Tr3yZ7oMoxX+zLd6RBpALYjaevy5DDxGG42GGWR\
bKMFZ1edLaYs0tJqfYme3yu6a0CLcH68XIV/QVqQsk23jlHR/mOO5pv6rDg8oxkR\
Bup7RdM3OTF0PRc1EF0qVqN9+hk8EnvGR+4pQDdwyNcrN60AGgq+y/4SV++qWsj1\
Fe9NfLnYRpCZWGI7IEetJyrhFvC0NCx3n1CaTQIDAQABAoIBAEbbqbr7j//RwB2P\
EwZmWWmh4mMDrbYBVYHrvB2rtLZvYYVxQiOexenK92b15TtbAhJYn5qbkCbaPwrJ\
E09eoQRI3u+3vKigd/cHaFTIS2/Y/qhPRGL/OZY5Ap6EEsMHYkJjlWh+XRosQNlw\
01zJWxbFsq90ib3E5k+ypdStRQ7JQ9ntvDAP6MDp3DF2RYf22Tpr9t3Oi2mUirOl\
piOEB55wydSylhSHusbms3sp2uvQBYJjZP7eENEQz55PebTzI9UF2dgJ0wJFS073\
rvp46fibcch1L7U6v8iUNaS47GTs3MMyO4zda73ufhYwZLU5gL8oEDY3tf/J8zuC\
mNurr0ECgYEA8i1GgstYBFSCH4bhd2mLu39UVsIvHaD38mpJE6avCNOUq3Cyz9qr\
NzewG7RyqR43HsrVqUSQKzIAGWqG7sf+jkiam3v6VW0y05yqDjs+SVW+ZN5CKyn3\
sMZV0ei4MLrfxWneQaKy/EUTJMIz3rLSDM/hpJoA/gOo9BIFRf2HPkkCgYEAxsGq\
LYU+ZEKXKehVesh8rlic4QXwzeDmpMF2wTq6GnFq2D4vWPyVGDWdORcIO2BojDWV\
EZ8e7F2SghbmeTjXGADIdYXQiQyt4Wtm+oJ6d+/juKSrQ1HIPzn1qgXDNLPfjd9o\
9IX5IGIRn49Jrx/kKQAPTcnCa1lirlcsmcdiy+UCgYBEbOBwUi3zQ0Fk0QJhb/Po\
LSjSPpI7YKDN4JP3NnBcKRPngLc1HU6lElny6gA/ombmj17hLZsia1GeHMg1LVLS\
NtdgOR5ZBrqGqcwuqzSFGfHqpBXEBl6SludmoL9yHUreh3QhzWuO9aFcEoNnl9Tb\
g9z4Wf8Pxk71byYlSYLt6QKBgERActjo3ZD+UPyCHQBp4m45B246ZQO9zFYdXVNj\
gE7eTatuR0lOkoBawN++6gPByoUDTWpcsvjF9S6ZAJH2E97ZR/KAfijh4r/66sTx\
k26mQRPB8FHQvqv/kj3NdsgdUJJeeqPEyEzPkcjyloJxuB7gN2El/I5wCRon3Qf9\
sQ6FAoGAfVOaROSAtq/bq9JIL60kkhA9sr3KmX52PnOR2hW0caWi96j+2jlmPT93\
4A2LIVUo6hCsHLSCFoWWiyX9plqyYTn5L1EmeBO0+E8BH9F/te9+ZZ53U+quwc/X\
AZ6Pseyhj7S9wkI5hZ9SO1gcK4rWrAK/UFOIzzIACr5INr723vw="

#define  APP_CFG_SECURE_CERT_LEN     (sizeof(APP_CFG_SECURE_CERT) - 1)
#define  APP_CFG_SECURE_KEY_LEN     (sizeof(APP_CFG_SECURE_KEY)  - 1)

int  AppServerInit (void)
{
    NET_SOCK_ID         sock_id;
    NET_SOCK_ADDR_IPv4  addr_server_ip;
    NET_SOCK_ADDR_LEN   addr_len;
    RTOS_ERR            err;

                                                    /* -------------- OPEN SOCK ---------------- */
```

```
    sock_id =NetApp_SockOpen(NET_SOCK_ADDR_FAMILY_IP_V4,
                             NET_SOCK_TYPE_STREAM,
                             NET_SOCK_PROTOCOL_TCP,3,5,&err);
    switch (err.Code){
      case RTOS_ERR_NONE:break;
      default:return(-1);}/* ----------- CFG SOCK SECURE OPT ----------- */(void)NetSock_CfgSecure(sock_id,
                           DEF_YES,&err);
    switch (err){
      case RTOS_ERR_NONE:break;
      default:NetApp_SockClose(sock_id,1,&err);return(-1);}(void)NetSock_CfgSecureServerCertKeyInstall(sock_id,
                                          APP_CFG_SECURE_CERT,
                                          APP_CFG_SECURE_CERT_LEN,
                                          APP_CFG_SECURE_KEY,
                                          APP_CFG_SECURE_KEY_LEN,
                                          NET_SOCK_SECURE_CERT_KEY_FMT_PEM,
                                          DEF_NO,&err);
    switch (err.Code){
      case RTOS_ERR_NONE:break;
      default:NetApp_SockClose(sock_id,1,&err);return(-1);}/* ---------------- BIND SOCK ---------------- */
    addr_len =sizeof(addr_server_ip);Mem_Set((void *)&addr_server_ip,0u,
                 addr_len);

    addr_server_ip.AddrFamily = NET_SOCK_ADDR_FAMILY_IP_V4;
    addr_server_ip.Port       =NET_UTIL_HOST_TO_NET_16(12345);
    addr_server_ip.Addr       =NET_UTIL_HOST_TO_NET_32(NET_IPv4_ADDR_NONE);(void)NetApp_SockBind(              sock_id,(NET_SOCK_ADDR
 *)&addr_server_ip,
                                  addr_len,3,5,&err);

    switch (err.Code){
      case RTOS_ERR_NONE:break;
      default:NetApp_SockClose(sock_id,1,&err);return(-1);}/* --------------- LISTEN SOCK ---------------
 */(void)NetApp_SockListen(sock_id,1,&err);switch(err.Code){
      case RTOS_ERR_NONE:break;
      default:NetApp_SockClose(sock_id,1,&err);return(-1);}return(sock_id);}
```

Client Sample

In order to achieve secure handshake connections, some keying material must be installed before performing any secure socket operation. The client side needs to install certificate authorities to validate the identity of the public key certificate sent by the server side. Certificate authorities can be found on the web site of Certificate authorities who delivered the Server's certificate ad key. As for the server's certificate it's possible to generate the CA certificate when generating self-signed certificate-key pair using a tool such as OpenSSL.

Listing - Client Sample

```
#include  <rtos/net/include/net_secure_mocana.h>
#include  <rtos/net/include/net_sock.h>
#include  <rtos/net/include/net_app.h>
#include  <rtos/net/include/net_ascii.h>
#include  <rtos/net/include/net_util.h>
#include  <rtos/common/include/lib_str.h>

CPU_CHAR *p_cert =
            "MIIDVDCCAjygAwIBAgIDAjRWMA0GCSqGSIb3DQEBBQUAMEIxCzAJBgNVBAYTAlVT"
            "MRYwFAYDVQQKEw1HZW9UcnVzdCBJbmMuMRswGQYDVQQDExJHZW9UcnVzdCBHbG9i"
            "YWwgQ0EwHhcNMDIwNTIxMDQwMDAwWhcNMjIwNTIxMDQwMDAwWjBCMQswCQYDVQQG"
            "EwJVUzEWMBQGA1UEChMNR2VvVHJ1c3QgSW5jLjEbMBkGA1UEAxMSR2VvVHJ1c3Qg"
            "R2xvYmFsIENBMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA2swYYzD9"
            "9BcjGlZ+W988bDjkcbd4kdS8odhM+KhDtgPpTSEHCIjaWC9mOSm9BXiLnTjoBbdq"
            "fnGk5sRgprDvgOSJKA+eJdbtg/OtppHHmMlCGDUUna2YRpIuT8rxh0PBFpVXLVDv"
            "iS2Aelet8u5fa9IAjbkU+BQVNdnARqN7csiRv8lVK83Qlz6cJmTM386DGXHKTubU"
            "1XupGc1V3sjs0l44U+VcT4wt/lAjNvxm5suOpDkZALeVAjmRCw7+OC7RHQWa9k0+"
            "bw8HHa8sHo9gOeL6NlMTOdReJivbPagUvTLrGAMoUgRx5aszPeE4uwc2hGKceeoW"
            "MPRfwCvocWvk+QIDAQABo1MwUTAPBgNVHRMBAf8EBTADAQH/MB0GA1UdDgQWBBTA"
            "ephojYn7qwVkDBF9qn1luMrMTjAfBgNVHSMEGDAWgBTAephojYn7qwVkDBF9qn1l"
            "uMrMTjANBgkqhkiG9w0BAQUFAAOCAQEANeMpauUvXVSOKVCUn5kaFOSPeCpilKIn"
            "Z57QzxpeR+nBsqTP3UEaBU6bS+5Kb1VSsyShNwrrZHYqLizz/Tt1kL/6cdjHPTfS"
            "tQWVYrmm3ok9Nns4d0iXrKYgjy6myQzCsplFAMfOEVEiIuCl6rYVSAlk6l5PdPcF"
            "PseKUgzbFbS9bZvlxrFUaKnjaZC2mqUPuLk/IH2uSrW4nOQdtqvmlKXBx4Ot2/Un"
            "hw4EbNX/3aBd7YdStysVAq45pmp06drE57xNNB6pXE0zX5IJL4hmXXeXxx12E6nV"
            "5fEWCRE11azbJHFwLJhWC9kXtNHjUStedejV0NxPNO3CBWaAocvmMw==";

CPU_BOOLEAN  AppClientCertTrustCallBackFnct (void                          *p_cert_dn,
                                             NET_SOCK_SECURE_UNTRUSTED_REASON  reason);

int  AppClientConnect (void)
{
    NET_SOCK_ID        sock_id;
    NET_SOCK_ADDR_IPv4  addr_server;
    CPU_INT32U         len_addr_server;
    CPU_INT32U         len;
    RTOS_ERR           err;
                                                        /* ---------- INSTALL CA CERTIFICATE --------- */
    len = Str_Len(p_cert);
    NetSecure_CA_CertIntall(p_cert, len, NET_SOCK_SECURE_CERT_KEY_FMT_PEM, &err);

                                                        /* -------------- OPEN THE SOCKET ------------ */
    sock_id = NetApp_SockOpen(NET_SOCK_ADDR_FAMILY_IP_V4,
                              NET_SOCK_TYPE_STREAM,
                              NET_SOCK_PROTOCOL_TCP,
                              3,
                              5,
                              &err);
    switch (err.Code) {
        case RTOS_ERR_NONE:
             break;
        default:
             return (-1);
    }

                                                        /* ----------- CFG SOCK SECURE OPT------------ */
   (void)NetSock_CfgSecure(sock_id,
                           DEF_YES,
                          &err);
    switch (err.Code) {
        case RTOS_ERR_NONE:
             break;
        default:
             NetApp_SockClose(sock_id, 1, &err);
             return (-1);
    }
```

```
(void)NetSock_CfgSecureClientCommonName(sock_id,"domain_name.com",&err);
  switch (err.Code){
    case RTOS_ERR_NONE:break;
    default:NetApp_SockClose(sock_id,1,&err);return(-1);}
(void)NetSock_CfgSecureClientTrustCallBack(sock_id,&AppClientCertTrustCallBackFnct,&err);
  switch (err.Code){
    case RTOS_ERR_NONE:break;
    default:NetApp_SockClose(sock_id,1,&err);return(-1);}/* --------- ESTABLISH TCP CONNECTION -------- */Mem_Clr((void *)&addr_server,
NET_SOCK_ADDR_SIZE);

  addr_server.AddrFamily =NetASCII_Str_to_IP("98.139.211.125",(void *)&addr_server.Addr,sizeof(addr_server.Addr),&err);
  addr_server.Port      =NET_UTIL_HOST_TO_NET_16(12345);
  len_addr_server       =sizeof(addr_server);(void)NetApp_SockConn(            sock_id,(NET_SOCK_ADDR *)&addr_server,
                          len_addr_server,3,5,5,&err);
  switch (err.Code){
    case RTOS_ERR_NONE:break;
    default:NetApp_SockClose(sock_id,1,&err);return(-1);}return(sock_id);}

CPU_BOOLEAN  AppClientCertTrustCallBackFnct (void                     *p_cert_dn,
                          NET_SOCK_SECURE_UNTRUSTED_REASON  reason){return(DEF_YES);}
```

Multicast Programmimg

This section describes how to use and control multicast options such:

- Joining and Leaving a Multicast Group
- Transmitting/Receiving to/from a Multicast IP Group

Configuration

Some parameters should be configured and/or optimized for your project requirements. See the section Multicast Configuration for further details.

Using Network Interface Wireless Programming API

Wherever you want to configure or access an Interface option you should include this file:

Include file	Description
rtos/net/include/net_igmp.h	Functions used for IPv4 Multicast API
rtos/net/include/net_mldp.h	Functions used for IPv6 Multicast API

API reference

All Multicast APIs for IPv4 are presented in the section IGMP Functions and in section MLDP Functions for IPv6.

Function Name	Description
NetIGMP_HostGrpJoin()	Join a host group.
NetIGMP_HostGrpLeave()	Leave a host group.
NetMLDP_HostGrpJoin()	Join a MLDP Multicast host group.
NetMLDP_HostGrpLeave()	Leave a MLDP host group.

Joining and Leaving a Multicast Group

IPv4 Multicasting

The Network module supports IPv4 multicasting with IGMP. In order to receive packets addressed to a given IPv4 multicast group address, the stack must have been configured to support multicasting in net_cfg.h, Multicast Configuration , and that host group has to be joined.

The following examples show how to join and leave an IPv4 multicast group:

Listing - Joining and leaving an IPv4 multicast group

```
NET_IF_NBR   if_nbr;
NET_IP_ADDR  group_ip_addr;
RTOS_ERR     err;

if_nbr        = NET_IF_NBR_BASE_CFGD;
group_ip_addr = NetASCII_Str_to_IP("233.0.0.1", &err);
if (err.Code != RTOS_ERR_NONE) {
  /* Handle error. */
}
NetIGMP_HostGrpJoin(if_nbr, group_ip_addr, &err);
if (err.Code != RTOS_ERR_NONE) {
  /* Handle error. */
}
[...]
NetIGMP_HostGrpLeave(if_nbr, group_ip_addr, &err);
if (err.Code != RTOS_ERR_NONE) {
  /* Handle error. */
}
```

Refer to the functions NetIGMP_HostGrpJoin() and NetIGMP_HostGrpLeave() for more details.

IPv6 Multicasting

The Network module supports IPv6 multicasting with MLDP. In order to receive packets addressed to a given IPv6 multicast group address, With IPv6 MLDP is always enabled but you might need to increase the number of group, see Multicast Configuration .

The following examples show how to join and leave an IPv6 multicast group:

Listing - Joining and leaving an IPv6 multicast group

```
NET_IF_NBR   if_nbr;
NET_IP_ADDR  group_ip_addr;
RTOS_ERR     err;

if_nbr        = NET_IF_NBR_BASE_CFGD;
group_ip_addr = NetASCII_Str_to_IP("FF03::1", &err);
if (err.Code != RTOS_ERR_NONE) {
  /* Handle error. */
}
NetMLDP_HostGrpJoin(if_nbr, group_ip_addr, &err);
if (err.Code != RTOS_ERR_NONE) {
  /* Handle error. */
}
[...]
NetMLDP_HostGrpLeave(if_nbr, group_ip_addr, &err);
if (err.Code != RTOS_ERR_NONE) {
  /* Handle error. */
}
```

Refer to the functions NetMLDP_HostGrpJoin() and NetMLDP_HostGrpLeave() for more details.

Transmitting/Receiving to/from a Multicast IP Group

Transmitting to a Multicast IP Group Address

Transmitting to an IP multicast group is identical to transmitting to a unicast or broadcast address. However, when using only IPv4, the stack must be configured to enable multicast transmit, see Multicast Configuration .

Receiving from a Multicast IP Group

An IP multicast group must be joined before packets can be received from it from it (see Joining and Leaving a Multicast Group for more information). Once this is done, receiving from a multicast group only requires a socket bound to the NET_SOCK_ADDR_IP_WILDCARD address, as shown in the following example:

Listing - Receiving from a multicast group

```
NET_SOCK_ID       sock;
NET_SOCK_ADDR_IPv4 sock_addr_ip;
NET_SOCK_ADDR     addr_remote;
NET_SOCK_ADDR_LEN  addr_remote_len;
CPU_CHAR          rx_buf[100];
CPU_INT16U        rx_len;
RTOS_ERR          err;

sock = NetSock_Open( NET_SOCK_ADDR_FAMILY_IP_V4,
                     NET_SOCK_TYPE_DATAGRAM,
                     NET_SOCK_PROTOCOL_UDP,
                    &err);
if (err.Code != RTOS_ERR_NONE) {
  /* Handle error. */
}
Mem_Set((void)&sock_addr_ip, 0, sizeof(sock_addr_ip));
sock_addr_ip.AddrFamily = NET_SOCK_ADDR_FAMILY_IP_V4;
sock_addr_ip.Addr       = NET_UTIL_HOST_TO_NET_32(NET_SOCK_ADDR_IP_WILDCARD);
sock_addr_ip.Port       = NET_UTIL_HOST_TO_NET_16(10000);
NetSock_Bind(                sock,
             (NET_SOCK_ADDR   *)&sock_addr_ip,
                              NET_SOCK_ADDR_SIZE,
                             &err);
if (err.Code != RTOS_ERR_NONE) {
  /* Handle error. */
}

rx_len = NetSock_RxDataFrom( sock,
                            &rx_buf [0],
                             BUF_SIZE,
                             NET_SOCK_FLAG_NONE,
                            &addr_remote,
                            &addr_remote_len,
                             0,
                             0,
                             0,
                            &err);
```

Wireless Programming

This section describe how to use and control Wireless options such:

- Creating a Wireless Access Point
- Scanning for a Wireless Access Point
- Joining A Wireless Access Point
- Leaving Wireless Access Point

Using Network Interface Wireless Programming API

Wherever you want to configure or access an Interface option you should include this file:

Include file	Description
rtos/net/include/net_if_wifi.h	Functions used for Interface Wireless API

API Reference

All Interface APIs are presented in the section Wireless Network Interface Functions .

Function Name	Description
NetIF_WiFi_Add()	Add & initialize a specific instance of a network WiFi interface.
NetIF_WiFi_Start()	Start a WiFi type interface.
NetIF_WiFi_Scan()	Scan available wireless access point.
NetIF_WiFi_Join()	Join an wireless access point.
NetIF_WiFi_Leave()	Leave the access point previously joined.
NetIF_WiFi_CreateAP()	Creates a wireless access point.
NetIF_WiFi_GetPeerInfo()	Gets the peer info connected to the access point (when acting as an access point).

Creating a Wireless Access Point

To turn your device into a wireless Access Point allowing other wireless devices to connect to it, you need to use the function NetIF_WiFi_CreateAP() .

A call to NetIF_WiFi_CreateAP() is shown below :

Listing - Call to NetIF_WiFi_CreateAdhoc()

```
RTOS_ERR  err;

ap_ctn = NetIF_WiFi_CreateAP(if_nbr,                         (1)
                             NET_IF_WIFI_NET_TYPE_INFRASTRUCTURE  (2)
                             NET_IF_WIFI_DATA_RATE_AUTO,         (3)
                             NET_IF_WIFI_SECURITY_WEP,           (4)
                             NET_IF_WIFI_PWR_LEVEL_HI,           (5)
                             NET_IF_WIFI_CH_1                    (6)
                            "ssid",                              (7)
                            "password",                          (8)
                            &err);                               (9)
```

NetIF_WiFi_CreateAP() requires nine arguments.

1. The interface number, which is acquired upon successfully adding and starting the interface.
2. The network type: ad hoc or infrastructure
3. The data rate used on the wireless network.
4. The wireless security type of the wireless network.
5. The radio power level used to communicate on the wireless network.
6. The wireless channel for the ad hoc network.
7. A pointer to a string that contains the SSID of the wireless access point.
8. A pointer to a string that contains the pre-shared key of the wireless access point.
9. A pointer to a RTOS_ERR that contains the return error code. The return error variable will contain the value RTOS_ERR_NONE if the create process has been completed successfully.

If an error occurs, you should always inspect the return error code and take the appropriate action.

Scanning for a Wireless Access Point

When a wireless network interface is started, it becomes an active interface that is not yet capable of transmitting and receiving data since no operational link to a network medium is configured. The first (but optional) step to joining a network to have an operational link is the *scan* operation which consists of finding the wireless networks available in the range of the wireless module.

A wireless network interface should be able to scan any time after the network interface has been successfully started. A successful call to NetIF_WiFi_Scan() returns the wireless networks available to join which can be joined by the wireless network interface. See NetIF_WiFi_Scan() for more information.

You can scan for a wireless network by calling the NetIF_WiFi_Scan() API function with the necessary parameters. A call to NetIF_WiFi_Scan() is shown below.

Listing - Calling NetIF_Start()

```
NET_IF_WIFI_AP  ap_buf[NB_AP_MAX]
CPU_INT16U      ap_ctn;
RTOS_ERR        err;

ap_ctn = NetIF_WiFi_Scan(if_nbr,           (1)
                    ap_buf,           (2)
                    NB_AP_MAX,        (3)
                    0,                (4)
                    NET_IF_WIFI_CH_ALL, (5)
                   &err);             (6)
```

1. NetIF_WiFi_Scan() requires six arguments. The first function argument is the interface number that the application wants to scan with. The interface number is acquired upon successful addition of the interface and upon the successful start of the interface.
2. The second argument is a pointer to a wireless access point buffer that contains the wireless network found in the range of the interface.
3. The third argument is the number of wireless access points that can be contained in the wireless access point buffer.
4. The fourth argument is a pointer to a string that contains the SSID of any hidden wireless access points to find.
5. The fifth argument is the wireless channel to scan.
6. The last argument is a pointer to a RTOS_ERR to contain the return error code. The return error variable will contain the value RTOS_ERR_NONE if the scan process has been completed successfully.

There are very few things that could cause a network interface to not scan properly. The application developer should always inspect the return error code and take the appropriate action if an error occurs. Once the error is resolved, the application may again attempt to call NetIF_WiFi_Scan().

Joining A Wireless Access Point

When a wireless network interface is started, it becomes an active interface that is not yet capable of transmitting and receiving data, since no operational link to a network medium is configured. Once the interface has found a wireless network, it must join it to get an operational link. A wireless network interface should be able to join any time after the network interface has been successfully started and before having joined any other wireless access points. See NetIF_WiFi_Join() for more information.

The application developer may join a wireless network by calling the NetIF_WiFi_Join() API function with the necessary parameters. A call to NetIF_WiFi_Join() is shown below.

Listing - Calling NetIF_Start()

```
RTOS_ERR  err;

ap_ctn = NetIF_WiFi_Join(if_nbr,                         (1)
                    NET_IF_WIFI_NET_TYPE_INFRASTRUCTURE, (2)
                    NET_IF_WIFI_DATA_RATE_AUTO,          (3)
                    NET_IF_WIFI_SECURITY_WPA2,           (4)
                    NET_IF_WIFI_PWR_LEVEL_HI,            (5)
                   "network_ssid",                       (6)
                   "network_password",                   (7)
                   &err);                                (8)
```

1. NetIF_WiFi_Join() requires height arguments. The first function argument is the interface number that the application wants to join with. The interface number is acquired upon successfully adding and starting the interface.
2. The second argument is the wireless network type.
3. The third argument is the data rate used to communicate on the wireless network.
4. The fourth argument is the wireless security configured for the wireless network to join.
5. The fifth argument is the wireless radio power level used to communicate on the wireless network.
6. The sixth argument is a pointer to a string that contains the SSID of the wireless access point to join.
7. The seventh argument is a pointer to a string that contains the pre-shared key of the wireless access point to join.
8. The last argument is a pointer to a RTOS_ERR that contains the return error code. The return error variable will contain the value RTOS_ERR_NONE if the join process has been completed successfully.

There are very few things that could cause a network interface to not join properly. The application developer should always inspect the return error code and take the appropriate action if an error occurs. Once the error is resolved, the application may again attempt to call NetIF_WiFi_Join().

Leaving Wireless Access Point

When an application needs to leave a wireless access point, it can do so by calling the NetIF_WiFi_Leave() API function with the necessary parameters.

A call to NetIF_WiFi_Leave() is shown below.

Listing - Call to NetIF_WiFi_Leave()

```
RTOS_ERR  err;

ap_ctn = NetIF_WiFi_Leave(if_nbr, (1)
                &err);   (2)
```

1. NetIF_WiFi_Leave() requires two arguments. The first function argument is the interface number. The interface number is acquired upon successful addition of the interface and upon the successful start of the interface.
2. The last argument is a pointer to a RTOS_ERR to contain the return error code. The return error variable will contain the value RTOS_ERR_NONE if the leave process has been completed successfully.

There are very few things that could cause a network interface to leave improperly. You should always inspect the return error code and take the appropriate action if an error occurs. Once the error is resolved, the application may again attempt to call NetIF_WiFi_Leave().

Network Tasks Programming

Core Task

The network core task is the main task implementing the TCP/IP core module.

The task is waiting for a signal from the ISR indicating a frame was received. The task will take care of going through each layer from the Link layer to the Transport layer and remove the encapsulation, retrieve the necessary information and take actions to comply with protocol's RFCs.

If data was received that is intended for a socket application, the task will signal that data has been received and is ready to be processed on the socket.

This task, when awoken, will also take care of updating the network timers list and call the callback functions of those that have timed out.

Core Services Task

A separate task exists for some of the core network services supported by the Network module, like DHCP and DNS client.

Custom Network Application Task

A customer could develop his own socket application task to transmit and receive data. Since the transmission operations will be handled by this task, it will require a certain amount of stack to create the packet and do the encapsulation for all the layers from the Transport to the Link layer.

Native Network Applications

Micrium OS Network module offers many network applications. Some applications may have their own task while others will be executed in the context of the task calling them. See the specific documentation of each network application for more details.

Network Core Hardware Porting Guide

Network Core Hardware Porting Guide

To work properly, Micrium OS Network module requires a hardware driver that is implemented for any specific Ethernet/WiFi controller IP.

Each network controller that your project will use requires a Board Support Package (BSP). The BSP has two purposes:

- It initializes and configures any resources needed by the network controller, but which are provided by external modules. These include the clock, GPIO, interrupts, etc.
- It provides hardware information to the network controller driver.

Micrium provides example BSPs for several popular platforms. If one is available for your platform, we recommend that you use it as a starting point. However, if no example BSP is available for your platform, the information in this section will help you understand how to create your own BSP and port the network module to your platform.

- Network Driver Selection Guide
- Network BSP Functions Guide
- Network Hardware Information
- Network Controller Registration to the Platform Manager

Network Driver Selection Guide

Network Interface Types

The Network module supports two types of network interfaces: Ethernet and Wi-Fi.

Ethernet

Typically, MCUs with network capabilities will have an internal Ethernet MAC controller, usually with DMA support. Therefore, only an external PHY controller is needed to implement an Ethernet network interface.

An MCU can have multiple Ethernet controllers, and so depending on the MCU's controllers, multiple drivers could be required (one per controller implementation). For example, if multiple, similar Ethernet controllers are present but with different base addresses, the driver will be the same but with individual configurations.

WiFi

The Micrium OS network module offers drivers for some external Wi-Fi controllers.

The network stack sends/receives commands and IEEE 802.4 type frames via a serial interface (e.g., SPI) between the MCU and the external Wi-Fi controller. Therefore, it is assumed that the external controller implements the IEEE 802.11 standard and takes care of the encryption/decryption of the wireless data.

Network BSP Functions Guide

The Board Support Package contains a set of functions that support your hardware platform on Micrium OS. These functions are called by the network controller driver and are meant to perform hardware configuration or initialization of I/O pins, interrupts, etc. It is your responsibility to either create these functions from scratch, or to modify example code to fit your needs and the specifics of your hardware.

A network driver knows the detail of a specific controller IP, such as the registers that it contains and how to properly read and write to them. But what a driver doesn't know is the context in which the controller exists in your hardware solution: what are the I/O and interrupt pins, clock source, etc. The purpose of the BSP is to provide a layer of abstraction between the driver and the context of the controller in your hardware. The functions of a BSP are what is subject to customization,

based on how the controller is wired to your board. Each network controller you intend to use will need its own BSP. Each BSP defines the same set of functions that the drivers will call.

Each of these functions is described below according to the network controller type.

- Ethernet Controller
  - Clock Configuration
  - Interrupt Control Configuration
  - GPIO Configuration
  - Get Clock Frequency
- Wi-Fi Controller
  - Start
  - Stop
  - GPIO Configuration
  - Interrupt Configuration
  - Interrupt Control
- ISR Handling

Ethernet Controller

Clock Configuration

The clock configuration function is called by the driver during network controller initialization. Listing - Clock Configuration Function Signature in the *Network BSP Functions Guide* page below shows the signature of this required BSP function.

Listing - Clock Configuration Function Signature

```
void  BSP_NetEther_ClkCfg (NET_IF    *p_if,
                           RTOS_ERR  *p_err);
```

Its purpose is to set the clock(s) for a specific network controller.

This function must configure and enable all clocks required by the controller in order for it to work properly. For example, on some controllers it may be necessary to enable clock gating for an Ethernet MAC, as well as various GPIO modules in order to configure Ethernet PHY pins for (R)MII mode and interrupts.

Interrupt Control Configuration

The interrupt control configuration function is called by the driver during network controller initialization. Listing - Interrupt Configuration Function Signature in the *Network BSP Functions Guide* page below shows the signature for this required BSP function.

Listing - Interrupt Configuration Function Signature

```
void  BSP_NetEther_CfgIntCtrl  (NET_IF    *p_if,
                                RTOS_ERR  *p_err);
```

Its purpose is to configure and enable all required interrupt sources for the network controller.

In addition, the function should store the network interface number to be assigned to the controller. This information is available inside the p_if argument passed to the function. It will be needed inside the controller's interrupt service routine to indicate to the TCP/IP stack which network interface the interrupt is associated to (see ISR Handling).

The function should enable only the interrupt sources for each controller, but not the local controller-level interrupts themselves, which are enabled by the driver only after the network controller has been fully configured and started.

GPIO Configuration

The GPIO configuration function is called by the driver during network controller initialization. Listing - GPIO Configuration Function Signature in the *Network BSP Functions Guide* page below shows the signature for this required BSP function.

Listing - GPIO Configuration Function Signature

```
void  BSP_NetEther_CfgGPIO  (NET_IF    *p_if,
                             RTOS_ERR  *p_err);
```

Its purpose is to configure the general-purpose input/output (GPIO) pins for the network controller.

This function should configure all GPIO pins required for the network controller in order for it to work properly. For Ethernet controllers, this function is necessary to configure the (R)MII bus pins, depending on whether the user has configured an Ethernet interface to operate in the RMII or MII mode, and optionally the Ethernet PHY interrupt pin.

Get Clock Frequency

The Get Clock Frequency function is called by the driver during network controller initialization. Listing - Get Clock Frequency Function Signature in the *Network BSP Functions Guide* page below shows the signature for this required BSP function.

Listing - Get Clock Frequency Function Signature

```
CPU_INT32U  BSP_NetEther_ClkFreqGet  (NET_IF    *p_if,
                                      RTOS_ERR  *p_err);
```

Its purpose is to return the network controller clock frequency (in Hz).

For Ethernet controllers, this is the clock frequency of the controller's (R)MII bus. The controller driver initialization process uses the returned clock frequency to configure an appropriate bus divider to ensure that the (R)MII bus logic operates within an allowable range. In general, the controller driver should not configure the divider such that the (R)MII bus operates faster than 2.5MHz.

Wi-Fi Controller

Start

The BSP start function is called by the Wi-Fi controller driver each time the Wi-Fi network interface is started. Listing - WiFi Start Function Signature in the *Network BSP Functions Guide* page below shows the signature for this required BSP function.

Listing - WiFi Start Function Signature

```
void BSP_NetWiFi_Start (NET_IF    *p_if,
                        RTOS_ERR  *p_err);
```

Its purpose is to power up the wireless chip. Therefore, the function must set the required GPIO pins to signal chip power up and reset.

Note that some wireless devices could require toggling the reset pin twice to be started or restarted correctly.

Stop

The BSP stop function is called by the Wi-Fi controller driver each time the Wi-Fi network interface is stopped. Listing - WiFi Stop Function Signature in the *Network BSP Functions Guide* page below shows the signature for this required BSP function.

Listing - WiFi Stop Function Signature

```
void BSP_NetWiFi_Stop (NET_IF    *p_if,
                RTOS_ERR  *p_err);
```

Its purpose is to power down the wireless chip. Therefore, the function must set the required GPIO pins for the chip power down to reduce the power consumption.

GPIO Configuration

The GPIO configuration function is called by the driver during wireless network controller initialization. Listing - WiFi GPIO Configuration Function Signature in the *Network BSP Functions Guide* page below shows the signature for this required BSP function.

Listing - WiFi GPIO Configuration Function Signature

```
void BSP_NetWiFi_CfgGPIO (NET_IF    *p_if,
                  RTOS_ERR  *p_err);
```

Its purpose is to configure the general-purpose input/output (GPIO) pins for the network controller. For wireless devices, this function is necessary to configure the power, reset and interrupt pins.

Interrupt Configuration

The interrupt configuration function is called by the driver during wireless network controller initialization. Listing - WiFi Interrupt Configuration Function Signature in the *Network BSP Functions Guide* page below shows the signature for this required BSP function.

Listing - WiFi Interrupt Configuration Function Signature

```
void BSP_NetWiFi_CfgIntCtrl (NET_IF    *p_if,
                     RTOS_ERR  *p_err);
```

Its main purpose is to configure and enable all required interrupt sources for the wireless network controller.

In addition, the function should store the network interface number to be assigned to the wireless controller. This information is available inside the p_if argument passed to the function. It will be needed inside the controller's interrupt service routine to indicate to the TCP/IP stack which network interface the interrupt is associated to (see ISR Handling).

The function should enable only each controller's interrupt sources, but not the local controller-level interrupts themselves, which are enabled by the driver only after the network controller has been fully configured and started.

Interrupt Control

The interrupt control function is called by the wireless controller driver while the controller is running. Listing - WiFi Interrupt Controller Function Signature in the *Network BSP Functions Guide* page below shows the signature for this required BSP function.

Listing - WiFi Interrupt Controller Function Signature

```
void BSP_NetWiFi_IntCtrl (NET_IF       *p_if,
                  CPU_BOOLEAN   en,
                  RTOS_ERR     *p_err);
```

Its purpose is to enable or disable all external interrupt sources for the wireless chip. This means the function will enable or disable its corresponding interrupt source based on the *enable* argument received.

ISR Handling

Each network controller driver has an ISR handler function that must be called each time a network controller interrupt is triggered. However, for most platforms, it will be necessary to implement an intermediate ISR handler in the BSP. This is

necessary, as some interrupt controllers may require the interrupt status to be cleared each time it is triggered. It is also necessary if your interrupt controller does not support passing an argument to the interrupt vector, as the driver's ISR handler needs to know the network controller/network interface associated with the interrupt.

This BSP ISR will call the function NetIF_ISR_Handler() and pass to it, among others, the network interface number that was previously saved during the BSP Interrupt configuration. It's the NetIF_ISR_Handler() function that will take care of calling the controller driver's ISR handler.

In most cases, each controller requires only a single BSP ISR function. This is possible when the controller's driver is able to determine the interrupt type via internal controller registers or the interrupt controller. In this case, NetIF_ISR_Handler() must be called with interrupt type code NET_DEV_ISR_TYPE_UNKNOWN. However, some controllers cannot determine the interrupt type when an interrupt occurs and may therefore require multiple, unique BSP ISR function's, each of which calls NetIF_ISR_Handler() with the appropriate interrupt type code.

Ethernet physical layer (PHY) interrupts should call NetIF_ISR_Handler() with interrupt type code NET_DEV_ISR_TYPE_PHY.

If using a Silicon Labs chip (that follows the CMSIS naming standard), the ISR Handler should be named ETH_IRQHandler().

Listing - ISR Handling Function Signature

```
void ETH_IRQHandler (void)
{
  RTOS_ERR err;

  /* TODO: Clear interrupt status, if needed. */

  NetIF_ISR_Handler(BSP_Net_<ctrlr>_IF_Nbr, NET_DEV_ISR_TYPE_UNKNOWN, &err);
}
```

Network Hardware Information

- Ethernet Interface
  - BSP API Structure
  - MAC Controller Configuration and Information
  - PHY Controller Configuration and Information
  - Hardware Information
- Wireless Interface
  - WiFi Part Information
  - BSP API Structure

Ethernet Interface

BSP API Structure

In order to provide a pointer to the BSP functions for the Ethernet controller driver, you must create a structure of type NET*DEV_BSP_ETHER. Each network controller you intend to use will need is own BSP API structure. You must define this in the network controller's BSP file (bsp_net_ether*&&&.c).

Table - NET_DEV_BSP_ETHER Structure in the *Network Hardware Information* page describes each field available in this structure.

Table - NET_DEV_BSP_ETHER Structure

Field	Description
.CfgClk	Pointer to the BSP clock configuration function.
.CfgIntCtrl	Pointer to the BSP Interrupt configuration function.
.CfgGPIO	Pointer to the BSP GPIO configuration function.
.ClkFreqGet	Pointer to the BSP get clock frequency function.

Listing - Example of BSP API Structure in the *Network Hardware Information* page shows an example of a BSP API structure.

Listing - Example of BSP API Structure

```
static  NET_DEV_BSP_ETHER  BSP_NetEther_<ctrlr>_API_Ptr = {
   .CfgClk      = BSP_NetEther_ClkCfg,
   .CfgIntCtrl = BSP_NetEther_CfgIntCtrl,
   .CfgGPIO     = BSP_NetEther_CfgGPIO,
   .ClkFreqGet = BSP_NetEther_ClkFreqGet
};
```

MAC Controller Configuration and Information

The network controller driver requires information about the Ethernet MAC controller(s) (typically on your MCU). You provide this information using a structure of type NET_DEV_CFG_ETHER, and each controller you intend to use will require its own configuration structure.

This structure will hold hardware information as well as more application-specific parameters, all of which are related to the specific Ethernet controller you want to use. This configuration structure is located in the BSP file of the Ethernet Controller (bsp*net_ether*&&&.c).

Table - NET_DEV_CFG_ETHER Configuration Structure in the *Network Hardware Information* page describes each configuration field available in this structure.

Table - NET_DEV_CFG_ETHER Configuration Structure

Field	Description
.RxBufPoolType	Specifies the memory location for the receive data buffers. Buffers may be located either in main memory or in a dedicated memory region. This setting is used by the Interface layer to initialize the Rx memory pool. This field must be set to one of two macros: NET_IF_MEM_TYPE_MAIN or NET_IF_MEM_TYPE_DEDICATED. You may want to set this field when using DMA with dedicated memory. It is possible that you might have to store descriptors within the dedicated memory if your device requires it.
.RxBufLargeSize	Specifies the size of all receive buffers. Specifying a value is required. The default buffer length is set to 1518 bytes, which corresponds to the Maximum Transmission Unit (MTU) size for an Ethernet network. For DMA-based Ethernet controllers, you must set the receive data buffer size to be greater or equal to the size of the largest receivable frame. If the size of the total buffer allocation is greater than the amount of available memory in the chosen memory region, a run-time error will be generated when the device is initialized.
.RxBufLargeNbr	Specifies the number of receive buffers that will be allocated to the device. There should be at least one receive buffer allocated, and it is recommended to have at least ten receive buffers. The optimal number of receive buffers depends on your application.
.RxBufAlignOctets	Specifies the required alignment of the receive buffers, in bytes. Some devices require that the receive buffers be aligned to a specific byte boundary. Additionally, some processor architectures do not allow multi-byte reads and writes across word boundaries and therefore may require buffer alignment. In general, it is probably a best practice to align buffers to the data bus width of the processor, which may improve performance. For example, a 32-bit processor may benefit from having buffers aligned on a four-byte boundary.
.RxBufIxOffset	Specifies the receive buffer offset in bytes. Most devices receive packets starting at base index zero in the network buffer data areas. However, some devices may buffer additional bytes prior to the actual received Ethernet packet. This setting configures an offset to ignore these additional bytes. If a device does not buffer any additional bytes ahead of the received Ethernet packet, then an offset of 0 must be specified. However, if a device does buffer additional bytes ahead of the received Ethernet packet, then you should configure this offset with the number of additional bytes.

Field	Description
.TxBufPoolType	Specifies the memory placement of the transmit data buffers. Buffers may be placed either in main memory or in a dedicated memory region. This field is used by the Interface layer, and it should be set to one of two macros: NET_IF_MEM_TYPE_MAIN or NET_IF_MEM_TYPE_DEDICATED. When DMA descriptors are used, they may be stored in dedicated memory.
.TxBufLargeSize	Specifies the size of the large transmit buffers in bytes. This field has no effect if the number of large transmit buffers is configured to zero. Setting the size of the large transmit buffers below 1594 bytes may hinder the TCP/IP module's ability to transmit full sized IP datagrams since IP transmit fragmentation is not yet supported. We recommend setting this field between 1594 and 1614 bytes in order to accommodate all of the maximum transmit packet sizes for TCP-IP's protocols.You can also optimize the transmit buffer if you know in advance what the maximum size of the packets the user will want to transmit through the device are.
.TxBufLargeNbr	Specifies the number of large transmit buffers allocated to the device. You may set this field to zero to make room for additional small transmit buffers, however, the size of the maximum transmittable packet will then depend on the size of the small transmit buffers.
.TxBufSmallSize	Specifies the small transmit buffer size. For devices with a minimal amount of RAM, it is possible to allocate small transmit buffers as well as large transmit buffers. In general, we recommend a small transmit buffer size of 64 bytes, however, you may adjust this value according to the application requirements. This field has no effect if the number of small transmit buffers is configured to zero.
.TxBufSmallNbr	Specifies the numbers of small transmit buffers. This field controls the number of small transmit buffers allocated to the device. You may set this field to zero to make room for additional large transmit buffers if required.
.TxBufAlignOctets	Specifies the transmit buffer alignment in bytes. Some devices require that the transmit buffers be aligned to a specific byte boundary. Additionally, some processor architectures do not allow multi-byte reads and writes across word boundaries, and therefore may require buffer alignment. In general, it's probably a best practice to align buffers to the data bus width of the processor which may improve performance. For example, a 32-bit processor may benefit from having buffers aligned on a four-byte boundary.
.TxBufIxOffset	Specifies the transmit buffer offset in bytes. Most devices only need to transmit the actual Ethernet packets as prepared by the higher network layers. However, some devices may need to transmit additional bytes prior to the actual Ethernet packet. This setting configures an offset to prepare space for these additional bytes. If a device does not transmit any additional bytes ahead of the Ethernet packet, the default offset of zero should be configured. However, if a device does transmit additional bytes ahead of the Ethernet packet then configure this offset with the number of additional bytes.
.MemAddr	Specifies the starting address of the dedicated memory region for devices with this memory type. For devices with non-dedicated memory, you can initialize this field to zero. You may use this setting to put DMA descriptors into the dedicated memory.
.MemSize	Specifies the size of the dedicated memory region in bytes for devices with this memory type. For devices with non-dedicated memory, you can initialize this field to zero. You may use this setting to put DMA descriptors into the dedicated memory.
.Flags	Specify the optional configuration flags. Configure (optional) device features by logically OR'ing bit-field flags:NET_DEV_CFG_FLAG_NONE — No device configuration flags selected.NET_DEV_CFG_FLAG_SWAP_OCTETS — Swap data bytes (i.e., swap data words' high-order bytes with data words' low-order bytes, and vice-versa) if required by device-to-CPU data bus wiring and/or CPU endian word order.
.RxDescNbr	Specifies the number of receive descriptors. For DMA-based devices, this value is used by the device driver during initialization in order to allocate a fixed-size pool of receive descriptors to be used by the device. The number of descriptors must be less than the number of configured receive buffers. We recommend setting this value to something within 40% and 70% of the number of receive buffers. Non-DMA based devices may configure this value to zero. You must use this setting with DMA based devices and at least two descriptors must be set.

Field	Description
.TxDescNbr	Specifies the number of transmit descriptors. For DMA based devices, this value is used by the device driver during initialization to allocate a fixed size pool of transmit descriptors to be used by the device. For best performance, it's recommended to set the number of transmit descriptors equal to the number of small, plus the number of large transmit buffers configured for the device. Non-DMA based devices may configure this value to zero. You must use this setting with DMA based devices and set at least two descriptors.
.BaseAddr	Specifies the base address of the device's hardware/registers.
.DataBusSizeNbrBits	Specifies the size of device's data bus (in bits), if available.
.HW_AddrStr	Specifies the desired device hardware address; may be NULL address or string if the device hardware address is configured or set at run-time.Depending on the driver, if this value is kept NULL or invalid, most of the device driver will automatically try to load and use the hardware address located in the memory of the device.
.CfgExtPtr	Specifies the pointer to an extension configuration structure if needed, else set to DEF_NULL.

PHY Controller Configuration and Information

Information about the PHY controller (typically outside the MCU) is also required by the network controller driver, which you provide using a structure of type NET*PHY_CFG_ETHER. This configuration structure is located in the BSP file of the Ethernet Controller (bsp_net_ether*&&&.c).

Table - NET_PHY_CFG_ETHER Configuration Structure in the *Network Hardware Information* page describes each configuration field available in this structure.

Table - NET_PHY_CFG_ETHER Configuration Structure

Field	Description
.BusAddr	Specifies the address of the PHY on the (R)MII bus. The value configured depends on the PHY and the state of the PHY pi consult the schematics for their board to determine the configured PHY address. Alternatively, the PHY address may be det NET_PHY_ADDR_AUTO; however, this will increase the initialization latency of the TCP/IP stack.
.BusMode	Specifies the PHY bus mode. This value should be set to one of the following values depending on the hardware capabilitie The network controller BSP should configure the PHY-level hardware based on this configuration value.NET_PHY_BUS_MODE_MIINET_PHY_BUS_MODE_RMIINET_PHY_BUS_MODE_SMIINET_PHY_BUS_MODE_GMIINET_PHY_
.Type	Specifies the PHY bus type. This field represents the type of electrical attachment of the PHY to the Ethernet controller. In network controller, while in other cases, it may be attached via an external MII or RMII bus. It is desirable to specify which a driver can initialize additional hardware resources if an external PHY is attached to a device that also has an internal PHY. A are:NET_PHY_TYPE_INTNET_PHY_TYPE_EXT
.Spd	Specifies the PHY initial link speed. This configuration setting will force the PHY to link to the specified link speed. Optionally must be set to one of the following values:NET_PHY_SPD_AUTONET_PHY_SPD_10NET_PHY_SPD_100NET_PHY_SPD_1000
.Duplex	Specifies the PHY initial link duplex. This configuration setting will force the PHY to link using the specified duplex. This setti values:NET_PHY_DUPLEX_AUTONET_PHY_DUPLEX_HALFNET_PHY_DUPLEX_FULL

Hardware Information

The last step is to create the main controller hardware information structure. This structure contains only pointers to other structures.

Table - Controller Hardware Information Structure in the *Network Hardware Information* page describes each configuration field available in this structure.

Field	Description
.DrvAPI_Ptr	Pointer to the driver API structure associated with your network controller IP.

Field	Description
.BSP_API_Ptr	Pointer to the BSP API structure you created at step BSP API Structure .
.PHY_API_Ptr	Pointer to the PHY driver API structure associated with your network PHY controller IP.
.DevCfgPtr	Pointer to the controller information structure you created at step Controller Configuration and Information .
.PHY_CfgPtr	Pointer to the PHY controller information structure you created at step PHY Configuration and Information .

Table - Controller Hardware Information Structure

Wireless Interface

WiFi Part Information

The Wi-Fi driver requires information about the Wi-Fi controller/part, which you provide using a structure of type NET*IF_WIFI_PART_INFO. Each Wi-Fi part has its own information structure. This information can be found in the manual of the Wi-Fi controller. For the Wi-Fi parts that are supported by the Micrium Network module, the information structures are already defined and available in the Wi-Fi driver (net_drv_wifi*&&&.c) associated with each WiFi controller.

Table - NET_IF_WIFI_PART_INFO Structure

Field	Description
.DrvAPI_Ptr	Pointer to the driver API structure associated with the Wi-Fi controller.
.RxBufIxOffset	Specifies the receive buffer offset in bytes. Most controllers receive packets starting at base index zero in the network buffer data areas. However, some controllers may buffer additional bytes prior to the actual received packet. This setting configures an offset to ignore these additional bytes. If a device does not buffer any additional bytes ahead of the received packet, then an offset of 0 must be specified. However, if a device does buffer additional bytes ahead of the received packet, then you should configure this offset with the number of additional bytes.
.TxBufIxOffset	Specifies the transmit buffer offset in bytes. Most controllers only need to transmit the actual packets as prepared by the higher network layers. However, some controllers may need to transmit additional bytes prior to the actual packet. This setting configures an offset to prepare space for these additional bytes. If a device does not transmit any additional bytes ahead of the packet, the default offset of zero should be configured. However, if a device does transmit additional bytes ahead of the packet then configure this offset with the number of additional bytes.

BSP API Structure

In order to provide a pointer to the BSP functions for the Wi-Fi controller driver, you must create a structure of type NET*DEV_BSP_WIFI. Each network controller you intend to use will need is own BSP API structure. This is must be defined within the network controller's BSP file (bsp_net_wifi*&&&.c).

Table - NET_DEV_BSP_WIFI Structure in the *Network Hardware Information* page describes each field available in this structure.

Table - NET_DEV_BSP_WIFI Structure

Field	Description
.Start	Pointer to the BSP start function.
.Stop	Pointer to the BSP stop function.
.CfgGPIO	Pointer to the BSP GPIO configuration function.
.CfgIntCtrl	Pointer to the BSP interrupt configuration function.
.IntCtrl	Pointer to the BSP interrupt control function.

Network Controller Registration to the Platform Manager

Once the hardware information structure for your network controller is ready, it must be registered with the Platform Manager. This should normally be done from the BSP_OS_Init() function that is located in the file bsp_os.c.

There are two different macros you can call to register a network controller; one for Ethernet controllers and one for Wi-Fi controllers.

Table - Network Controller Register Macros

Macro	Description	Definition Location
NET_CTRLR_ETHER_REG()	Registers an Ethernet controller with the Platform Manager.	net_if_ether.h
NET_CTRLR_WIFI_SPI_REG()	Registers an external Wi-Fi controller that communicates with the MCU via SPI with the Platform Manager.	net_if_wifi.h

Ethernet Controller Registration

Listing - Example of Ethernet Controller Registration in the *Network Controller Registration to the Platform Manager* page shows an example of how to register a network Ethernet controller.

Listing - Example of Ethernet Controller Registration

```
#include  <rtos_description.h>

#ifdef  RTOS_MODULE_NET_IF_ETHER_AVAIL                          (1)
BSP_HW_INFO_EXT(const  NET_IF_ETHER_CTRLR_INFO, BSP_Ether_<ctrlr0>_Info);
#endif

void  BSP_OS_Init (void)
{

    /* ... */
                        /* --------- REGISTER ETHER CONTROLLER ------- */
#ifdef  RTOS_MODULE_NET_IF_ETHER_AVAIL
    NET_CTRLR_ETHER_REG("eth0", &BSP_Ether_<ctrlr0>_Info);          (2)
#endif
}
```

(1) Since the global variable for Ethernet controller information is declared in another file, you must declare it as external in your bsp_os.c file. Always use the BSP_HW_INFO_EXT() macro.

(2) Registering an Ethernet controller with the tag "eth0". The tag will be used later on when calling the NetIF_Ether_Add() function.

Wi-Fi Controller Registration

Listing - Example of WiFi Controller Registration in the *Network Controller Registration to the Platform Manager* page shows an example of how to register an external network Wi-Fi controller that is connected to the MCU via an SPI bus.

Listing - Example of WiFi Controller Registration

```
#include  <rtos_description.h>

#if defined(RTOS_MODULE_IO_SERIAL_AVAIL)                         (1)
BSP_HW_INFO_EXT(const  SERIAL_CTRLR_DRV_INFO,    BSP_Serial_SPI_HwInfo_<bus>__DrvInfo);
#endif
#if defined(RTOS_MODULE_NET_IF_WIFI_AVAIL)
BSP_HW_INFO_EXT(const  NET_IF_WIFI_PART_INFO,    NetWiFi_Info_<ctrlr>);
BSP_HW_INFO_EXT(const  NET_DEV_BSP_WIFI,         BSP_NetWiFi_<ctrlr>_BSP_API);
#endif
```

```
void  BSP_OS_Init (void){/* ... *//* ------ REGISTER SPI & WiFi CONTROLLER ----- */
#ifdefined(RTOS_MODULE_IO_SERIAL_AVAIL)IO_SERIAL_CTRLR_REG("spi0",(2)&BSP_Serial_SPI_HwInfo_<bus>__DrvInfo);
#endif

#ifdefined(RTOS_MODULE_NET_IF_WIFI_AVAIL)NET_CTRLR_WIFI_SPI_REG("wifi0",(3)"spi0",&NetWiFi_Info_<ctrlr>,&BSP_NetWiFi_<ctrlr>_BSP_API,0u);
#endif
}
```

(1) Since the global variables for the Wi-Fi controller information are declared in another file, you must declare them as external in your bsp_os.c file. Always use the BSP_HW_INFO_EXT() macro.

(2) Registering the SPI Bus used by the Wi-Fi controller with the appropriate tag. The tag will be used later on when calling the SPI_BusAdd() function.

(3) Registering a Wi-Fi controller with the tag "wifi0". The tag will be used later on when calling the NetIF_WiFi_Add() function.

Network Core Troubleshooting

Network Core Troubleshooting

- Network Interface
- Socket Operations
- Performance Issues
  - Number of RX & TX Buffers to Configure
  - Number of DMA Descriptors to Configure
  - Configuring Window Sizes
  - Reducing the Number of Transitory Errors
- Wi-Fi Interface
  - An RTOS_ERR_IO error is returned when I try to join a Wi-Fi network
  - I am getting an RTOS_ERR_INVALID_HANDLE in SPI_SlaveOpen() when adding a Wi-Fi interface
  - I am getting an RTOS_ERR_NOT_FOUND when adding a Wi-Fi interface

Network Interface

When adding a new network interface with NetIF_xx_Add()**, I never return from the function.**

- Verify that your controller Base Address is correct (see section).
- Could also be a problem with your BSP Clock or GPIO configuration (see section).

When adding a new network interface with NetIF_xx_Add()**, the target crashes before returning**.

- Verify that your controller Base Address is correct (the target should crash in the driver initialization function NetDev_Init() if the Base Address is incorrect).
- Could be a problem in your BSP (the target should crash in NetDev_Init() after a BSP call).
- Your task stack size could be insufficient.

When adding a new network interface with NetIF_xx_Add()**, I get an error.**

These are the more common errors:

- RTOS_ERR_NOT_FOUND : Before creating a new network interface, you need to have previously registered the network controller associated with this network interface inside the BSP_OS_Init() function.
- RTOS_ERR_NO_MORE_RSRC : Be sure to have set the maximum network interface (NET_IF_CFG_MAX_NBR_IF inside file net_cfg.h) according to your application needs.
- RTOS_ERR_SEG_OVF : The memory space required for the new network interface is insufficient. You can increase your general HEAP memory size or the memory segment used for the network module. You can also try to reduce the number of buffers configured for this network interface.

When starting a network interface with NetIF_xx_Start()**, I end up in a debug assert with the error code** RTOS_ERR_NOT_AVAIL**.**

You are trying to setup the network interface with a module that is not enabled in your configuration; e.g., your NET_IF_xx_CFG argument enables an IPv6 address configuration process but the NET_IPv6_CFG_EN inside your *net_cfg.h* file is set to DEF_DISABLED.

When starting a network interface with NetIF_xx_Start()**, I set the** NET_IF_xx_CFG **argument to enable DHCP client, but then I just exit** NetIF_xx_Start() **without knowing if the DHCP process was successful. (This can also apply to the IPv6 setup)**

The NetIF_xx_Start() function is non-blocking. If you want to be notified when the DHCP client process is done, you need to define a callback function and refer to it in the DHCPc_ON_COMPLETE_HOOK field of the NET_IF_xx_CFG argument. The program call the hook function when the DHCP process has completed. You can also use the API function

NetIF_WaitSetupReady() that allows waiting for all the setup of the network interface to be done after the Start function. When using this API the different callback functions inside the NET_IF_xx_CFG argument are not mandatory.

When using the function NetIF_WaitSetupReady () **to wait for the setup completion after the network interface start, the function returns with the error code** RTOS_ERR_TIMEOUT**.**

By default, the function NetIF_WaitSetupReady() will time out after a delay of 30 seconds even if not all the setup steps have been completed. This delay should be enough on most networks. You can pass a bigger timeout value if you feel that your network needs more time to complete all the address setups. You can also use the argument of type NET_IF_APP_INFO to get more information on the setup progress. By passing a pointer to a NET_IF_APP_INFO variable, the function will populate the structure with the information of the different address setups gradually. You can, therefore, see which ones was completed and the ones still in progress. If no address setups seem to have happened during the waiting process, you should validate that your network interface is well configured because it could mean that the interface is never getting is link state to UP.

Socket Operations

When binding a socket to an IP address using the NetSock_Bind() **function, I get an error.**

These are the more common errors:

- RTOS_ERR_INVALID_HANDLE : The socket ID passed to the function is invalid. Check if you did not previously close the socket.
- RTOS_ERR_NOT_FOUND : The IP address must have previously been configured on the network interface.You can validate if an IP address is configured with the function NetIPv4_IsAddrHostCfgd() or NetIPv6_IsAddrHostCfgd().
- RTOS_ERR_ALREADY_EXISTS : A socket binding with the same IP local/remote addresses and local/remote ports already exist. The network stack doesn't support to have multiple sockets with the exact same binding of local/remote addresses/ports.

When calling NetSock_Conn() **to open a TCP connection to a remote host, I get an error.**

These are the more common errors:

- RTOS_ERR_INVALID_HANDLE : The socket ID passed to the function is invalid. Check if you did not previously close the socket.
- RTOS_ERR_ALREADY_EXISTS : A socket binding with the same IP local/remote addresses and local/remote ports already exist. The network stack doesn't support to have multiple sockets with the exact same binding of local/remote addresses/ports.
- RTOS_ERR_NET_INVALID_ADDR_SRC : No valid configured IP address was found to send the connection request. You should check that you have a valid IP address configured on each of your network interfaces.

Transitory errors are also possible. Most of the time, in those cases, you can retry the operation. It is recommended to add a short delay before the retry especially if your socket is in a non-blocking mode:

- RTOS_ERR_POOL_EMPTY : You are referring to a memory pool that doesn't have any element left inside. In this context, it will refer most of the time to the network buffer pool, therefore no more buffer is available to send the TCP SYN message.

When my TCP socket is in listening mode and I call NetSock_Accept() **to accept new incoming connections, I get an error.**

These are the more common errors:

- RTOS_ERR_INVALID_HANDLE : The socket ID passed to the function is invalid. Check if you did not previously close the socket.

Transitory errors are also possible. Most of the time, in those cases, you can retry the operation. It is recommended to add a short delay before the retry especially if your socket is in a non-blocking mode:

1. RTOS_ERR_POOL_EMPTY : You are referring to a memory pool that doesn't have any element left inside. In this context, it will refer most of the time to the socket pool since a child socket will be created for the new connection. Therefore be sure that your socket configuration includes enough TCP socket type for your needs.
2. RTOS_ERR_TIMEOUT : When the socket is in blocking mode, this means that no incoming connections requests were received even after the waiting inside the NetSock_Accept() function. The timeout value can be set by the macro

NET_SOCK_DFLT_TIMEOUT_CONN_ACCEPT_MS in the *net_cfg.h* file or can be set for via API functions (see NetSock_CfgTimeoutConnAcceptDflt() and NetSock_CfgTimeoutConnAcceptSet()).
3. RTOS_ERR_WOULD_BLOCK : When the socket is in non-blocking mode, this means that no incoming connection requests were received at the time NetSock_Accept() was called.

When my TCP socket is in listening mode and multiple new incoming connection requests are sent to it, not all connections seem to be received by the socket.

Different reasons can explain why some TCP connection requests are dropped by the stack. The two main reasons are:

- Your socket accept queue is already full when a new incoming connection request is received. You can increase the size of your socket accept queue with the configuration macro NET_SOCK_CFG_CONN_ACCEPT_Q_SIZE_MAX in *net_cfg.h* file.
- You don't have any TCP connection objects left when a new incoming connection request is received. The number of TCP connection objects you have comes from the configuration macro NET_SOCK_CFG_SOCK_NBR_TCP, but you can also have additional TCP connection objects from the configuration macro NET_TCP_CFG_NBR_CONN.

To validate if either of those cases are happening, you can check the error counters Net_ErrCtrs.TCP.RxListenQ_FullCtr and Net_ErrCtrs.TCP.NoneAvailCtr.

When calling NetSock_RxDataFrom() **/** NetSock_RxData() **to receive data, I get an error.**

These are the more common errors:

- RTOS_ERR_INVALID_HANDLE : The socket ID passed to the function is invalid. Check if you did not previously close the socket.
- RTOS_ERR_NOT_FOUND : The destination address passed as argument when using API NetSock_RxDataFrom() is not associated with any known connections.
- RTOS_ERR_NET_CONN_CLOSE_RX : In case of a TCP socket, this means a FIN was received from the other side of the connection. Therefore, when ready, you should close your socket.
- RTOS_ERR_NET_CONN_CLOSED_FAULT : In case of a TCP socket, this means that a RESET was received. Therefore you should close your socket as soon as possible.
- RTOS_ERR_WOULD_OVF : In case of a UDP socket, the size of the reception buffer is not enough for the size of the datagram received.

Transitory errors are also possible. Most of the time, in those cases, you can retry the operation. It is recommended to add a short delay before the retry especially if your socket is in a non-blocking mode:

- RTOS_ERR_TIMEOUT : When the socket is in blocking mode, this means that no data was received even after the waiting inside the receive function. The timeout value can be set by the macro NET_SOCK_DFLT_TIMEOUT_RX_Q_MS in the *net_cfg.h* file or can be set for via API functions (see NetSock_CfgTimeoutRxQ_Dflt() and NetSock_CfgTimeoutRxQ_Set()).
- RTOS_ERR_WOULD_BLOCK : When the socket is in non-blocking mode, this means that no data was received at the time the receive function was called.

When calling NetSock_TxDataTo() **/** NetSock_TxData() **to transmit data, I get an error.**

These are the more common errors:

- RTOS_ERR_INVALID_HANDLE : The socket ID passed to the function is invalid. Check if you did not previously close the socket.
- RTOS_ERR_NOT_FOUND : The destination address passed as argument when using API NetSock_TxDataTo() is not associated with any known connections.
- RTOS_ERR_NET_CONN_CLOSE_RX : In case of a TCP socket, this means a FIN was received from the other side of the connection. Therefore, when ready, you should close your socket.
- RTOS_ERR_NET_CONN_CLOSED_FAULT : In case of a TCP socket, this means that a RESET was received. Therefore you should close your socket as soon as possible.
- RTOS_ERR_NET_INVALID_ADDR_SRC : In case of a UDP socket, no valid configured IP address was found to sent the data. You should check that you have a valid IP address configured on each of your network interfaces.
- RTOS_ERR_NET_NEXT_HOP : No valid next hop was found to sent the packet to. You should check that a valid gateway was configured if you wish to send data outside the local network.
- RTOS_ERR_WOULD_OVF : In case of a UDP socket, this means that the length of the data to send is too big for the size of the transmit buffers.

Transitory errors are also possible. Most of the time, in those cases, you can retry the operation. It is recommended to add a short delay before the retry especially if your socket is in a non-blocking mode:

- RTOS_ERR_TIMEOUT : When the socket is in blocking mode and for TCP socket, this means that the transmit window is full and no data can be queued for transmission. The timeout value can be set by the macro NET_TCP_DFLT_TIMEOUT_CONN_TX_Q_MS in the net_cfg.h file or can be set for via API functions (see NetSock_CfgTimeoutTxQ_Dflt() and NetSock_CfgTimeoutTxQ_Set()).
- RTOS_ERR_WOULD_BLOCK : When the socket is in non-blocking mode, this means that the transmit window is full at the time the transmit function was called.
- RTOS_ERR_POOL_EMPTY : Refer to a memory pool who doesn't have any element left inside. In this context, it will refer most of the time to the network buffer pool, therefore no more buffer is available to send the data.
- RTOS_ERR_NET_ADDR_UNRESOLVED : The destination MAC address of the next hop is not resolved. It could be that the ARP or NDP process was just not completed yet.

Performance Issues

Number of RX & TX Buffers to Configure

The number of large receive, small transmit and large transmit buffers you will configure for a specific interface depends on several factors:

1. Desired level of performance.
2. Amount of data to be either transmitted or received.
3. Ability of the target application to either produce or consume transmitted or received data.
4. Average CPU utilization.
5. Average network utilization.
6. Type of connection (UDP or TCP)
7. Number of simultaneous connection.
8. Application/connection priorities

The discussion on the bandwidth-delay product is always valid. In general, the more buffers the better. However, the number of buffers can be tailored based on the application. For example, if an application receives a lot of data but transmits very little, then it may be sufficient to define a number of small transmit buffers for operations such as TCP acknowledgments and allocate the remaining memory to large receive buffers. Similarly, if an application transmits and receives little, then the buffer allocation emphasis should be on defining more transmit buffers. However, there is a caveat:

If the application is written such that the task that consumes receive data runs infrequently or the CPU utilization is high and the receiving application task(s) becomes starved for CPU time, then more receive buffers will be required.

To ensure the highest level of performance possible, it makes sense to define as many buffers as possible and use the interface and pool statistics data in order to refine the number after having run the application for a while. A busy network will require more receive buffers in order to handle the additional broadcast messages that will be received.

In general, at least two large and two small transmit buffers should be configured. This assumes that neither the network or CPU are very busy.

Many applications will receive properly with four or more large receive buffers. However, for TCP applications that move a lot of data between the target and the peer, this number may need to be higher.

Specifying too few transmit or receive buffers may lead to stalls in communication and possibly even dead-lock. Care should be taken when configuring the number of buffers. The Network module is often tested with configurations of 10 or more small transmit, large transmit, and large receive buffers.

Number of DMA Descriptors to Configure

If the hardware device is an Ethernet MAC that supports DMA, then the number of configured receive descriptors will play an important role in determining overall performance for the configured interface.

For applications with 10 or less large receive buffers, it is desirable to configure the number of receive descriptors to that of 60% to 70% of the number of configured receive buffers.

In this example, 60% of 10 receive buffers allows for four receive buffers to be available to the stack waiting to be processed by application tasks. While the application is processing data, the hardware may continue to receive additional

frames up to the number of configured receive descriptors.

There is, however, a point at which configuring additional receive descriptors no longer greatly impacts performance. For applications with 20 or more buffers, the number of descriptors can be configured to 50% of the number of configured receive buffers. After this point, only the number of buffers remains a significant factor; especially for slower or busy CPUs and networks with higher utilization.

In general, if the CPU is not busy and the Network Receive task has the opportunity to run often, the ratio of receive descriptors to receive buffers may be reduced further for very high numbers of available receive buffers (e.g., 50 or more).

The number of transmit descriptors should be configured such that it is equal to the number of small plus the number of large transmit buffers.

These numbers only serve as a starting point. The application and the environment that the device will be attached to will ultimately dictate the number of required transmit and receive descriptors necessary for achieving maximum performance.

Specifying too few descriptors can cause communication delays. See section Network Interface Controller Configuration for descriptors configuration.

Configuring Window Sizes

Receive and transmit queue size must be properly configured to optimize performance. It represents the number of bytes that can be queued by one socket. It's important that all socket are not able to queue more data than what the device can hold in its buffers. The size should be also a multiple of the maximum segment size (MSS) to optimize performance. UDP MSS is 1470 and TCP MSS is 1460.

RX and TX maximum queue size are configured using #defines in net_cfg.h , see Network Core Compile-Time Configurations .

RX and TX queue size can be reduced at runtime using socket option API (NetTCP_ConnCfgRxWinSize() and NetTCP_ConnCfgTxWinSize()).

The following listing shows a calculation example:

```
Number of TCP connection  : 2
  Number of UDP connection  : 0
  Number of RX large buffer : 10
  Number of TX Large buffer : 6
  Number of TX small buffer : 2
  Size of RX large buffer   : 1518
  Size of TX large buffer   : 1518
  Size of TX small buffer   : 60

  TCP MSS RX                = 1460
  TCP MSS TX large buffer   = 1460
  TCP MSS TX small buffer   = 0

  Maximum receive  window   = (10 * 1460)           = 14600 bytes
  Maximum transmit window   = (6  * 1460) + (2 * 0) = 8760  bytes

  RX window size per socket = (14600 / 2) =  7300 bytes
  TX window size per socket = (8760  / 2) =  4380 bytes
```

Reducing the Number of Transitory Errors

The number of transmit buffer should be increased. Additionally, it may be helpful to add a short delay between successive calls to socket transmit functions.

Wi-Fi Interface

An RTOS_ERR_IO error is returned when I try to join a WiFi network

Make sure the SSID and Password sent to NetIF_WiFi_Join() are correct. Also note that some WiFi adapters require you to perform a scan before attempting to join an access point.

I am getting an RTOS_ERR_INVALID_HANDLE in SPI_SlaveOpen() when adding a WiFi interface

Make sure you are specifying the correct SPI bus handle when registering the WiFi controller in the BSP. Also make sure that it corresponds to the handle of the serial controller that was registered beforehand in the BSP. If you are using the provided examples, don't forget to update the value of EX_SPI_CTRLR_NAME in the SPI initialization example file.

I am getting an RTOS_ERR_NOT_FOUND when adding a WiFi interface

Make sure that the WiFi controller is properly registered at the BSP level (bsp_os.c). If multiple WiFi controllers are supported on a development board you are using, you may have to select it in bsp_net.h.

Application Modules

Application Modules

This section introduces you to the various network application modules that are included as part of Micrium OS.

Complementary information is also available in the Network section of the Technologies Overview documentation page. The following sections will refer to it when appropriate.

- HTTP Client Module
- HTTP Server Module
- MQTT Client Module
- SNTP Client Module
- SMTP Client Module
- FTP Client Module
- IPerf Module
- Telnet Server Module
- TFTP Client Module
- TFTP Server Module

HTTP Client Module

The Micrium OS HTTP Client can be used to access HTTP servers and perform actions on resources maintained by a server.

In an embedded environment, an HTTP client will rarely be used to display web pages like a web browser does. It will primarily be used to get and update specific data, download and upload files or to interface with web services.

- HTTP Client Overview
- HTTP Client Example Applications
- HTTP Client Configuration
  - HTTP Client Compile-Time Configurations
  - HTTP Client Run-Time Application Specific Configurations
    - HTTP Client Connection Configurations
    - HTTP Client Quantity Configurations
- HTTP Client Programming Guide
  - HTTP Client Control Structure
  - HTTP Client Connection Object Setup
  - HTTP Client Request Object Setup
  - HTTP Client Open Secure Connection with SSL-TSL
  - HTTP Client Add Additional Header fields to an HTTP Request
  - HTTP Client Request's Body Standard Transfer
  - HTTP Client Chunked Transfer Encoding
  - HTTP Client Query String
  - HTTP Client Form Submission
  - HTTP Client Persistent Connection
  - HTTP Client Retrieve HTTP Response Data
  - HTTP Client WebSocket
    - HTTP Client Upgrading an HTTP Connection
    - HTTP Client Message reception
      - HTTP Client Auto Mode Example
      - HTTP Client Normal Mode Example
    - HTTP Client Sending Message

- HTTP Client Closing a Connection

HTTP Client Protocol Recognized Fields

HTTP Client Overview

- Specifications
- Features
- Design
  - Memory Usage
  - Internal Task
  - Auto Select Remote IP address
- Limitations

Specifications

- Complies with the following RFC:
  - RFC 2616 - Hypertext Transfer Protocol -- HTTP/1.1
  - RFC 6455 - The WebSocket Protocol
- Implements the following methods:
  - GET
  - POST
  - PUT
  - DELETE
  - HEAD

Features

- Scalable to contain only required features and minimize memory footprint
- Supports persistent connection
- Supports form submission
- Supports chunked transfer encoding for both reception and transmission
- Supports application and multipart forms
- Supports reception of HTTP headers in requests and addition of HTTP headers in responses.
- Supports HTTP HTTP Query String processing
- Supports HTTP WebSocket message processing

Design

The HTTP Client module has been designed to suit many different types of HTTP client applications. With the right configuration it can be used with a very low footprint to send a simple HTTP request and wait for the HTTP server's response. Another benefit of its design is that it can also be used to make simultaneous HTTP requests by opening multiple connections which allows for consistent usage of the same API calls.

Memory Usage

Each HTTP client application can have very different memory needs. For example, some applications will only require one connection whereas others will require more; some applications will prefer to allocate objects on their own stack while others will want to use a memory pool on the heap. Therefore, to conform with all of these different requirements, HTTP Client does not have its own memory module. All the objects required by the HTTP Client **MUST** be passed by the application. Three HTTP Client Objects must be supplied by the Application: a Connection object , a Request Object and a Response Object . Also, a buffer per connection must be supplied for the transmission/reception operations. Additional objects may be required when some available features are enabled.

> **It is very important that every object passed by the application to the HTTP Client stack remain valid for the duration of the HTTP transaction or until the connection is closed for the HTTP Client Connection Object.**

Internal Task

To support multiple simultaneous connections, HTTP Client has an internal task. This task also allows the queuing of multiple requests on a given connection. When the HTTP Client Task is present, API calls to HTTP Client's stack can be set to the

blocking or non-blocking mode.

For simpler applications and to reduce memory footprint, the internal HTTP Client task can be disabled. In that case, HTTP Client operations will be executed directly in the application task and will always be blocking.

Auto Select Remote IP address

When the DNS Client that is part of the core network is properly configured, HTTP Client can accept a host name string to connect to the remote server. It will perform the IP address resolution by communicating with a DNS server. If the DNS server sends back IPv4 and IPv6 addresses, the first attempt will be to use the IPv6 address to connect to the remote HTTP server. If this connection fails the IPv4 address will be used instead.

Limitations

- Only supports HTTP v1.1.
- HTTP range requests are not supported.

HTTP Client Example Applications

This section describes the examples that are related to the HTTP Client module of Micrium OS.

- HTTP Client Initialization Example
  - Description
  - Configuration
    - Mandatory
    - Optional
  - Location
  - API
  - Notes
- HTTP Client GET Request Examples
  - Description
  - Configuration
    - Optional
  - Location
  - API
  - Notes
- HTTP Client POST Request Examples
  - Description
  - Configuration
    - Optional
  - Location
  - API
  - Notes
- HTTP Client PUT Request Examples
  - Description
  - Configuration
    - Optional
  - Location
  - API
  - Notes
- HTTP Client Persistent Connection Examples
  - Description
  - Configuration
    1. Optional
  - Location
  - API
  - Notes
- HTTP Client Multi-connection Examples
  - Description
  - Configuration
    - Optional

- Location
- API
- Notes

HTTP Client Initialization Example

Description

This is a generic example that shows how to initialize the HTTP client module. It accomplishes the following tasks:

- Initialize the HTTP client module
- Create a RAM disk for file exchange:
  - Store a file (index.html) to be sent to a server

Configuration

Mandatory

The following #define must be added in ex_description.h to allow other examples to initialize the HTTP Client module correctly:

#define	Description
EX_HTTP_CLIENT_INIT_AVAIL	Lets the upper example layer know that the HTTP Client Initialization example is present and must be called by other examples.

Optional

The following #define can be added to ex_description.h, as described in Example Applications section, to change default configuration value used by the example:

#define	Default value	Description
EX_HTTP_CLIENT_RAMDISK_SEC_SIZE	512 u	Specify the sector size of the RAM disk
EX_HTTP_CLIENT_RAMDISK_SEC_NBR	60 u	Specify the number of sector for the RAM disk
EX_HTTP_CLIENT_FILE_RAM_MEDIA_NAME	"ram_httpc"	Specify the RAM media name
EX_HTTP_CLIENT_FILE_VOL_NAME	"ram_httpc"	Specify the RAM volume name
EX_HTTP_CLIENT_WRK_DIR	EX_HTTP_CLIENT_FILE_VOL_NAME	Specify the working directory used by the example
EX_HTTP_CLIENT_FILE_NAME	"index.html"	Specify the name of the internal file.
EX_HTTP_CLIENT_CLIENT_HOSTNAME	"httpbin.org"	Specify the server host name used by the client
EX_HTTP_CLIENT_CONN_NBR_MAX	5 u	Specify the maximum of active connection
EX_HTTP_CLIENT_REQ_NBR_MAX	5 u	Specify the maximum of requests
EX_HTTP_CLIENT_CONN_BUF_SIZE	512 u	Specify the connection buffer size
EX_HTTP_CLIENT_CFG_QUERY_STR_NBR_MAX	6 u	Specify the maximum number of query string
EX_HTTP_CLIENT_CFG_QUERY_STR_KEY_LEN_MAX	20 u	Specify the maximum of query string key length

#define	Default value	Description
EX_HTTP_CLIENT_CFG_QUERY_STR_VAL_LEN_MAX	50 u	Specify the maximum of query string value length
EX_HTTP_CLIENT_CFG_HDR_NBR_MAX	6 u	Specify the maximum of header object
EX_HTTP_CLIENT_CFG_HDR_VAL_LEN_MAX	100 u	Specify the maximum of header value length
EX_HTTP_CLIENT_CFG_FORM_BUF_SIZE	256 u	Specify the form buffer size.
EX_HTTP_CLIENT_CFG_FORM_FIELD_NBR_MAX	10 u	Specify the maximum number of form's field
EX_HTTP_CLIENT_CFG_FORM_FIELD_KEY_LEN_MAX	100 u	Specify the maximum of form key length
EX_HTTP_CLIENT_CFG_FORM_FIELD_VAL_LEN_MAX	200 u	Specify the maximum of form value length
EX_HTTP_CLIENT_CFG_FORM_MULTIPART_NAME_LEN_MAX	100 u	Specify the maximum of multipart name length
EX_HTTP_CLIENT_CFG_FORM_MULTIPART_FILENAME_LEN_MAX	100 u	Specify the maximum of multipart filename length

Location

```
/examples/net/http/client/ex_http_client.c

/examples/net/http/client/ex_http_client.h
```

API

API	Description
Ex_HTTP_Client_Init()	Initialize the HTTP Client stack for the example application.

Notes

None.

HTTP Client GET Request Examples

Description

This is a generic example that shows how to send GET request. It accomplishes the following tasks:

- Set connection option
- Open an HTTP connection
- Send GET request
- Close HTTP Connection

Configuration

Optional

The following #define can be added to ex_description.h, as described in Example Applications section, to change default configuration value used by the example:

#define	Default value	Description
EX_HTTP_CLIENT_RAMDISK_SEC_SIZE	512 u	Specify the sector size of the RAM disk
EX_HTTP_CLIENT_RAMDISK_SEC_NBR	60 u	Specify the number of sector for the RAM disk

#define	Default value	Description
EX_HTTP_CLIENT_FILE_RAM_MEDIA_NAME	"ram_httpc"	Specify the RAM media name
EX_HTTP_CLIENT_FILE_VOL_NAME	"ram_httpc"	Specify the RAM volume name
EX_HTTP_CLIENT_WRK_DIR	EX_HTTP_CLIENT_FILE_VOL_NAME	Specify the working directory used by the example
EX_HTTP_CLIENT_FILE_NAME	"index.html"	Specify the name of the internal file.
EX_HTTP_CLIENT_CLIENT_HOSTNAME	"httpbin.org "	Specify the server host name used by the client
EX_HTTP_CLIENT_CONN_NBR_MAX	5 u	Specify the maximum of active connection
EX_HTTP_CLIENT_REQ_NBR_MAX	5 u	Specify the maximum of requests
EX_HTTP_CLIENT_CONN_BUF_SIZE	512 u	Specify the connection buffer size
EX_HTTP_CLIENT_CFG_QUERY_STR_NBR_MAX	6 u	Specify the maximum number of query string
EX_HTTP_CLIENT_CFG_QUERY_STR_KEY_LEN_MAX	20 u	Specify the maximum of query string key length
EX_HTTP_CLIENT_CFG_QUERY_STR_VAL_LEN_MAX	50 u	Specify the maximum of query string value length
EX_HTTP_CLIENT_CFG_HDR_NBR_MAX	6 u	Specify the maximum of header object
EX_HTTP_CLIENT_CFG_HDR_VAL_LEN_MAX	100 u	Specify the maximum of header value length
EX_HTTP_CLIENT_CFG_FORM_BUF_SIZE	256 u	Specify the form buffer size.
EX_HTTP_CLIENT_CFG_FORM_FIELD_NBR_MAX	10 u	Specify the maximum number of form's field
EX_HTTP_CLIENT_CFG_FORM_FIELD_KEY_LEN_MAX	100 u	Specify the maximum of form key length
EX_HTTP_CLIENT_CFG_FORM_FIELD_VAL_LEN_MAX	200 u	Specify the maximum of form value length

#define	Default value	Description
EX_HTTP_CLIENT_CFG_FORM_MULTIPART_NAME_LEN_MAX	100 u	Specify the maximum of multipart name length
EX_HTTP_CLIENT_CFG_FORM_MULTIPART_FILENAME_LEN_MAX	100 u	Specify the maximum of multipart filename length

Location

> /examples/net/http/client/ex_http_client_files.c
>
> /examples/net/http/client/ex_http_client_files.h
>
> /examples/net/http/client/ex_http_client_hooks.c
>
> /examples/net/http/client/ex_http_client_hooks.h
>
> /examples/net/http/client/ex_http_client.c
>
> /examples/net/http/client/ex_http_client.h

API

API	Description
Ex_HTTP_Client_ReqSendGet_NoTask()	Send a GET request synchronously, i.e., returns only when the transaction is completed.
Ex_HTTP_Client_ReqSendGet()	Send a GET request asynchronously. Show the callback system to be notified when the transaction is completed.

Notes

None.

HTTP Client POST Request Examples

Description

This is a generic example that shows how to send POST request. It accomplishes the following tasks:

- Set connection option
- Open an HTTP connection
- Send POST request
- Close HTTP Connection

Configuration

Optional

The following #define can be added to ex_description.h, as described in Example Applications section, to change default configuration value used by the example:

#define	Default value	Description
EX_HTTP_CLIENT_RAMDISK_SEC_SIZE	512 u	Specify the sector size of the RAM disk
EX_HTTP_CLIENT_RAMDISK_SEC_NBR	60 u	Specify the number of sector for the RAM disk
EX_HTTP_CLIENT_FILE_RAM_MEDIA_NAME	"ram_httpc"	Specify the RAM media name

#define	Default value	Description
EX_HTTP_CLIENT_FILE_VOL_NAME	"ram_httpc"	Specify the RAM volume name
EX_HTTP_CLIENT_WRK_DIR	EX_HTTP_CLIENT_FILE_VOL_NAME	Specify the working directory used by the example
EX_HTTP_CLIENT_FILE_NAME	"index.html"	Specify the name of the internal file.
EX_HTTP_CLIENT_CLIENT_HOSTNAME	"httpbin.org "	Specify the server host name used by the client
EX_HTTP_CLIENT_CONN_NBR_MAX	5 u	Specify the maximum of active connection
EX_HTTP_CLIENT_REQ_NBR_MAX	5 u	Specify the maximum of requests
EX_HTTP_CLIENT_CONN_BUF_SIZE	512 u	Specify the connection buffer size
EX_HTTP_CLIENT_CFG_QUERY_STR_NBR_MAX	6 u	Specify the maximum number of query string
EX_HTTP_CLIENT_CFG_QUERY_STR_KEY_LEN_MAX	20 u	Specify the maximum of query string key length
EX_HTTP_CLIENT_CFG_QUERY_STR_VAL_LEN_MAX	50 u	Specify the maximum of query string value length
EX_HTTP_CLIENT_CFG_HDR_NBR_MAX	6 u	Specify the maximum of header object
EX_HTTP_CLIENT_CFG_HDR_VAL_LEN_MAX	100 u	Specify the maximum of header value length
EX_HTTP_CLIENT_CFG_FORM_BUF_SIZE	256 u	Specify the form buffer size.
EX_HTTP_CLIENT_CFG_FORM_FIELD_NBR_MAX	10 u	Specify the maximum number of form's field
EX_HTTP_CLIENT_CFG_FORM_FIELD_KEY_LEN_MAX	100 u	Specify the maximum of form key length
EX_HTTP_CLIENT_CFG_FORM_FIELD_VAL_LEN_MAX	200 u	Specify the maximum of form value length

#define	Default value	Description
EX_HTTP_CLIENT_CFG_FORM_MULTIPART_NAME_LEN_MAX	100 u	Specify the maximum of multipart name length
EX_HTTP_CLIENT_CFG_FORM_MULTIPART_FILENAME_LEN_MAX	100 u	Specify the maximum of multipart filename length

Location

> /examples/net/http/client/ex_http_client_files.c
>
> /examples/net/http/client/ex_http_client_files.h
>
> /examples/net/http/client/ex_http_client_hooks.c
>
> /examples/net/http/client/ex_http_client_hooks.h
>
> /examples/net/http/client/ex_http_client.c
>
> /examples/net/http/client/ex_http_client.h

API

API	Description
Ex_HTTP_Client_ReqSendPost()	Send a POST request with a pre-formatted form as body.
Ex_HTTP_Client_ReqSendAppForm()	Send an Application type form.
Ex_HTTP_Client_ReqSendMultipartForm()	Send a multipart type form.

Notes

None.

HTTP Client PUT Request Examples

Description

This is a generic example that shows how to send PUT request. It accomplishes the following tasks:

- Set connection option
- Open an HTTP connection
- Send PUT request
- Close HTTP Connection

Configuration

Optional

The following #define can be added to ex_description.h, as described in Example Applications section, to change default configuration value used by the example:

#define	Default value	Description
EX_HTTP_CLIENT_RAMDISK_SEC_SIZE	512 u	Specify the sector size of the RAM disk
EX_HTTP_CLIENT_RAMDISK_SEC_NBR	60 u	Specify the number of sector for the RAM disk
EX_HTTP_CLIENT_FILE_RAM_MEDIA_NAME	"ram_httpc"	Specify the RAM media name

#define	Default value	Description
EX_HTTP_CLIENT_FILE_VOL_NAME	"ram_httpc"	Specify the RAM volume name
EX_HTTP_CLIENT_WRK_DIR	EX_HTTP_CLIENT_FILE_VOL_NAME	Specify the working directory used by the example
EX_HTTP_CLIENT_FILE_NAME	"index.html"	Specify the name of the internal file.
EX_HTTP_CLIENT_CLIENT_HOSTNAME	"httpbin.org "	Specify the server host name used by the client
EX_HTTP_CLIENT_CONN_NBR_MAX	5 u	Specify the maximum of active connection
EX_HTTP_CLIENT_REQ_NBR_MAX	5 u	Specify the maximum of requests
EX_HTTP_CLIENT_CONN_BUF_SIZE	512 u	Specify the connection buffer size
EX_HTTP_CLIENT_CFG_QUERY_STR_NBR_MAX	6 u	Specify the maximum number of query string
EX_HTTP_CLIENT_CFG_QUERY_STR_KEY_LEN_MAX	20 u	Specify the maximum of query string key length
EX_HTTP_CLIENT_CFG_QUERY_STR_VAL_LEN_MAX	50 u	Specify the maximum of query string value length
EX_HTTP_CLIENT_CFG_HDR_NBR_MAX	6 u	Specify the maximum of header object
EX_HTTP_CLIENT_CFG_HDR_VAL_LEN_MAX	100 u	Specify the maximum of header value length
EX_HTTP_CLIENT_CFG_FORM_BUF_SIZE	256 u	Specify the form buffer size.
EX_HTTP_CLIENT_CFG_FORM_FIELD_NBR_MAX	10 u	Specify the maximum number of form's field
EX_HTTP_CLIENT_CFG_FORM_FIELD_KEY_LEN_MAX	100 u	Specify the maximum of form key length
EX_HTTP_CLIENT_CFG_FORM_FIELD_VAL_LEN_MAX	200 u	Specify the maximum of form value length

#define	Default value	Description
EX_HTTP_CLIENT_CFG_FORM_MULTIPART_NAME_LEN_MAX	100 u	Specify the maximum of multipart name length
EX_HTTP_CLIENT_CFG_FORM_MULTIPART_FILENAME_LEN_MAX	100 u	Specify the maximum of multipart filename length

Location

```
/examples/net/http/client/ex_http_client_files.c

/examples/net/http/client/ex_http_client_files.h

/examples/net/http/client/ex_http_client_hooks.c

/examples/net/http/client/ex_http_client_hooks.h

/examples/net/http/client/ex_http_client.c

/examples/net/http/client/ex_http_client.h
```

API

API	Description
Ex_HTTP_Client_ReqSendPut()	Send PUT request.

Notes

None.

HTTP Client Persistent Connection Examples

Description

This is a generic example that shows how to send multiple GET request on the same connection. It accomplishes the following tasks:

- Set connection option
- Open an HTTP connection
- Send GET request #1
- Send GET request #2
- Close HTTP Connection

Configuration

Optional

The following #define can be added to ex_description.h, as described in Example Applications section, to change default configuration value used by the example:

#define	Default value	Description
EX_HTTP_CLIENT_RAMDISK_SEC_SIZE	512 u	Specify the sector size of the RAM disk
EX_HTTP_CLIENT_RAMDISK_SEC_NBR	60 u	Specify the number of sector for the RAM disk
EX_HTTP_CLIENT_FILE_RAM_MEDIA_NAME	"ram_httpc"	Specify the RAM media name
EX_HTTP_CLIENT_FILE_VOL_NAME	"ram_httpc"	Specify the RAM volume name

#define	Default value	Description
EX_HTTP_CLIENT_WRK_DIR	EX_HTTP_CLIENT_FILE_VOL_NAME	Specify the working directory used by the example
EX_HTTP_CLIENT_FILE_NAME	"index.html"	Specify the name of the internal file.
EX_HTTP_CLIENT_CLIENT_HOSTNAME	"httpbin.org "	Specify the server host name used by the client
EX_HTTP_CLIENT_CONN_NBR_MAX	5 u	Specify the maximum of active connection
EX_HTTP_CLIENT_REQ_NBR_MAX	5 u	Specify the maximum of requests
EX_HTTP_CLIENT_CONN_BUF_SIZE	512 u	Specify the connection buffer size
EX_HTTP_CLIENT_CFG_QUERY_STR_NBR_MAX	6 u	Specify the maximum number of query string
EX_HTTP_CLIENT_CFG_QUERY_STR_KEY_LEN_MAX	20 u	Specify the maximum of query string key length
EX_HTTP_CLIENT_CFG_QUERY_STR_VAL_LEN_MAX	50 u	Specify the maximum of query string value length
EX_HTTP_CLIENT_CFG_HDR_NBR_MAX	6 u	Specify the maximum of header object
EX_HTTP_CLIENT_CFG_HDR_VAL_LEN_MAX	100 u	Specify the maximum of header value length
EX_HTTP_CLIENT_CFG_FORM_BUF_SIZE	256 u	Specify the form buffer size.
EX_HTTP_CLIENT_CFG_FORM_FIELD_NBR_MAX	10 u	Specify the maximum number of form's field
EX_HTTP_CLIENT_CFG_FORM_FIELD_KEY_LEN_MAX	100 u	Specify the maximum of form key length
EX_HTTP_CLIENT_CFG_FORM_FIELD_VAL_LEN_MAX	200 u	Specify the maximum of form value length
EX_HTTP_CLIENT_CFG_FORM_MULTIPART_NAME_LEN_MAX	100 u	Specify the maximum of multipart name length

#define	Default value	Description
EX_HTTP_CLIENT_CFG_FORM_MULTIPART_FILENAME_LEN_MAX	100 u	Specify the maximum of multipart filename length

Location

> /examples/net/http/client/ex_http_client_files.c
>
> /examples/net/http/client/ex_http_client_files.h
>
> /examples/net/http/client/ex_http_client_hooks.c
>
> /examples/net/http/client/ex_http_client_hooks.h
>
> /examples/net/http/client/ex_http_client.c
>
> /examples/net/http/client/ex_http_client.h

API

API	Description
Ex_HTTP_Client_PersistentConn()	Send multiple requests on same connection.

Notes

None.

HTTP Client Multi-connection Examples

Description

This is a generic example that shows how to open multiple connections and how to send requests in parallel. It accomplishes the following tasks:

- Set connection #1 option
- Set connection #2 option
- Open an HTTP connection #1
- Open an HTTP connection #2
- Send GET request #1
- Send GET request #2

Configuration

Optional

The following #define can be added to ex_description.h, as described in Example Applications section, to change default configuration value used by the example:

#define	Default value	Description
EX_HTTP_CLIENT_RAMDISK_SEC_SIZE	512 u	Specify the sector size of the RAM disk
EX_HTTP_CLIENT_RAMDISK_SEC_NBR	60 u	Specify the number of sector for the RAM disk
EX_HTTP_CLIENT_FILE_RAM_MEDIA_NAME	"ram_httpc"	Specify the RAM media name
EX_HTTP_CLIENT_FILE_VOL_NAME	"ram_httpc"	Specify the RAM volume name

#define	Default value	Description
EX_HTTP_CLIENT_WRK_DIR	EX_HTTP_CLIENT_FILE_VOL_NAME	Specify the working directory used by the example
EX_HTTP_CLIENT_FILE_NAME	"index.html"	Specify the name of the internal file.
EX_HTTP_CLIENT_CLIENT_HOSTNAME	"httpbin.org "	Specify the server host name used by the client
EX_HTTP_CLIENT_CONN_NBR_MAX	5 u	Specify the maximum of active connection
EX_HTTP_CLIENT_REQ_NBR_MAX	5 u	Specify the maximum of requests
EX_HTTP_CLIENT_CONN_BUF_SIZE	512 u	Specify the connection buffer size
EX_HTTP_CLIENT_CFG_QUERY_STR_NBR_MAX	6 u	Specify the maximum number of query string
EX_HTTP_CLIENT_CFG_QUERY_STR_KEY_LEN_MAX	20 u	Specify the maximum of query string key length
EX_HTTP_CLIENT_CFG_QUERY_STR_VAL_LEN_MAX	50 u	Specify the maximum of query string value length
EX_HTTP_CLIENT_CFG_HDR_NBR_MAX	6 u	Specify the maximum of header object
EX_HTTP_CLIENT_CFG_HDR_VAL_LEN_MAX	100 u	Specify the maximum of header value length
EX_HTTP_CLIENT_CFG_FORM_BUF_SIZE	256 u	Specify the form buffer size.
EX_HTTP_CLIENT_CFG_FORM_FIELD_NBR_MAX	10 u	Specify the maximum number of form's field
EX_HTTP_CLIENT_CFG_FORM_FIELD_KEY_LEN_MAX	100 u	Specify the maximum of form key length
EX_HTTP_CLIENT_CFG_FORM_FIELD_VAL_LEN_MAX	200 u	Specify the maximum of form value length
EX_HTTP_CLIENT_CFG_FORM_MULTIPART_NAME_LEN_MAX	100 u	Specify the maximum of multipart name length

#define	Default value	Description
EX_HTTP_CLIENT_CFG_FORM_MULTIPART_FILENAME_LEN_MAX	100 u	Specify the maximum of multipart filename length

Location

```
/examples/net/http/client/ex_http_client_files.c

/examples/net/http/client/ex_http_client_files.h

/examples/net/http/client/ex_http_client_hooks.c

/examples/net/http/client/ex_http_client_hooks.h

/examples/net/http/client/ex_http_client.c

/examples/net/http/client/ex_http_client.h
```

API

API	Description
Ex_HTTP_Client_MultiConn()	Open multiple Connections to send HTTP requests in parallel.

Notes

None.

HTTP Client Configuration

In order to address your application's needs, the HTTP Client module must be properly configured. There are two groups of configuration parameters:

- Compile time configurations
- Run-time application specific configurations

HTTP Client Compile-Time Configurations

Micrium OS HTTP Client is configurable at compile time via several #defines located in http_client_cfg.h file. HTTP Client module uses #defines when possible, because they allow code and data sizes to be scaled at compile time based on enabled features. This allows the Read-Only Memory (ROM) and Random-Access Memory (RAM) footprints of the HTTP Client module to be adjusted based on application requirements.

It is recommended that the configuration process begins with the default configuration values which in the next sections will be shown in **bold**.

Task Configuration

HTTP Client module has an internal task that can be enabled or disabled. Using this internal task allows for simultaneous connection processing and using the API in a non-blocking mode.

Table - HTTPc Task Configuration

Constant	Description	Possible Values
HTTPc_CFG_MODE_ASYNC_TASK_EN	Enables/Disables the internal asynchronous task.When ENABLED, the internal task can accept simultaneous connections and can queue requests. API functions can be called with the non-blocking flag.When DISABLED, the API functions will always be blocking.	DEF_ENABLED or DEF_DISABLED
HTTPc_CFG_MODE_BLOCK_EN	Enables/Disables the blocking option when the internal task is active.When the internal task is enabled, API functions can also be blocking but only if this configuration is enabled.	DEF_ENABLED or DEF_DISABLED

Persistent Connection Configuration

Table - HTTPc Persistent Connection Configuration

Constant	Description	Possible Values
HTTPc_CFG_PERSISTENT_EN	Enables/Disables the Persistent Connection Feature.	DEF_ENABLED or DEF_DISABLED

Chunked Transfer Encoding Configuration

Table - HTTPc Chunked Transfer Encoding Configuration

Constant	Description	Possible Values
HTTPc_CFG_CHUNK_TX_EN	Enables/Disables the Chunked Transfer Encoding feature for transmission	DEF_ENABLED or DEF_DISABLED

Query String Configuration

Table - HTTPc Query String Configuration

Constant	Description	Possible Values
HTTPc_CFG_QUERY_STR_EN	Enables/Disables the Query String Feature.	DEF_ENABLED or DEF_DISABLED

Header Field Configuration

Table - HTTPc Header Field Configuration

Constant	Description	Possible Values
HTTPc_CFG_HDR_RX_EN	Enables/Disables the addition of header fields to HTTP Requests when transmitting.	DEF_ENABLED or DEF_DISABLED
HTTPc_CFG_HDR_TX_EN	Enables/Disables the copy and processing of header fields for HTTP Responses received.	DEF_ENABLED or DEF_DISABLED

Form Submission Configuration

Table - HTTPc Form Submission Configuration

Constant	Description	Possible Values
HTTPc_CFG_FORM_EN	Enables/Disables the HTTP Form Submission Feature.	DEF_ENABLED or DEF_DISABLED

User Data Configuration

Table - HTTPc User Data Configuration

Constant	Description	Possible Values
HTTPc_CFG_USER_DATA_EN	Enables/Disables the addition of a void pointer in the HTTPc_CONN_OBJ and HTTPc_REQ_OBJ objects.The additional void pointer can be used by the upper application to store an application data pointer relative to the object.	DEF_ENABLED or DEF_DISABLED

WebSocket Configuration

Table - HTTPc WebSocket Configuration

Constant	Description	Possible Values
HTTPc_CFG_WEBSOCKET_EN	Enables/Disables the WebSocket Feature.	DEF_ENABLED or DEF_DISABLED

> The WebSocket feature requires HTTPc_CFG_MODE_ASYNC_TASK_EN to be set to DEF_ENABLED.

HTTP Client Run-Time Application Specific Configurations

This section defines the configurations related to HTTP Client but that are specified at run-time, during the initialization process.

- Initialization
- Optional configurations
- Post-Init Configurations

Initialization

Initializing the HTTP Client module is done by calling the function HTTPc_Init(). This function takes no configuration argument. Unless you override an optional configuration before calling the function HTTPc_Init(), the default configurations will be used.

Optional configurations

This section describes the configurations that are optional. If you do not set them in your application, the default configurations will apply.

The default values can be retrieved via the structure HTTPc_InitCfgDflt.

> Note that these configurations must be set *before* you call the function HTTPc_Init().

Table - HTTP Client Optional Configurations

Configurations	Description	Type	Function to call	Default	Field from default configuration structure
Task's stack	The HTTP client module can use a task depending on the compile-time configuration. This configuration allows you to set the stack pointer and the stack size (in quantity of elements).	CPU_INT32Uvoid *	HTTPc_ConfigureTaskStk()	A stack of 768 elements allocated on Common 's memory segment.	.StkSizeElements.StkPtr
Memory segment	This module allocates some control data. You can specify the memory segment from where such data should be allocated.	MEM_SEG *	HTTPc_ConfigureMemSeg()	General-purpose heap .	.MemSegPtr
Quantity parameters	The HTTP client task use a message queue. You can overwrite the maximum element in the message pool.	HTTPc_QTY_CFG	HTTPc_ConfigureQty()	Unlimited queue size.	.QtyCfg
Connection parameters	The HTTP client allows to configure the connection timeouts.	HTTPc_CONN_CFG	HTTPc_ConfigureConnParam()	Timeout of 30 milliseconds for connection timeout and 30 seconds for inactivity.	.ConnCfg

Post-Init Configurations

This section describes the configurations that can be set at any time during execution *after* you called the function HTTPc_Init().

These configurations are optional. If you do not set them in your application, the default configurations will apply.

Table - HTTP Client Post-init Configurations

Configurations	Description	Type	Function to call	Default
Task priority	The HTTP client module will create a task that handles the HTTP requests. You can change the priority of the created task at any time.	RTOS_TASK_PRIO	HTTPc_TaskPrioSet()	See Appendix A - Internal Tasks .
Task Delay	If the HTTP client module uses a task, you can change the delay inside this task to allow a period of time for other tasks to run.	CPU_INT08U	HTTPc_TaskDlySet()	1u

HTTP Client Connection Configurations

Table - HTTPc_CONN_CFG configuration structure in the *HTTP Client Connection Configurations* page describes each configuration field available in this configuration structure.

Table - HTTPc_CONN_CFG configuration structure

Field	Description
.ConnConnectTimeout_ms	Connection connect timeout, in milliseconds.
.ConnInactivityTimeout_s	Connection inactivity timeout, in seconds.

HTTP Client Quantity Configurations

Table - HTTPc_QTY_CFG configuration structure in the *HTTP Client Quantity Configurations* page describes each configuration field available in this configuration structure.

Table - HTTPc_QTY_CFG configuration structure

Field	Description
.MsgQ_Size	Maximum size of the message queue.

HTTP Client Programming Guide

This Section regroups topics to help developing a custom HTTP client application with the MIcrium OS HTTP Client module. Examples are include in many sub-sections.

- HTTP Client Control Structure
- HTTP Client Connection Object Setup
- HTTP Client Request Object Setup
- HTTP Client Open Secure Connection with SSL-TSL
- HTTP Client Add Additional Header fields to an HTTP Request
- HTTP Client Request's Body Standard Transfer
- HTTP Client Chunked Transfer Encoding
- HTTP Client Query String
- HTTP Client Form Submission
- HTTP Client Persistent Connection
- HTTP Client Retrieve HTTP Response Data
- HTTP Client WebSocket

HTTP Client Control Structure

The HTTP Client module offers many structures that the application can use to create objects to interact with the API.

All the HTTP Client Objects required by the application must be allocated by the application and passed to the HTTP Client API.

> All objects or strings passed to the HTTP Client API MUST be persistent and unmodified for the duration of the HTTP Transaction for Request-oriented parameters and objects; or until the HTTP connection is closed for Connection-oriented parameters and objects.

HTTP Client Connection (HTTPc_CONN_OBJ)

This structure is the main one used by the HTTP Client module. It contains all parameters relative to an HTTP connection (server port number, server address, server host name, etc.) and also many internal parameters for the HTTP connection and HTTP transaction processing.

Each configurable parameter SHOULD be set up with the function HTTPc_ConnSetParam(). The list of available parameters for a connection can be viewed here .

The members of an HTTP Client object should never be directly tampered with at any time. To ensure this, the HTTPc_CONN_OBJ structure's members have all been declared constant with the *const* keyword and the suffix *_reserved* has been added.

Listing - HTTPc_CONN Structure

```
struct httpc_conn_obj {
                                            /* --------- CONNECTION PARAMETERS --------- */
  const  NET_SOCK_ID              SockID_reserved;           /* Connection's Socket ID.                */
  const  HTTPc_FLAGS              SockFlags_reserved;        /* Connection's Socket flags.             */
#ifdef NET_SECURE_MODULE_EN
  const  NET_APP_SOCK_SECURE_CFG   SockSecureCfg_reserved;    /* Connection's Socket Secure Cfg.        */
#endif
  const  CPU_INT16U               ConnectTimeout_ms_reserved;  /* Connection Connect Timeout.          */
  const  CPU_INT16U               InactivityTimeout_s_reserved; /* Connection Inactivity Timeout.      */
  const  NET_PORT_NBR             ServerPort_reserved;
  const  NET_SOCK_ADDR            ServerSockAddr_reserved;    /* Server socket address.               */
  const  CPU_CHAR                *HostNamePtr_reserved;       /* Pointer to HTTP server hostname string.  */
  const  CPU_INT16U               HostNameLen_reserved;
  const  HTTPc_CONN_STATE         State_reserved;             /* Connection State.                    */
  const  HTTPc_FLAGS              Flags_reserved;             /* Set of flags related to HTTP connection.  */
  const  HTTPc_CONN_CLOSE_STATUS  CloseStatus_reserved;       /* Status of connection closed.          */
  const  HTTPc_ERR                ErrCode_reserved;           /* Error code of connection.             */
#ifdef HTTPc_TASK_MODULE_EN
  const  HTTPc_CONNECT_CALLBACK    OnConnect_reserved;        /* Connection Connect callback function.   */
  const  HTTPc_CONN_CLOSE_CALLBACK OnClose_reserved;          /* Connection Close callback function.     */
  const  HTTPc_CONN_ERR_CALLBACK   OnErr_reserved;            /* Connection Error callback function.     */
#endif
#ifdef HTTPc_SIGNAL_TASK_MODULE_EN
  const  KAL_SEM_HANDLE           ConnectSignal_reserved;     /* HTTP Socket Connect Done Signal.       */
  const  KAL_SEM_HANDLE           TransDoneSignal_reserved;   /* HTTP Transaction Done Signal.          */
  const  KAL_SEM_HANDLE           CloseSignal_reserved;       /* HTTP Socket Close Done Signal.         */
#endif
                                            /* ----------- REQUEST PARAMETERS ---------- */
  const  HTTPc_REQ               *ReqListHeadPtr_reserved;    /* Head of the Request list.             */
  const  HTTPc_REQ               *ReqListEndPtr_reserved;     /* End of the Request list.              */
  const  HTTPc_FLAGS              ReqFlags_reserved;          /* Set of flags related to the HTTP request. */
#if (HTTPc_CFG_QUERY_STR_EN == DEF_ENABLED)
  const  CPU_INT16U               ReqQueryStrTxIx_reserved;   /* Tbl index of the transmitted strings.   */
  const  HTTPc_KEY_VAL           *ReqQueryStrTempPtr_reserved; /* Temporary Query String to Tx.          */
#endif
#if (HTTPc_CFG_HDR_TX_EN == DEF_ENABLED)
  const  CPU_INT08U               ReqHdrTxIx_reserved;        /* Index in Req hdr table to Tx headers.   */
  const  HTTP_HDR                *ReqHdrTempPtr_reserved;     /* Temporary Header field to Tx.           */
#endif
#if (HTTPc_CFG_FORM_EN == DEF_ENABLED)
  const  CPU_INT16U               ReqFormDataTxIx_reserved;   /* Index in form table to Tx form fields.   */
#endif
  const  CPU_INT16U               ReqDataOffset_reserved;     /* Offset in Request Data Pointer to Tx data */
                                            /* ----------- RESPONSE PARAMETERS --------- */
  const  HTTPc_FLAGS              RespFlags_reserved;         /* Set of flags related to the HTTP response.*/
                                            /* ------ CONNECTION BUFFER PARAMETERS ----- */
```

```
  const  void                *TxDataPtr_reserved;/* Pointer to data to transmit.            */
  const  CPU_CHAR            *BufPtr_reserved;/* Pointer to conn buffer.                  */
  const  CPU_INT16U           BufLen_reserved;/* Conn buffer length.                      */
  const  CPU_CHAR            *RxBufPtr_reserved;/* Pointer to Buffer where to Rx data.       */
  const  CPU_INT16U           RxDataLenRem_reserved;/* Remaining data in the RX buffer.          */
  const  CPU_INT32U           RxDataLen_reserved;/* Data length received.                    */
  const  CPU_CHAR            *TxBufPtr_reserved;/* Pointer to Buffer where to Tx data.       */
  const  CPU_INT16U           TxDataLen_reserved;/* Length of data to Tx.                    *//* ------- CONNECTION LIST PARAMETERS ------ */
  const  HTTPc_CONN          *NextPtr_reserved;/* Pointer to next connection.               */
#if(HTTPc_CFG_USER_DATA_EN == DEF_ENABLED)/* ---------- USER DATA PARAMETER ---------- */
         void                *UserDataPtr;
#endif
};
```

HTTP Client Request (HTTPc_REQ_OBJ)

This structure regroups parameters and flags related to the configuration of an HTTP request.

Each configurable parameter SHOULD be set up with the function HTTPc_ReqSetParam(). The list of available parameters for a connection can be viewed here .

The members of an HTTP Request object should never be directly tempered with at any time. To ensure this, the HTTPc_REQ_OBJ structure's members have all been declared constant with the *const* keyword and the suffix *_reserved* has been added.

Listing - HTTPc_REQ Structure

```
struct httpc_req_obj {
  const  HTTPc_FLAGS              Flags_reserved;
  const  HTTPc_FLAGS              HdrFlags_reserved;
  const  HTTP_METHOD              Method_reserved;
  const  CPU_CHAR                *ResourcePathPtr_reserved;
  const  CPU_INT16U               ResourcePathLen_reserved;
  const  HTTP_CONTENT_TYPE        ContentType_reserved;
  const  CPU_INT32U               ContentLen_reserved;
  const  void                    *DataPtr_reserved;
#if (HTTPc_CFG_QUERY_STR_EN == DEF_ENABLED)
  const  HTTPc_KEY_VAL           *QueryStrTbl_reserved;
  const  CPU_INT16U               QueryStrNbr_reserved;
  const  HTTPc_REQ_QUERY_STR_HOOK  OnQueryStrTx_reserved;
#endif
#if (HTTPc_CFG_HDR_TX_EN == DEF_ENABLED)
  const  HTTP_HDR                *HdrTbl_reserved;
  const  CPU_INT16U               HdrNbr_reserved;
  const  HTTPc_REQ_HDR_HOOK       OnHdrTx_reserved;
#endif
#if (HTTPc_CFG_FORM_EN == DEF_ENABLED)
  const  HTTPc_FORM_TBL_FIELD    *FormFieldTbl_reserved;
  const  CPU_INT16U               FormFieldNbr_reserved;
#endif
#if (HTTPc_CFG_CHUNK_TX_EN == DEF_ENABLED)
  const  HTTPc_REQ_BODY_HOOK      OnBodyChunkTx_reserved;
#endif
#if (HTTPc_CFG_HDR_RX_EN == DEF_ENABLED)
  const  HTTPc_RESP_HDR_HOOK      OnHdrRx_reserved;        /* Resp Hdr Hook fnct to choose which hdr to keep*/
#endif
  const  HTTPc_RESP_BODY_HOOK     OnBodyRx_reserved;       /* Resp Body Hook fnct to pass Rx data to app.   */
#ifdef HTTPc_TASK_MODULE_EN
  const  HTTPc_COMPLETE_CALLBACK  OnTransComplete_reserved; /* Response received callback function.          */
  const  HTTPc_TRANS_ERR_CALLBACK  OnErr_reserved;          /* Transaction Error callback function.          */
#endif
  const  HTTPc_CONN              *ConnPtr_reserved;        /* Pointer to Connection Object.                 */
  const  HTTPc_RESP              *RespPtr_reserved;        /* Pointer to Conn Response Object.              */
  const  HTTPc_REQ               *NextPtr_reserved;
#if (HTTPc_CFG_USER_DATA_EN == DEF_ENABLED)
         void                    *UserDataPtr;
#endif
};
```

HTTP Client Response (HTTPc_RESP_OBJ)

This structure will be filled by the HTTP Client core with the data received in the HTTP response; except for the body part that is retrieved by the application with the hook function On Response Body .

Listing - HTTPc_RESP Structure

```
struct httpc_resp_obj {
         HTTP_PROTOCOL_VER        ProtocolVer;        /* HTTP version received in response message.   */
         HTTP_STATUS_CODE         StatusCode;         /* Status code received in response.            */
   const  CPU_CHAR               *ReasonPhrasePtr;    /* Pointer to received reason phrase.           */
         HTTP_CONTENT_TYPE        ContentType;        /* Content type received in response.           */
         CPU_INT32U               ContentLen;         /* Content length received in response if any.  */
}
```

HTTP Client Key-Value Pair (HTTPc_KEY_VAL)

Key-Value Pair Objects can be used with the HTTP Query String feature and the HTTP Form feature.

The structure's object allows for storing the pointer to an application's Key string and the pointer to its Value string's equivalent key. The string's length must also be saved in the object.

Listing - HTTPc_KEY_VAL Structure

```
struct  httpc_key_val {
    CPU_CHAR      *KeyPtr;   /* Pointer to Key String.   */
    CPU_INT16U     KeyLen;   /* Length of Key String.    */
    CPU_CHAR      *ValPtr;   /* Pointer to Value String. */
    CPU_INT16U     ValLen;   /* Length of Value String.  */
};
```

HTTP Client Extended Key-Value Pair (HTTPc_KEY_VAL_EXT)

Extended Key-Value Pair Objects can be used with the HTTP Form feature. More specifically, with a multipart type form.

Extended Key-Value Pair Objects are very similar to the Key-Value Pair Objects except that the value pointer is replaced by a hook function that will allow the application to directly copy the value into the HTTP Client buffer.

Listing - HTTPc_KEY_VAL_EXT Structure

```
struct  httpc_key_val_ext {
    CPU_CHAR               *KeyPtr;    /* Pointer to Key String.                 */
    CPU_INT16U              KeyLen;    /* Length of Key String.                  */
    HTTPc_KEY_VAL_EXT_HOOK  OnValTx;   /* Pointer to Hook Function to set Value. */
    CPU_INT32U              ValLen;    /* Length of the Value String.            */
};
```

HTTP Client Multipart File (HTTPc_MULTIPART_FILE)

Multipart File Objects can be used with the HTTP Form feature. More specifically, with a multipart type form.

The structure's object allows it to store a pointer to the strings file name, the HTTP Content Type of the file and the hook function pointer that will be used to read the file into the transmit buffer.

Listing - HTTPc_MULTIPART_FILE Structure

```
struct  http_multipart_file {
    CPU_CHAR                   *NamePtr;       /* Pointer to Name of form field.                 */
    CPU_INT16U                  NameLen;       /* Length of the form field's name.               */
    CPU_CHAR                   *FileNamePtr;   /* Pointer to File Name String.                   */
    CPU_INT16U                  FileNameLen;   /* Length of the File Name String.                */
    CPU_INT32U                  FileLen;       /* File Length.                                   */
    HTTP_CONTENT_TYPE           ContentType;   /* HTTP Content Type of the file.                 */
    HTTPc_MULTIPART_FILE_HOOK   OnFileTx;      /* Pointer to Hook Function to Transmit file's data. */
};
```

HTTP Client Header (HTTPc_HDR)

This structure is used by the application to add Header Fields to an HTTP request.

Listing - HTTP_HDR Structure

```
struct  httpc_hdr {
    HTTP_HDR_FIELD    HdrField;   /* HTTP Header Field Type.         */
    CPU_CHAR         *ValPtr;     /* Pointer to Header Field Value. */
    CPU_INT16U        ValLen;     /* Length of the value.            */
};
```

HTTP Client Connection Object Setup

Connection's Parameters

HTTP Client provides the function HTTPc_ConnSetParam() to configure parameters related to the HTTP Client Connection object . The function takes as argument the type of parameter and the pointer to the parameter. The below parameter's types are available:

Table - HTTPc Connection's Parameters

Parameter Type	Description
HTTPc_PARAM_TYPE_SERVER_PORT	Sets a specific port for the remote HTTP server.By default, the port is set to 80 or to 443 when the Secure module is used.
HTTPc_PARAM_TYPE_PERSISTENT	Enables the persistent connection mode.The HTTP client will not close the connection at the end of a transaction and will also notify the HTTP server to do the same. The HTTP server must also support persistent connections for the mode to be functional. Enabling this feature on the HTTP client is no guaranty that a persistent connection with the server will be established but that a best effort will be made.
HTTPc_PARAM_TYPE_CONNECT_TIMEOUT	Sets the Connect Timeout of the connection.This value represents the time allowed for the connection process to complete.
HTTPc_PARAM_TYPE_INACTIVITY_TIMEOUT	Sets the Inactivity Timeout of the connectionThis value represents the time of inactivity allowed on a connection before it is automatically closed.
HTTPc_PARAM_TYPE_SECURE_COMMON_NAME	Sets the Common Name linked to the HTTP server's SSL Certificate.The Common Name is associated with the SSL Certificate of a web server. The host + domain name is commonly used for the Common Name, ex: "www.example.com " or "example.com ". When the HTTP client attempts a secure connection, by default the host name will be used as Common Name. You can use this parameter to specify a different Common Name for a secure connection.
HTTPc_PARAM_TYPE_SECURE_TRUST_CALLBACK	Sets the callback function to notify the application when a secure connection with a server was made to ensure that server certificate is trustworthy.This parameter is mandatory if a secure connection to the server is desired. Therefore, when this parameter is set up, the connection with the server will be automatically a secure one.
HTTPc_PARAM_TYPE_CONN_CONNECT_CALLBACK	Sets the callback function to notify the application when a connection attempt with the HTTP server is completed.This parameter is mandatory when the HTTPc_CFG_MODE_ASYNC_TASK_EN configuration is enabled and when the function HTTPc_ConnOpen() is called in the none-blocking mode.
HTTPc_PARAM_TYPE_CONN_CLOSE_CALLBACK	Sets the callback function to notify the application when an HTTP connection was closed.A connection can be closed at any moment by the HTTP server, or by the client when an unexpected error occurred. A connection will also be closed after any transaction when the persistent mode is not enabled.This parameter is mandatory when the HTTPc_CFG_MODE_ASYNC_TASK_EN configuration is enabled.

Example

Listing - HTTP Client Connection

```
CPU_BOOLEAN  HTTPcApp_ConnPrepare (HTTPc_CONN_OBJ  *p_conn)
{
 CPU_BOOLEAN  persistent;
 RTOS_ERR     err;

                        /* ---------------- INIT NEW CONNECTION --------------- */
  HTTPc_ConnClr(p_conn, &err);
  if (err.Code != RTOS_ERR_NONE) {
     return (DEF_FAIL);
  }
                                              /* ----------------- SET SERVER PORT ------------------ */
  port = 8080;
  HTTPc_ConnSetParam(p_conn,
                   HTTPc_PARAM_TYPE_SERVER_PORT,
                  &port,
                  &err);
  if (err.Code != RTOS_ERR_NONE) {
     return (DEF_FAIL);
  }

                        /* --------------- SET PERSISTENT MODE ---------------- */
  persistent = DEF_YES;
  HTTPc_ConnSetParam(p_conn,
                   HTTPc_PARAM_TYPE_PERSISTENT,
                  &persistent,
                  &err);
  if (err.Code != RTOS_ERR_NONE) {
     return (DEF_FAIL);
  }

                        /* --------------- SET CONN'S CALLBACKS --------------- */
#if (HTTPc_CFG_MODE_ASYNC_TASK_EN == DEF_ENABLED)
  HTTPc_ConnSetParam(p_conn,
                   HTTPc_PARAM_TYPE_CONN_CONNECT_CALLBACK,
                  &AppHTTPc_ConnConnectCallback,
                  &err);
  if (err.Code != RTOS_ERR_NONE) {
     return (DEF_FAIL);
  }

  HTTPc_ConnSetParam(p_conn,
                   HTTPc_PARAM_TYPE_CONN_CLOSE_CALLBACK,
                  &AppHTTPc_ConnCloseCallback,
                  &err);
  if (err.Code != RTOS_ERR_NONE) {
     return (DEF_FAIL);
  }
#endif

 return (DEF_OK);
}
```

HTTP Client Request Object Setup

Request's Parameters

HTTP Client provides the function HTTPc_ReqSetParam() to configure parameters related to the HTTP Request. The function takes as argument the type of parameter and the pointer to the parameter. The below parameter's types are available:

Table - HTTPc Request's Parameters

Parameter Type	Description
HTTPc_PARAM_TYPE_REQ_QUERY_STR_TBL	Sets the Request's Query String Table.
HTTPc_PARAM_TYPE_REQ_QUERY_STR_HOOK	Sets the Request's Query String Hook Function.
HTTPc_PARAM_TYPE_REQ_HDR_TBL	Sets the Request's Header Table.
HTTPc_PARAM_TYPE_REQ_HDR_HOOK	Sets the Request's Header Hook Function.
HTTPc_PARAM_TYPE_REQ_FORM_TBL	Sets the Request's Form Table.
HTTPc_PARAM_TYPE_REQ_BODY_CONTENT_TYPE	Sets the Request's Body Content Type.
HTTPc_PARAM_TYPE_REQ_BODY_CONTENT_LEN	Sets the Request's Body Content Length.
HTTPc_PARAM_TYPE_REQ_BODY_HOOK	Sets the Request's Hook Function for retrieving the data body.
HTTPc_PARAM_TYPE_RESP_HDR_HOOK	Sets the Response's Header Hook Function.
HTTPc_PARAM_TYPE_RESP_BODY_HOOK	Sets the Response's Hook Function for the Body received.
HTTPc_PARAM_TYPE_TRANS_COMPLETE_CALLBACK	Sets the HTTP Transaction Complete Callback to notify the application that the Transaction is completed.This parameter is mandatory when the HTTPc_CFG_MODE_ASYNC_TASK_EN configuration is enabled.
HTTPc_PARAM_TYPE_TRANS_ERR_CALLBACK	Sets the HTTP Transaction Error Callback to notify the application when an unexpected error occurred with an HTTP Transaction.This parameter is mandatory when the HTTPc_CFG_MODE_ASYNC_TASK_EN configuration is enabled.

Example

Listing - Request Setup Example Code

```
CPU_BOOLEAN  HTTPcApp_ReqPrepare (HTTPc_CONN_OBJ  *p_req)
{
 RTOS_ERR  err;

                              /* ------------- INIT NEW REQUEST ------------ */
   HTTPc_ReqClr(p_req, &err);
   if (err.Code != RTOS_ERR_NONE) {
      return (DEF_FAIL);
   }

                                              /* -------- SET STRING QUERY PARAMETERS ------ */
   HTTPc_ReqSetParam(p_req,
                     HTTPc_PARAM_TYPE_REQ_QUERY_STR_HOOK,
                    &HTTPcApp_ReqQueryStrHook,
                    &err);
   if (err.Code != RTOS_ERR_NONE) {
      return (DEF_FAIL);
   }
                                              /* ------ SET REQUEST ADDITIONAL HEADERS ----- */
   HTTPc_ReqSetParam(p_req,
                     HTTPc_PARAM_TYPE_REQ_HDR_HOOK,
                    &HTTPcApp_ReqHdrHook,
                    &err);
   if (err.Code != RTOS_ERR_NONE) {
      return (DEF_FAIL);
   }

                                              /* -------- SET REQ/RESP HOOK FUNCTIONS ------ */
   HTTPc_ReqSetParam(p_req,
                     HTTPc_PARAM_TYPE_RESP_HDR_HOOK,
                    &HTTPcApp_RespHdrHook,
                    &err);
   if (err.Code != RTOS_ERR_NONE) {
      return (DEF_FAIL);
   }
```

```
HTTPc_ReqSetParam(p_req,
            HTTPc_PARAM_TYPE_RESP_BODY_HOOK,&HTTPcApp_RespBodyHook,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}/* -----
SET REQ/RESP CALLBACK FUNCTIONS ----- */
#if(HTTPc_CFG_MODE_ASYNC_TASK_EN == DEF_ENABLED)HTTPc_ReqSetParam(p_req,
            HTTPc_PARAM_TYPE_TRANS_COMPLETE_CALLBACK,&HTTPcApp_TransDoneCallback,&err);if(err.Code != RTOS_ERR_NONE)
{return(DEF_FAIL);}HTTPc_ReqSetParam(p_req,
            HTTPc_PARAM_TYPE_TRANS_ERR_CALLBACK,&HTTPcApp_TransErrCallback,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}
#endif

  return(DEF_OK);}
```

HTTP Client Open Secure Connection with SSL-TSL

Requirements

To open a secure HTTP connection with the HTTP Client module, the below requirements are needed:

- A network security module (such as Mocana - NanoSSL) is required.
- The Micrium OS Network core stack needs to be configured accordingly (see section Transport Layer Security Configuration).
- The client side needs to install certificate authorities to authenticate the identity of each public key certificate sent by servers.

Refer to section Secure Sockets (TLS or SSL) for an example on how to use the Network module to open a secure socket for a client application.

HTTP Client parameters

HTTP Client offers two parameters to configure the secure connection:

- HTTPc_PARAM_TYPE_SECURE_COMMON_NAME
- HTTPc_PARAM_TYPE_SECURE_TRUST_CALLBACK

The first parameter is optional and gives the option to specify the Common Name linked to the Secure Certificate to identify it with the certificate authorities. If this parameter is not set the server host name will be used.

The second parameter is mandatory to open a secure connection and therefore when this parameter is specified, the HTTP Client stack will assume that the connection to open must be secured. The parameter specified the hook function used by the secure module to ask the upper application to verify and validate the public key certificate send by the server.

Both parameters must be configured with the function HTTPc_ConnSetParam().

Example

Listing - Secure Connection Example Code

```
HTTPc_CONN_OBJ  conn;

/*
*********************************************************************************************************
*                                   HTTPcEx_ConnPrepare()
*
* Description : Example function to prepare the HTTPc Connection.
*
* Argument(s) : p_conn  Pointer to HTTPc Connection object to set up.
*
* Return(s)   : DEF_YES, if Connection preparation successful.
*             DEF_NO,  otherwise.
*********************************************************************************************************
*/

CPU_BOOLEAN  HTTPcEx_ConnPrepare (HTTPc_CONN_OBJ  *p_conn)
{
 RTOS_ERR  err;

                                                      /* ------------ INIT NEW CONNECTION ---------- */
   HTTPc_ConnClr(p_conn, &err);
   if (err.Code != RTOS_ERR_NONE) {
      return (DEF_FAIL);
   }

                                                      /* ----------- SET CONN'S CALLBACKS ---------- */
#if (HTTPc_CFG_MODE_ASYNC_TASK_EN == DEF_ENABLED)
   HTTPc_ConnSetParam(p_conn,
                  HTTPc_PARAM_TYPE_CONN_CONNECT_CALLBACK,
                  &HTTPcEx_ConnConnectCallback,
                  &err);
   if (err.Code != RTOS_ERR_NONE) {
      return (DEF_FAIL);
   }

   HTTPc_ConnSetParam(p_conn,
                  HTTPc_PARAM_TYPE_CONN_CLOSE_CALLBACK,
                  &HTTPcEx_ConnCloseCallback,
                  &err);
   if (err.Code != RTOS_ERR_NONE) {
      return (DEF_FAIL);
   }
#endif

                                                      /* ----- SET CONNECTION SECURE PARAMETERS ---- */
 HTTPc_ConnSetParam(p_conn,
                  HTTPc_PARAM_TYPE_SECURE_TRUST_CALLBACK
                  &HTTPcEx_ConnSecureCallback,
                  &err);
   if (err.Code != RTOS_ERR_NONE) {
      return (DEF_FAIL);
   }

   return (DEF_OK);
}

/*
*********************************************************************************************************
*                                   HTTPcEx_ConnCloseCallback()
*
* Description : Example Callback Function to validate public key from server.
*
* Argument(s) : p_cert_dn   Pointer to certificate distinguished name.
*
*             reason      Reason why certificate is not trusted:
*                         NET_SOCK_SECURE_UNTRUSTED_BY_CA
*                         NET_SOCK_SECURE_EXPIRE_DATE
```

```
*                       NET_SOCK_SECURE_INVALID_DATE
*                       NET_SOCK_SECURE_SELF_SIGNED
*
* Return(s)   : DEF_YES, if certificate is valid.
*               DEF_NO,  otherwise.
*********************************************************************************************************
*/

CPU_BOOLEAN  HTTPcEx_ConnCloseCallback (void                               *p_cert_dn,
                                        NET_SOCK_SECURE_UNTRUSTED_REASON   reason){return(DEF_YES);}
```

HTTP Client Add Additional Header fields to an HTTP Request

HTTP Client supports the addition of header fields to an HTTP request:

- The preprocessor macro HTTPc_CFG_HDR_TX_EN must be set to DEF_ENABLED.
- HTTPc_HDR objects must be used to set up headed fields to add.
- Two methods are offered by the HTTP Client stack to add header fields:
  - Passing a header fields table to to the HTTP Client stack.
  - Using a hook function that will ask the application for each header field to add to the request.

Some header fields are taken care of by the HTTP Client module and must therefore not be added by the application :

- Host
- Content-Type
- Content-Length
- Transfer-Encoding
- Connection

Using a Header Fields Table

With the first option, before sending the HTTP request, the application can prepare a table of HTTPc_HDR objects that the HTTP Client will include in the request. The API function HTTPc_ReqSetParam() must be used with the parameter type HTTPc_PARAM_TYPE_REQ_HDR_TBL to pass out the table to the HTTP Client stack.

Listing - Adding Header Fields with a Table Example Code

```
#define  HTTPc_CFG_HDR_NBR_MAX      10
#define  HTTPc_CFG_HDR_VAL_LEN_MAX  100

HTTPc_HDR      HTTPc_ReqHdrTbl[HTTPc_CFG_HDR_NBR_MAX];
CPU_CHAR       HTTPc_ReqHdrValTbl[HTTPc_CFG_HDR_NBR_MAX][HTTPc_CFG_HDR_VAL_LEN_MAX];

/*
*********************************************************************************************************
*                                         HTTPc_ReqPrepareHdr()
*
* Description : Prepare the HTTP header table.
*
* Argument(s) : p_tbl    Variable that will received the pointer to header table.
*
* Return(s)   : Number of Header fields in the table.
*********************************************************************************************************
*/
static  CPU_INT08U  HTTPc_ReqPrepareHdr (HTTPc_KEY_VAL  **p_tbl)
{
  HTTPc_HDR  *p_hdr;

    p_hdr           = &HTTPc_ReqHdrTbl[0];
    p_hdr->ValPtr   = &HTTPc_ReqHdrValTbl[0][0];
    p_hdr->HdrField =  HTTP_HDR_FIELD_ACCEPT;

    Str_Copy_N(p_hdr->ValPtr, "text/*", HTTPc_CFG_HDR_VAL_LEN_MAX);

    p_hdr->ValLen = Str_Len_N(p_hdr->ValPtr, HTTPc_CFG_HDR_VAL_LEN_MAX);

   *p_ext_hdr_tbl = &HTTPc_ReqHdrTbl[0];

    return (1);
}

/*
*********************************************************************************************************
*                                         HTTPc_ReqPrepare()
*
* Description : Prepare the HTTP request to send.
*
* Argument(s) : p_req    Pointer to the HTTP Request object to configure.
*
* Return(s)   : DEF_OK,   if the request was configured successfully.
*               DEF_FAIL, otherwise.
*********************************************************************************************************
*/
static  CPU_BOOLEAN  HTTPc_ReqPrepare (HTTPc_REQ_OBJ  *p_req)
{
    HTTPc_PARAM_TBL   tbl_obj;
    HTTPc_HDR        *p_hdr_tbl;
    CPU_INT08U        hdr_nbr;
    HTTPc_ERR         err;

                                             /* ---------- SET HEADER FIELDS DATA --------- */
    hdr_nbr = HTTPc_ReqPrepareHdr(&p_hdr_tbl);
    if (hdr_nbr <= 0) {
        return (DEF_FAIL);
    }

                                             /* -------- SET HEADER FIELDS PARAMETER ------ */
    tbl_obj.EntryNbr = hdr_nbr;
    tbl_obj.TblPtr   = (void *)p_hdr_tbl;
    HTTPc_ReqSetParam(p_req, HTTPc_PARAM_TYPE_REQ_HDR_TBL, &tbl_obj, &err);
    if (err != HTTPc_ERR_NONE) {
        return (DEF_FAIL);
    }
```

```
.../* TODO other operations on request if necessary. */return(DEF_OK);
```

Using a Hook Function

The Second option is to set up a hook function. The hook function will be called by the HTTP Client core to add a Header Field to the request. The hook function will be called until it return DEF_YES to notified the HTTP Client stack that all the fields have been added. The API function HTTPc_ReqSetParam() must be used with the parameter type HTTPc_PARAM_TYPE_REQ_HDR_HOOK to pass out hook function pointer to the HTTP Client stack.

Listing - Adding Header Fields with a Hook Example Code

```
#define  HTTPc_CFG_HDR_NBR_MAX      10
#define  HTTPc_CFG_HDR_VAL_LEN_MAX  100

CPU_INT08U     HTTPc_ReqHdrIx;
HTTPc_HDR      HTTPc_ReqHdrTbl[HTTPc_CFG_HDR_NBR_MAX];
CPU_CHAR       HTTPc_ReqHdrValTbl[HTTPc_CFG_HDR_NBR_MAX][HTTPc_CFG_HDR_VAL_LEN_MAX];

/*
*********************************************************************************************************
*                                       HTTPc_ReqPrepare()
*
* Description : Prepare the HTTP request to send.
*
* Argument(s) : p_req    Pointer to the HTTP Request object to configure.
*
* Return(s)   : DEF_OK,   if the request was configured successfully.
*               DEF_FAIL, otherwise.
*********************************************************************************************************
*/
static  CPU_BOOLEAN  HTTPc_ReqPrepare (HTTPc_REQ_OBJ  *p_req)
{
    HTTPc_ERR      err;

                                                  /* -------- SET STRING QUERY PARAMETER ------- */
    HTTPc_ReqSetParam(p_req, HTTPc_PARAM_TYPE_REQ_HDR_HOOK, &HTTPc_ReqHdrHook, &err);
    if (err != HTTPc_ERR_NONE) {
        return (DEF_FAIL);
    }

    p_req->UserDataPtr = (void *)&HTTPc_ReqHdrIx;

    ...  /* TODO other operations on request if necessary. */

    return (DEF_OK);
}

/*
*********************************************************************************************************
*                                       HTTPc_ReqHdrHook()
*
* Description : Hook function to retrieve the header fields to include in the HTTP request.
*
* Argument(s) : p_conn   Pointer to the HTTP Connection object.
*
*               p_req    Pointer to the HTTP Request object.
*
*               p_hdr    Variable that will received the pointer to the Header field object.
*
* Return(s)   : DEF_YES, if all the header fields to include have been passed.
*               DEF_NO,  otherwise.
*********************************************************************************************************
*/
static  CPU_BOOLEAN  HTTPc_ReqHdrHook (HTTPc_CONN_OBJ   *p_conn,
                                       HTTPc_REQ_OBJ    *p_req,
                                       HTTPc_HDR       **p_hdr)
{
    HTTPc_HDR      *p_hdr_item;
    CPU_INT08U      index;

    index = *(CPU_INT08U)p_req->UserDataPtr;

    switch(index) {
        case 0:
             p_hdr_item          = &HTTPc_ReqHdrTbl[0];
             p_hdr_item->ValPtr   = &HTTPc_ReqHdrValTbl[0][0];
             p_hdr_item->HdrField =  HTTP_HDR_FIELD_ACCEPT;
            (void)Str_Copy_N(p_hdr_item->ValPtr, "text/*", HTTPc_CFG_HDR_VAL_LEN_MAX);
```

```
        p_hdr_item        =&HTTPc_ReqHdrTbl[0];
        p_hdr_item->ValPtr   =&HTTPc_ReqHdrValTbl[0][0];
        p_hdr_item->HdrField =  HTTP_HDR_FIELD_ACCEPT;(void)Str_Copy_N(p_hdr_item->ValPtr,"text/*", HTTPc_CFG_HDR_VAL_LEN_MAX);
        p_hdr_item->ValLen =Str_Len_N(p_hdr_item->ValPtr, HTTPc_CFG_HDR_VAL_LEN_MAX);*p_hdr = p_hdr_item;
        index++;*(CPU_INT08U)p_req->UserDataPtr = index;return(DEF_NO);

    case 1:
        p_hdr_item        =&HTTPc_ReqHdrTbl[1];
        p_hdr_item->ValPtr   =&HTTPc_ReqHdrValTbl[1][0];
        p_hdr_item->HdrField =  HTTP_HDR_FIELD_DATE;(void)Str_Copy_N(p_hdr_item->ValPtr,"Thursday, 21-Aug-14 02:15:31 GMT",
HTTPc_CFG_HDR_VAL_LEN_MAX);
        p_hdr_item->ValLen =Str_Len_N(p_hdr_item->ValPtr, HTTPc_CFG_HDR_VAL_LEN_MAX);*p_hd = p_hdr_item;
        index =0;*(CPU_INT08U)p_req->UserDataPtr = index;return(DEF_YES);

    default:*p_hdr = DEF_NULL;
        index =0;*(CPU_INT08U)p_req->UserDataPtr = index;return(DEF_YES);}}
```

HTTP Client Request's Body Standard Transfer

The POST and PUT methods can be used to transfer data to the HTTP server. The data is included in the body part of the HTTP request.

Additional HTTP header fields must be included when a body is present. The content type of the data must be specified with the *Content-Type* header field, and for a standard transfer, the length of the data must be specified with the *Content-Length* header field. Those headers will be added automatically by the HTTP Client stack after the required parameters are configured.

The function HTTPc_ReqSetParam() must be used to configure the additional parameters necessary when data must be transmitted the standard way. Three parameters must be set up :

- HTTPc_PARAM_TYPE_REQ_BODY_CONTENT_TYPE
- HTTPc_PARAM_TYPE_REQ_BODY_CONTENT_LEN
- HTTPc_PARAM_TYPE_REQ_BODY_HOOK

The first parameter sets the content-type of the data, the second the length of the data, and the third sets the hook function that will be used by the HTTP Client module to retrieve the data.

Example

> **Since all the API function are in blocking mode in the example below, the HTTPc objects and string can be allocated on the application stack without problems.**

The example below implements a function to send a PUT request :

- An HTTP Connection is set up
- An HTTP request is configured with the three body parameters: content-type, content-length, and hook function.
- The Connection is open.
- The request is sent.

Listing - Standard Body Transfer Function Example Code

```
CPU_CHAR  index_html[1024];

/*
*********************************************************************************************************
*                              HTTPcEx_ReqSendPUT_Standard()
*
* Description : Example function to send a HTTP PUT request with the standard transfer (not chunked).
*
* Argument(s) : none.
*
* Return(s)   : DEF_YES, is request sent successfully.
*               DEF_NO,  otherwise.
```

```
*****************************************************************************************************
*/

CPU_BOOLEAN  HTTPcEx_ReqSendPUT_Standard (void){
   HTTPc_CONN_OBJ     conn;
   HTTPc_REQ_OBJ      req;
   HTTPc_RESP_OBJ     resp;
   CPU_CHAR           buf[512];
   HTTPc_FLAGS        flags;
   HTTP_CONTENT_TYPE  content_type;
   CPU_SIZE_T         content_len;
   CPU_SIZE_T         str_len;
   CPU_BOOLEAN        result;
   RTOS_ERR           err;/* ------------ INIT NEW CONNECTION ---------- */HTTPc_ConnClr(&conn,&err);if(err.Code != RTOS_ERR_NONE)
{return(DEF_FAIL);}/* ---------- SET CONNECTION PARAM ----------- */
#if(HTTPc_CFG_MODE_ASYNC_TASK_EN == DEF_ENABLED)HTTPc_ConnSetParam(&conn,
                  HTTPc_PARAM_TYPE_CONN_CLOSE_CALLBACK,&HTTPcEx_ConnCloseCallback,&err);if(err.Code != RTOS_ERR_NONE)
{return(DEF_FAIL);}
#endif

                                                /* -------------- INIT NEW REQUEST------------ */HTTPc_ReqClr(&req,&err);if(err.Code != RTOS_ERR_NONE)
{return(DEF_FAIL);}/* ------- SET REQUEST BODY PARAMETERS ------- */
   content_type = HTTP_CONTENT_TYPE_HTML;HTTPc_ReqSetParam(&req,
                 HTTPc_PARAM_TYPE_REQ_BODY_CONTENT_TYPE,&content_type,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}

   content_len = STATIC_INDEX_HTML_LEN;HTTPc_ReqSetParam(&req,
                 HTTPc_PARAM_TYPE_REQ_BODY_CONTENT_LEN,&content_len,&err);if(err.Code != RTOS_ERR_NONE)
{return(DEF_FAIL);}HTTPc_ReqSetParam(&req,
                 HTTPc_PARAM_TYPE_REQ_BODY_HOOK,&HTTPcEx_ReqBodyHook,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}/* --------
SET TRANSACTION CALLBACKS -------- */
#if(HTTPc_CFG_MODE_ASYNC_TASK_EN == DEF_ENABLED)HTTPc_ReqSetParam(&req,
                 HTTPc_PARAM_TYPE_TRANS_ERR_CALLBACK,&HTTPcEx_TransErrCallback,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}
#endif

                                                /* ------------- OPEN CONNECTION ------------- */
   str_len =Str_Len("www.example.com");
   flags   = HTTPc_FLAG_NONE;
   result  =HTTPc_ConnOpen(&conn,&buf,
                       APP_HTTPc_CFG_BUF_SIZE,"www.example.com",
                       str_len,
                       flags,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}if(result != DEF_OK){return(DEF_FAIL);}/* ------------ SEND HTTP
REQUEST ------------ */
   str_len =Str_Len("/index.html");
   flags   = HTTPc_FLAG_NONE;
   result  =HTTPc_ReqSend(&conn,&req,&resp,
                      HTTP_METHOD_PUT,"/index.html",
                      str_len,
                      flags,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}return(DEF_OK);}
```

The example below is a first implementation of the hook function to retrieve the request body data from the application. In this example, the data is retrieved from an existing application buffer (index_html) and the hook function only set up the data pointer and the length of the data.

Listing - Request Body Hook Function Example Code #1

```
/*
****************************************************************************************************
*                              HTTPcEx_ReqBodyHook()
*
* Description : Example Hook function to transfer data in request with the standard transfer.
*
*             This function let you the choice to set the pointer to the application data (with p_data)
*             that the HTTP Client stack will take care of transferring; or to directly copy the data
*             in the HTTP transmit buffer (with p_buf).
*
* Argument(s) : p_conn      Pointer to the current HTTPc Connection object.
*
*             p_req       Pointer to the current HTTPc Request object.
*
*             p_data      Pointer to the application data.
*
*             p_buf       Pointer to the HTTP buffer.
*
*             buf_len     Buffer's length.
*
*             p_data_len  Variable that will return the data length passed by the application.
*
* Return(s)   : DEF_YES, if all data as been passed.
*             DEF_NO,  if data still remaining to be passed by the application.
****************************************************************************************************
*/

static  CPU_BOOLEAN  HTTPcEx_ReqBodyHook (HTTPc_CONN_OBJ   *p_conn,
                                          HTTPc_REQ_OBJ    *p_req,
                                          void            **p_data,
                                          CPU_CHAR         *p_buf,
                                          CPU_INT16U        buf_len,
                                          CPU_INT16U       *p_data_len)
{
 CPU_INT16U  size;
  CPU_INT16U  str_len;

 *p_data = &index_html[0];

  str_len = Str_Len(&index_html[0]);
  Str_Copy_N(p_buf, &index_html[0], str_len);

 *p_data_len = str_len;

  return (DEF_YES);
}
```

The example below is a second implementation of the hook function to retrieve the request body data from the application. In this example, the data is retrieved from a file that is accessed using Micrium OS File System and directly copied into the HTTP transmit buffer.

Listing - Request Body Hook Function Example Code #2

```
static  CPU_BOOLEAN  HTTPcEx_ReqBodyHook (HTTPc_CONN_OBJ   *p_conn,
                                          HTTPc_REQ_OBJ    *p_req,
                                          void            **p_data,
                                          CPU_CHAR         *p_buf,
                                          CPU_INT16U        buf_len,
                                          CPU_INT16U       *p_data_len)
{
    CPU_BOOLEAN     finish;
    CPU_SIZE_T      file_rem;
    CPU_SIZE_T      size;
    CPU_SIZE_T      size_rd;
    FS_FILE_HANDLE  file_handle;
    FS_ENTRY_INFO   file_info;
    FS_FILE_SIZE    file_pos;
    RTOS_ERR        err;

   *p_data = DEF_NULL;

    file_handle = FSFile_Open(FSWrkDir_NullHandle,
                              "ram0/index.html",
                               FS_FILE_ACCESS_MODE_RD,
                              &err);
    if (err.Code != RTOS_ERR_NONE) {
        // TODO: Handle error.

       *p_data_len = 0u;
        finish     = DEF_YES;
        return (finish);
    }

    FSFile_Query(file_handle,
                &file_info,
                &err);
    if (err.Code != RTOS_ERR_NONE) {
        // TODO: Handle error.

       *p_data_len = 0;
        finish     = DEF_YES;
        goto exit_close;
    }

    file_pos = FSFile_PosGet(file_handle,
                             &err);
    if (err.Code != RTOS_ERR_NONE) {
        // TODO: Handle error.

       *p_data_len = 0;
        finish     = DEF_YES;
        goto exit_close;
    }

    file_rem = file_info.Size - file_pos;
    if (file_rem <= 0) {
       *p_data_len = 0;
        finish     = DEF_YES;
        goto exit_close;
    }

    size = DEF_MIN(file_rem, buf_len);

    size_rd = FSFile_Rd(file_handle,
                        p_buf,
                        size,
                       &err);
    if (err.Code != RTOS_ERR_NONE) {
        // TODO: Handle error.
```

```
*p_data_len =0u;
    finish    = DEF_YES;
    goto exit_close;}*p_data_len = size_rd;
  finish    = DEF_NO;

exit_close:FSFile_Close(file_handle,&err);if(err.Code != RTOS_ERR_NONE){// TODO: Handle error.}return(finish);}
```

HTTP Client Chunked Transfer Encoding

The POST and PUT methods can be used to transfer data to the HTTP server. The data is included in the body part of the HTTP request.

When the size of the data to transmit is unknown before starting the HTTP request, the Chunked Transfer Encoding can be used. It will split the data into chunks of known size and will send an empty chunk to advertise the end of the data.

Additional HTTP header fields must be included when a body is present. The content type of the data must be specified with the *Content-Type* header field and for the Chunked Transfer encoding the *Transfer-Encoding* header field must be specified. Those headers will be added automatically by the HTTP Client module after the required parameters are configured. For more details, refer to section Chunked Transfer Encoding.

The function HTTPc_ReqSetParam() must be used to configure the additional parameters necessary when data is transmitted in chunks. Three parameters must be set up :

- HTTPc_PARAM_TYPE_REQ_BODY_CONTENT_TYPE
- HTTPc_PARAM_TYPE_REQ_BODY_CHUNK
- HTTPc_PARAM_TYPE_REQ_BODY_HOOK

The first parameter is to set the content-type of the data, the second is a Boolean to enable the Chunked Transfer Encoding and the third is to set the hook function that will be used by the HTTP Client module to retrieve the data.

Example

> Since all the API functions are in blocking mode in the example below, the HTTPc objects and string can be allocated on the application stack without problems.

The example below implements a function to send a PUT request :

- An HTTP Connection is configured.
- An HTTP request is configured with the three body parameters: content-type, chunk enabled and hook function.
- The Connection is open.
- The request is sent.

Listing - Chunked Transfer Encoding Example Code

```
/*
*********************************************************************************************************
*                               HTTPcEx_ReqSendPUT_Chunked()
*
* Description : Example function to send a HTTP PUT request with the Chunked Transfer Encoding.
*
* Argument(s) : none.
*
* Return(s)   : DEF_YES, is request sent successfully.
*             DEF_NO,  otherwise.
*********************************************************************************************************
*/

CPU_BOOLEAN  HTTPcEx_ReqSendPUT_Chunked (void)
{
  CPU_BOOLEAN       result;
  CPU_BOOLEAN       chunk_en;
  CPU_CHAR         buf[512];
  CPU_SIZE_T       str_len;
  HTTPc_CONN_OBJ     conn;
```

```
    HTTPc_REQ_OBJ      req;
    HTTPc_RESP_OBJ     resp;
    HTTP_CONTENT_TYPE  content_type;
    RTOS_ERR           err;/* ---------- INIT NEW CONNECTION --------- */HTTPc_ConnClr(&conn,&err);if(err.Code != RTOS_ERR_NONE)
{return(DEF_FAIL);}/* -------- SET CONNECTION PARAM ---------- */
#if(HTTPc_CFG_MODE_ASYNC_TASK_EN == DEF_ENABLED)HTTPc_ConnSetParam(&conn,
                HTTPc_PARAM_TYPE_CONN_CLOSE_CALLBACK,&HTTPcEx_ConnCloseCallback,&err);if(err.Code != RTOS_ERR_NONE)
{return(DEF_FAIL);}
#endif

                                 /* ----- SET REQUEST BODY PARAMETERS ------ */
    content_type = HTTP_CONTENT_TYPE_JSON;HTTPc_ReqSetParam(&req,
                HTTPc_PARAM_TYPE_REQ_BODY_CONTENT_TYPE,&content_type,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}

    chunk_en = DEF_YES;HTTPc_ReqSetParam(&req,
                HTTPc_PARAM_TYPE_REQ_BODY_CHUNK,&chunk_en,&err);if(err.Code != RTOS_ERR_NONE)
{return(DEF_FAIL);}HTTPc_ReqSetParam(&req,
                HTTPc_PARAM_TYPE_REQ_BODY_HOOK,&HTTPcEx_ReqBodyHook,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}

#if(HTTPc_CFG_MODE_ASYNC_TASK_EN == DEF_ENABLED)HTTPc_ReqSetParam(&req,
                HTTPc_PARAM_TYPE_TRANS_ERR_CALLBACK,&HTTPcEx_TransErrCallback,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}
#endif

                                 /* ----------- OPEN CONNECTION ------------ */
    str_len = Str_Len("www.example.com");
    result  = HTTPc_ConnOpen(&conn,&buf,
                         APP_HTTPc_CFG_BUF_SIZE,"www.example.com",
                         str_len,
                         HTTPc_FLAG_NONE,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}if(result != DEF_OK){return(DEF_FAIL);}/* ----------
SEND HTTP REQUEST ----------- */
    str_len = Str_Len("/");
    result  = HTTPc_ReqSend(&conn,&req,&resp,
                        HTTP_METHOD_PUT,"/",
                        str_len,
                        HTTPc_FLAG_NONE,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}return(DEF_OK);}
```

For examples on the On Request Body hook function, refer to section Request's Body Standard Transfer since the hook function is used regardless of the transfer mode.

HTTP Client Query String

A Query String is a set of Key-Value pairs or single values added at the end of the request's URL after a question mark character "?". Each field of the Query is separated by the "&" character. For more details, see section Query String .

The HTTP Client module supports the addition of a Query String to an HTTP request:

1. The preprocessor macro HTTPc_CFG_QUERY_STR_EN must be set to DEF_ENABLED.
2. HTTPc_KEY_VAL type's objects must be used to setup a Query String field. When the field is a single string and not a pair of key-value, the pointer to the value can be left blank and only the pointer to the key will be used.
3. Two methods are offered by the HTTP Client module to add Query String:

- Passing a Key-Value Pairs' Table to the HTTP Client stack.
- Using a hook function that will ask the application for each Key-Value Pair.

Using a Key-Value Pairs' Table

For the first option, before sending the HTTP request, the application can prepare a table of HTTPc_KEY_VAL objects that the HTTP Client will include in the Query String. The API function HTTPc_ReqSetParam() must be used with the parameter type HTTPc_PARAM_TYPE_REQ_QUERY_STR_TBL to pass out the table to the HTTP Client stack.

Listing - Adding a Query String with Table Example Code

```
#define  HTTPc_CFG_QUERY_STR_NBR_MAX      10
#define  HTTPc_CFG_QUERY_STR_KEY_LEN_MAX  50
#define  HTTPc_CFG_QUERY_STR_VAL_LEN_MAX  100

HTTPc_KEY_VAL  HTTPc_ReqQueryStrTbl[HTTPc_CFG_QUERY_STR_NBR_MAX];
CPU_CHAR       HTTPc_ReqQueryStrKeyTbl[HTTPc_CFG_QUERY_STR_NBR_MAX][HTTPc_CFG_QUERY_STR_KEY_LEN_MAX];
CPU_CHAR       HTTPc_ReqQueryStrValTbl[HTTPc_CFG_QUERY_STR_NBR_MAX][HTTPc_CFG_QUERY_STR_VAL_LEN_MAX];

/*
*********************************************************************************************************
*                                    HTTPc_ReqPrepareQueryStr()
*
* Description : Prepare the Query String Table.
*
* Argument(s) : p_tbl    Variable that will received the pointer to the Query String table.
*
* Return(s)   : Number of fields in the Query String table.
*********************************************************************************************************
*/
static  CPU_INT08U  HTTPc_ReqPrepareQueryStr (HTTPc_KEY_VAL  **p_tbl)
{
    HTTPc_KEY_VAL  *p_key_val;

                                                /* ------------- SET FIRST QUERY ------------- */
    p_key_val         = &HTTPc_ReqQueryStrTbl[0];
    p_key_val->KeyPtr = &HTTPc_ReqQueryStrKeyTbl[0][0];
    p_key_val->ValPtr = &HTTPc_ReqQueryStrValTbl[0][0];

   (void)Str_Copy_N(p_key_val->KeyPtr, "name", HTTPc_CFG_QUERY_STR_KEY_LEN_MAX);
   (void)Str_Copy_N(p_key_val->ValPtr, "Jonh", HTTPc_CFG_QUERY_STR_VAL_LEN_MAX);

    p_key_val->KeyLen = Str_Len_N(p_key_val->KeyPtr, HTTPc_CFG_QUERY_STR_KEY_LEN_MAX);
    p_key_val->ValLen = Str_Len_N(p_key_val->ValPtr, HTTPc_CFG_QUERY_STR_VAL_LEN_MAX);

                                                /* ------------ SET SECOND QUERY ------------- */
    p_key_val         = &HTTPc_ReqQueryStrTbl[1];
    p_key_val->KeyPtr = &HTTPc_ReqQueryStrKeyTbl[1][0];
    p_key_val->ValPtr =  DEF_NULL;

   (void)Str_Copy_N(p_key_val->KeyPtr, "active", HTTPc_CFG_QUERY_STR_KEY_LEN_MAX);

    p_key_val->KeyLen = Str_Len_N(p_key_val->KeyPtr, HTTPc_CFG_QUERY_STR_KEY_LEN_MAX);

   *p_tbl = &HTTPc_ReqQueryStrTbl[0];

    return (2);
}

/*
*********************************************************************************************************
*                                       HTTPc_ReqPrepare()
*
* Description : Prepare the HTTP request to send.
*
* Argument(s) : p_req    Pointer to the HTTP Request object to configure.
*
* Return(s)   : DEF_OK,   if the request was configured successfully.
*               DEF_FAIL, otherwise.
*********************************************************************************************************
*/
static  CPU_BOOLEAN  HTTPc_ReqPrepare (HTTPc_REQ_OBJ  *p_req)
{
    HTTPc_PARAM_TBL   tbl_obj;
    HTTPc_KEY_VAL    *p_query_str_tbl;
    CPU_INT08U        query_nbr;
    RTOS_ERR          err;
```

```
/* -------- SET STRING QUERY DATA ------------ */
    query_nbr =HTTPc_ReqPrepareQueryStr(&p_query_str_tbl);if(query_nbr <=0){return(DEF_FAIL);}/* ------ SET STRING QUERY PARAMETERS ------
-- */
    tbl_obj.EntryNbr = query_nbr;
    tbl_obj.TblPtr   =(void *)p_query_str_tbl;HTTPc_ReqSetParam(p_req, HTTPc_PARAM_TYPE_REQ_QUERY_STR_TBL,&tbl_obj,&err);if(err.Code !=
RTOS_ERR_NONE){return(DEF_FAIL);}.../* TODO other operations on request if necessary. */return(DEF_OK);}
```

Using a Hook Function

The Second option is to set up a hook function. The hook function will be called by the HTTP Client module to add a field to the Query String. The hook function will be called until it returns DEF_YES to notified the HTTP Client module that all the fields have been added. The API function HTTPc_ReqSetParam() must be used with the parameter type HTTPc_PARAM_TYPE_REQ_QUERY_STR_HOOK to pass out hook function pointer to the HTTP Client module.

Listing - Adding Query String with a Hook Example Code

```
#define  HTTPc_CFG_QUERY_STR_NBR_MAX      10
#define  HTTPc_CFG_QUERY_STR_KEY_LEN_MAX  50
#define  HTTPc_CFG_QUERY_STR_VAL_LEN_MAX  100

CPU_INT08U     HTTPc_ReqQueryStrIx;
HTTPc_KEY_VAL  HTTPc_ReqQueryStrTbl[HTTPc_CFG_QUERY_STR_NBR_MAX];
CPU_CHAR       HTTPc_ReqQueryStrKeyTbl[HTTPc_CFG_QUERY_STR_NBR_MAX][HTTPc_CFG_QUERY_STR_KEY_LEN_MAX];
CPU_CHAR       HTTPc_ReqQueryStrValTbl[HTTPc_CFG_QUERY_STR_NBR_MAX][HTTPc_CFG_QUERY_STR_VAL_LEN_MAX];

/*
*********************************************************************************************************
*                                    HTTPc_ReqPrepare()
*
* Description : Prepare the HTTP request to send.
*
* Argument(s) : p_req    Pointer to the HTTP Request object to configure.
*
* Return(s)   : DEF_OK,   if the request was configured successfully.
*               DEF_FAIL, otherwise.
*********************************************************************************************************
*/
static  CPU_BOOLEAN  HTTPc_ReqPrepare (HTTPc_REQ_OBJ  *p_req)
{
    RTOS_ERR  err;

                                                        /* -------- SET STRING QUERY PARAMETER ------- */
    HTTPc_ReqSetParam(p_req,
                      HTTPc_PARAM_TYPE_REQ_QUERY_STR_HOOK,
                     &HTTPc_ReqQueryStrHook,
                     &err);
    if (err.Code != RTOS_ERR_NONE) {
        return (DEF_FAIL);
    }

    p_req->UserDataPtr = (void *)&HTTPc_ReqQueryStrIx;

    ...  /* TODO other operations on request if necessary. */

    return (DEF_OK);
}

/*
*********************************************************************************************************
*                                    HTTPc_ReqQueryStrHook()
*
* Description : Hook function to retrieve the data of the Query String.
*
* Argument(s) : p_conn      Pointer to the HTTP Connection object.
*
*               p_req       Pointer to the HTTP Request object.
*
*               p_key_val   Variable that will receive the pointer to the Key-Val object.
*
* Return(s)   : DEF_YES, if all the fields of the Query String have been passed.
*               DEF_NO,  otherwise.
*********************************************************************************************************
*/
static  CPU_BOOLEAN  HTTPc_ReqQueryStrHook (HTTPc_CONN_OBJ   *p_conn,
                                            HTTPc_REQ_OBJ    *p_req,
                                            HTTPc_KEY_VAL   **p_key_val)
{
    HTTPc_KEY_VAL  *p_kvp;
    CPU_INT08U      index;

    index = *(CPU_INT08U)p_req->UserDataPtr;

    switch(index) {
```

```
    case 0:
        p_kvp       =&HTTPc_ReqQueryStrTbl[0];
        p_kvp->KeyPtr =&HTTPc_ReqQueryStrKeyTbl[0][0];
        p_kvp->ValPtr =&HTTPc_ReqQueryStrValTbl[0][0];(void)Str_Copy_N(p_kvp->KeyPtr,"name", HTTPc_CFG_QUERY_STR_KEY_LEN_MAX);
(void)Str_Copy_N(p_kvp->ValPtr,"Jonh", HTTPc_CFG_QUERY_STR_VAL_LEN_MAX);
        p_kvp->KeyLen =Str_Len_N(p_kvp->KeyPtr, HTTPc_CFG_QUERY_STR_KEY_LEN_MAX);
        p_kvp->ValLen =Str_Len_N(p_kvp->ValPtr, HTTPc_CFG_QUERY_STR_VAL_LEN_MAX);*p_key_val = p_kvp;
        index++;*(CPU_INT08U)p_req->UserDataPtr = index;return(DEF_NO);

    case 1:
        p_kvp       =&HTTPc_ReqQueryStrTbl[1];
        p_kvp->KeyPtr =&HTTPc_ReqQueryStrKeyTbl[1][0];
        p_kvp->ValPtr =  DEF_NULL;(void)Str_Copy_N(p_kvp->KeyPtr,"active", HTTPc_CFG_QUERY_STR_KEY_LEN_MAX);
        p_kvp->KeyLen =Str_Len_N(p_kvp->KeyPtr, HTTPc_CFG_QUERY_STR_KEY_LEN_MAX);*p_key_val = p_kvp;
        index =0;*(CPU_INT08U)p_req->UserDataPtr = index;return(DEF_YES);

    default:*p_key_val = DEF_NULL;
        index =0;*(CPU_INT08U)p_req->UserDataPtr = index;return(DEF_YES);}}
```

HTTP Client Form Submission

HTTP Client offers several structures and API functions to simplify the formatting and transmission of Forms for the development of an HTTP Client application.

For more details on HTTP Forms, see section Form .

Form Field Types

HTTP Client offers three types of form field. Each type is associated with a structure that the application can used to create objects:

- Key-Value Pair : Simple pair of key and value strings. Can be used with the Application type form and the Multipart type form.
- Extended Key-Value Pair : Key-Value pair where the string value is not specified but a hook function is to allow the HTTP Client stack to retrieve the value when the form is formatted. Can be used with the Multipart type form only.
- Multipart File : Use to upload file to an HTTP server inside a Multipart type form. Can be used with the Multipart type form only.

Form API

The application should incorporate all the form fields into a table using the API functions offered below :

- HTTPc_FormAddKeyVal() to add a simple Key-Value pair object to a form table.
- HTTPc_FormAddKeyValExt() to add a Extended Key-Value pair object to a form table.
- HTTPc_FormAddFile() to add a Multipart File object to a form table.

Once the form fields' table is created, two possibilities are offered.

The first option is valid only if the table contains simple Key-Value pairs and consists of formatting the form on the application side with one of the API functions below:

- HTTPc_FormAppFmt() to format the table into an Application-type form.
- HTTPc_FormMultipartFmt() to format the table into a Multipart-type form.

Afterwards the application can transmit the formatted form like any other body types.

The second option is to pass the form fields table to the HTTP Client module so that the formatting can be done internally by the HTTP Client core. This option allows you to have a different form field type in the table. The HTTP Client module will take care of calling the hook functions to recover data from the application for extended Key-Value pair object and Multipart File objects.

The form fields table can be passed to the HTTP Client module by calling the HTTPc_ReqSetParam() function will the parameter type HTTPc_PARAM_TYPE_REQ_FORM_TBL and the macro configuration HTTPc_CFG_FORM_EN must be set to DEF_ENABLED.

Examples

Example #1: Application Type Form with the form formatting on the application side

Listing - Form Submission Example Code #1

```
#define  HTTP_SERVER_HOSTNAME            "www.example.com"
#define  HTTPc_EX_CFG_CONN_BUF_SIZE        512
#define  HTTPc_EX_CFG_FORM_FIELD_NBR_MAX    10
#define  HTTPc_EX_CFG_FORM_BUF_SIZE       4096
#define  HTTPc_EX_CFG_KEY_LEN_MAX           50
#define  HTTPc_EX_CFG_VAL_LEN_MAX          100

HTTPc_CONN_OBJ       HTTPcEx_Conn;
CPU_CHAR             HTTPcEx_ConnBuf[HTTPc_CFG_CONN_BUF_SIZE];
HTTPc_REQ_OBJ        HTTPcEx_Req;
HTTPc_RESP_OBJ       HTTPcEx_Resp;
HTTPc_FORM_TBL_FIELD  HTTPcEx_ReqFormTbl[HTTPc_EX_CFG_FORM_FIELD_NBR_MAX];
CPU_CHAR             HTTPcEx_FormBufTbl[HTTPc_EX_CFG_FORM_BUF_SIZE];
HTTPc_KEY_VAL        HTTPcEx_KeyVal_Tbl[HTTPc_EX_CFG_FORM_FIELD_NBR_MAX];
CPU_CHAR             HTTPcEx_KeyVal_KeyTbl[HTTPc_EX_CFG_FORM_FIELD_NBR_MAX][HTTPc_EX_CFG_KEY_LEN_MAX];
CPU_CHAR             HTTPcEx_KeyVal_ValTbl[HTTPc_EX_CFG_FORM_FIELD_NBR_MAX][HTTPc_EX_CFG_VAL_LEN_MAX];

/*
*********************************************************************************************************
*                                    HTTPcEx_ReqSendFormApp()
*
* Description : Example function to send an HTTP POST request containing an Application type form.
*
* Argument(s) : None.
*
* Return(s)   : DEF_OK,   if the HTTP transaction was successful.
*               DEF_FAIL, otherwise.
*********************************************************************************************************
*/

CPU_BOOLEAN  HTTPcEx_ReqSendFormApp (void)
{
    HTTPc_CONN_OBJ        *p_conn;
    HTTPc_REQ_OBJ         *p_req;
    HTTPc_RESP_OBJ        *p_resp;
    CPU_CHAR              *p_buf;
    CPU_CHAR              *p_data;
    HTTPc_FORM_TBL_FIELD  *p_form_tbl;
    CPU_INT08U             form_nbr;
    CPU_INT16U             content_len;
    RTOS_ERR               err;

    p_conn = &HTTPcEx_ConnTbl[0];
    p_req  = &HTTPcEx_ReqTbl[0];
    p_resp = &HTTPcEx_RespTbl[0];
    p_buf  = &HTTPcEx_BufTbl[0][0];

                                                                /* ------------- SET FORM TO SEND ------------ */
    form_nbr = HTTPcEx_ReqPrepareFormApp(&p_form_tbl);
    if (form_nbr <= 0) {
        return (DEF_FAIL);
    }

    p_data      = &HTTPcEx_FormBufTbl[0];
    content_len = HTTPc_FormMultipartFmt(p_data,
                                         APP_HTTPc_CFG_FORM_BUF_SIZE,
                                         p_form_tbl,
                                         form_nbr,
                                        &err);
    if (err.Code != RTOS_ERR_NONE) {
        return (DEF_FAIL);
    }

                                                                /* ------------ INIT NEW CONNECTION ---------- */
    HTTPc_ConnClr(p_conn, &err);
    if (err.Code != RTOS_ERR_NONE) {
```

```
}/* ----------- SET CONN'S CALLBACKS ---------- */
#if(HTTPc_CFG_MODE_ASYNC_TASK_EN == DEF_ENABLED)HTTPc_ConnSetParam(p_conn,
                    HTTPc_PARAM_TYPE_CONN_CLOSE_CALLBACK,&HTTPcEx_ConnCloseCallback,&err);if(err.Code != RTOS_ERR_NONE)
{return(DEF_FAIL);}
#endif

                                                        /* ------------- INIT NEW REQUEST ------------ */HTTPc_ReqClr(p_req,&err);if(err.Code != RTOS_ERR_NONE)
{return(DEF_FAIL);}/* ------- SET REQUEST BODY PARAMETERS ------- */
   content_type = HTTP_CONTENT_TYPE_MULTIPART_FORM;HTTPc_ReqSetParam(p_req,
                   HTTPc_PARAM_TYPE_REQ_BODY_CONTENT_TYPE,&content_type,&err);if(err.Code != RTOS_ERR_NONE)
{return(DEF_FAIL);}HTTPc_ReqSetParam(p_req,
                   HTTPc_PARAM_TYPE_REQ_BODY_CONTENT_LEN,&content_len,&err);if(err.Code != RTOS_ERR_NONE)
{return(DEF_FAIL);}HTTPc_ReqSetParam(p_req,
                   HTTPc_PARAM_TYPE_REQ_BODY_HOOK,&HTTPcEx_ReqBodyHook,&err);if(err != HTTPc_ERR_NONE){return(DEF_FAIL);}/* -----------
SET REQ'S CALLBACKS ----------- */
#if(HTTPc_CFG_MODE_ASYNC_TASK_EN == DEF_ENABLED)HTTPc_ReqSetParam(p_req,
                   HTTPc_PARAM_TYPE_TRANS_ERR_CALLBACK,&HTTPcEx_TransErrCallback,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}
#endif

                                                       /* ------------- OPEN CONNECTION ------------- */
   str_len =Str_Len(HTTP_SERVER_HOSTNAME);
   result  =HTTPc_ConnOpen(p_conn,
                      p_buf,
                      HTTPc_EX_CFG_BUF_SIZE,
                      HTTP_SERVER_HOSTNAME,
                      str_len,
                      HTTPc_FLAG_NONE,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}if(result == DEF_OK){printf("Connection to server
succeeded.\n\r");}else{printf("Connection to server failed.\n\r");}/* ------------ SEND HTTP REQUEST ------------ */
   str_len =Str_Len("/form");
   result  =HTTPc_ReqSend(p_conn,
                     p_req,
                     p_resp,
                     HTTP_METHOD_POST,"/form",
                     str_len,
                     HTTPc_FLAG_NONE,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}if(result == DEF_OK){printf("%s\n\r", p_resp-
>ReasonPhrasePtr);}return(DEF_OK);}/*
*********************************************************************************************************
*                                    HTTPcEx_ReqPrepareFormApp()
*
* Description : Example function to prepare the form table for an application type form
*
* Argument(s) : p_form_tbl  Variable that will received the pointer to the form table.
*
* Return(s)   : Number of fields in the table.
*********************************************************************************************************
*/

static  CPU_INT08U  HTTPcEx_ReqPrepareFormApp (HTTPc_FORM_TBL_FIELD  **p_form_tbl){
   HTTPc_FORM_TBL_FIELD  *p_tbl;
   HTTPc_KEY_VAL         *p_kvp;
   RTOS_ERR              err;

   p_tbl =&HTTPcEx_ReqFormTbl[0];/* ---------- ADD FIRST FORM FIELD ----------- */
   p_kvp          =&HTTPcEx_KeyVal_Tbl[0];
   p_kvp->KeyPtr  =&HTTPcEx_KeyVal_KeyTbl[0][0];
   p_kvp->ValPtr  =&HTTPcEx_KeyVal_ValTbl[0][0];
   p_kvp->KeyLen  =Str_Len("Name");
   p_kvp->ValLen  =Str_Len("John Smith");(void)Str_Copy_N(p_kvp->KeyPtr,"Name",       p_kvp->KeyLen);(void)Str_Copy_N(p_kvp->ValPtr,"Jonh
Smith", p_kvp->ValLen);HTTPc_FormAddKeyVal(p_tbl, p_kvp,&err);/* ---------- ADD SECOND FORM FIELD ---------- */
   p_tbl++;
   p_kvp++;
   p_kvp->KeyPtr   =&HTTPcEx_KeyVal_KeyTbl[1][0];
   p_kvp->ValPtr   =&HTTPcEx_KeyVal_ValTbl[1][0];
   p_kvp->KeyLen =Str_Len("Book");
   p_kvp->ValLen =Str_Len("Implementing IPv6 Second Edition");(void)Str_Copy_N(p_kvp->KeyPtr,"Book",                     p_kvp->KeyLen);
(void)Str_Copy_N(p_kvp->ValPtr,"Implementing IPv6 Second Edition", p_kvp->ValLen);HTTPc_FormAddKeyVal(p_tbl, p_kvp,&err);*p_form_tbl
=&HTTPcEx_ReqFormTbl[0];return(2);}
```

HTTP Client Persistent Connection

Persistent Connection allows you to send multiple HTTP requests one after another on the same opened connection before closing it. For more details, refer to section Persistent Connection .

Example

In this example, two HTTP requests are queued on an HTTP Connection before the client closes the connection.

The function HTTPc_ReqSend() is called with the flag HTTPc_FLAG_REQ_NO_BLOCK, so the function is not blocking and will return before the transaction completes. Therefore the On Transaction Complete hook must be set to retrieve the HTTP response.

Listing - Persistent Connection Example Code

```
#define  HTTP_SERVER_HOSTNAME                        "www.example.com"

#define  HTTPc_EX_CFG_CONN_NBR_MAX                    5u
#define  HTTPc_EX_CFG_REQ_NBR_MAX                     5u
#define  HTTPc_EX_CFG_BUF_SIZE                      512u

HTTPc_CONN_OBJ  HTTPcEx_ConnTbl[HTTPc_EX_CFG_CONN_NBR_MAX];
HTTPc_REQ_OBJ   HTTPcEx_ReqTbl[HTTPc_EX_CFG_REQ_NBR_MAX];
HTTPc_RESP_OBJ  HTTPcEx_RespTbl[HTTPc_EX_CFG_REQ_NBR_MAX];
CPU_CHAR        HTTPcEx_BufTbl[HTTPc_EX_CFG_CONN_NBR_MAX][HTTPc_EX_CFG_BUF_SIZE ];

CPU_BOOLEAN    close_conn = DEF_NO;
CPU_BOOLEAN    req1_done = DEF_NO;
CPU_BOOLEAN    req2_done = DEF_NO;

/*
*********************************************************************************************************
*                                   HTTPcEx_ReqSendGET()
*
* Description : Example function to Send Requests on persistent connection.
*
* Argument(s) : None.
*
* Return(s)   : DEF_YES, if request successfully sent.
*               DEF_NO,  otherwise.
*********************************************************************************************************
*/

CPU_BOOLEAN  HTTPcEx_ReqSendGET (void)
{
    HTTPc_CONN_OBJ  *p_conn;
    HTTPc_REQ_OBJ   *p_req1;
    HTTPc_REQ_OBJ   *p_req2;
    HTTPc_RESP_OBJ  *p_resp1;
    HTTPc_RESP_OBJ  *p_resp2;
    CPU_CHAR        *p_buf;
    CPU_BOOLEAN      persistent;
    CPU_BOOLEAN      result;
    RTOS_ERR         err;

    p_conn  = &HTTPcEx_ConnTbl[0];
    p_buf   = &HTTPcEx_BufTbl[0][0];
    p_req1  = &HTTPcEx_ReqTbl[0];
    p_req2  = &HTTPcEx_ReqTbl[1];
    p_resp1 = &HTTPcEx_RespTbl[0];
    p_resp2 = &HTTPcEx_RespTbl[1];

                                                    /* ------------ INIT NEW CONNECTION ---------- */
    HTTPc_ConnClr(p_conn, &err);
    if (err.Code != RTOS_ERR_NONE) {
        return (DEF_FAIL);
    }

                                                    /* ----------- SET PERSISTENT MODE ----------- */
    persistent = DEF_YES;
    HTTPc_ConnSetParam(p_conn,
                       HTTPc_PARAM_TYPE_PERSISTENT,
                      &persistent,
                      &err);
    if (err.Code != RTOS_ERR_NONE) {
        return (DEF_FAIL);
    }

                                                    /* ---------- SET CONN'S CALLBACKS ----------- */
#if (HTTPc_CFG_MODE_ASYNC_TASK_EN == DEF_ENABLED)
    HTTPc_ConnSetParam(p_conn,
```

```
&HTTPcEx_ConnCloseCallback,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}
#endif

                          /* ------------ INIT NEW REQUESTS ------------ */HTTPc_ReqClr(p_req1,&err);if(err.Code !=
RTOS_ERR_NONE){return(DEF_FAIL);}HTTPc_ReqClr(p_req2,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}/* ---- SET REQ/RESP
CALLBACK FUNCTIONS ------ */
#if(HTTPc_CFG_MODE_ASYNC_TASK_EN == DEF_ENABLED)HTTPc_ReqSetParam(p_req1,
         HTTPc_PARAM_TYPE_TRANS_COMPLETE_CALLBACK,&HTTPcEx_TransDoneCallback,&err);if(err.Code != RTOS_ERR_NONE)
{return(DEF_FAIL);}HTTPc_ReqSetParam(p_req2,
         HTTPc_PARAM_TYPE_TRANS_COMPLETE_CALLBACK,&HTTPcEx_TransDoneCallback,&err);if(err.Code != RTOS_ERR_NONE)
{return(DEF_FAIL);}HTTPc_ReqSetParam(p_req1,
         HTTPc_PARAM_TYPE_TRANS_ERR_CALLBACK,&HTTPcEx_TransErrCallback,&err);if(err.Code != RTOS_ERR_NONE)
{return(DEF_FAIL);}HTTPc_ReqSetParam(p_req2,
         HTTPc_PARAM_TYPE_TRANS_ERR_CALLBACK,&HTTPcEx_TransErrCallback,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}
#endif

                          /* ------------ OPEN CONNECTION -------------- */
  str_len = Str_Len(HTTP_SERVER_HOSTNAME);
  result  = HTTPc_ConnOpen(p_conn,
                           p_buf,
                           HTTPc_EX_CFG_BUF_SIZE,
                           HTTP_SERVER_HOSTNAME,
                           str_len,
                           HTTPc_FLAG_NONE,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}if(result == DEF_OK){printf("Connection to server
succeeded.\n\r");}else{printf("Connection to server failed.\n\r");}/* ----------- SEND HTTP REQUESTS ------------ */
  str_len = Str_Len("//");
  result  = HTTPc_ReqSend(p_conn,
                          p_req1,
                          p_resp1,
                          HTTP_METHOD_GET,"//",
                          str_len,
                          HTTPc_FLAG_REQ_NO_BLOCK,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}(void)HTTPc_ReqSend(p_conn,
                        p_req2,
                        p_resp2,
                        HTTP_METHOD_GET,"//",
                        str_len,
                        HTTPc_FLAG_REQ_NO_BLOCK,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}while(close_conn == DEF_NO){/* Do OS
delay. */};/* --------- CLOSE HTTP CONNECTION ----------- */
  close_conn = DEF_NO;HTTPc_ConnClose(p_conn,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}return(DEF_OK);}/*
*********************************************************************************************************
*                                 HTTPcEx_TransDoneCallback()
*
* Description : Example callback function to be notified when an HTTP transaction is completed.
*
* Argument(s) : p_conn_obj   Pointer to current HTTPc Connection object.
*
*               p_req_obj    Pointer to current HTTPc Request object.
*
*               p_resp_obj   Pointer to current HTTPc Response object.
*
*               status       Status of the HTTP transaction:
*                            DEF_OK,   transaction succeeded.
*                            DEF_FAIL, transaction failed.
*
* Return(s)   : None.
*********************************************************************************************************
*/

void  HTTPcEx_TransDoneCallback (HTTPc_CONN_OBJ  *p_conn_obj,
                                 HTTPc_REQ_OBJ   *p_req_obj,
                                 HTTPc_RESP_OBJ  *p_resp_obj,
                                 CPU_BOOLEAN      status){
  HTTPc_REQ_OBJ     *p_req1;
  HTTPc_REQ_OBJ     *p_req2;

  p_req1  = &HTTPcEx_ReqTbl[0];
  p_req2  = &HTTPcEx_ReqTbl[1];if(p_req_obj == p_req1){if(status == DEF_OK){printf("Transaction #1: %s\n\r", p_resp_obj-
>ReasonPhrasePtr);}else{printf("Transaction #1 failed\n\r");}
```

```
    req1_done = DEF_YES;}elseif(p_req_obj == p_req2){if(status == DEF_OK){printf("Transaction #2: %s\n\r", p_resp_obj->ReasonPhrasePtr);}else{printf("Transaction #2 failed\n\r");}
    req2_done = DEF_YES;}else{printf("Unexpected error.\n\r");}

  close_conn = req1_done & req2_done;}
```

HTTP Client Retrieve HTTP Response Data

The HTTPc Response object will be filled by the HTTP Client stack with the fields below received in the HTTP response:

- The Server HTTP version
- Status Code
- Reason phrase
- Content type (if body present)
- Content Length (if body present)

If the function HTTPc_ReqSend() was called in the blocking mode and the function returned with no errors, then the HTTPc Response object can be read by the application after the function call. Otherwise, the application must wait until the On Transaction Complete function is called by the HTTP Client stack to read the HTTPc Response object.

Two hook functions allow the application to retrieve more data from the HTTP response :

- On Response Header
- On Response Body

The first hook allows the application to retrieve header fields contained in the HTTP response. If the hook function is set via the function HTTPc_ReqSetParam() with the parameter type HTTPc_PARAM_TYPE_RESP_HDR_HOOK, the hook will be called by the HTTP Client module for each header field parsed and recognized by the stack. The application as therefore the liberty to copy the data received on its side inside the hook function.

The second hook allows the application to retrieve the data contained in the HTTP response body. If the hook function is set via the function HTTPc_ReqSetParam() with the parameter type HTTPc_PARAM_TYPE_RESP_BODY_HOOK, the hook will be called by the HTTP Client module for each piece of data received until all is received.

Examples

Blocking Mode

In the first example below, the function HTTPc_ReqSend() is called in blocking mode. Therefore, the HTTPc Response object can be read just after the function call.

Listing - Retrieve HTTP Response Data Example Code #1

```
#define  HTTP_SERVER_HOSTNAME  "www.example.com"
#define  HTTPc_EX_CFG_BUF_SIZE  1024

/*
*********************************************************************************************************
*                                   HTTPcEx_ReqSendGET()
*
* Description : Example function to send a GET request and retrieve HTTP response.
*
* Argument(s) : None.
*
* Return(s)   : DEF_OK,   if HTTP transaction was successful.
*               DEF_FAIL, otherwise.
*********************************************************************************************************
*/

CPU_BOOLEAN  HTTPcEx_ReqSendGET (void)
{
    HTTPc_CONN_OBJ   conn;
    HTTPc_REQ_OBJ    req;
    HTTPc_RESP_OBJ   resp;
    CPU_CHAR         buf[HTTPc_EX_CFG_BUF_SIZE];
    HTTPc_FLAGS      flags;
    CPU_SIZE_T       str_len;
    CPU_BOOLEAN      result;
    RTOS_ERR         err;

                                                  /* ------------ INIT NEW CONNECTION ---------- */
    HTTPc_ConnClr(&conn, &err);
    if (err.Code != RTOS_ERR_NONE) {
        return (DEF_FAIL);
    }

                                                  /* ----------- SET CONN'S CALLBACKS ---------- */
#if (HTTPc_CFG_MODE_ASYNC_TASK_EN == DEF_ENABLED)
    HTTPc_ConnSetParam(&conn,
                        HTTPc_PARAM_TYPE_CONN_CLOSE_CALLBACK,
                       &HTTPcEx_ConnCloseCallback,
                       &err);
    if (err.Code != RTOS_ERR_NONE) {
        return (DEF_FAIL);
    }
#endif

                                                  /* ------------- INIT NEW REQUEST ------------ */
    HTTPc_ReqClr(&req, &err);
    if (err.Code != RTOS_ERR_NONE) {
        return (DEF_FAIL);
    }

                                                  /* --------- SET TRANS HOOK FUNCTIONS -------- */
    HTTPc_ReqSetParam(&req,
                       HTTPc_PARAM_TYPE_RESP_HDR_HOOK,
                      &HTTPcEx_RespHdrHook,
                      &err);
    if (err.Code != RTOS_ERR_NONE) {
        return (DEF_FAIL);
    }

    HTTPc_ReqSetParam(&req,
                       HTTPc_PARAM_TYPE_RESP_BODY_HOOK,
                      &HTTPcEx_RespBodyHook,
                      &err);
    if (err.Code != RTOS_ERR_NONE) {
        return (DEF_FAIL);
    }
```

```
/* ------ SET TRANS CALLBACK FUNCTIONS ------- */
#if(HTTPc_CFG_MODE_ASYNC_TASK_EN == DEF_ENABLED)HTTPc_ReqSetParam(&req,
            HTTPc_PARAM_TYPE_TRANS_ERR_CALLBACK,&HTTPcEx_TransErrCallback,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}
#endif

                    /* ------------- OPEN CONNECTION ------------- */
  str_len = Str_Len(HTTP_SERVER_HOSTNAME);
  result  = HTTPc_ConnOpen(&conn,&buf,
                HTTPc_EX_CFG_BUF_SIZE,
                HTTP_SERVER_HOSTNAME,
                str_len,
                HTTPc_FLAG_NONE,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}if(result == DEF_OK){printf("Connection to server
succeeded.\n\r");}else{printf("Connection to server failed.\n\r");}/* ------------ SEND HTTP REQUEST ------------ */
  str_len = Str_Len("/");
  result  = HTTPc_ReqSend(&conn,&req,&resp,
                HTTP_METHOD_GET,"/",
                str_len,
                HTTPc_FLAG_NONE,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}if(result == DEF_OK){printf("%s\n\r",
resp.ReasonPhrasePtr);}return(DEF_OK);}
```

No-Blocking Mode

In this second example, the function HTTPc_ReqSend() is called in non-blocking mode. Therefore, the HTTPc Response object can be read once the On Transaction Complete function is called.

Listing - Retrieve HTTP Response Data Example Code #2

```
#define  HTTP_SERVER_HOSTNAME  "www.example.com"
#define  HTTPc_EX_CFG_BUF_SIZE  1024

HTTPc_CONN_OBJ  HTTPcEx_Conn;
HTTPc_REQ_OBJ   HTTPcEx_Req;
HTTPc_RESP_OBJ  HTTPcEx_Resp;
CPU_CHAR        HTTPcEx_ConnBuf[HTTPc_EX_CFG_BUF_SIZE];

/*
*********************************************************************************************************
*                                    HTTPcEx_ReqSendGET()
*
* Description : Example function to send a GET request and retrieve HTTP response.
*
* Argument(s) : None.
*
* Return(s)   : DEF_OK,   if HTTP transaction was successful.
*               DEF_FAIL, otherwise.
*********************************************************************************************************
*/

CPU_BOOLEAN  HTTPcEx_ReqSendGET (void)
{
    HTTPc_CONN_OBJ   *p_conn;
    HTTPc_REQ_OBJ    *p_req;
    HTTPc_RESP_OBJ   *p_resp;
    CPU_CHAR         *p_buf;
    CPU_SIZE_T        str_len;
    CPU_BOOLEAN       result;
    HTTPc_ERR         err;

    p_conn = &HTTPcEx_Conn;
    p_req  = &HTTPcEx_Req;
    p_resp = &HTTPcEx_Resp;
    p_buf  = &HTTPcEx_ConnBuf[0];

                                                   /* ------------ INIT NEW CONNECTION ---------- */
    HTTPc_ConnClr(p_conn, &err);
    if (err.Code != RTOS_ERR_NONE) {
        return (DEF_FAIL);
    }

                                                   /* ----------- SET CONN'S CALLBACKS ---------- */
#if (HTTPc_CFG_MODE_ASYNC_TASK_EN == DEF_ENABLED)
    HTTPc_ConnSetParam(p_conn,
                       HTTPc_PARAM_TYPE_CONN_CLOSE_CALLBACK,
                      &HTTPcEx_ConnCloseCallback,
                      &err);
    if (err.Code != RTOS_ERR_NONE) {
        return (DEF_FAIL);
    }
#endif

                                                   /* ------------- INIT NEW REQUEST ------------ */
    HTTPc_ReqClr(p_req, &err);
    if (err.Code != RTOS_ERR_NONE) {
        return (DEF_FAIL);
    }

                                                   /* --------- SET TRANS HOOK FUNCTIONS -------- */
    HTTPc_ReqSetParam(p_req,
                      HTTPc_PARAM_TYPE_RESP_HDR_HOOK,
                     &HTTPcEx_RespHdrHook,
                     &err);
    if (err.Code != RTOS_ERR_NONE) {
        return (DEF_FAIL);
    }
```

```
HTTPc_ReqSetParam(p_req,
                  HTTPc_PARAM_TYPE_RESP_BODY_HOOK,&HTTPcEx_RespBodyHook,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}/* ------
SET TRANS CALLBACK FUNCTIONS ------- */
#if(HTTPc_CFG_MODE_ASYNC_TASK_EN == DEF_ENABLED)HTTPc_ReqSetParam(p_req,
                  HTTPc_PARAM_TYPE_TRANS_COMPLETE_CALLBACK,&HTTPcEx_TransDoneCallback,&err);if(err.Code != RTOS_ERR_NONE)
{return(DEF_FAIL);}HTTPc_ReqSetParam(p_req,
                  HTTPc_PARAM_TYPE_TRANS_ERR_CALLBACK,&HTTPcEx_TransErrCallback,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}
#endif

                                                     /* ------------- OPEN CONNECTION ------------- */
   str_len = Str_Len(HTTP_SERVER_HOSTNAME);
   result  = HTTPc_ConnOpen(p_conn,
                         p_buf,
                         HTTPc_EX_CFG_BUF_SIZE,
                         HTTP_SERVER_HOSTNAME,
                         str_len,
                         HTTPc_FLAG_NONE,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}if(result == DEF_OK){printf("Connection to server
succeeded.\n\r");}else{printf("Connection to server failed.\n\r");}/* ------------ SEND HTTP REQUEST ------------ */
   str_len = Str_Len("/");
   result  = HTTPc_ReqSend(p_conn,
                        p_req,
                        p_resp,
                        HTTP_METHOD_GET,"/",
                        str_len,
                        HTTPc_FLAG_REQ_NO_BLOCK,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}return(DEF_OK);}/*
*********************************************************************************************************
*                             HTTPcEx_TransDoneCallback()
*
* Description : Example callback function to be notified when an HTTP transaction was completed.
*
*             Once this callback is called, the HTTPc Response object is ready to be read by the application
*             if no error occurred.
*
* Argument(s) : p_conn     Pointer to current HTTPc Connection object.
*
*             p_req      Pointer to current HTTPc Request object.
*
*             p_resp     Pointer to current HTTPc Response object.
*
*             status     DEF_OK, if transaction was complete successfully.
*                        DEF_NO, otherwise.
*
* Return(s)   : None.
*********************************************************************************************************
*/

static  void  HTTPcEx_TransDoneCallback(HTTPc_CONN_OBJ    *p_conn,
                                        HTTPc_REQ_OBJ     *p_req,
                                        HTTPc_RESP_OBJ    *p_resp,
                                        CPU_BOOLEAN        status){if(status == DEF_OK){printf("Transaction response status: %s\n\r", p_resp-
>ReasonPhrasePtr);}else{printf("Transaction failed\n\r");}}
```

Response Header and Body Hook Functions

The example code below is an implementation of the On Response Header hook function and the On Response Body hook function.

Listing - Response Hook Functions Example Code

```
#define  HTTPc_EX_CFG_HDR_VAL_BUF_LEN  100

CPU_CHAR  HTTPcEx_HdrValBuf[HTTPc_EX_CFG_HDR_VAL_BUF_LEN];

/*
*********************************************************************************************************
*                                     HTTPcEx_RespHdrHook()
*
* Description : Example hook function to retrieve Header Fields received in the HTTP response.
*
* Argument(s) : p_conn      Pointer to current HTTPc Connection object.
*
*               p_req       Pointer to current HTTPc Request object.
*
*               hdr_field   Header field type received.
*
*               p_hdr_val   Pointer to Header field value string received.
*
*               val_len     Length of value string.
*
* Return(s)   : None.
*********************************************************************************************************
*/

static  void  HTTPcEx_RespHdrHook(HTTPc_CONN_OBJ    *p_conn,
                                  HTTPc_REQ_OBJ     *p_req,
                                  HTTP_HDR_FIELD     hdr_field,
                                  CPU_CHAR          *p_hdr_val,
                                  CPU_INT16U         val_len)
{
    CPU_CHAR    *p_buf;
    CPU_INT16U   len;

    p_buf = &HTTPcEx_HdrValBuf[0];

    switch (hdr_field) {
        case HTTP_HDR_FIELD_DATE:
        case HTTP_HDR_FIELD_SERVER:
             len = DEF_MIN((HTTPc_EX_CFG_HDR_VAL_BUF_LEN - 1), val_len);
             Mem_Copy(p_buf, p_hdr_val, len);
             p_buf += len;
            *p_buf  = '\0';
             break;

        default:
             break;
    }
}

/*
*********************************************************************************************************
*                                     HTTPcEx_RespBodyHook()
*
* Description : Example hook function to retrieve body data.
*
*               In this example, a html file is received and is copied into a file.
*
* Argument(s) : p_conn        Pointer to current HTTPc Connection object.
*
*               p_req         Pointer to current HTTPc Request object.
*
*               content_type  Content type of the body received in the HTTP response.
*
*               p_data        Pointer to the data piece received.
*
*               data_len      Length of data piece received.
*
```

```
*           last_chunk    DEF_YES, if this represent the last piece of the data body to receive.
*                         DEF_NO,  if data still remains to be received.
*
* Return(s)   : None.
*******************************************************************************************************
*/

static  void  HTTPcEx_RespBodyHook (HTTPc_CONN_OBJ     *p_conn,
                                    HTTPc_REQ_OBJ      *p_req,
                                    HTTP_CONTENT_TYPE   content_type,
                                    void               *p_data,
                                    CPU_INT32U          data_len,
                                    CPU_BOOLEAN         last_chunk){
    FS_FILE_HANDLE  file_handle;
    CPU_SIZE_T      size_wr;
    CPU_SIZE_T      size_wr_tot;
    CPU_BOOLEAN     is_open;
    RTOS_ERR        err;

    file_handle =FSFile_Open(FSWrkDir_NullHandle,"ram0/index.html",(FS_FILE_ACCESS_MODE_WR | FS_FILE_ACCESS_MODE_CREATE |
FS_FILE_ACCESS_MODE_APPEND),&err);if(err.Code != RTOS_ERR_NONE){return;}

    switch (content_type){
        case HTTP_CONTENT_TYPE_HTML:if(p_data != DEF_NULL){
             size_wr     =0u;
             size_wr_tot =0u;while(size_wr < data_len){
                 size_wr =FSFile_Wr(file_handle,
                                    p_data,
                                    data_len,&err);if(err.Code != RTOS_ERR_NONE){return;}
                 size_wr_tot += size_wr;}}break;

        case HTTP_CONTENT_TYPE_OCTET_STREAM:
        case HTTP_CONTENT_TYPE_PDF:
        case HTTP_CONTENT_TYPE_ZIP:
        case HTTP_CONTENT_TYPE_GIF:
        case HTTP_CONTENT_TYPE_JPEG:
        case HTTP_CONTENT_TYPE_PNG:
        case HTTP_CONTENT_TYPE_JS:
        case HTTP_CONTENT_TYPE_PLAIN:
        case HTTP_CONTENT_TYPE_CSS:
        case HTTP_CONTENT_TYPE_JSON:
        default:break;}FSFile_Close(file_handle,&err);}
```

HTTP Client WebSocket

WebSocket allows to create a full duplex communication channel over a single TCP/HTTP connection and enables streams of structured messages. It use simple and efficient headers to describe those messages which offer a better solution when it come to send small data at higher rate. For more details, refer to section WebSocket .

This section regroups topics to help developing a custom WebSocket application with the HTTP Client module. Examples are include in many sub-sections.

- HTTP Client Upgrading an HTTP Connection
- HTTP Client Message reception
- HTTP Client Sending Message
- HTTP Client Closing a Connection

HTTP Client Upgrading an HTTP Connection

Because it uses many HTTP functionalities for opening handshake, the WebSocket protocol has been integrated has part of Micrium OS HTTP Client module. Furthermore, all the HTTP Client request features, such as query string, are available during the opening handshake request. Here are the usual steps to upgrade a HTTP connection to WebSocket:

1. First, create a HTTPc Connection to the desired server using HTTPc_ConnOpen(). If successful, it will return a valid HTTPc_CONN_OBJ object that can be used for the next request.
2. Before sending an Upgrade Request to the server, an HTTPc_WEBSOCK_OBJ object may be configured by using HTTPc_WebSockSetParam() function. Note that WebSocket Message Reception is done through hook functions that are configured in HTTPc_WEBSOCK_OBJ object passed during the WebSocket Upgrade.
3. Because the WebSocket opening handshake is essentially a specific HTTP Request with specific headers, HTTPc_WebSockUpgrade() function is available to simplify the connection upgrade. It will automatically set the mandatory headers and validates the response from the server. Because it also uses a HTTPc_REQ_OBJ object, you can use all the features available during a normal HTTP Request.
4. Finally, when the opening handshake is successful, the connection that has been established in step 1 has now upgraded to WebSocket and is no more related to HTTP. At this point, HTTPc_WebSockSend() can be used for WebSocket Message Transmission and the WebSocket Message Reception will be processed through the RX hook configured in HTTPc_WEBSOCK_OBJ.

Related Hooks

Name	Description	Object	Parameter Type
On Open	Called when a WebSocket Upgrade is successful.	HTTPc_WEBSOCK_OBJ	HTTPc_PARAM_TYPE_WEBSOCK_ON_OPEN
On Close	Called when a connection that has been successfully upgraded close.	HTTPc_WEBSOCK_OBJ	HTTPc_PARAM_TYPE_WEBSOCK_ON_CLOSE

Example

The following example simply shows how to upgrade an HTTP connection to Websocket. As you can see below, it will first open an HTTP connection to the remote server and then upgrade it. Note that there's no Message Reception mechanism configured in this example.

Listing - Upgrading an HTTP Connection

```
#include <rtos/net/include/http_client.h>

/*
********************************************************************************************************
*                                    LOCAL DEFINES
********************************************************************************************************
*/
#define  HTTPc_EXAMPLE_URI            "server.example.com"
#define  HTTPc_EXAMPLE_RESSOURCE        "/echo"
#define  HTTPc_EXAMPLE_CONN_BUF_LEN     1024

/*
********************************************************************************************************
*                                 LOCAL GLOBAL VARIABLES
********************************************************************************************************
*/

static  HTTPc_CONN_OBJ        HTTPcExample_Conn;
static  CPU_CHAR              HTTPcExample_ConnBuf[HTTPc_EXAMPLE_CONN_BUF_LEN];
static  HTTPc_REQ_OBJ         HTTPcExample_Req;
static  HTTPc_RESP_OBJ        HTTPcExample_Resp;
static  HTTPc_WEBSOCK_OBJ     HTTPcExample_WebSock;

/*
********************************************************************************************************
*                               LOCAL FUNCTION PROTOTYPES
********************************************************************************************************
*/

static void  HTTPcExample_ConnOnClose   (HTTPc_CONN_OBJ           *p_conn,
                                         HTTPc_CONN_CLOSE_STATUS    close_status,
                                         HTTPc_ERR                  err);
```

```
static void  HTTPcExample_WebSockOnOpen  (HTTPc_CONN_OBJ          *p_conn);

static void  HTTPcExample_WebSockOnClose (HTTPc_CONN_OBJ           *p_conn,
                                          HTTPc_WEBSOCK_CLOSE_CODE     close_code,
                                          HTTPc_WEBSOCK_CLOSE_REASON  *p_reason);/*
*********************************************************************************************************
*                                               EXAMPLE
*********************************************************************************************************
*/

CPU_BOOLEAN  HTTPcExample_OpenWebSocket (void){
    CPU_BOOLEAN  result;
    RTOS_ERR     err;/* ----------- INIT NEW CONNECTION & REQUEST ----- *//* Always Clear the object.
*/HTTPc_ConnClr(&HTTPcExample_Conn,&err);if(err.Code != RTOS_ERR_NONE)
{return(DEF_FAIL);}HTTPc_ReqClr(&HTTPcExample_Req,&err);if(err.Code != RTOS_ERR_NONE)
{return(DEF_FAIL);}HTTPc_WebSockClr(&HTTPcExample_WebSock,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}/* ------------- SET ON
CONNECTION CLOSE --------- *//* This callback is always required.             */HTTPc_ConnSetParam(&HTTPcExample_Conn,
                                HTTPc_PARAM_TYPE_CONN_CLOSE_CALLBACK,(void *) HTTPcExample_ConnOnClose,&err);if(err.Code != RTOS_ERR_NONE)
{return(DEF_FAIL);}/* ----------------- OPEN CONNECTION ------------- */
    result =HTTPc_ConnOpen(&HTTPcExample_Conn,
                              HTTPcExample_ConnBuf,
                              HTTPc_EXAMPLE_CONN_BUF_LEN,
                              HTTPc_EXAMPLE_URI,sizeof(HTTPc_EXAMPLE_URI),
                              HTTPc_FLAG_NONE,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}/* --------------- SET WEBSOCK ON OPEN -----------
*/HTTPc_WebSockSetParam(&HTTPcExample_WebSock,
                                   HTTPc_PARAM_TYPE_WEBSOCK_ON_OPEN,(void *) HTTPcExample_WebSockOnOpen,&err);if(err.Code != RTOS_ERR_NONE)
{return(DEF_FAIL);}/* --------------- SET WEBSOCK ON CLOSE ---------- */HTTPc_WebSockSetParam(&HTTPcExample_WebSock,
                                   HTTPc_PARAM_TYPE_WEBSOCK_ON_CLOSE,(void *) HTTPcExample_WebSockOnClose,&err);if(err.Code != RTOS_ERR_NONE)
{return(DEF_FAIL);}/* ------- UPGRADE HTTP CONNECTION TO WEBSOCKET -- */
    result =HTTPc_WebSockUpgrade(&HTTPcExample_Conn,&HTTPcExample_Req,&HTTPcExample_Resp,&HTTPcExample_WebSock,
                                   HTTPc_EXAMPLE_RESSOURCE,sizeof(HTTPc_EXAMPLE_RESSOURCE),
                                   HTTPc_FLAG_NONE,&err);if(result != DEF_OK){return(DEF_FAIL);}return(DEF_OK);}/*
*********************************************************************************************************
*                                  CONNECTION ON CLOSE CALLBACK
*********************************************************************************************************
*/

static void  HTTPcExample_ConnOnClose (HTTPc_CONN_OBJ          *p_conn,
                                       HTTPc_CONN_CLOSE_STATUS   close_status,
                                       RTOS_ERR                  err){/* Called  when a connection closes.             */}/*
*********************************************************************************************************
*                                    WEBSOCK ON OPEN CALLBACK
*********************************************************************************************************
*/

static void  HTTPcExample_WebSockOnOpen (HTTPc_CONN_OBJ          *p_conn){/* Called  when a WebSocket Upgrade is successful. */}/*
*********************************************************************************************************
*                                    WEBSOCK ON CLOSE CALLBACK
*********************************************************************************************************
*/

static void  HTTPcExample_WebSockOnClose (HTTPc_CONN_OBJ           *p_conn,
                                          HTTPc_WEBSOCK_CLOSE_CODE     close_code,
                                          HTTPc_WEBSOCK_CLOSE_REASON  *p_reason){/* Called when a WebSocket Connection has closed.  */}
```

HTTP Client Message reception

In the WebSocket Protocol, data is exchanged using sequences of frame to form WebSocket Messages. The user doesn't need to know how the WebSocket data framing works since the HTTP Client WebSocket module manages it internally. The HTTP Client WebSocket API simplifies the application development by using the concept of message which can be summarized by its Type, Payload Length and Contents.

WebSocket Messages can have 5 different types divided in 2 groups.

Control Message

- Close
- Ping
- Pong

Data Message

- Text
- Binary

Control Message

Control messages are intended to be used for protocol-level signaling. Note that their payload length cannot exceed 125 bytes and they cannot be fragmented (only one frame per message).

Since control messages are protocol related, they are managed directly by the WebSocket Module which provides hook to notify the application.

Message Type	WebSocket Module Behavior
Close	When a Close message is received from the remote server, the WebSocket module will start the closing handshake. Once it's done, successfully or not, it will call the hook On Close
Ping	When a Ping message is received from the remote server, the WebSocket module will reply the correct Pong message. The application won't be notify.
Pong	When a Pong message is received from the remote server, the WebSocket module will simply call the hook On Pong .

Data Messages

Data messages are intended to transport information. Their length can vary from 0 to 2^64 and can be fragmented. They can be divided in two type.

Message Type	
Text	The content must respected UTF-8 standard.
Binary	The content is not restricted.

Note that the HTTP Client WebSocket module doesn't validate UTF-8 and must be done by the user application if necessary.

The reception of any Data Messages are done through hooks that are configured in the HTTPc_WEBSOCK_OBJ and that must be set before the connection upgrade request (HTTPc_WebSockUpgrade()).

Name	Description	Object	Parameter Type
On Message Reception Initialization	Called at the beginning of the WebSocket Data Type message reception.	HTTPc_WEBSOCK_OBJ	HTTPc_PARAM_TYPE_WEBSOCK_ON_MSG_RX_INIT
On Message RX Data	Called each time a chunk of the message is received is available.	HTTPc_WEBSOCK_OBJ	HTTPc_PARAM_TYPE_WEBSOCK_ON_MSG_RX_DATA

Name	Description	Object	Parameter Type
On Message RX Complete	Called when the Data message is completely received.	HTTPc_WEBSOCK_OBJ	HTTPc_PARAM_TYPE_WEBSOCK_ON_MSG_RX_COMPLETE

Note that only one Data Message is processed at a time.

Data Message Reception Mode

The hooks described in the previous section can be use in many different ways. However, those can be group in two different mode.

Normal mode

Normal Mode uses all of the three Data Message Reception hooks. It allow the user to be notify every time a message payload chunk is available with the On Message RX Data hook. Thus, the application can parse a message during its reception without having to copying it all in a buffer.

This mode is useful when the memory footprints is critical and the message length can be long or undefined.

Normal Mode Example

Auto mode

Auto Mode only use two hooks. During On Message Reception Initialization , it is possible to set a pointer to the buffer where the message payload must be set. If the application do so, the message will be automatically copied to the specified buffer. Once the message is completely received, On Message RX Complete hook will be called and data will be available is the buffer previously set.

This mode is useful for simpler application which the message length received from other endpoint are small and size-limited

Auto Mode Example

HTTP Client Auto Mode Example

This example shows how to open a WebSocket with Data message Reception hooks configured in Auto Mode.

Listing - HTTP Client Auto Mode

```
/*
*********************************************************************************************************
*                                          LOCAL DEFINES
*********************************************************************************************************
*/

#define  HTTPc_EXAMPLE_URI            "server.example.com"
#define  HTTPc_EXAMPLE_RESSOURCE       "/echo"
#define  HTTPc_EXAMPLE_CONN_BUF_LEN    1024
#define  HTTPc_EXAMPLE_MSG_MAX_LEN     512

/*
*********************************************************************************************************
*                                       LOCAL GLOBAL VARIABLES
*********************************************************************************************************
*/

static  HTTPc_CONN_OBJ       HTTPcExample_Conn;
static  CPU_CHAR             HTTPcExample_ConnBuf[HTTPc_EXAMPLE_CONN_BUF_LEN];
static  HTTPc_REQ_OBJ        HTTPcExample_Req;
static  HTTPc_RESP_OBJ       HTTPcExample_Resp;
static  HTTPc_WEBSOCK_OBJ    HTTPcExample_WebSock;
static  CPU_CHAR             HTTPcExample_RxBuf[HTTPc_EXAMPLE_MSG_MAX_LEN];

/*
```

```
*********************************************************************************************************
*                                  LOCAL FUNCTION PROTOTYPES
*********************************************************************************************************
*/

static void      HTTPcExample_ConnOnClose          (HTTPc_CONN_OBJ          *p_conn,
                                     HTTPc_CONN_CLOSE_STATUS    close_status,
                                     HTTPc_ERR                  err);

static void      HTTPcExample_WebSockOnMsgRxInit    (HTTPc_CONN_OBJ          *p_conn,
                                     HTTPc_WEBSOCK_MSG_TYPE     msg_type,
                                     CPU_INT32U                 msg_len,
                                     void                     **p_data);

static void      HTTPcExample_WebSockOnMsgRxComplete (HTTPc_CONN_OBJ          *p_conn);/*
*********************************************************************************************************
*                                        EXAMPLE
*********************************************************************************************************
*/

CPU_BOOLEAN  HTTPcExample_OpenWebSocket (void){
   CPU_BOOLEAN  result;
   RTOS_ERR     err;/* ----------- INIT NEW CONNECTION & REQUEST ---------- *//* Always Clear the object.
*/HTTPc_ConnClr(&HTTPcExample_Conn,&err);if(err.Code != RTOS_ERR_NONE)
{return(DEF_FAIL);}HTTPc_ReqClr(&HTTPcExample_Req,&err);if(err.Code != RTOS_ERR_NONE)
{return(DEF_FAIL);}HTTPc_WebSockClr(&HTTPcExample_WebSock,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}/* ------------- SET ON
CONNECTION CLOSE -------------- *//* This callback is always required.              */HTTPc_ConnSetParam(&HTTPcExample_Conn,
                        HTTPc_PARAM_TYPE_CONN_CLOSE_CALLBACK,(void *) HTTPcExample_ConnOnClose,&err);if(err.Code != RTOS_ERR_NONE)
{return(DEF_FAIL);}/* ----------------- OPEN CONNECTION ------------------ */
   result =HTTPc_ConnOpen(&HTTPcExample_Conn,
                     HTTPcExample_ConnBuf,
                     HTTPc_EXAMPLE_CONN_BUF_LEN,
                     HTTPc_EXAMPLE_URI,sizeof(HTTPc_EXAMPLE_URI),
                     HTTPc_FLAG_NONE,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}/* ----------- SET WEBSOCK HOOKS/CALLBACKS ----
-------- *//* Set OnMsgRxInit hooks.                    */HTTPc_WebSockSetParam(&HTTPcExample_WebSock,
                         HTTPc_PARAM_TYPE_WEBSOCK_ON_MSG_RX_INIT,(void *) HTTPcExample_WebSockOnMsgRxInit,&err);if(err.Code !=
RTOS_ERR_NONE){return(DEF_FAIL);}/* Set OnMsgRxComplete hooks.                 */HTTPc_WebSockSetParam(&HTTPcExample_WebSock,
                         HTTPc_PARAM_TYPE_WEBSOCK_ON_MSG_RX_COMPLETE,(void *)
HTTPcExample_WebSockOnMsgRxComplete,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}/* ------- UPGRADE HTTP CONNECTION TO
WEBSOCKET ------- */
   result =HTTPc_WebSockUpgrade(&HTTPcExample_Conn,&HTTPcExample_Req,&HTTPcExample_Resp,&HTTPcExample_WebSock,
                         HTTPc_EXAMPLE_RESSOURCE,sizeof(HTTPc_EXAMPLE_RESSOURCE),
                         HTTPc_FLAG_NONE,&err);if(result != DEF_OK){return(DEF_FAIL);}return(DEF_OK);}/*
*********************************************************************************************************
*                                  CONNECTION ON CLOSE CALLBACK
*********************************************************************************************************
*/

static void  HTTPcExample_ConnOnClose (HTTPc_CONN_OBJ          *p_conn,
                               HTTPc_CONN_CLOSE_STATUS   close_status,
                               HTTPc_ERR                 err){/* Called  when a connection close.               */}/*
*********************************************************************************************************
*                                 WEBSOCK ON MSG RX INIT HOOK
*********************************************************************************************************
*/

static void  HTTPcExample_WebSockOnMsgRxInit (HTTPc_CONN_OBJ          *p_conn,
                                    HTTPc_WEBSOCK_MSG_TYPE     msg_type,
                                    CPU_INT32U                 msg_len,
                                    void                     **p_data){/* Validate that we have enough space in the buffer...  *//* ...to get all the message.
*/if(msg_len <= HTTPc_EXAMPLE_MSG_MAX_LEN){*p_data = HTTPcExample_RxBuf;/* Set the pointer where the message data must be set.  */}}/*
*********************************************************************************************************
*                               WEBSOCK ON MSG RX COMPLETE HOOK
*********************************************************************************************************
*/

static void  HTTPcExample_WebSockOnMsgRxComplete (HTTPc_CONN_OBJ *p_conn){/* Once the message is completely received...        *//*
...Do Something.                          */}
```

HTTP Client Normal Mode Example

This example show how to open a WebSocket with Data message Reception hooks configured in Normal Mode.

Listing - HTTP Client Normal Mode

```
#include <rtos/net/include/http_client.h>

/*
*********************************************************************************************************
*                                            LOCAL DEFINES
*********************************************************************************************************
*/

#define  HTTPc_EXAMPLE_URI              "server.example.com"
#define  HTTPc_EXAMPLE_RESSOURCE        "/echo"
#define  HTTPc_EXAMPLE_CONN_BUF_LEN     1024
#define  HTTPc_EXAMPLE_MSG_MAX_LEN      512

/*
*********************************************************************************************************
*                                        LOCAL GLOBAL VARIABLES
*********************************************************************************************************
*/

static  HTTPc_CONN_OBJ        HTTPcExample_Conn;
static  CPU_CHAR              HTTPcExample_ConnBuf[HTTPc_EXAMPLE_CONN_BUF_LEN];
static  HTTPc_REQ_OBJ         HTTPcExample_Req;
static  HTTPc_RESP_OBJ        HTTPcExample_Resp;
static  HTTPc_WEBSOCK_OBJ     HTTPcExample_WebSock;

static  CPU_CHAR              HTTPcExample_RxBuf[HTTPc_EXAMPLE_MSG_MAX_LEN];
static  CPU_INT32U            HTTPcExample_RxMsgLenToGet ;

/*
*********************************************************************************************************
*                                      LOCAL FUNCTION PROTOTYPES
*********************************************************************************************************
*/

static void        HTTPcExample_ConnOnClose            (HTTPc_CONN_OBJ            *p_conn,
                                                        HTTPc_CONN_CLOSE_STATUS    close_status,
                                                        HTTPc_ERR                  err);

static void        HTTPcExample_WebSockOnMsgRxInit     (HTTPc_CONN_OBJ            *p_conn,
                                                        HTTPc_WEBSOCK_MSG_TYPE     msg_type,
                                                        CPU_INT32U                 msg_len,
                                                        void                     **p_data);

static CPU_INT32U  HTTPcExample_WebSockOnMsgRxData     (HTTPc_CONN_OBJ            *p_conn,
                                                        HTTPc_WEBSOCK_MSG_TYPE     msg_type,
                                                        void                      *p_data,
                                                        CPU_INT16U                 data_len);

static void        HTTPcExample_WebSockOnMsgRxComplete (HTTPc_CONN_OBJ            *p_conn);

/*
*********************************************************************************************************
*                                               EXAMPLE
*********************************************************************************************************
*/

CPU_BOOLEAN  HTTPcExample_OpenWebSocket (void)
{
    CPU_BOOLEAN  result;
    RTOS_ERR     err;
                                         /* ----------- INIT NEW CONNECTION & REQUEST ---------- */
                                         /* Always Clear the object.                              */
    HTTPc_ConnClr(&HTTPcExample_Conn, &err);
    if (err.Code != RTOS_ERR_NONE) {
        return (DEF_FAIL);
    }
```

```
if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}HTTPc_WebSockClr(&HTTPcExample_WebSock,&err);if(err.Code != RTOS_ERR_NONE)
{return(DEF_FAIL);}/* ------------- SET ON CONNECTION CLOSE -------------- *//* This callback is always required.
*/HTTPc_ConnSetParam(&HTTPcExample_Conn,
                        HTTPc_PARAM_TYPE_CONN_CLOSE_CALLBACK,(void *) HTTPcExample_ConnOnClose,&err);if(err.Code != RTOS_ERR_NONE)
{return(DEF_FAIL);}/* ----------------- OPEN CONNECTION ------------------ */
   result =HTTPc_ConnOpen(&HTTPcExample_Conn,
                      HTTPcExample_ConnBuf,
                      HTTPc_EXAMPLE_CONN_BUF_LEN,
                      HTTPc_EXAMPLE_URI,sizeof(HTTPc_EXAMPLE_URI),
                      HTTPc_FLAG_NONE,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}/* ----------- SET WEBSOCK HOOKS/CALLBACKS ----
-------- *//* Set OnMsgRxInit hooks.                        */HTTPc_WebSockSetParam(&HTTPcExample_WebSock,
                          HTTPc_PARAM_TYPE_WEBSOCK_ON_MSG_RX_INIT,(void *) HTTPcExample_WebSockOnMsgRxInit,&err);if(err.Code !=
RTOS_ERR_NONE){return(DEF_FAIL);}/* Set OnMsgRxData hooks.                        */HTTPc_WebSockSetParam(&HTTPcExample_WebSock,
                          HTTPc_PARAM_TYPE_WEBSOCK_ON_MSG_RX_DATA,(void *) HTTPcExample_WebSockOnMsgRxData,&err);if(err.Code !=
RTOS_ERR_NONE){return(DEF_FAIL);}/* Set OnMsgRxComplete hooks.                    */HTTPc_WebSockSetParam(&HTTPcExample_WebSock,
                          HTTPc_PARAM_TYPE_WEBSOCK_ON_MSG_RX_COMPLETE,(void *)
HTTPcExample_WebSockOnMsgRxComplete,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}/* ------- UPGRADE HTTP CONNECTION TO
WEBSOCKET ------- */
   result =HTTPc_WebSockUpgrade(&HTTPcExample_Conn,&HTTPcExample_Req,&HTTPcExample_Resp,&HTTPcExample_WebSock,
                          HTTPc_EXAMPLE_RESSOURCE,sizeof(HTTPc_EXAMPLE_RESSOURCE),
                          HTTPc_FLAG_NONE,&err);if(result != DEF_OK){return(DEF_FAIL);}return(DEF_OK);}/*
*********************************************************************************************************
*                               CONNECTION ON CLOSE CALLBACK
*********************************************************************************************************
*/

static void  HTTPcExample_ConnOnClose (HTTPc_CONN_OBJ         *p_conn,
                              HTTPc_CONN_CLOSE_STATUS   close_status,
                              HTTPc_ERR               err){/* Called  when a connection close.                  */}/*
*********************************************************************************************************
*                              WEBSOCK ON MSG RX INIT HOOK
*********************************************************************************************************
*/

static void  HTTPcExample_WebSockOnMsgRxInit (HTTPc_CONN_OBJ          *p_conn,
                                   HTTPc_WEBSOCK_MSG_TYPE     msg_type,
                                   CPU_INT32U              msg_len,
                                   void                   **p_data){/* Validate that we have enough space in the buffer...  *//* ...to get all the message.
*/if(msg_len <= HTTPc_EXAMPLE_MSG_MAX_LEN){
       HTTPcExample_RxMsgLenToGet = msg_len;}else{
       HTTPcExample_RxMsgLenToGet =0u;}}/*
*********************************************************************************************************
*                              WEBSOCK ON MSG RX DATA HOOK
*********************************************************************************************************
*/

static CPU_INT32U  HTTPcExample_WebSockOnMsgRxData (HTTPc_CONN_OBJ          *p_conn,
                                      HTTPc_WEBSOCK_MSG_TYPE     msg_type,
                                      void                  *p_data,
                                      CPU_INT16U              data_len){
   CPU_INT16U  data_len_used;

   data_len_used =0u;if(HTTPcExample_RxMsgLenToGet !=0u){if(HTTPcExample_RxMsgLenToGet == data_len){/* In this example, wait to have all
the message...    *//* ...before copying it in the reception buffer.      */Mem_Copy(HTTPcExample_RxBuf, p_data, data_len);
         HTTPcExample_RxMsgLenToGet =0u;
         data_len_used            = data_len;}}return(data_len_used);/* Return that number of bytes used.                 */}/*
*********************************************************************************************************
*                             WEBSOCK ON MSG _RX COMLETE HOOK
*********************************************************************************************************
*/

static void  HTTPcExample_WebSockOnMsgRxComplete (HTTPc_CONN_OBJ  *p_conn){/* Once the message is completely received...          *//*
...Do Something.                        */}
```

HTTP Client Sending Message

The user can send a WebSocket message with HTTPc_WebSockSend() function and a properly initialized HTTPc_WEBSOCK_MSG_OBJ. To set the type, the payload content and length of a message, it is possible to either configure it with HTTPc_WebSockSend() parameters or with hooks configured for each message.

For message transmission, it's possible to send all the message type available :

- HTTPc_WEBSOCK_MSG_TYPE_TXT
- HTTPc_WEBSOCK_MSG_TYPE_BIN
- HTTPc_WEBSOCK_MSG_TYPE_CLOSE
- HTTPc_WEBSOCK_MSG_TYPE_PING
- HTTPc_WEBSOCK_MSG_TYPE_PONG

Here a list of the available Transmission hooks.

Name	Description	Object	Parameter Type
On Message TX Initialization	Called when the message is ready to be sent.	HTTPc_WEBSOCK_OBJ	HTTPc_PARAM_TYPE_WEBSOCK_ON_MSG_TX_INIT
On Message TX Data	Called to set the internal connection buffer of a data chunk of a message to send.	HTTPc_WEBSOCK_OBJ	HTTPc_PARAM_TYPE_WEBSOCK_ON_MSG_TX_DATA
On Message TX Complete	Called when the message is completely transmitted.	HTTPc_WEBSOCK_OBJ	HTTPc_PARAM_TYPE_WEBSOCK_ON_MSG_TX_COMPLETE

Transmission Mode

The hooks described in the previous section can be use in many different ways. However, those can be group in three different mode.

Mode	Buffer used	Message Length	On Message TX Initialization	On Message TX Data	On Message TX Complete
Normal	External data buffer set when HTTPc_WebSockSend() is called.	Defined at the beginning when HTTPc_WebSockSend() is called.	Not Used	Not Used	Optional
Hooks	Internal Connection data buffer is used to construct message to transmit.	Defined either when HTTPc_WebSockSend() is called or with On Message TX Initialization hook.	Optional	Used	Optional
Dynamic	Internal Connection data buffer is used to construct message to transmit.	Defined either when HTTPc_WebSockSend() is called or with On Message TX Initialization hook and *must* be HTTPc_WEBSOCK_TX_MSG_LEN_NOT_DEFINED.	Optional	Used	Optional

Normal Mode

Normal mode only uses only HTTPc_WebSockSend() and doesn't require a hook. The application must provide a pointer to the message payload data and its length that will be used during the transmission.

This mode is simple to use and suitable for a non-complex application.

Hook Mode

Hook mode allows you to use directly the connection buffer to send a message using hooks.

- The application must provide the length of the message at the beginning either with HTTPc_WebSockSend() or with the On Message TX Initialization hook.
- Then, On Message TX Data is called and provide the application a pointer to the internal connection buffer and the available length.
- Once the application has set the buffer, it must return the total number of bytes used in this buffer.
- On Message TX Data is called until the message is completely sent, thus the length set at the beginning matches the total length sent.

This mode allows you to save memory space by using directly the connection buffer. However, the message length must be known at the beginning.

Dynamic Mode

Dynamic mode use the message fragmentation mechanism that the WebSocket protocol provides to transfer message. It allows to send a message without knowing the total message and to use directly the connection buffer to construct the message to send, thereby reducing the memory footprints.

To use this mode, two points are important:

- The length of the message must be set to HTTPc_WEBSOCK_TX_MSG_LEN_NOT_DEFINED. It possible to either set it during HTTPc_WebSockSend() is called or using the On Message TX Initialization hook.
- Then, On Message TX Data is called and provide the application a pointer to the internal connection buffer and the available length.
- Each time the On Message TX Data is called, the application must set the buffer and return the numbers of bytes to be sent.
- To complete the message, 0 value must be returned from this function.

This mode helps saving memory space by using directly the connection buffer to construct messages and allows to start the data transmission without knowing the total message length.

> Note that it is not possible to send Control Message (Close, Ping or Pong) in Dynamic Mode. Those messages must be sent either in Hook Mode or Normal Mode.

Non-Blocking Option

An non-blocking option is available when HTTPc_WebSockSend() is called by using the flag HTTPc_FLAG_WEBSOCK_NO_BLOCK. If used, On Message TX Complete hook must be set. It allows to queue message without having to wait the completion of each message transmission.

HTTP Client Closing a Connection

WebSockets can be closed properly by sending a Close Message. Close Messages have a special format which the first 2 bytes is the close code defined by the standard. The rest of the data concatenated is usually defined by the user for debug purpose. When the other end receive the close message, it should reply with the exact same message. However, it is possible that endpoints don't agree on the value of the close code, but the connection will be closed when both have received and transmitted a Close Messages.

Closing messages are sent in the same way that other message are sent using HTTPc_WebSockSend() (see Sending Message). Note that Control message payload length cannot exceed 125 bytes and cannot be sent in Dynamic Mode. Once the closing handshake is done, the WebSocket Module will notify the application if the On Close hook is properly set.

Finally, HTTPc_WebSockFmtCloseMsg() allows the user to properly format a Close Message into a buffer before sending it.

HTTP Client Protocol Recognized Fields

This section regroups lists of all the HTTP fields recognized by Micrium OS HTTP Client module.

- HTTP Versions
- HTTP Methods

- HTTP Header Fields
- HTTP Content Types
- HTTP Status Codes

HTTP Versions

HTTP Version	Micrium Type
0.9	HTTP_PROTOCOL_VER_0_9
1.0	HTTP_PROTOCOL_VER_1_0
1.1	HTTP_PROTOCOL_VER_1_1

HTTP Methods

HTTP Method	Micrium Type
CONNECT	HTTP_METHOD_CONNECT
DELETE	HTTP_METHOD_DELETE
GET	HTTP_METHOD_GET
HEAD	HTTP_METHOD_HEAD
OPTIONS	HTTP_METHOD_OPTIONS
POST	HTTP_METHOD_POST
PUT	HTTP_METHOD_PUT
TRACE	HTTP_METHOD_TRACE

Those are the methods recognized by the HTTP Server and Client stacks. It doesn't mean that all those methods are supported.

HTTP Header Fields

Header Field Type	Micrium Type
Content-Type	HTTP_HDR_FIELD_CONTENT_TYPE
Content-Length	HTTP_HDR_FIELD_CONTENT_LEN
Content-Disposition	HTTP_HDR_FILED_CONTENT_DISPOSITION
Host	HTTP_HDR_FIELD_HOST
Location	HTTP_HDR_FIELD_LOCATION
Connection	HTTP_HDR_FIELD_CONN
Transfer-Encoding	HTTP_HDR_FIELD_TRANSFER_ENCODING
Accept	HTTP_HDR_FIELD_ACCEPT
Accept-Charset	HTTP_HDR_FIELD_ACCEPT_CHARSET
Accept-Encoding	HTTP_HDR_FIELD_ACCEPT_ENCODING
Accept-Language	HTTP_HDR_FIELD_ACCEPT_LANGUAGE
Accept-Ranges	HTTP_HDR_FIELD_ACCEPT_RANGES
Age	HTTP_HDR_FIELD_AGE
Allow	HTTP_HDR_FIELD_ALLOW
Authorization	HTTP_HDR_FIELD_AUTHORIZATION
Content-Encoding	HTTP_HDR_FIELD_CONTENT_ENCODING

Header Field Type	Micrium Type
Content-Language	HTTP_HDR_FIELD_CONTENT_LANGUAGE
Content-Location	HTTP_HDR_FIELD_CONTENT_LOCATION
Content-MD5	HTTP_HDR_FIELD_CONTENT_MD5
Content-Range	HTTP_HDR_FIELD_CONTENT_RANGE
Cookie	HTTP_HDR_FIELD_COOKIE
Cookie2	HTTP_HDR_FIELD_COOKIE2
Date	HTTP_HDR_FIELD_DATE
ETag	HTTP_HDR_FEILD_ETAG
Expect	HTTP_HDR_FIELD_EXPECT
Expires	HTTP_HDR_FIELD_EXPIRES
From	HTTP_HDR_FIELD_FROM
If-Modified-Since	HTTP_HDR_FIELD_IF_MODIFIED_SINCE
If-Match	HTTP_HDR_FIELD_IF_MATCH
If-None-Match	HTTP_HDR_FIELD_IF_NONE_MATCH
If-Range	HTTP_HDR_FIELD_IF_RANGE
If-Unmodified-Since	HTTP_HDR_FIELD_IF_UNMODIFIED_SINCE
Last-Modified	HTTP_HDR_FIELD_LAST_MODIFIED
Range	HTTP_HDR_FIELD_RANGE
Referrer	HTTP_HDR_FIELD_REFERER
Retry-After	HTTP_HDR_FIELD_RETRY_AFTER
Server	HTTP_HDR_FIELD_SERVER
Set-Cookie	HTTP_HDR_FIELD_SET_COOKIE
Set-Cookie2	HTTP_HDR_FIELD_SET_COOKIE2
TE	HTTP_HDR_FIELD_TE
Trailer	HTTP_HDR_FIELD_TRAILER
Upgrade	HTTP_HDR_FIELD_UPGRATE
User-Agent	HTTP_HDR_FIELD_USER_AGENT
Vary	HTTP_HDR_FIELD_VARY
Via	HTTP_HDR_FIELD_VIA
Warning	HTTP_HDR_FIELD_WARNING
WWW-Authenticate	HTTP_HDR_FIELD_WWW_AUTHENTICATE

If a header is missing for your application, contact your sales representative so that it can be included in subsequent release of Micrium OS.

HTTP Content Types

MIME Content Type	File Extension	Micrium Type
text/html	.html	HTTP_CONTENT_TYPE_HTML
application/octet-stream	.bin	HTTP_CONTENT_TYPE_OCTET_STREAM
application/pdf	.pdf	HTTP_CONTENT_TYPE_PDF
application/zip	.zip	HTTP_CONTENT_TYPE_ZIP

MIME Content Type	File Extension	Micrium Type
image/gif	.gif	HTTP_CONTENT_TYPE_GIF
image/jpeg	.jpeg, .jpg	HTTP_CONTENT_TYPE_JPEG
image/png	.png	HTTP_CONTENT_TYPE_PNG
application/javascript	.js	HTTP_CONTENT_TYPE_JS
text/plain	.txt	HTTP_CONTENT_TYPE_PLAIN
text/css	.css	HTTP_CONTENT_TYPE_CSS
application/json	.json	HTTP_CONTENT_TYPE_JSON
application/x-www-form-urlencoded	-	HTTP_CONTENT_TYPE_APP_FORM
multipart/form-data	-	HTTP_CONTENT_TYPE_MULTIPART_FORM

If a Content Type is missing for your application, contact your sales representative so that it can be included in subsequent release of Micrium OS.

HTTP Status Codes

HTTP Status Code	Micrium Type
200	HTTP_STATUS_OK
201	HTTP_STATUS_CREATED
202	HTTP_STATUS_ACCEPTED
204	HTTP_STATUS_NO_CONTENT
205	HTTP_STATUS_RESET_CONTENT
301	HTTP_STATUS_MOVED_PERMANENTLY
302	HTTP_STATUS_FOUND
303	HTTP_STATUS_SEE_OTHER
304	HTTP_STATUS_NOT_MODIFIED
305	HTTP_STATUS_USE_PROXY
307	HTTP_STATUS_TEMPORARY_REDIRECT
400	HTTP_STATUS_BAD_REQUEST
401	HTTP_STATUS_UNAUTHORIZED
403	HTTP_STATUS_FORBIDDEN
404	HTTP_STATUS_NOT_FOUND
405	HTTP_STATUS_METHOD_NOT_ALLOWED
406	HTTP_STATUS_NOT_ACCEPTABLE
408	HTTP_STATUS_REQUEST_TIMEOUT
409	HTTP_STATUS_CONFLICT
410	HTTP_STATUS_GONE
411	HTTP_STATUS_LENGTH_REQUIRED
412	HTTP_STATUS_PRECONDITION_FAILED
413	HTTP_STATUS_REQUEST_ENTITY_TOO_LARGE
414	HTTP_STATUS_REQUEST_URI_TOO_LONG
415	HTTP_STATUS_UNSUPPORTED_MEDIA_TYPE
416	HTTP_STATUS_REQUESTED_RANGE_NOT_SATISFIABLE

HTTP Status Code	Micrium Type
417	HTTP_STATUS_EXPECTATION_FAILED
500	HTTP_STATUS_INTERNAL_SERVER_ERR
501	HTTP_STATUS_NOT_IMPLEMENTED
503	HTTP_STATUS_SERVICE_UNAVAILABLE
505	HTTP_STATUS_HTTP_VERSION_NOT_SUPPORTED

If a Status Code is missing for your application, contact your sales representative so that it can be included in subsequent release of Micrium OS.

HTTP Server Module

The Micrium HTTP Server can be used in the more traditional way, meaning used to store web pages and web resources that can be fetched through Web Browsers. In that view, HTTP Server is compliant with all the available browsers (Firefox, Chrome, Safari, etc.) for computer systems.

HTTP Server can also be used in a more web service style, i.e., to fetch, modify and create web resources that are not necessarily files.

- HTTP Server Overview
- HTTP Server Example Applications
- HTTP Server Configuration
  - HTTP Server Compile-time Configuration
    - HTTP Server Compile-Time Configuration for Static File System
  - HTTP Server Run-Time Application Specific Configuration
  - HTTP Server Instance Configuration
    - HTTP Server Instance Secure Configuration
    - HTTP Server Instance File System Configuration
    - HTTP Server Instance Hook Configuration
    - HTTP Server Instance Header Configuration
    - HTTP Server Instance Query String Configuration
    - HTTP Server Instance Form Configuration
    - HTTP Server Instance Token Configuration
- HTTP Server Programming Guide
  - HTTP Server Interface with File System
  - HTTP Server Control Structures
  - HTTP Server Query String
  - HTTP Server Hook Functions
  - HTTP Server Header Fields
  - HTTP Server Request Body Data
  - HTTP Server Form
  - HTTP Server Response Body Data
  - HTTP Server Token Replacement
  - HTTP Server Proxy
  - HTTP Server Add-on
    - HTTP Server Control Layer
    - HTTP Server Authentication module
    - HTTP Server REST module
- HTTP Server Protocol Recognized Fields

HTTP Server Overview

- Specifications
- Features
- Limitations

Specifications

Complies with the following RFC:
  - RFC 2616 - Hypertext Transfer Protocol -- HTTP/1.1
- Implemented methods:
  - GET
  - POST
  - PUT
  - DELETE
  - HEAD

Features

- Scalable to contain only required features and minimize memory footprint.
- Add-on modules are also given to enhance and simplify the development of your HTTP server application.
- Supports file storage via Micrium OS File System or can be used via a "Static File System" which is provided inside the HTTP Server source code. The server can also be used without any File System.
- Supports multiple instances, i.e., it's possible to launch many web server on different TCP ports. This allows, for example, to have a part of a web application without security and another part with SSL/TLS security.
- Supports up to 255 connections simultaneously (using only one task).
- Allows application programmers to define "hook" functions, which are called by the HTTP Server stack when processing a request.
- Provides many (optional) built-in statistics and error counters that can be used for debugging or for logging usage statistics.
- Allows to specify a working directory for each HTTP server instance (Only when Micrium OS File System is used as a File System).
- Supports secure Web Servers (HTTPs).
- Supports persistent connections .
- Supports HTTP REST .
- Supports Dynamic Token Replacement in web pages and text files.
- Supports HTTP Query String
- Supports reception of HTTP headers in requests and addition of HTTP headers in responses.
- Supports application and multipart forms .
- Supports chunked transfer encoding in transmission.

Limitations

- No support for WebSockets yet.

HTTP Server Example Applications

This section describes the examples that are related to the HTTP Server module of Micrium OS.

- HTTP Server Initialization Example
  - Description
  - Configuration
    - Mandatory
  - Location
  - API
  - Notes
- Simple Server That Uses No File System
  - Description
  - Configuration
  - Location
  - API
- Basic Server That Uses the HTTP Static File System
  - Description
  - Configuration
  - Location
  - API
- Basic Server That Uses Micrium OS File System
  - Description
  - Configuration

- Location
- API
- Server That Handles REST Requests
  - Description
  - Configuration
  - Location
  - API
- Server That Handles Web pages, REST Requests and Authentication Support
  - Description
  - Configuration
  - Location
  - API
- Basic Secure Server That Uses SSL-TLS and the HTTP Static File System
  - Description
  - Configuration
  - Location
  - API

HTTP Server Initialization Example

Description

This is a generic example that shows how to initialize the HTTP Server module. It accomplishes the following tasks:

- Initialize the HTTP Server module
- Create a RAM disk for file exchange

Configuration

Mandatory

The following #define must be added in ex_description.h to allow other examples to initialize the HTTP Server module correctly:

#define	Description
EX_HTTP_SERVER_INIT_AVAIL	Lets the upper example layer know that the HTTP Server Initialization example is present and must be called by other examples.

Location

```
/examples/net/http/server/ex_http_server_init.c

/examples/net/http/server/ex_http_server.h
```

API

API	Description
Ex_HTTP_Server_Init()	Initialize the HTTP Server stack for the example application.

Notes

None.

Simple Server That Uses No File System

Description

This is a simple HTTP server example that demonstrates how to create a web page server without the use of a File System (Micrium OS File System or HTTP static FS).

The web server simply displays a "Hello World!" message on the web page once it is loaded.

Configuration

None.

Location

```
/examples/net/http/server/ex_http_server_no_fs.c

/examples/net/http/server/ex_http_server_hooks.c

/examples/net/http/server/ex_http_server_hooks.h

/examples/net/http/server/ex_http_server_init.c

/examples/net/http/server/ex_http_server.h
```

API

API	Description
Ex_HTTP_Server_InstanceCreateNoFS()	Initializes and starts a basic web server instance.

Basic Server That Uses the HTTP Static File System

Description

This is a basic HTTP server example that demonstrates how to create a web page server using the HTTP server static file system.

The web server displays some web pages and a form once it is loaded.

Configuration

None.

Location

```
/examples/net/http/server/ex_http_server_basic_static_fs.c

/examples/net/http/server/ex_http_server_hooks.c

/examples/net/http/server/ex_http_server_hooks.h

/examples/net/http/server/ex_http_server_init.c

/examples/net/http/server/ex_http_server.h
```

The content of the folder /examples/net/http/server/files

API

API	Description
Ex_HTTP_Server_InstanceCreateStaticFS()	Initializes and starts a basic web server instance. Retrieves files from the built-in static file system.

Basic Server That Uses Micrium OS File System

Description

This is a basic HTTP server example that demonstrates how to create a web page server. The functionalities are similar to the "Basic Server that Uses the HTTP static File System" example. However, the web pages are retrieved using Micrium OS File System.

The web server displays some web pages and a form once it is loaded.

Configuration

None.

Location

```
/examples/net/http/server/ex_http_server_basic_fs.c

/examples/net/http/server/ex_http_server_hooks.c

/examples/net/http/server/ex_http_server_hooks.h

/examples/net/http/server/ex_http_server_init.c

/examples/net/http/server/ex_http_server.h
```

- The content of the folder /examples/net/http/server/files

API

API	Description
Ex_HTTP_Server_InstanceCreateBasic()	Initializes and starts a basic web server instance. Retrieves files from the Micrium OS file system.

Server That Handles REST Requests

Description

This is a simple HTTP server example that demonstrates how to handle REST requests. It uses the HTTP static file system to store the web pages.

The web server displays some web pages and a form once it is loaded. It also offers a form that allows adding entries to a database via REST requests.

Configuration

None.

Location

```
/examples/net/http/server/ex_http_server_rest.c

/examples/net/http/server/ex_http_server_rest_hooks.c

/examples/net/http/server/ex_http_server_rest_hooks.h

/examples/net/http/server/ex_http_server_init.c

/examples/net/http/server/ex_http_server.h
```

- The content of the folder /examples/net/http/server/files

API

API	Description
Ex_HTTP_Server_InstanceCreateREST()	Initialize HTTPs REST Example application.

Server That Handles Webpages, REST Requests and Authentication Support

Description

This is an HTTP server example that demonstrates how to handle web pages, RESTrequestsand authentication in a single instance. It requires the use of the Control Layer. It uses the HTTP static file system to store the web pages.

The web server displays some web pages and a form once it is loaded. It also offers a form that allows adding entries to a database via REST requests.

Note that you will have to login first before accessing the pages. The example will create the following user accounts:

Username	admin	user0	user1
Password	password	<empty>	user1
Rights	Manager of HTTP user access right	HTTP user	HTTP user

Configuration

None.

Location

```
/examples/net/http/server/ex_http_server_ctrl_layer.c

/examples/net/http/server/ex_http_server_hooks.c

/examples/net/http/server/ex_http_server_hooks.h

/examples/net/http/server/ex_http_server_rest_hooks.c

/examples/net/http/server/ex_http_server_rest_hooks.h

/examples/net/http/server/ex_http_server_init.c

/examples/net/http/server/ex_http_server.h
```

- The content of the folder /examples/net/http/server/files

API

API	Description
Ex_HTTP_Server_InstanceCreateCtrlLayer()	Initialize HTTPs REST Example application.

Basic Secure Server That Uses SSL-TLS and the HTTP Static File System

Description

This is a basic HTTP server example that demonstrates how to create asecurewebserverusingSSL-TLS and the HTTP server static file system.

The web server displays some web pages and a form once it is loaded.

Configuration

None.

Location

```
/examples/net/http/server/ex_http_server_ssl_tls_static_fs.c

/examples/net/http/server/ex_http_server_hooks.c

/examples/net/http/server/ex_http_server_hooks.h

/examples/net/http/server/ex_http_server_init.c

/examples/net/http/server/ex_http_server.h
```

- The content of the folder /examples/net/http/server/files

API

API	Description
Ex_HTTP_Server_InstanceCreateSecure()	Initialize HTTPs REST Example application.

Notes

None

HTTP Server Configuration

In order to address your application's needs, the HTTP Server module must be properly configured. There are three groups of configuration parameters:

- HTTP Server Compile-time Configuration
- HTTP Server Run-Time Application Specific Configuration
- HTTP Server Instance Configuration

HTTP Server Compile-time Configuration

The HTTP Server module is configurable at compile time via several #defines located in http_server_cfg.h file. It uses #defines when possible, because they allow code and data sizes to be scaled at compile time based on enabled features. This allows the Read-Only Memory (ROM) and Random-Access Memory (RAM) footprints of the HTTP Server to be adjusted based on application requirements.

It is recommended that the configuration process begins with the default configuration values which in the next sections will be shown in **bold**.

- Debug Configuration
- Counter Configuration
- File System Configuration
- Persistent Connection Configuration
- Header Field Configuration
- Query String Configuration
- Form Submission Configuration
- Token Replacement Configuration
- Proxy Configuration

Debug Configuration

A fair amount of code in the HTTP Server module has been included to simplify debugging. There are several configuration constants used to aid debugging.

Table - Argument Checking and Debug Configuration

Constant	Description	Possible Values
HTTPs_CFG_DBG_INFO_EN	Enable/disable HTTP Server debug information: Internal constants assigned to global variables.	DEF_ENABLED or DEF_DISABLED

Counter Configuration

The HTTP Server contains code that increments counters to keep track of statistics such as the number of request received, the number of connection processed, etc. Also, the HTTP Server module contains counters that are incremented when error conditions are detected. The following constants enable or disable HTTP counters.

Table - Counter Configuration

Constant	Description	Possible Values
HTTPs_CFG_CTR_STAT_EN	Determines whether the code and data space used to keep track of statistics will be included.	DEF_ENABLED or DEF_DISABLED
HTTPs_CFG_CTR_ERR_EN	Determines whether the code and data space used to keep track of errors will be included.	DEF_ENABLED or DEF_DISABLED

File System Configuration

The HTTP Server can be used with a standard file system like Micrium OS File System.

Table - File System Configuration

Constant	Description	Possible Values
HTTPs_CFG_FS_PRESENT_EN	Determines whether or not a File System is present and must be used with HTTP server.	DEF_DISABLED or DEF_ENABLED

Persistent Connection Configuration

The HTTP Server supports persistent connection when HTTP protocol version 1.1 is used.

The configuration related to Persistent Connection must also be enabled in the run-time configuration of each server instance for which this feature is desired. See section Instance Configuration for more details.

See section Persistent Connection for more details on HTTP Persistent Connection.

Table - Persistent Connection Configuration

Constant	Description	Possible Values
HTTPs_CFG_PERSISTENT_CONN_EN	Determines whether the code and data used to support the persistent connection feature will be included.	DEF_DISABLED or DEF_ENABLED

Header Field Configuration

The HTTP Server contains code that allow the upper application to receive header field received in the request message and add header field to be transmitted with the response message.

The header parameters must also be defined in the run-time configuration of each server instance for which this feature is wished. See section Instance Configuration and HTTP Header Configuration for more details.

See section Headers for more details on HTTP header fields concept.

Table - Header Field Configuration

Constant	Description	Possible Values
HTTPs_CFG_HDR_RX_EN	Determines whether the code and data space used to support additional header field in request will be included.	DEF_ENABLED or DEF_DISABLED
HTTPs_CFG_HDR_TX_EN	Determines whether the code and data space used to support additional header field in response will be included.	DEF_ENABLED or DEF_DISABLED

Query String Configuration

The HTTP Server contains code to allow the parsing and saving of Query String received in HTTP requests.

To enable this feature for a specific server instance, the parameter associated with the Query String must also be enabled in the the run-time configuration of the instance. See section Instance Configuration and Query String Configuration for more details on the configuration.

See section Query String for additional details on the HTTP Query String concept.

Table - Query String Configuration

Constant	Description	Possibles Values
HTTPs_CFG_QUERY_STR_EN	Determines whether the code and data space used to support Query String in HTTP request will be included.	DEF_ENABLED or DEF_DISABLED

Form Submission Configuration

The HTTP Server can support the reception of HTML forms using the HTTP POST method. The HTTP Server module is designed to manage different type of form encoding and also to upload files from an HTML form.

To enable this feature for a specific server instance, the parameter associated with the Form feature must also be enabled in the the run-time configuration of the instance. See section Instance Configuration and HTTP Form Configuration for more details on the configuration.

See section Form for additional details on the HTML Form concept.

Table - Dynamic Content Configuration

Constant	Description	Possible Values
HTTPs_CFG_FORM_EN	Determines whether the code and data space used to support forms submission will be included. If the server does not host form, this feature should be disabled in order to speed up processing. If this feature is set as disabled all other Form's configurations are automatically disabled.	DEF_ENABLED or DEF_DISABLED
HTTPs_CFG_FORM_MULTIPART_EN	Determines whether the code and data space used to support forms submission with “multipart/form-data” encoding will be included.	DEF_ENABLED or DEF_DISABLED

Token Replacement Configuration

A considerable amount of code in the HTTP Server has been included to parse, replace and transmit HTML document with dynamic content using token replacement and the HTTP Chunked Transfer encoding.

To enable this feature for a specific server instance, the parameter associated with the Token feature must also be enabled in the the run-time configuration of the instance. See section Instance Configuration and Token Configuration for more details on the configuration.

Table - Dynamic Content Configuration

Constant	Description	Possible Values
HTTPs_CFG_TOKEN_PARSE_EN	Determines whether the code and data space used to support the parsing of the tokens found in HTML files for dynamic content will be included. If the server does not host dynamic content, the feature should be disabled in order to speed up processing.	DEF_DISABLED or DEF_ENABLED

Proxy Configuration

The HTTP Server module can be accessed even if it is located behind a proxy.

Table - Proxy Configuration

Constant	Description	Possible Values
HTTPs_CFG_ABSOLUTE_URI_EN	Determines whether the code and data space used to support header field for absolute URI.	DEF_ENABLED or DEF_DISABLED

HTTP Server Compile-Time Configuration for Static File System

HTTP Server's static file system is configurable at compile time via three #defines located in http_server_fs_port_static_cfg.h file. The HTTP Server module uses #defines when possible, because they allow code and data sizes to be scaled at compile time based on enabled features. This allows the Read-Only Memory (ROM) and Random-Access Memory (RAM) footprints of the HTTP Server to be adjusted based on application requirements.

It is recommended that the configuration process begins with the default configuration values which in the next sections will be shown in **bold**.

These configurations are necessary *only* if you use the static file system in your application.

Constant	Description	Possible Values
HTTPs_FS_CFG_NBR_FILES	This value defines the maximum number of files the static file system could manage.	1 or more. Default is **15**.
HTTPs_FS_CFG_NBR_DIRS	This value defines the maximum number of directories the static file system could manage.	1 or more. Default is **1**.
HTTPs_FS_CFG_MAX_FILE_NAME_LEN	This value defines the maximum file name length.	1 or more. Default is **256**.

HTTP Server Run-Time Application Specific Configuration

This section defines the configurations related to the HTTP Server module but that are specified at run-time, during the initialization process.

- Initialization
- Optional Configurations
- Post-Init Configurations

Initialization

Initializing the HTTP Server module is done by calling the function HTTPs_Init(). This function takes no configuration argument. Unless you override an optional configuration before calling the function HTTPs_Init(), the default configurations will be used.

Optional Configurations

This section describes the configurations that are optional. If you do not set them in your application, the default configurations will apply.

The default values can be retrieved via the structure HTTPs_InitCfgDflt.

Note that these configurations must be set *before* you call the function HTTPs_Init().

Table - HTTP Server Optional Configurations

Configurations	Description	Type	Function to call	Default	Field from default configuration structure
Memory segment	This module allocates some control data. You can specify the memory segment from where such data should be allocated.	MEM_SEG*	HTTPs_ConfigureMemSeg()	General-purpose heap .	.MemSegPtr

Post-Init Configurations

This section describes the configurations that can be set at any time during execution AFTER you called the function HTTPs_Init().

These configurations are optional. If you do not set them in your application, the default configurations will apply.

Table - HTTP Server Post-init Configurations

Configurations	Description	Type	Function to call	Default
HTTP server instance task priority	Each HTTP server instance created via the function HTTPs_InstanceInit() will create a task. You can change the priority of the created task at any time..	RTOS_TASK_PRIO	HTTPs_InstanceTaskPrioSet()	N/A

HTTP Server Instance Configuration

This section defines the configurations related to the HTTP Server module that are specified at run-time and that are specific to each added HTTP server instance.

To add a HTTP server instance, call the HTTPs_InstanceInit() function. This function requires two configurations arguments (described below).

- p_cfg
  - Task
  - Listen Socket
  - Connection
  - File System
  - Proxy
  - Hook Functions
  - Header Field
  - Query String
  - Form
  - Dynamic Token Replacement
- p_task_cfg

p_cfg

`p_cfg` is a pointer to a configuration structure of type HTTPs_CFG. The HTTP Server instance configuration is based on a large structure that contains different configuration sections with many parameter settings. This section should provide you an in-depth understanding of all instance parameter configuration. You will also discover which settings to modify in order to enhance the functionalities and the performances. Refer to the configuration field description section for further details. The HTTPs_CFG object must be passed to the HTTPs_InstanceInit() API function during the initialization of the HTTP server instance.

Task

Table - Task Configuration

Structure Field	Type	Description	Possible Values
.OS_TaskDly_ms	CPU_INT32U	Configure instance task delay in integer millisecondsThe web server can delay his task periodically to allow another task with lower priority to run.	MUST be >= 0

Listen Socket

Table - Listen Socket Configuration

Structure Field	Type	Description	Possible Values
.SockSel	HTTPs_SOCK_SEL	Configure socket type. Select which kind of IP addresses can be accepted by the web server instance.When only one IP type is selected, only one socket and TCP connection will be reserved to listen for incoming connections.When the two IP types are selected, two sockets and TCP connections will be reserved for listening.	- HTTPs_SOCK_SEL_IPv4 (Accept Only IPv4)- HTTPs_SOCK_SEL_IPv6 (Accept Only IPv6)- HTTPs_SOCK_SEL_IPv4_IPv6 (Accept IPv4 and IPv6)

Structure Field	Type	Description	Possible Values		
.SecurePtr	HTTPs_SECURE_CFG	Configure instance secure (SSL/TLS) configuration structure.'Secure' field is used to enabled or disable SSL/TLS:1. DEF_NULL, the web server instance is not secure and doesn't use SSL/TLS.2. Point to a secure configuration structure, the web server is secure and use SSL/TLS.The secure web server can be enabled *only* if the application project contains a secure module supported by the Network module such as:1. NanoSSL provided by Mocana.2. CyaSSL provided by YaSSL.	- DEF_NULL for a non-secure web server- Pointer to the secure configuration (Instance^Secure^Configuration) to be used.		
.Port	CPU_INT16U	Configure instance server port.1. Default HTTP port used by all web browser is 80. The default port number is defined by the following value:HTTPs_CFG_DFLT_PORTWhen default port is used the web server instance can be accessed using the IP address of the target from any web browser:http://*\\	<target ip address\	> *If the web server instance is configured with the non-default port, the instance server should be accessed via this kind of address:http://*	<target ip address\

Connection

Table - Connection Configuration

Structure Field	Type	Description	Possible Values

Structure Field	Type	Description	Possible Values						
.Port	CPU_INT16U	Configure instance server port.1. Default HTTP port used by all web browser is 80. The default port number is defined by the following value:HTTPs_CFG_DFLT_PORTWhen default port is used the web server instance can be accessed using the IP address of the target from any web browser:http://*\\	<target ip address\	>*If the web server instance is configured with the non-default port, the instance server should be accessed via this kind of address:http://*	<target ip address\	>:\	<port number\	>*Where:* \	<targe ip addre

Connection

Table - Connection Configuration

Structure Field	Type	Description	Possible Values
.ConnNbrMax	CPU_INT08U	Configure the maximum number of simultaneous connections.'ConnNbrMax' is used to configure the maximum number of connections that the web server will be able to serve simultaneously.The maximum number of connections must be configured following your requirements about the memory usage and the number of connection:- Each connection requires memory space. So the memory required by the web server is greatly affected by the number of connection configured.- When a client downloads an item such as *HTML* document, image, *CSS* file, *Javascript* file, it should open a new connection for each of these items when the Persistent Connection feature is disabled. Also, most common web server can open up to 15 simultaneous connections. As an example, for an *HTML* document which includes 2 images + 1 *CSS* file, 4 connections could be opened simultaneously.The number of connection and the Network core module configuration must be configured accordingly. Each connection requires 1 TCP socket and the server requires 1 or 2 TCP socket depending if IPv4 and IPv6 connection can be accepted, see TCP Layer Configuration and Socket Layer Configuration to properly configure the web instance and Micrium OS Network module.	MUST be >= 1
.ConnInactivityTimeout_s	CPU_INT16U	Configure connection maximum inactivity timeout in integer seconds.For each connection when the inactivity timeout occurs the connection is automatically closed whatever what the state of the connection was.	SHOULD be >= 1
.BufLen	CPU_INT16U	Configure connection buffer length.Each connection has a buffer to receive & transmit data and to read file. If the memory is limited the buffer size can be reduced, but the performance could be impacted.	

Connection

Table - Connection Configuration

Structure Field	Type	Description	Possible Values
.ConnNbrMax	CPU_INT08U	Configure the maximum number of simultaneous connections.'ConnNbrMax' is used to configure the maximum number of connections that the web server will be able to serve simultaneously.The maximum number of connections must be configured following your requirements about the memory usage and the number of connection:- Each connection requires memory space. So the memory required by the web server is greatly affected by the number of connection configured.- When a client downloads an item such as *HTML* document, image, *CSS* file, *Javascript* file, it should open a new connection for each of these items when the Persistent Connection feature is disabled. Also, most common web server can open up to 15 simultaneous connections. As an example, for an *HTML* document which includes 2 images + 1 *CSS* file, 4 connections could be opened simultaneously.The number of connection and the Network core module configuration must be configured accordingly. Each connection requires 1 TCP socket and the server requires 1 or 2 TCP socket depending if IPv4 and IPv6 connection can be accepted, see TCP Layer Configuration and Socket Layer Configuration to properly configure the web instance and Micrium OS Network module.	MUST be >= 1
.ConnInactivityTimeout_s	CPU_INT16U	Configure connection maximum inactivity timeout in integer seconds.For each connection when the inactivity timeout occurs the connection is automatically closed whatever what the state of the connection was.	SHOULD be >= 1
.BufLen	CPU_INT16U	Configure connection buffer length.Each connection has a buffer to receive & transmit data and to read file. If the memory is limited the buffer size can be reduced, but the performance could be impacted.	MUST be >= 512
.ConnPersistentEn	CPU_BOOLEAN	Configure Persistent Connection feature.See section Persistent Connection for more details on the Persistent Connection concept.	- DEF_DISABLED- DEF_ENABLED

File System

See section File System Configuration for additional details on the File System configuration and configuration structures.

Table - File System Configuration

Structure Field	Type	Description	Possible Valu
.FS_Type	HTTPs_FS_TYPE	Configure the Type of File System to use with the HTTP server:HTTPs_FS_TYPE_NONE : No File System is present.HTTPs_FS_TYPE_STATIC : Use the HTTP Static File System offered inside the HTTP-server.HTTPs_FS_TYPE_DYN : Use a standard dynamic File System (e.g., Micrium OS File System)	HTTPs_FS_TY
.FS_CfgPtr	const void*	Configure Pointer to the File System Configuration object.Each File System Type has its configuration structure:HTTPs_CFG_FS_NONEHTTPs_CFG_FS_STATICHTTPs_CFG_FS_DYN	MUST be a va
.DfltResourceNamePtr	CPU_CHAR*	Configure instance default HTML document.The default HTML document is returned when no file is specified in the request of the client, i.e., accessing with only the web server address. Most of the time this file should be "index.html".	MUST be a st

Proxy

Table - Proxy Configuration

Structure Field	Type	Description	Possible Values
.HostNameLenMax	CPU_INT16U	Configure maximum host name length.When an HTTP Server is behind an HTTP Proxy, the HTTP client must send its requests with an absolute URI. For example :GET http://example.com/index.html HTTP/1.1When the absolute URI feature is enabled, the HTTP server will support absolute URI in the first line of the HTTP request messages (see the example just above). The server will also look for the 'Host' header field in the received request messages and save it in the HostPtr field of the HTTPs_CONN structure.The maximum host name length is the maximum length the server will allow for the received host name in a request message.Proxy support must be enabled to allow the web server to support absolute URI. See Proxy Configuration for further information.	SHOULD be >= 1.

Hook Functions

See Section Hook Configuration for additional details on the hook functions configuration structure.

Table - Hook Functions Configuration

Structure Field	Type	Description	Possible Values
.HookPtr	const HTTPs_HOOK_CFG Pointer	Pointer to an HTTPs_HOOK_CFG type object. This object contains all the hook function pointers available to personalize the processing of HTTP requests received.	Pointer to Hook object or DEF_NULL.
.Hook_CfgPtr	void*	Data associated with the Hook set-up that will be available inside each hook function.	Pointer to Hook Configuration object or DEF_NULL.

Header Field

See Section HTTP Header Configuration for additional details on the Header Configuration structures.

Table - Header Field Configuration

Structure Field	Type	Description	Possible Values
.HdrRxCfgPtr	const HTTPs_HDR_RX_CFG Pointer	Configure pointer to HTTP Request (RX) Header Configuration object structure.	Pointer to Configuration object structure.DEF_NULL, if feature not necessary.
.HdrTxCfgPtr	const HTTPs_HDR_TX_CFG Pointer	Configure pointer to HTTP Response (TX) Header Configuration object structure.	Pointer to Configuration object structure.DEF_NULL, if feature not necessary.

Query String

See section Query String Configuration for additional details on Query String Configuration structure.

Table - Query String Configuration

Structure Field	Type	Description	Possible Values
.QueryStrCfgPtr	const HTTPs_QUERY_STR_CFG Pointer	Configure pointer to HTTP Query String Configuration object structure.	Pointer to Configuration object structure.DEF_NULL, if feature not necessary.

Form

See section HTTP Form Configuration for additional details on Form Configuration structure.

Table - Form Configuration

Structure Field	Type	Description	Possible Values
.FormCfgPtr	const HTTPs_FORM_CFG Pointer	Configure pointer to HTML Form Configuration object structure.	Pointer to Configuration object structure.DEF_NULL, if feature not necessary.

Dynamic Token Replacement

See section Token Configuration for additional details on Token Configuration structure.

Table - Token Configuration

Structure Field	Type	Description	Possible Values
.Token_CfgPtr	const HTTPs_TOKEN_CFG Pointer	Configure pointer to Token Replacement Configuration object structure.	Pointer to Configuration object structure.DEF_NULL, if feature not necessary.

p_task_cfg

Each HTTP server instance has a kernel task associated. p_task_cfg is a pointer to a configuration structure of type RTOS_TASK_CFG that allows you to configure the priority, stack base pointer, and stack size for that task.

Guidelines on How to Properly Set the Priority and Stack Size

The priority of an HTTP Server instance's task greatly depends on the requirements of your application. For some applications, it might be better to set it at a high priority, especially if your application requires a lot of tasks and is CPU intensive.

Task Stack Sizes

The arbitrary stack size of **1024** is a good starting point for most applications.

The only guaranteed method of determining the required task stack sizes is to calculate the maximum stack usage for each task. Obviously, the maximum stack usage for a task is the total stack usage along the task's most-stack-greedy function path. Note that the most-stack-greedy function path is not necessarily the longest or deepest function path.

The easiest and best method for calculating the maximum stack usage for any task/function should be performed statically by the compiler or by a static analysis tool since these can calculate function/task maximum stack usage based on the compiler's actual code generation and optimization settings. So for optimal task stack configuration, we recommend that you invest in a task stack calculator tool compatible with your build toolchain.

HTTP Server Instance Secure Configuration

- Configuration Field Description
- Structure Example

The HTTPs_SECURE_CFG structure referenced in instance configuration must exist throughout the lifetime of the HTTPs instance since the certificate and the key are not copied internally and are directly referenced throughout the HTTPs_SECURE_CFG pointer.

SSL/TLS certificate and key can be acquired either:

1. From a certificate authority. Acquiring the certificate from an authority should ensure to avoid the untrusted warning message to be displayed when accessing the web server.
2. Generated from a SSL tool such as OpenSSL. This kind of tool generates self-signed certificate and the untrusted warning message will be displayed every time the web server is accessed.

Configuration Field Description

Table - Secure Configuration

Structure Field	Type	Description	Possible Values
.CertPtr	CPU_CHAR *	Pointer to the public certificate's character string.	String
.CertLen	CPU_INT32U	Length of the public certificate.	MUST BE > 0
.KeyPtr	CPU_CHAR *	Pointer to the private key's character string.	String
.KeyLen	CPU_INT32U	Length of the private key.	MUST BE > 0
.Fmt	NET_SOCK_SECURE_CERT_KEY_FMT	Format of the key and certificate. Supported formats are PEM and DER.If the PEM format is used, do not include the "-----BEGIN CERTIFICATE-----", "-----END CERTIFICATE-----", "-----BEGIN RSA PRIVATE KEY-----" or "-----END RSA PRIVATE KEY-----" sections.	- NET_SOCK_SECURE_CERT_KEY_FMT_PEM- NET_SOCK_SECURE_CERT_KEY_FMT_DER
.CertChain	CPU_BOOLEAN	Flag to set if the certificate is chained to another one.	- DEF_NO- DEF_YES

Structure Example

Listing - Secure Configuration Structure Example in the *HTTP Server Instance Secure Configuration* page demonstrates how to create an HTTP Server secure configuration structure.

Listing - Secure Configuration Structure Example

```
#define HTTPs_CFG_SECURE_CERT                                    \
"MIIEEjCCAvqgAwIBAgIBBzANBgkqhkiG9w0BAQUFADAaMRgwFgYDVQQDEw9WYWxp\
Y29yZS1EQzEtQ0EwHhcNMTEwMzE4MTcwMTQyWhcNMjEwMzE1MTcwMTQyWjCBkDEL\
MAkGA1UEBhMCVVMxCzAJBgNVBAgTAkNBMQ8wDQYDVQQHEwZJcnZpbmUxHjAcBgNV\
BAoTFVZhbGljb3JlIFRlY2hub2xvZ2llczEhMB8GA1UEAxMYbGFuLWZ3LTAxLnZh\
bGljb3JlLmxvY2FsMSAwHgYJKoZIhvcNAQkBFhFhZG1pbkBsb2NhbGRvbWFpbjCC\
ASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBALwGOahytiwshzz1s/ngxy1+\
+VrXZYjKSEzMYbJCUhK9xA5fz8pGtOZIXI+CasZPSbXv+ZDLGpSpeFnOL49pIYRs\
vmTxg2n3AIZbP6pD9OPU8rmufsTvXAmQGxxIkdmWiXYJk0pbj+U698me6DKMV/sy\
3ekQaQC2I2nr8uQw8RhuNhhlkWyjBWdXnS2mLNLSan2Jnt8rumtAi3B+vF5Vf0Fa\
kLJNt45R0f5jjuab+qw4PKMZEQbqe0XTNzkxdD0XNRBdKlajffoZPBJ7xkfuKUA3\
cMjXKzetABoKvsv+ElfvqIrl9RXvTXy52EaQmVhiOyBHrScq4RbwtDQsd59Qmk0C\
AwEAAaOB6zCB6DAJBgNVHRMEAjAAMBEGCWCGSAGG+EIBAQQEAwIGQDA0BglghkgB\
hvhCAQ0EJxYlRWFzeS1SU0EgR2VuZXJhdGVkIFNlcnZlciBDZXJ0aWZpY2F0ZTAd\
BgNVHQ4EFgQUrq5KF11M9rpKm75nAs+MaiK0niYwUQYDVR0jBEowSIAU2Q9eGjzS\
LZhvIRRKO6c4Q5ATtuChHqQcMBoxGDAWBgNVBAMTD1ZhbGljb3JlLURDMS1DQYIQ\
T9aBcT0uXoxJmC0ohp7oSTATBgNVHSUEDDAKBggrBgEFBQcDATALBgNVHQ8EBAMC\
BaAwDQYJKoZIhvcNAQEFBQADggEBAAUMm/9G+mhxVlYK4anc34FMqu88NQy8Irh0\
loNfHhlEKnerzMz+nQGidf+KBg5K5U2Jo8e9gVnrzz1gh2RtUFvDjgosGIrgYZMN\
yreNUD2I7sWtuWFQyEuewbs8h2MECs2xVktkqp5KPmJGCYGhXbi+zuqi/19cIsly\
yS01kmexwcFMXyX4YOVbG+JFHy1b4zFvWgSDULj14AuKfc8RiZNvMRMWR/JqIpr5\
xWQRSmkjuzQMFavs7soZ+kHp9vnFtY2D6gF2caiIk0sdG0uuyPBVxEJ2meifG6eb\
o3FQzdtIrB6oMFHEU00P38SJq+mrDItPDRXNLa2Nrtc1EJtmjws="

#define HTTPs_CFG_SECURE_KEY                                     \
"MIIEogIBAAKCAQEAvAY5qHK2LCyHPPWz+eDHLX75WtdIiMpITMxhskJSEr3EDI/P\
yka05khcj4Jqxk9Jte/5kMsalKI4Wc4vj2mVhGy+ZPGDafcCVls/qkP049Tyua5+\
xO9cCZAbHEiR2ZaJdgmTSluP5Tr3yZ7oMoxX+zLd6RBpALYjaevy5DDxGG42GGWR\
bKMFZ1edLaYs0tJqfYme3yu6a0CLcH68XIV/QVqQsk23jlHR/mOO5pv6rDg8oxkR\
Bup7RdM3OTF0PRc1EF0qVqN9+hk8EnvGR+4pQDdwyNcrN60AGgq+y/4SV++qWsj1\
Fe9NfLnYRpCZWGI7lEetJyrhFvC0NCx3n1CaTQIDAQABAoIBAEbbqbr7j//RwB2P\
EwZmWWmh4mMDrbYBVYHrvB2rtLZvYYVxQiOexenK92b15TtbAhJYn5qbkCbaPwrJ\
E09eoQRl3u+3vKigd/cHaFTIS2/Y/qhPRGL/OZY5Ap6EEsMHYkJjlWh+XRosQNlw\
01zJWxbFsq90ib3E5k+ypdStRQ7JQ9ntvDAP6MDp3DF2RYf22Tpr9t3Oi2mUirOI\
piOEB55wydSylhSHusbms3sp2uvQBYJjZP7eENEQz55PebTzI9UF2dgJ0wJFS073\
rvp46fibcch1L7U6v8iUNaS47GTs3MMyO4zda73ufhYwZLU5gL8oEDY3tf/J8zuC\
mNurr0ECgYEA8i1GgstYBFSCH4bhd2mLu39UVslvHaD38mpJE6avCNOUq3Cyz9qr\
NzewG7RyqR43HsrVqUSQKzIAGWqG7sf+jkiam3v6VW0y05yqDjs+SVW+ZN5CKyn3\
sMZV0ei4MLrfxWneQaKy/EUTJMlz3rLSDM/hpJoA/gOo9BIFRf2HPkkCgYEAxsGq\
LYU+ZEKXKehVesh8rlic4QXwzeDmpMF2wTq6GnFq2D4vWPyVGDWdORcIO2BojDWV\
EZ8e7F2SghbmeTjXGADldYXQiQyt4Wtm+oJ6d+/juKSrQ1HIPzn1qgXDNLPfjd9o\
9IX5IGIRn49Jrx/kKQAPTcnCa1lirlcsmcdiy+UCgYBEbOBwUi3zQ0Fk0QJhb/Po\
LSjSPpl7YKDN4JP3NnBcKRPngLc1HU6lElny6gA/ombmj17hLZsia1GeHMg1LVLS\
NtdgOR5ZBrqGqcwuqzSFGfHqpBXEBl6SludmoL9yHUreh3QhzWuO9aFcEoNnl9Tb\
g9z4Wf8Pxk71byYISYLt6QKBgERActjo3ZD+UPyCHQBp4m45B246ZQO9zFYdXVNj\
gE7eTatuR0lOkoBawN++6gPByoUDTWpcsvjF9S6ZAJH2E97ZR/KAfijh4r/66sTx\
k26mQRPB8FHQvqv/kj3NdsgdUJJeeqPEyEzPkcjyloJxuB7gN2El/I5wCRon3Qf9\
sQ6FAoGAfVOaROSAtq/bq9JIL60kkhA9sr3KmX52PnOR2hW0caWi96j+2jlmPT93\
4A2LlVUo6hCsHLSCFoWWiyX9pIqyYTn5L1EmeBO0+E8BH9F/te9+ZZ53U+quwc/X\
AZ6Pseyhj7S9wkI5hZ9SO1gcK4rWrAK/UFOIzzIACr5INr723vw="

#define  HTTPs_CFG_SECURE_CERT_LEN      (sizeof(HTTPs_CFG_SECURE_CERT) - 1)
#define  HTTPs_CFG_SECURE_KEY_LEN       (sizeof(HTTPs_CFG_SECURE_KEY)  - 1)

HTTPs_SECURE_CFG  HTTPs_Cfg_InstanceSecure = {
                                    HTTPs_CFG_SECURE_CERT,
                                    HTTPs_CFG_SECURE_CERT_LEN,
                                    HTTPs_CFG_SECURE_KEY,
                                    HTTPs_CFG_SECURE_KEY_LEN,
                                    NET_SOCK_SECURE_CERT_KEY_FMT_PEM,
                                    DEF_NO,
                                  };
```

HTTP Server Instance File System Configuration

The HTTP Server module can run with or without Micrium OS File System module.

If you don't use Micrium's File System and still need to store files on your HTTP server, the HTTP Server module comes with a "Static File System". This pseudo File System allows to store your files in constant C arrays, read them and navigate inside them. A script allows you to convert your files to constant C arrays that should then be added to your project. Afterwards, the function `HTTPs_FS_AddFile()` allows you to store those files in the "Static File System".

Configuration

Three types of File System can be configured with the HTTP Server. Each type has its own File System configuration structure.

In the HTTP server Instance configuration, the FS Type and the Pointer to the associated FS configuration must be passed. See section Instance Configuration .

HTTPs_FS_TYPE	Configuration Structure	Description
HTTPs_FS_TYPE_NONE	None	Use when no File System is present.
HTTPs_FS_TYPE_STATIC	HTTPs_CFG_FS_STATIC	Use when the HTTP Server Static File System is used.
HTTPS_FS_TYPE_DYN	HTTPs_CFG_FS_DYN	Use when Micrium OS File System module is used.

No File System Configuration

No configuration is required when no File System is present.

HTTP Server Static File System Configuration

The configurations structure is of type HTTPs_CFG_FS_STATIC. Table - HTTPs_CFG_FS_STATIC Configuration Structure Description in the *HTTP Server Instance File System Configuration* page describes this configuration structure.

Table - HTTPs_CFG_FS_STATIC Configuration Structure Description

Field	Type	Description	Possible Values
.FS_API_Ptr	const NET_FS_API*	Configure the pointer to the File System Network API object structure.	HTTPs_FS_API_Static located in http_server_fs_port_static.h.

Dynamic File System Configuration

The configurations structure is of type HTTPs_CFG_FS_DYN. Table - HTTPs_CFG_FS_DYN Configuration Structure Description in the *HTTP Server Instance File System Configuration* page describes this configuration structure.

Table - HTTPs_CFG_FS_DYN Configuration Structure Description

Field	Type	Description	Possible Values
.FS_API_Ptr	const NET_FS_API*	Configure the pointer to the File System Network API object structure.	NetFS_API_Native located in net_fs.h.
.WorkingFolderPtr	CPU_CHAR*	Pointer to the Working Folder name.Web server instances can use a working folder where files and subfolders are located. It can be set as a null pointer (DEF_NULL) if the file system doesn't support 'set working folder' functionality, but HTML documents and files must be located in the default path used by the file system.	SHOULD be a string pointer.

HTTP Server Instance Hook Configuration

- .OnInstanceInitHook
- .OnReqHdrRxHook
- .OnReqHook

- .OnReqBodyRxHook
- .OnReqRdySignalHook
- .OnReqRdyPollHook
- .OnRespHdrTxHook
- .OnRespTokenHook
- .OnRespChunkHook
- .OnTransCompleteHook
- .OnErrHook
- .OnErrFileGetHook
- .OnConnCloseHook

The HTTP Server module offers multiple hook functions during the processing of an HTTP transaction.Those hooks allow the HTTP server to be customized according to your application requirements. The hook configuration is part of the HTTP server Instance configuration and must be included in the HTTPs_CFG object (see section Instance Configuration).

See section the Hook Functions to see at what point of the process each hook is called by the server stack.

The configuration structure is of type HTTPs_HOOK_CFG. It contains only function pointers. The sections below describe this configuration structure.

.OnInstanceInitHook

When the instance is created, a hook function can be called to initialize connection objects used by the instance (see Hook Functions page to see when this function is called).

If the hook is not required by the upper application, it can be set as DEF_NULL and no function will be called.

See Connection Objects Initialization for further details.

```
CPU_BOOLEAN App_OnInstanceInit (const  HTTPs_INSTANCE  *p_instance,
                                const  void            *p_hook_cfg);
```

.OnReqHdrRxHook

Configure request header field receive callback function.

Each time a header field other than the default one is received, a hook function is called allowing to choose which header field(s) to keep for further processing.

If the hook is not required by the upper application, it can be set as DEF_NULL and no function will be called.

See Receive Request Header Field for further details.

```
CPU_BOOLEAN  App_OnReqHdrRx (const  HTTPs_INSTANCE  *p_instance,
                             const  HTTPs_CONN      *p_conn,
                             const  void            *p_hook_cfg,
                                    HTTP_HDR_FIELD   hdr_field);
```

.OnReqHook

Configure request hook function.

For each new incoming request, a hook function can be called by the HTTP server to authenticate the request and accept or reject it. At this point, the Start Line and the HTTP headers of the request have been parsed. The data received can be access in the HTTPs_CONN object. This function has also the right to modify some of the parameters of the HTTP transaction inside the HTTPs_CONN object.

If the hook is not required by the upper application, it can be set as DEF_NULL and no function will be called.

See Connection Request for further details.

```
CPU_BOOLEAN  App_OnReq (const  HTTPs_INSTANCE  *p_instance,
                        HTTPs_CONN      *p_conn,
                  const  void          *p_hook_cfg);
```

.OnReqBodyRxHook

Configure request body data received hook function.

For some applications, it is required to parse itself the data as soon as received and not let the HTTP server core do it. It that case, this hook function allows the application to retrieve the received data directly.

If the hook is not required by the upper application, it can be set as DEF_NULL and no function will be called.

See On Request Body Received Hook for further details.

```
CPU_BOOLEAN  App_OnReqBodyRx (const  HTTPs_INSTANCE       *p_instance,
                              HTTPs_CONN          *p_conn,
                        const  void               *p_hook_cfg,
                              void             *p_buf,
                        const  CPU_SIZE_T          buf_size,
                              CPU_SIZE_T         *p_buf_size_used);
```

.OnReqRdySignalHook

Configure request ready signal hook function.

The Signal hook function occurs after the request has been parsed completely (header + body). In the case of a POST request containing a form, all the data have been saved in the CGI key-value pairs block. The list of CGI data is available in the HTTPs_CONN object. The callback function *should* not be blocking and *should* return quickly. A time-consuming function will block the processing of the other connections and reduce the HTTP server performance.

In case the CGI data processing is time-consuming, the Poll hook function *should* be enabled to allow the server to periodically verify if the upper application has finished the CGI data processing.

If the CGI form feature is not enabled, this field is not used and can be set as `DEF_NULL`.

See Request Ready Signal Hook for further details.

```
CPU_BOOLEAN  App_OnReqRdySignal (const  HTTPs_INSTANCE  *p_instance,
                                 HTTPs_CONN      *p_conn,
                           const  void          *p_hook_cfg,
                           const  HTTPs_KEY_VAL   *p_data);
```

.OnReqRdyPollHook

Configure request ready poll hook function.

The Poll hook function *should* be enabled in case the request data processing require lots of time. It allows the HTTP server to periodically poll the upper application and verify if the request data processing has finished.

If the Poll feature is not required, this field *should* be set as DEF_NULL.

See Request Ready Poll Hook for further details.

```
CPU_BOOLEAN  App_OnReqRdyPoll (const  HTTPs_INSTANCE  *p_instance,
                               HTTPs_CONN      *p_conn,
                         const  void          *p_hook_cfg);
```

.OnRespHdrTxHook

Configure response header field transmit hook function.

Before an HTTP response message is sent, a hook function is called allowing to add header field(s) to the message before it is sent.

If the hook is not required by the upper application, it can be set as DEF_NULL and no function will be called.

See Add Response Header Field for further details.

```
CPU_BOOLEAN  App_OnRespHdrTx (const  HTTPs_INSTANCE  *p_instance,
                                     HTTPs_CONN      *p_conn,
                              const  void            *p_hook_cfg);
```

.OnRespTokenHook

Configure response dynamic token replacement hook function.

The hook function is called by the HTTP server when a token is found. Basically, the callback function must fill a buffer with the value to be sent instead of the token.

If the header feature is not enabled, this field is not used and can be set as `DEF_NULL`.

See Get Token Value for further information.

```
CPU_BOOLEAN  App_OnRespToken (const  HTTPs_INSTANCE  *p_instance,
                                     HTTPs_CONN      *p_conn,
                              const  void            *p_hook_cfg,
                              const  CPU_CHAR        *p_token,
                                     CPU_INT16U       token_len,
                                     CPU_CHAR        *p_val,
                                     CPU_INT16U       val_len_max);
```

.OnRespChunkHook

Configure response chunked data hook function.

The hook function must be set up when the application needs to send data in chunked transfer encoding. In that case, the HTTP server core will call the hook when the response body is being built.

If the hook is not required by the upper application layer, it can be set as `DEF_NULL` and no function will be called.

See On Response Chunk Hook for further details.

```
CPU_BOOLEAN  App_OnRespChunk (const  HTTPs_INSTANCE  *p_instance,
                                     HTTPs_CONN      *p_conn,
                              const  void            *p_hook_cfg,
                                     void            *p_buf,
                                     CPU_SIZE_T       buf_len_max,
                                     CPU_SIZE_T      *p_tx_len);
```

.OnTransCompleteHook

Configure HTTP Transaction complete hook function.

This hook function is called after an HTTP transaction has been completed. When the persistent connection mode is enabled, it is possible that the application needs to free some memory objects related to a transaction and this hook function is there for that.

If the hook is not required by the upper application layer, it can be set as DEF_NULL and no function will be called.

See On Transaction Complete Hook for further details.

```
void  App_OnTransCmpl (const  HTTPs_INSTANCE  *p_instance,
                            HTTPs_CONN      *p_conn,
                      const  void          *p_hook_cfg);
```

.OnErrHook

Configure error hook function.

When an internal error occurred during the processing of a transaction a hook function can be called to change the behavior such as the status code and the page returned.

If the hook is not required by the upper application layer, it can be set as DEF_NULL and no function will be called.

See Connection Error for further details.

```
void  App_OnErr (const  HTTPs_INSTANCE  *p_instance,
                      HTTPs_CONN      *p_conn,
                const  void          *p_hook_cfg,
                      HTTPs_ERR       err);
```

.OnErrFileGetHook

Configure Get error file hook function.

Get error file hook can be called every time an error has occurred when processing a transaction, i.e., status code is not equal to 'OK'. This function can set the web page that should be transmitted instead of the default error page defined in http-s_cfg.h.

If set to DEF_NULL the default error page will be used for every error.

See Get Error Document for further details.

```
void  App_OnErrFileGet (const  void                *p_hook_cfg,
                             HTTP_STATUS_CODE       status_code,
                             CPU_CHAR              *p_file_str,
                             CPU_INT32U             file_len_max,
                             HTTPs_BODY_DATA_TYPE  *p_file_type,
                             HTTP_CONTENT_TYPE     *p_content_type,
                             void                 **p_data,
                             CPU_INT32U            *p_date_len);
```

.OnConnCloseHook

Configure Connection close hook function.

Once a connection is closed a hook function can be called to let the upper application layer know that a connection is not yet active. This hook function could be used to free some previously allocated memory.

If the hook is not required by the upper application layer, it can be set as DEF_NULL and no function will be called.

See Connection Close for further details.

```
void  App_OnConnClose (const  HTTPs_INSTANCE  *p_instance,
                            HTTPs_CONN      *p_conn,
                      const  void          *p_hook_cfg);
```

HTTP Server Instance Header Configuration

HTTP headers can be added to HTTP request and response messages to specify additional parameters for the HTTP transaction. See section Headers for more details on HTTP headers.

The HTTP Server module supports both reception and transmission of additional headers. However, note that some header fields are already taken care of by the HTTP Server stack and must therefore not be added by the application :

- Host
- Content-Type
- Content-Length
- Transfer-Encoding
- Connection
- Location

Two distinct structures are available to configure the request and response part. Therefore, header fields received have their own configuration as the headers fields to transmit an HTTP response. This allows more flexibility in customizing your own HTTP server application. For example, your server can be more interested in receiving many headers from clients and not adding any headers of its own in the HTTP response.

The Header configurations are part of the HTTP server Instance configuration and must be included in the HTTPs_CFG object (see section Instance Configuration).

Fields Description

Structure Field	Type	Description	Possible Values
.NbrPerConnMax	CPU_INT16U	Maximum number of headers for each HTTP request or response.	SHOULD be >= 1LIB_MEM_BLK_QTY_UNLIMITED to set an unlimited pool.
.DataLenMax	CPU_INT16U	Maximum length of the header field value accepted by the HTTP server.	SHOULD be >= 1

Since the HTTP server only processed an HTTP Transaction at a time on a given Connection, the .NbrPerConnMax field set the number of headers available for an HTTP transaction and connection.

HTTP Server Instance Query String Configuration

To enable the Query String feature, the compile-time configuration HTTPs_CFG_QUERY_STR_EN must be set accordingly (See section Module Configuration).

Also a configuration object (HTTPs_QUERY_STR_CFG) MUST be added to the HTTP Instance configuration (See section Instance Configuration).

For more information on HTTP Query String, see section Query String .

Fields Description

Structure Field	Type	Description	Possible Values
.NbrPerConnMax	CPU_INT16U	Maximum number of Query String fields accepted for each HTTP request.	SHOULD be >= 1LIB_MEM_BLK_QTY_UNLIMITED to set an unlimited pool.
.KeyLenMax	CPU_INT16U	Maximum length of the key part accepted by the HTTP server.	SHOULD be >= 1
.ValLenMax	CPU_INT16U	Maximum length of the value part accepted by the HTTP server.	SHOULD be >= 1

HTTP Server Instance Form Configuration

To enable the Form feature, the compile-time configuration HTTPs_CFG_FORM_EN must be set accordingly (See section Module Configuration).

Also a configuration object (HTTPs_FORM_CFG) MUST be added to the HTTP Instance configuration (See section Instance Configuration).

For more information on HTTP Form Submission, see section HTTP Form .

Fields Description

Structure Field	Type	Description	Possible Values
.NbrPerConnMax	CPU_INT16U	Maximum number of Form fields accepted for each HTTP request.Number must be greater than or equal to the maximum number of input fields which can be transmitted inside a received HTML form.	SHOULD be >= 1LIB_MEM_BLK_QTY_UNLIMITED to set an unlimited pool.
.KeyLenMax	CPU_INT16U	Maximum length of the key part accepted by the HTTP server.	SHOULD be >= 1
.ValLenMax	CPU_INT16U	Maximum length of the value part accepted by the HTTP server.	SHOULD be >= 1
.MultipartEn	CPU_BOOLEAN	Enable/Disable Multipart Form support.	DEF_ENABLED or DEF_DISABLED
.MultipartFileUploadEn	CPU_BOOLEAN	Enable/Disable File Upload feature inside Multipart Form.If File Upload feature is enabled, the web server will store the file received. If the feature is not enabled and a file is received the file will simply be dropped.File upload is not yet possible with the Static File System.	DEF_ENABLED or DEF_DISABLED
.MultipartFileUploadOverWrEn	CPU_BOOLEAN	Enable/Disable File overwrite when file is uploaded to server.File overwrite must be enabled to allow a file to be received if the file already exists in the folder.	DEF_ENABLED or DEF_DISABLED
.MultipartFileUploadFolderPtr	CPU_CHAR *	Configures the folder where the uploaded files will be stored.A folder name needs to be specified to indicate where the uploaded files will be save.If it is wished to save uploaded files directly in the root web directory, the name folder needs to be set as "\".I f uploaded files need to be saved inside a subfolder of the root web directory, the folder MUST already exist when the HTTP server tries to access it.	SHOULD be a string pointer

HTTP Server Instance Token Configuration

To enable the Dynamic Token Replacement feature, the compile-time configuration HTTPs_CFG_TOEKN_EN must be set accordingly (See section Module Configuration).

Also a configuration object (HTTPs_TOKEN_CFG) MUST be added to the HTTP Instance configuration (See section Instance Configuration).

Fields Description

Structure Field	Type	Description	Possible Values
.NbrPerConnMax	CPU_INT16U	Maximum number of tokens accepted for each HTTP transaction.	SHOULD be >= 1LIB_MEM_BLK_QTY_UNLIMITED to set an unlimited pool.
.ValLenMax	CPU_INT16U	Maximum length of the token replacement value accepted by the HTTP server.	SHOULD be >= 1

HTTP Server Programming Guide

This section regroups topics to help developing a custom HTTP server application with the Micrium OS HTTP Server stack. Examples are include in many sub-sections.

- HTTP Server Interface with File System
- HTTP Server Control Structures
- HTTP Server Query String
- HTTP Server Hook Functions
- HTTP Server Header Fields
- HTTP Server Request Body Data
- HTTP Server Form
- HTTP Server Response Body Data
- HTTP Server Token Replacement
- HTTP Server Proxy
- HTTP Server Add-on

HTTP Server Interface with File System

Although the HTTP Server module can be used without a File System, when file operations are required by the HTTP server, Micrium OS File System module is needed. This will allow you to dynamically load the web pages and enable your web server to receive files. However, the HTTP Server comes with a minimalist in-house Static File system which only stores static files (stored in the code space memory). Obviously, it is only possible to read them and thus not possible to upload files to a web server when the static file system is used.

- Configuration
  - Compile-Time Configuration
  - Run-Time Configuration
- Usage
  - Transferring Files to Use with the HTTP Server
  - Using the HTTP Server with the Static File System
    - Static File System Module Configuration

Configuration

Compile-Time Configuration

The configuration macro HTTPs_CFG_FS_PRESENT_EN must be set to DEF_ENABLED in the `http_server_cfg.h` file when a File System (the in-house static or a standard FS) must be used with the HTTP server.

See section File System Compile-time Configuration for more details.

Run-Time Configuration

Three types of configuration structure exist for the File System Configuration: one when No FS is used, one for the Static FS and one when Micrium OS File System is used. Depending on your setup, an object of one of those structures type must be defined in the application and passed as a pointer inside the instance configuration object structure (HTTPs_CFG).

See section File System Run-time Configuration for additional details.

Usage

Transferring Files to Use with the HTTP Server

Transferring files on the file system for use with the HTTP Server module is outside the scope of this document. For this purpose, you could use an FTP client, such as Micrium OS FTP Client , or alternatively, use a mass storage media to access your file system.

Note that the HTTP Server can support file upload when Micrium OS File System is used. Thus it's possible to create scripts to transmit files using the POST method even if no file is yet loaded. Also, many web browsers now have plugins for testing purposes. They allow you to send custom HTTP requests to a server. You can therefore use some of those tools to upload files to the HTTP server. Micrium has tested the Postman plugin with Chrome.

Using the HTTP Server with the Static File System

The HTTP Server provides a small file system that could allow it to be used without Micrium OS File System.

The static file system relies on C arrays to represent the various files, and those files are accessible only for reading. The HTTP Server ships with a very basic web page consisting of a .gif and two .html files.

A python script named **GenerateFS.py** is offered in the HTTP Server code allowing to convert files into C arrays. The script can be useful if many files are to be converted since it supports recursion inside a folder. The script also generates C files (generated_fs.c/.h) with the function to call to include all the C arrays generated.

This script has been created to ease your life as a developer. It is not intended to be a production tool.

The script should be called in a command line shell like shown below:

```
python GenerateFS.py [source directory or file name] [destination directory] -f
```

The first argument is the path directory where to files to be converted are located or the file name of the single file to convert. The second argument is the directory path where to save the generated file(s). An optional *-f* argument can be used to force the re-writing if the destination file already exists.

Static File System Module Configuration

A configuration file will be created if you select a static file system in your project. The configuration file name is http_server_fs_port_static_cfg.h. Refer to Configuration for Static File System for more information.

HTTP Server Control Structures

Many hook functions receive pointers to control structures which can be used by upper layer to change the behavior of the server and for debugging purpose.

HTTPs_INSTANCE

The instance control structure should be only used for debugging purpose. The upper application **MUST** not modify this structure, it must only read data except for the .DataPtr field. Here is a list of fields of interest:

Field	Description
.Started	The HTTP Server instance is running or is stopped.
.CfgPtr	Pointer to the instance's configuration.
.SockListenID_IPv4	Socket ID used to listen on server port for IPv4 socket family.
.SockListenID_IPv6	Socket ID used to listen on server port for IPv6 socket family.
.ConActiveCtr	Current number of connection active.
.StatsCtr	Pointer to instance statistics counter structure.
.ErrsCtr	Pointer to instance errors counter structure.
.DataPtr	This field is available for the upper application to store memory location relative to the instance that will be accessible across all hook functions.

HTTPs_CONN

The connection control structure may be used to change the behavior on a particular HTTP transaction occurring on a connection and for connection processing and debugging purpose.

Fields to Read

Those are the relevant connection fields that can be read inside the hook functions by the application. They can, among others things, be used to demultiplex a connection:

`.ClientAddr`

Contains the IP address and port of the client.

`.SockID`

Contains the socket ID used to receive and transmit data.

`.Method`

This field contains the method of the request received:

- HTTP_METHOD_GET
- HTTP_METHOD_POST
- HTTP_METHOD_HEAD
- HTTP_METHOD_PUT
- HTTP_METHOD_DELETE

`.ReqContentType`

If the HTTP request has a body, this field will contains the content type of the body.

`.QueryStrListPtr`

This field is the head of the list of key-value pairs received inside a Query String if one was present after the request URL.

`.HdrListPtr`

This field is the head of the list of header fields received inside the HTTP request.

`.FormDataListPtr`

This field is the head of the list of key-value pairs received in a form inside a POST request.

NULL, if no key-value pairs was received.

Fields to Read/Write

The following is a list of field that can be modified via some hook functions:

`.StatusCode`

HTTP status code of the connection. When it changes, the web server transmits the response (status code and status phrase) with the status code of the connection. See section HTTP Server Recognized Status Codes for a list of Status Code recognized by the HTTP Server module.

`.PathPtr`

At the request level, this field contains the path to the resource received in the HTTP request URL. It can be used for comparison inside hooks to do some operations according to the URL received.

At the response level, if the resource is a file (see RespBodyDataType field), this field will be used to fetch the file from the File System whose location is indicated by the PathPtr. This file will then be copied in the response body. This field will also be used to construct the location header if necessary.

Before the response level, the upper application can modify the PathPtr contents via the hook functions if its needs to modify the file to send in the response. If the resource to send back in the response is not a file, the field can still be used to set the path of the resource in case a location header also needs to be sent.

The pointer **MUST NOT** be changed at any moment. Only the buffer content pointed by PathPtr can be modified to write a new string.

`.DataPtr`

In the case where the resource to send in the response is a file contained in a FS (RespBodyDataType == FILE), this field is used by the server to store the file data to transmit. Therefore, It **MUST NOT** be modified by the application.

In the case where the resource to send in the response is static data located in memory (RespBodyDataType == STATIC DATA), this field **MUST** be set by the application via the hook functions. It should be set the point to the data to send.

In the case where the resource to send is also data located in memory (RespBodyDataType == STATIC DATA) but that must be sent in Chunked Transfer Encoding with the Chunk Hook , the DataPtr **MUST** be set to NULL by the application via the hooks functions. That way the Chunk Hook will be called later.

`.DataLen`

In the case where the resource to send is a file contain in a FS (RespBodyDataType == FILE), this field will be used by the server to store the length of the file. Therefore, it **MUST NOT** be modified by the application via the hook functions.

In the case where the resource to send is static data located in memory (RespBodyDataType == STATIC DATA) and that the DataPtr was set to the memory address of the data, the DataLen field **MUST** be set to the data length by the application.

When transferring data via the Chunk Hook because the DataPtr was previously set to NULL, the DataLen field is not taken into account by the server.

`.RespContentType`

This field will contains the content type of the response body.

If a file (that is in a File System) is to be sent and the RespContentType field is set to "Unknown", the server will parse the file extension to set the Content Type. Therefore, the application could set the field via the hook functions, but doesn't need to in the case of a file.

If static data is to be sent (RespBodyDataType == STATIC DATA), this field **MUST** absolutely be set by the application via the hook functions.

`.RespBodyDataType`

This field set the type of data of the response body. Three types are available:

- HTTPs_BODY_DATA_TYPE_FILE: When data body is from a file that was stored in a File System.
- HTTPs_BODY_DATA_TYPE_STATIC_DATA: When data body is from memory allocation or is sent via Chunk Hook .
- HTTPs_BODY_DATA_TYPE_NONE: When no body will be sent with the response.

By default this field is set to HTTPs_BODY_DATA_TYPE_FILE for GET and HEAD methods and to HTTPs_BODY_DATA_TYPE_NONE for POST, DELETE and PUT method. The field can be modified by the application via the hook functions.

`.ConnDataPtr`

This parameter is available for the upper application to store memory location for connection processing across all hook functions.

> All other fields MUST NOT be modified but they could be read for the connection processing and for debugging.

> We strongly recommend to use the three following API functions: HTTPs_RespBodySetParamFile () , HTTPs_RespBodySetParamStaticData() and HTTPs_RespBodySetParamNoBody() , to set the Response Body parameters adequately instead of changing the fields directly. Those functions will change the right fields for you according to the type of body you want to send.

HTTP Server Query String

A Query String is a set of Key-Value pairs or single values added at the end of the request's URL after a question mark character "?". Each fields of the Query are separated by the "&" character.

For more details, see section Query String .

The HTTP Server can supports the reception of a Query String in the HTTP request.

Configuration

Compile-Time Configuration

The configuration macro HTTPs_CFG_QUERY_STR_EN must be set to DEF_ENABLED in the http_server_cfg.h file to enable this feature.

See section Query String Configuration for more details.

Run-Time Configuration

The run-time configuration for Query String is defined as a HTTPs_QUERY_STR_CFG structure. Your application must declare an HTTPs_QUERY_STR_CFG object and include a pointer to this object in the HTTP server instance configuration HTTPs_CFG object where the QueryStrCfgPtr parameter is defined.

See section Query String Configuration for additional details.

Hook Function

The HTTP Server Instance configuration structure (HTTPs_CFG) contains a pointer to an HTTPs_HOOK_CFG object. This object indexes all the hook functions required by your application.

For additional details on the configuration of hook functions, see section Hook Configuration .

To visualize when each hook function is called by the HTTP server core, see section Hook Functions .

To retrieve the Query String data saved by the HTTP server, two hook functions can be used: OnReqHook and OnReqRdySignalHook .

Usage

When the feature is enabled, query strings received in an HTTP request are saved by the HTTP server core under a list of key-value pair. This list is available using the field .QueryStrListPtr inside the HTTPs_CONN structure. This list is accessible to your application inside the hook functions OnReqHook and OnReqRdySignalHook.

Single value fields inside the Query String are also supported. The value will still be saved in a Key-Value pair block, but only the value field will be meaningful.

HTTP Server Hook Functions

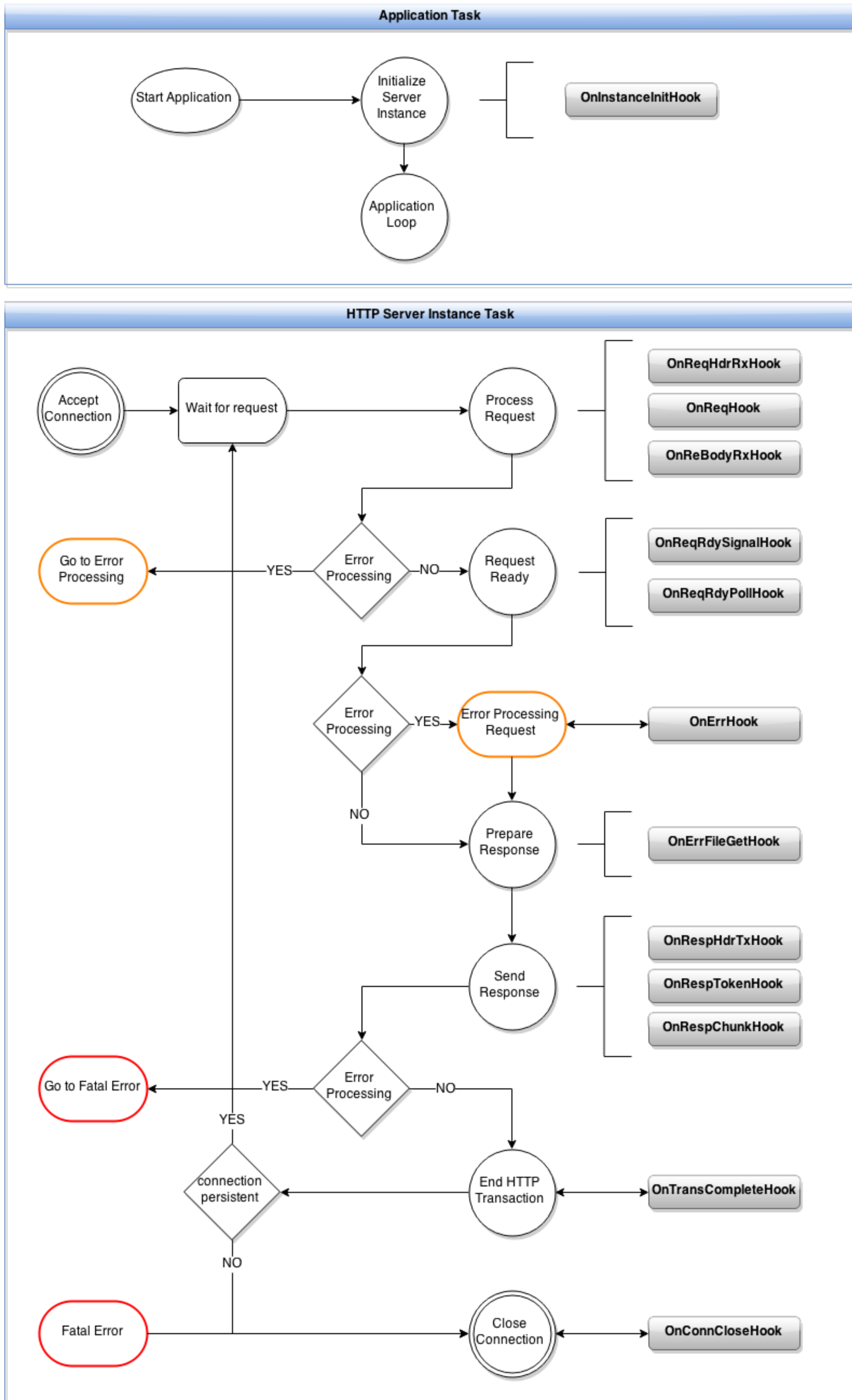
Each Web server is highly customizable using hook functions. Many hook functions can be called by the HTTP Server module during the processing of each client's request. Those hook functions allow the application to respond differently based on the application context. For example, it's possible to implement functionality such as dynamic content replacement, authentication, adding HTTP headers to the HTTP response (i.e., cookie), redirect, generate run-time data page, etc.

For more information on how to configure the hook functions for a given HTTP server instance, see the section Hook Configuration .

Also for the complete list of all the hook functions available and their prototype, see the section Hook Functions API .

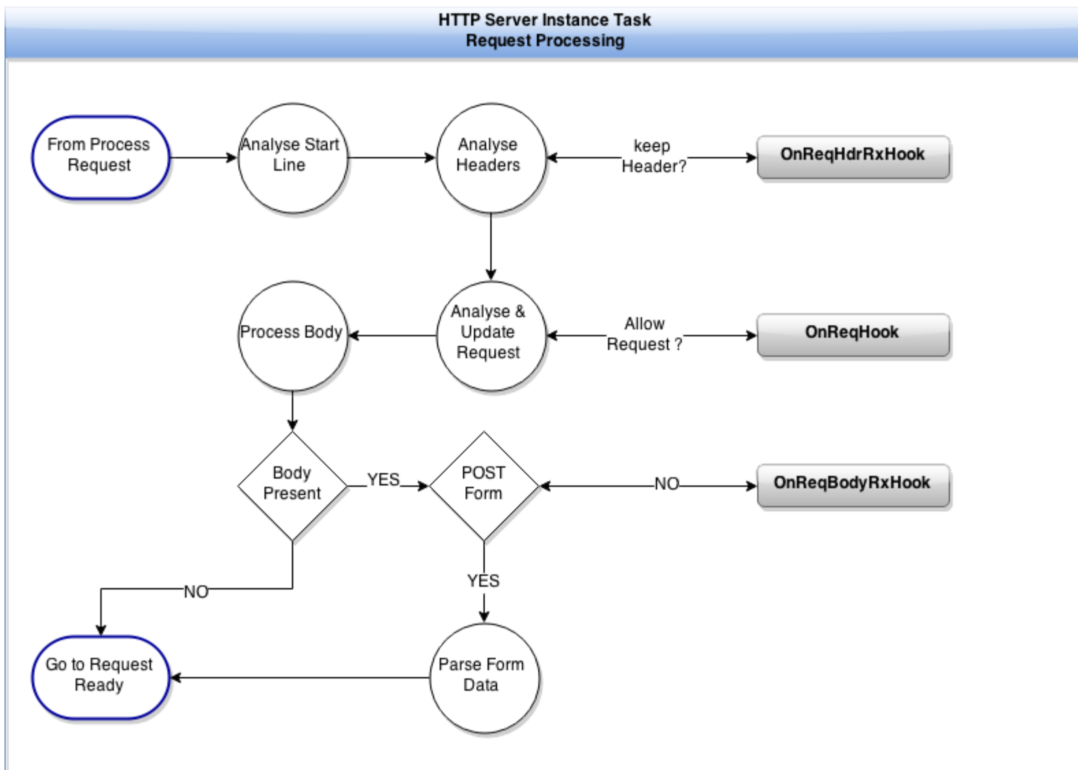
HTTP Tasks Global Overview

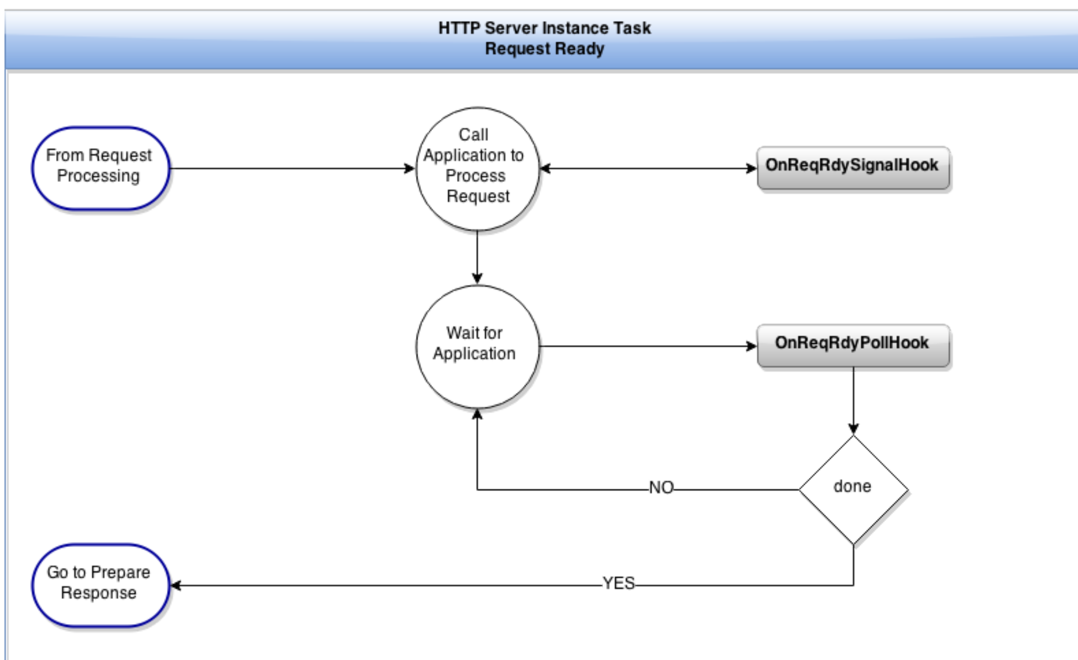
The diagrams below show a simplified state diagram of the HTTP server transaction processing but also when and where the hook functions are called.



HTTP Request Processing

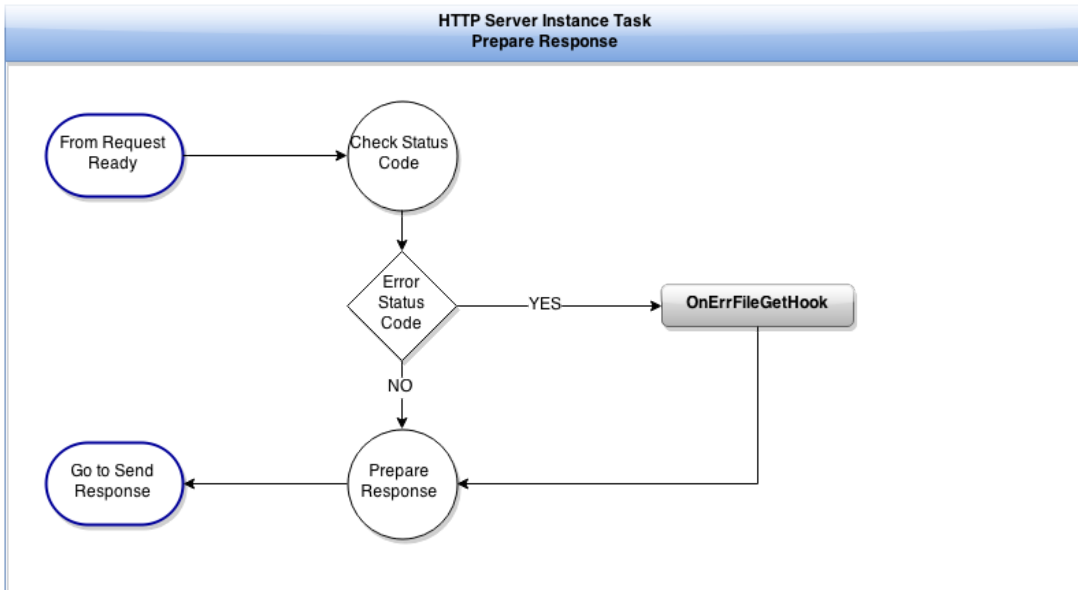
The diagrams below shows in more detail the HTTP received request processing section.

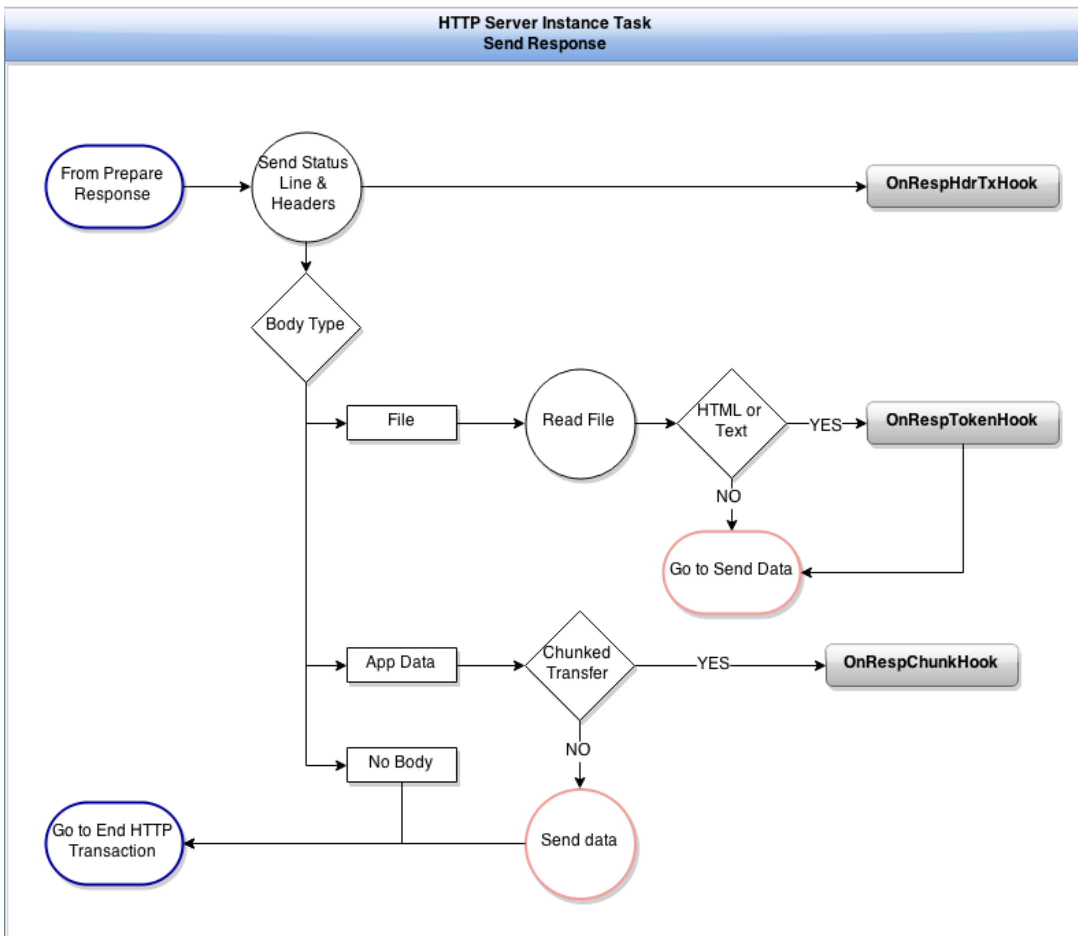




HTTP Response Processing

The diagrams below show in more details the HTTP response processing section.





HTTP Server Header Fields

The Header Field feature allows the HTTP Server module to receive and transmit additional HTTP headers to the one already processed internally by the HTTP server core.

The header fields listed below are already taken care of by the HTTP Server stack and must therefore not be added or processed by the application:

- Host

- Content-Type
- Content-Length
- Transfer-Encoding
- Connection
- Location

See section HTTP Header Fields Recognized for a list of all the header field recognized by the HTTP Server product.

To learn more on the HTTP header concepts, see section Headers .

Configuration

For the HTTP server to support processing of additional headers, two types of configuration must be set up: the compile-time and the run-time configuration. Also, some hook functions must be defined to allow the upper application to use this feature.

The Header Field feature is divided in two independent parts: the reception and the transmission. Each part has its own configuration as it will be shown below.

The reception is associated with HTTP header fields received in HTTP requests and the transmission is associated with HTTP header fields to send in HTTP responses.

Compile-time Configuration

Two configuration macros are available for the Header Field feature: HTTPs_CFG_HDR_RX_EN for the reception and HTTPs_CFG_HDR_TX_EN for the transmission. Each can be enabled and disabled separately.

See section Header Field Configuration for more details.

Run-time Configuration

The run-time configuration for the Header Field feature has defined two structures: HTTPs_HDR_RX_CFG and HTTPs_HDR_TX_CFG. If your application needs to save headers received in HTTP requests, it must declare an HTTPs_HDR_RX_CFG object and include a pointer to this object in the HTTP server instance configuration HTTPs_CFG object where the HdrRxCfgPtr parameter is defined. The same must be done with an HTTPs_HDR_TX_CFG object if your application wishes to add headers to HTTP responses. The HdrTxCfgPtr parameter of the HTTPs_CFG object must therefore be defined.

See section HTTP Header Configuration for additional details.

Hook Function

The general HTTP Server Instance configuration structure (HTTPs_CFG) contains a pointer to an HTTPs_HOOK_CFG object. This object indexes all the hook functions required by your application.

For additional details on the configuration of hook functions, see section Hook Configuration .

To visualize when each hook function is called by the HTTP server core, see section Hook Functions .

In reception, the hook OnReqHdrRxHook() can be used to set which header field the application wishes to keep.

In transmission, the hook OnRespHdrTxHook() can be used to add a header field to the header list that the server will use to construct the HTTP response to send.

Usage

Reception

Once a HTTP request is received by the HTTP server, the server will call, if defined, the OnReqHdrRxHook() for each header fields received. The upper application must implement this hook to the return status DEF_YES when the header field must be saved and DEF_NO otherwise.

Transmission

When the server is constructing an HTTP response to send, it will call the hook function OnRespHdrTxHook() when adding headers to the response. In this hook, the upper application can add header fields to the HTTP headers list by using the API function HTTPs_RespHdrGet(). This function will find a Header Field block available in the pool and add it to the list. The upper application just needs to fill the block fields afterwards.

HTTP Server Request Body Data

The HTTP Server module has the capacity to parse forms received in a POST request body when the Form feature is enabled on the server instance. For more information, see section Form .

Forms are the only body parsing integrated in the server core. For your application to retrieve the body data of other requests (or even form's data when the feature is disabled) the server offers a hook function: On Request Body Received Hook .

```
CPU_BOOLEAN  HTTPs_ReqBodyRxHook (const  HTTPs_INSTANCE  *p_instance,
                                  HTTPs_CONN      *p_conn,
                               const  void          *p_hook_cfg,
                                  void          *p_buf,
                               const  CPU_SIZE_T      buf_size,
                                  CPU_SIZE_T      *p_buf_size_used);
```

This hook, if defined, will be called by the HTTP server each time new data is received in the server's internal buffer. With this hook function, the application has the liberty to copy the data received or to read it directly in the HTTP buffer to retrieve only the information needed. The application doesn't have the obligation to read all the data contained in the buffer. It **MUST** return the length of the data that was read (p_buf_size_used) and therefore, the next time around, the data not read will still be in the reception buffer. If the application doesn't want to receive more data, it can return DEF_NO at any time and data received afterward will be dropped by the server.

HTTP Server Form

The HTTP Server module supports HTML form submissions through the POST method. The following POST Internet media type are supported:

- "application/x-www-form-urlencoded" is the default format for encoding key-value pairs when a web browser sends a POST request from a web form element.
- "multipart/form-data" is the format that must be used when uploading large amount of data such as a file on a web server.

For more information on HTML Forms, see section HTTP Form.

Configuration

For the HTTP server to support processing of incoming form submissions, two types of configuration must be set up: the compile-time and the run-time configuration. Also, some hook functions must be defined to allow the application to retrieve the data received in the form.

Compile-time Configuration

Two compile-time configuration macros are available for the form submissions: HTTPs_CFG_FORM_EN and HTTPs_CFG_FORM_MULTIPART_EN.

The first macro, if set to DEF_ENABLED, enables the Form feature and all the functions and data associated with it. Moreover, if the multipart form must be supported the second macro should also be enabled.

See section Module Configuration for more details.

Run-time Configuration

The run-time configuration for Form submission is defined as a HTTPs_FORM_CFG structure. Your application must declare an HTTPs_FORM_CFG object and include a pointer to this object in the HTTP server instance configuration HTTPs_CFG object where the FormCfgPtr parameter is defined.

See section HTTP Form Configuration for additional details.

Hook Functions

The general HTTP Server Instance configuration structure (HTTPs_CFG) contains a pointer to an HTTPs_HOOK_CFG object. This object indexes all the hook functions required by your application.

For additional details on the configuration of hook functions, see section Hook Configuration.

To visualize when each hook function is called by the HTTP server core, see section Hook Functions.

The hook function `OnReqRdySignalHook()` can be used to access the data received inside the form. The data is located under a list of key-value pairs (parameter FormDataListPtr) inside the HTTPs_CONN object. If a lot of processing must be done with the data received, it is recommend not to do it inside the `OnReqRdySignalHook()` hook because it will slow down all the other connection processing. Instead, the `OnReqRdySignalHook()` hook should be used to signal another task that the data was received and is ready to be processed. In that case, the hook function OnReqRdyPollHook () should be used to indicate to the server when the data processing is completed. When defined, the server will call the OnReqRdyPollHook () hook until the function returns the status Ok to indicate that the processing is finished.

Usage

When the form is posted, the web server will process the POST action and will invoke the callback with a list of key-value pairs transmitted.

Assuming the HTML page that look like this:

Listing - HTML Page Example Application Form

```
<html>
 <body>
  <form action="form_return_page.htm" method="post">
   Text Box 1: <input type="text" name="textbox1" />
   Text Box 2: <input type="text" name="textbox1" />
   <input type="submit" name="submit" value="Submit"></input>
  </form>
 </body>
</html>
```

The HTTP server core will receive the POST request and save the form data inside a key-value pairs list (FormDataListPtr parameter of the HTTPs_CONN structure). The server will then call the hook function OnReqRdySignalHook() (of the HTTPs_HOOK_CFG object inside the HTTPs_CFG object) to allow the application to retrieve and process the data received. The key-value pairs inside the list will be:

Key: "textbox1", Value: "Text Box 1 value"

Key: "textbox2", Value: "Text Box 2 value"

Key: "submit", Value: "Submit"

Since all connections are processing within only one task it is very important that all hook functions are non-blocking and can be executed quickly.

If the upper application takes a while to complete the data processing, the upper application should implement a task where the data can be process. The task should be polled to know if the processing is completed with the OnReqRdyPollHook () hook.

Uploading files using the HTTP Server

It is possible to use a HTML form to upload files on the file system using the HTTP Server. The following listing shows an example of HTML code that can be used to implement that functionality:

Listing - HTML Page Example Multipart Form

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN">
<html>
<head>
  <title>File Upload</title>
</head>
<body>
 <form action="File_Upload.htm"
      ENCTYPE="multipart/form-data"
      method="post">
  <p><b>Choose The file To Upload:</b>
    <input type="file" name="file_form_field_name" ></input></p>
  <p><input type="submit" name="Submit" value="Submit"></input></p>
 </form>
</body>
</html>
```

Multipart form type must absolutely be specified in the form tag. For example: `\<form action="upload.html" ENCTYPE="multipart/form-data" method="post"\>`

When the file upload feature is enabled (parameter MultipartFileUploadEn inside the run-time configuration), the HTTP Server instance writes the file to the default location specified by the run-time configuration. If no error occurred when writing the file a key-value pair will be added to the list to specify the HTML control name of the field and the absolute location of the uploaded file. Thus, the upper application can easily retrieve or even move any file transmitted when the OnReqRdySignalHook() function is called. For example, for the upper listing, the key value pair list will contain the following:

Key: "file_form_field_name", Value: "*//Path/RemoteFileName.html*"

Key: "submit", Value: "Submit"

If an error occurs during the file transfer such as inability to open or write the file, the `OnReqRdySignalHook()` function is not called, and the upper application is notified by the Error hook function.

Note that the file upload is not possible using the static file system.

HTTP Server Response Body Data

To configure and edit the response body data, the HTTP Server module offers three API functions:

HTTPs_RespBodySetParamFile()	This function can be used to transmit in the response body a file contained in a File System (a standard FS or the HTTP Static FS) . See section Specify File to Send in Response Body .
HTTPs_RespBodySetParamStaticData()	This function can be used to transmit data contained in an application buffer or also to specify that the data should be sent with the Chunked Transfer Encoding. See sections Send data contained in Application Buffer and Send data with the Chunk Hook .
HTTPs_RespBodySetParamNoBody()	This function can be used to specify that no body should be added in the HTTP response.

Those three API functions should be used by the upper application inside one of two hook functions: Connection Request and Request Ready Signal .

Specify File to Send in Response Body

```
CPU_BOOLEAN  App_OnReqHook (const  HTTPs_INSTANCE  *p_instance,                                              HTTPs_CONN     *p_conn,                    const  void         *p_hook_cfg){    RTOS_ERR  err_https;     HTTPs_RespBodySetParamFile(p_instance,                               p_conn,                            "index.html",                         HTTP_CONTENT_TYPE_HTML,                        DEF_NO,                               &err_https);    return (DEF_YES);}
```

Send data contained in Application Buffer

```
#define  APP_BUF_LEN  20CPU_CHAR  AppBuf[APP_BUF_LEN]; CPU_BOOLEAN  App_OnReqHook (const  HTTPs_INSTANCE  *p_instance,                                   HTTPs_CONN    *p_conn,                    const  void         *p_hook_cfg){   RTOS_ERR   err_https;    Str_Copy_N(&AppBuf[0], "Hello World", APP_BUF_LEN);   data_len = Str_Len_N(&AppBuf[0], APP_BUF_LEN);   HTTPs_RespBodySetParamStaticData (p_instance,                                     p_conn,                          HTTP_CONTENT_TYPE_PLAIN,                          &AppBuf[0],                          data_len,                          DEF_NO,                          &err_https);    return (DEF_YES);}
```

Send data with the Chunk Hook

```
CPU_BOOLEAN  App_OnReqHook (const  HTTPs_INSTANCE  *p_instance,                                    HTTPs_CONN    *p_conn,                    const  void         *p_hook_cfg){   RTOS_ERR  err_https;    HTTPs_RespBodySetParamStaticData (p_instance,                                     p_conn,                          HTTP_CONTENT_TYPE_JSON,                          DEF_NULL,                          0,                          DEF_NO,                          &err_https);    return (DEF_YES);} CPU_BOOLEAN  HTTPs_RespChunkDataGetHook(const  HTTPs_INSTANCE  *p_instance,                          HTTPs_CONN     *p_conn,                          const  void         *p_hook_cfg,                          void          *p_buf,                          CPU_SIZE_T      buf_len_max,                          CPU_SIZE_T      *p_tx_len){}
```

HTTP Server Token Replacement

The HTTP Server module allows dynamic content to be inserted in HTML and plain text documents by using special tokens being substituted before the page is actually sent to the web browser. There are two type of token: internal and external. The value of external token must be provided by the upper application via a hook function. Internal tokens are defined to print some data related to HTTP such as status lines and error codes.

Configuration

Compile-time Configuration

The macro configuration HTTPs_CFG_TOKEN_PARSE_EN inside the http_server_cfg.h file must be set to DEF_ENABLED to enable the token substitution feature.

See section Dynamic Content Configuration for more details on compile-time configuration.

Run-time Configuration

The run-time configuration for Token Replacement is defined as a HTTPs_TOKEN_CFG structure. Your application must declare an HTTPs_TOKEN_CFG object and include a pointer to this object in the HTTP server instance configuration HTTPs_CFG object where the TokenCfgPtr parameter is defined.

See section Token Configuration for additional details.

Hook Configuration

The HTTP Server Instance configuration structure (HTTPs_CFG) contains a pointer to an HTTPs_HOOK_CFG object. This object indexes all the hook functions required by your application.

For additional details on the configuration of hook functions, see section Hook Configuration .

To visualize when each hook function is called by the HTTP server core, see section Hook Functions .

The hook function OnRespTokenHook() must be used to set the value that will be used to replace the token found. Each time a token is found in the HTML/text file, the hook function will be called.

Usage

External Token

External tokens are represented in the HTML/text files as:

```
${TOKEN_NAME}
```

Assuming we have an HTML page that look like this:

```
<html><body><center> This system's IP address is ${My_IP_Address} </center></body></html>
```

When a web client requests this file, the HTTP Server will parse the file, find ${My_IP_Address} token, and pass the string "My_IP_Address" into the OnRespTokenHook() hook function. That function will copy the token value then the HTTP Server instance will send the token value instead of the token:

```
<html><body><center> This system's IP address is 135.17.115.215 </center></body></html>
```

Note that if the upper application doesn't copy a valid data, the token is automatically replaced by '~' characters. See Response Token Transmit Hook for more information on how the callback function has to be implemented to support this.

Internal Token

Internal tokens are represented in the HTML files as:

#{TOKEN_NAME}

For every HTML document that must be transmitted, the HTTP Server parses the document and replaces all internal tokens automatically without calling any callback function. Internal token can be use in any HTML/text documents. Here is the list of built-in token replacement:

Token	Value
#{STATUS_CODE}	Replaced by status code number of the connection.
#{REASON_PHRASE}	Replaced by the reason phrase based on the status code of the connection

HTTP Server Proxy

The HTTP Server module can be accessed behind a proxy. When an HTTP Server is behind a HTTP Proxy, the HTTP client must send its requests using an absolute URI. For example,

GET http://example.com/index.html HTTP/1.1

When the absolute URI feature is enable, the HTTP server will support absolute URI in the first line of the HTTP request messages.

The server will also look for the Host header field in the received request messages and save it in the .HostPtr field of the HTTPs_CONN structure.

HTTP Server Add-on

The HTTP Server module offers different add-on to simplify complex HTTP server application development.

Add-on Module	Description
Control Layer	The Control Layer is a module bound to the HTTP Server by its hook functions and used to manage multiple groups of authentication process and application process.
Authentication	This module basically implements an authentication and authorization process based on the HTTP cookie header. It is made to work with the Control Layer only.
REST	This module was created to facilitate the development of RESTful applications. Its allows you to create REST resources based on a specific URL. Each resource has its own set of hook functions with a different hook for each HTTP method. It can be used with or without the Control Layer module.

HTTP Server Control Layer

- Introduction
- Authentications

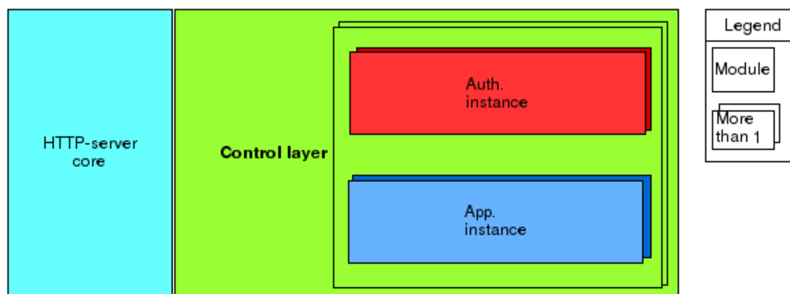
Applications
- Configuration
  - Demystify the HooksCfgPtr Parameter
  - HTTPs_CTRL_LAYER_CFG and HTTPs_CTRL_LAYER_CFG_LIST
- HTTP Server Hook Binding

Introduction

The control layer is a module bound to the HTTP Server used to manage multiple groups of authentication process and application process. There are many cases where one would like to have multiple functions bound to the same hook but with different goals. This is a the role of the Control layer.

The control layer maps each hook function available in the HTTP Server on its own hook. The configuration of those hooks has two layers: The Authentications and the Applications.

It provides an abstraction over the HTTPs_CONN's and HTTPs_INSTANCE's contextual data holders: p_instance->DataPtr and p_conn->ConnDataPtr. The Control Layer will automatically substitute them. Those contextual information are saved after each hook call and restored before each hook call. Which means that each Control Layer instance has its own connection and instance data pointers. The following diagram shows these context switches through color variance:



Authentications

The Authentications layer is the part of the configuration where one can specify how to hide an application, e.g., put your application behind a Login form. Each Control Layer configuration can have a list of authentications which must succeed in order to access any application of the same configuration.

Listing - Authentication structure

```
typedef struct https_ctrl_layer_auth_hooks {
    HTTPs_INSTANCE_INIT_HOOK   OnInstanceInit;
    HTTPs_REQ_HDR_RX_HOOK      OnReqHdrRx;
    HTTPs_REQ_HOOK             OnReqAuth;
    HTTPs_RESP_HDR_TX_HOOK     OnRespHdrTx;
    HTTPs_TRANS_COMPLETE_HOOK  OnTransComplete;
    HTTPs_CONN_CLOSE_HOOK      OnConnClose;
} HTTPs_CTRL_LAYER_AUTH_HOOKS;

typedef struct https_ctrl_layer_auth_inst {
    HTTPs_CTRLLAYER_AUTH_HOOKS    *HooksPtr;
    void                          *HooksCfgPtr;
} HTTPs_CTRL_LAYER_AUTH_INST;
```

Therefore, for a given Control Layer configuration, many Authentication modules can be defined. Each module must declared its own HTTPs_CTRL_LAYER_AUTH_INST object that regroups a pointer to its own hook configuration

(HTTPs_CTRL_LAYER_AUTH_HOOKS object) and a pointer to other configurations required by the module if necessary (See section HooksCfgPtr Parameter).

Applications

The Application layer is the part where HTTP requests can be analyze and HTTP responses will be built. At this point, the authentications required have passed already. The application layer handles both body parsing and body formatting. The application has full control over the HTTP Server response and is called every time the server needs to poll the hook to complete its task.

Listing - Application structures

```
typedef struct https_ctrl_layer_app_hooks {
    HTTPs_INSTANCE_INIT_HOOK    OnInstanceInit;
    HTTPs_REQ_HDR_RX_HOOK       OnReqHdrRx;
    HTTPs_REQ_HOOK              OnReq;
    HTTPs_REQ_BODY_RX_HOOK      OnReqBodyRx;
    HTTPs_REQ_RDY_SIGNAL_HOOK   OnReqSignal;
    HTTPs_REQ_RDY_POLL_HOOK     OnReqPoll;
    HTTPs_RESP_HDR_TX_HOOK      OnRespHdrTx;
    HTTPs_RESP_TOKEN_HOOK       OnRespToken;
    HTTPs_RESP_CHUNK_HOOK       OnRespChunk;
  HTTPs_TRANS_COMPLETE_HOOK   OnTransComplete;
    HTTPs_ERR_HOOK              OnError;
    HTTPs_CONN_CLOSE_HOOK       OnConnClose;
} HTTPs_CTRL_LAYER_APP_HOOKS;

typedef struct https_ctrl_layer_app_inst {
    HTTPs_CTRL_LAYER_APP_HOOKS    *HooksPtr;
    void                          *HooksCfgPtr;
} HTTPs_CTRL_LAYER_APP_INST;
```

As for the Authentication Layer, a given Control Layer configuration can have many Application modules. each module must declared its own HTTPs_CTRL_LAYER_APP_INST object that regroups a pointer to its own hook configuration (HTTPs_CTRL_LAYER_APP_HOOKS object) and a pointer to other configurations required by the module if necessary (See section HooksCfgPtr Parameter).

Configuration

Demystify the HooksCfgPtr Parameter

The HooksCfgPtr parameter will be passed as argument with each hook function from HooksPtr. Therefore, HooksCfgPtr parameter is offered to pass information necessary to specify the behavior of a given authentication instance or application instance. This information highly depends on the instance nature. If not necessary, the HooksCfgPtr parameter can be set to DEF_NULL.

HTTPs_CTRL_LAYER_CFG and HTTPs_CTRL_LAYER_CFG_LIST

The control layer takes an HTTPs_CTRL_LAYER_CFG_LIST structure pointer as configuration. Below is the definition of the structures for the configuration of the Control Layer:

Listing - HTTPs_CTRLL_CFG and HTTPs_CTRLL_CFG_LIST

```
/* Controlled service layer. */
typedef  struct  https_ctrl_layer_cfg {
    HTTPs_CTRL_LAYER_AUTH_INST     **AuthInstsPtr;
    CPU_SIZE_T                     AuthInstsNbr;
    HTTPs_CTRL_LAYER_APP_INST      **AppInstsPtr;
    CPU_SIZE_T                     AppInstsNbr;
} HTTPs_CTRL_LAYER_CFG;

/* List of controlled services. */
typedef  struct https_ctrl_layer_cfg_List {
    HTTPs_CTRL_LAYER_CFG  **CfgsPtr;
    CPU_SIZE_T            Size;
} HTTPs_CTRL_LAYER_CFG_LIST;
```

Here's an example how to declare a Control Layer configuration:

Listing - Control layer config example

```
/* Define some authentication hook set. */
HTTPs_CTRL_LAYER_AUTH_HOOKS  AuthHooks    = { ... };

/* Create an auth instance based on the hook set and a user config pointer */
HTTPs_CTRL_LAYER_AUTH_INST   AuthInst     = { &AuthHooks, ... };

/* Define some application hook set */
HTTPs_CTRL_LAYER_APP_HOOKS   SomeAppHooks = { ... };

/*Create an app instance based on the hook set and a user config pointer */
HTTPs_CTRL_LAYER_APP_INST    SomeAppInst  = { &SomeAppHooks, ... };

/* List both authentications and applications */
HTTPs_CTRL_LAYER_AUTH_INST  *AuthInsts[]  = { &AuthInst };
HTTPs_CTRL_LAYER_APP_INST   *AppInsts[]   = { &SomeAppInst };

/* Create a config for the control layer */
HTTPs_CTRL_LAYER_CFG        SomeCfg      = {
        authInsts,
        sizeof(authInsts) / sizeof(HTTPs_CTRL_LAYER_AUTH_INST *),
        appInsts,
        sizeof(appInsts)  / sizeof(HTTPs_CTRL_LAYER_APP_INST *),
};

/* List the configurations. */
HTTPs_CTRL_LAYER_CFG        *CtrlLayerCfgs[] = { &SomeCfg };

/* Build the cfg structure for the control layer */
HTTPs_CTRL_LAYER_CFG_LIST    CtrlLayerCfgList= {
        CtrlLayerCfgs,
        sizeof(CtrlLayerCfgs) / sizeof(HTTPs_CTRL_LAYER_CFG *)
};
```

HTTP Server Hook Binding

Here's an example of how to bind it to an HTTPs_CFG:

Listing - HTTPs_CFG using ControlLayer

```
const  HTTPs_CFG  foo = {
    ...,

/*
*--------------------------------------------------------------------------------------------------------
*                                    HOOKS CONFIGURATION
*--------------------------------------------------------------------------------------------------------
*/
                                  /* Declared in HTTP/Server/Add-on/CtrlLayer/http-s_ctrl_layer.h  */
  &HTTPsCtrlLayer_HookCfg,           /* .HooksPtr                                                    */

                                /* Previously declared in the configuration section              */
  &CtrlLayerCfgList,              /* .Hooks_CfgPtr                                               */

    ...
};
```

HTTP Server Authentication module

- Introduction
- Authentication Layer : Cookie Checker
- Application Layer
  - Unprotected Requests: Cookie Creator
  - Protected Requests
- Session Timeout
- How to Configure
- Hook Functions
  - Get Required Rights Hook
  - Login Hook
  - Logout Hook
- Example

Introduction

The HTTP Server authentication system is built on top of the Control Layer. Without the control layer it wont work. It also requires the Authentication module of the Micrium OS Common libraries that defines the User Management.

There is two subsystems for this module: The **Control Layer Authentication Layer** and the **Control Layer Application Layer**.

Authentication Layer : Cookie Checker

The cookie checker is basically a Control Layer instance bound to the Authentication layer. When a request is made without a valid cookie, it refuses to let it go and will take on the request. When a cookie is present and valid, the request is then checked to know which rights are required to continue. If the user bound to the cookie provided has the rights to continue, the request is marked as accepted.

Application Layer

Unprotected Requests: Cookie Creator

The cookie creator work after the cookie checker has done it's work and could not find a valid cookie. In that case, if the request is a POST with the "*logme form*", the request is "catched" and the parsed body is inspected to find a username and a password.This happens at the application phase of the Control Layer. If a username and a password is found within the request, it is then validated with known credentials from the user management layer. If a valid match is found, a session is created, the user is associated with it, and the cookie identifier is sent to the HTTP client.

Protected Requests

Once the session was created, requests related to *log in* should not occurred, but for safety the Authentication module will still "catched" them. A request to the login page will be redirect to the default page. A POST request with the *"logme form"* will be parsed even though a session is already existing and if the credentials are incorrect the session will be terminated.

Session Timeout

The Authentication module also configures a timer to periodically check if any created sessions have timed out. After a session timed out, a cookie received with the session ID number will be considered invalid and any not authorized requests will be redirect to the configured page for unauthorized requests.

How to Configure

To configure the HTTP Server authentication, here are the steps to follow:

1. Create users with their appropriate credentials and rights.
2. Define an hook function which will return the right necessary to process a given request.
3. Define an hook function which will check the requests received for a *log in* request but also set the pages for redirection for all of those requests.
4. Define an hook function which will check the requests received for a *log out* request.
5. Create the appropriate configurations and bind those three hook functions.
6. Bind those configurations to their respective Control Layer Instance configuration.
7. Use the three instances created in the Control Layer configuration.

Hook Functions

Get Required Rights Hook

This hook will be called by the Authentication Layer of the Control Layer when a request is received to check with the upper application what rights are associated with this request.

```
AUTH_RIGHT  HTTPsAuth_GetRequiredRightsHook (const  HTTPs_INSTANCE  *p_instance,
                                            const  HTTPs_CONN     *p_conn);
```

Login Hook

This hook will be called by the Application Layer of the Control Layer to found requests related to the log in process, e.g., the GET requests for the resources of the *log in* page (html, css, images) and the POST request with the *form log in*. This hook functions will be called at two occasions : when the URL and headers of the request were received and parse by the server (state HTTPs_AUTH_STATE_REQ_URL), and also when the request body was received and parse (state HTTPs_AUTH_STATE_REQ_COMPLETE).

```
CPU_BOOLEAN  AppGlobal_Auth_ParseLoginHook (const  HTTPs_INSTANCE    *p_instance,
                                           const  HTTPs_CONN        *p_conn,
                                                  HTTPs_AUTH_STATE   state,
                                                  HTTPs_AUTH_RESULT  *p_result);
```

Logout Hook

This hook will be called by the Application Layer of the Control Layer

```
CPU_BOOLEAN  AppGlobal_Auth_ParseLogoutHook (const  HTTPs_INSTANCE    *p_instance,
                                            const  HTTPs_CONN        *p_conn,
                                                   HTTPs_AUTH_STATE   state);
```

Example

The output of the "How to" steps should look like the example provided:

Listing - Authentication Functions Examples

```
/*
****************************************************************************************************
*                                AppGlobal_UsersInit()
*
* Description : Adds three users to the authentication system.
*
* Argument(s) : None.
*
* Return(s)   : DEF_OK,   if users initialization was successful.
*               DEF_FAIL, otherwise.
*
* Note(s)     : None.
****************************************************************************************************
*/

CPU_BOOLEAN  AppGlobal_UsersInit(void)
{
    AUTH_USER    admin;
    AUTH_USER    user;
    AUTH_RIGHT   right;
    RTOS_ERR     err_auth;
    CPU_BOOLEAN  result;

    result = Auth_Init(&err_auth);
    if (result != DEF_OK) {
        return (DEF_FAIL);
    }

    result = Auth_CreateUser("admin",
                             "password",
                             &admin,
                             &err_auth);
    if (result != DEF_OK) {
        return (DEF_FAIL);
    }

    right = (HTTP_USER_ACCESS | AUTH_RIGHT_MNG);
    result = Auth_GrantRight(right,
                            &admin,
                            &Auth_RootUser,
                            &err_auth);
    if (result != DEF_OK) {
        return (DEF_FAIL);
    }
    result = Auth_CreateUser("user0",
                             "",
                             &user,
                             &err_auth);
    if (result != DEF_OK) {
        return (DEF_FAIL);
    }

    result = Auth_GrantRight(HTTP_USER_ACCESS,
                            &user,
                            &admin,
                            &err_auth);
    if (result != DEF_OK) {
        return (DEF_FAIL);
    }

    result = Auth_CreateUser("user1",
                             "user1",
                             &user,
                             &err_auth);
    if (result != DEF_OK) {
        return (DEF_FAIL);
    }
```

```
    result = Auth_Init(&err_auth);if(result != DEF_OK){return(DEF_FAIL);}

    result = Auth_CreateUser("admin","password",&admin,&err_auth);if(result != DEF_OK){return(DEF_FAIL);}

    right = (HTTP_USER_ACCESS | AUTH_RIGHT_MNG);
    result = Auth_GrantRight(right,&admin,&Auth_RootUser,&err_auth);if(result != DEF_OK){return(DEF_FAIL);}
    result = Auth_CreateUser("user0","",&user,&err_auth);if(result != DEF_OK){return(DEF_FAIL);}

    result = Auth_GrantRight(HTTP_USER_ACCESS,&user,&admin,&err_auth);if(result != DEF_OK){return(DEF_FAIL);}

    result = Auth_CreateUser("user1","user1",&user,&err_auth);if(result != DEF_OK){return(DEF_FAIL);}

    result = Auth_GrantRight(HTTP_USER_ACCESS,&user,&admin,&err_auth);if(result != DEF_OK){return(DEF_FAIL);}return(DEF_OK);}/*
*********************************************************************************************************
*                           AppGlobal_Auth_GetRequiredRightsHook()
*
* Description : Returns the rights required to fulfill the needs of a given request.
*
* Argument(s) : p_instance  Pointer to the HTTP server instance object.
*
*               p_conn      Pointer to the HTTP connection object.
*
* Return(s)   : returns the authorization right level for this example application.
*
* Note(s)     : none.
*********************************************************************************************************
*/

AUTH_RIGHT  AppGlobal_Auth_GetRequiredRightsHook (const  HTTPs_INSTANCE  *p_instance,
                                                  const  HTTPs_CONN      *p_conn){return(HTTP_USER_ACCESS);}/*
*********************************************************************************************************
*                           AppGlobal_Auth_ParseLoginHook()
*
* Description : (1) Check the HTTP requests received to see if they are related to resources of the login page.
*                   (a) Check if the POST login is received.
*                   (b) For each request set the redirect paths on no, invalid & valid credentials.
*                   (c) Parse the form fields received in the body for the user and password.
*
*
* Argument(s) : p_instance  Pointer to the HTTP server instance object.
*
*               p_conn      Pointer to the HTTP connection object.
*
*               state       State of the Authentication module:
*                               HTTPs_AUTH_STATE_REQ_URL:      The URL and the headers were received and parse.
*                               HTTPs_AUTH_STATE_REQ_COMPLETE: All the request (URL + headers + body)
*                                                              was received and parse.
*
*               p_result    Pointer to the authentication result structure to fill.
*
* Return(s)   : DEF_YES, if the request is the POST login.
*               DEF_NO,  otherwise.
*
* Note(s)     : (2) This hook will be called twice for a request processed by the Authentication module:
*                   (a) When the Start line of the request (with the URL) and the headers have been
*                       received and parse -> HTTPs_AUTH_STATE_REQ_URL state.
*                   (b) When all the request has been completely received and parse including the body
*                       -> HTTPs_AUTH_STATE_REQ_COMPLETE state.
*
*               (3) for each request received the redirect paths is set as follow:
*                   (a) RedirectPath_OnValidCred    INDEX_PAGE_URL
*                   (b) RedirectPath_OnInvalidCred  LOGIN_PAGE_URL
*                   (c) RedirectPath_OnNoCred
*                       (1) if the path is an unprotected path, let it go. (DEF_NULL) (i.e., the logo)
*                       (2) otherwise               LOGIN_PAGE_URL
*********************************************************************************************************
```

```
*/

CPU_BOOLEAN  AppGlobal_Auth_ParseLoginHook (const  HTTPs_INSTANCE    *p_instance,
                                 const  HTTPs_CONN       *p_conn,
                                     HTTPs_AUTH_STATE   state,
                                     HTTPs_AUTH_RESULT  *p_result){
   CPU_INT16S      cmp_val;
   CPU_BOOLEAN     is_login = DEF_NO;
#if(HTTPs_CFG_FORM_EN == DEF_ENABLED)
   HTTPs_KEY_VAL  *p_current;
   CPU_INT16S      cmp_val_username;
   CPU_INT16S      cmp_val_password;
#endif
   p_result->UsernamePtr = DEF_NULL;
   p_result->PasswordPtr = DEF_NULL;/* Set redirect paths for each requests.      */
   p_result->RedirectPathOnInvalidCredPtr = LOGIN_PAGE_URL;
   p_result->RedirectPathOnValidCredPtr   = INDEX_PAGE_URL;

   switch (state){/* ----------- REQUEST URL RECEIVED ---------- */
      case HTTPs_AUTH_STATE_REQ_URL:/* Set redirect paths for each requests.      */
         cmp_val =Str_Cmp(p_conn->PathPtr, LOGIN_PAGE_URL);if(cmp_val !=0){
            cmp_val =Str_Cmp(p_conn->PathPtr, MICRIUM_LOGO_URL);if(cmp_val !=0){
               cmp_val =Str_Cmp(p_conn->PathPtr, MICRIUM_CSS_URL);}}
         p_result->RedirectPathOnNoCredPtr =(cmp_val ==0)? DEF_NULL : LOGIN_PAGE_URL;/* Check if POST login received.            */
         cmp_val =Str_Cmp(p_conn->PathPtr, LOGIN_PAGE_CMD);if(cmp_val ==0){
            is_login = DEF_YES;}break;/* ---------- REQUEST BODY RECEIVED ---------- */
      case HTTPs_AUTH_STATE_REQ_COMPLETE:
#if(HTTPs_CFG_FORM_EN == DEF_ENABLED)/* Parse form fields received for user/pass.   */
         p_current = p_conn->FormDataListPtr;while((p_current           != DEF_NULL)&&((p_result->UsernamePtr == DEF_NULL)||(p_result-
>PasswordPtr == DEF_NULL))){if(p_current->DataType == HTTPs_KEY_VAL_TYPE_PAIR){
               cmp_val_username =Str_CmpIgnoreCase_N(p_current->KeyPtr,
                                          FORM_USERNAME_FIELD_NAME,
                                          p_current->KeyLen);
               cmp_val_password =Str_CmpIgnoreCase_N(p_current->KeyPtr,
                                          FORM_PASSWORD_FIELD_NAME,
                                          p_current->KeyLen);if(cmp_val_username ==0){
                  p_result->UsernamePtr = p_current->ValPtr;
                  is_login = DEF_YES;}elseif(cmp_val_password ==0){
                  p_result->PasswordPtr = p_current->ValPtr;}}
            p_current = p_current->NextPtr;}
#endif
         break;

      default:break;}return(is_login);}/*
*********************************************************************************************************
*                             AppGlobal_Auth_ParseLogoutHook()
*
* Description : Parse requests received for logout URL and form data logout info.
*
* Argument(s) : p_instance  Pointer to HTTPs instance object.
*
*             p_conn      Pointer to HTTPs connection object.
*
*             state       State of the Authentication module:
*                         HTTPs_AUTH_STATE_REQ_URL:      The URL and the headers were received and parse.
*                         HTTPs_AUTH_STATE_REQ_COMPLETE: All the request (URL + headers + body) w
*                                                       as received and parse.
*
* Return(s)   : DEF_YES, if Logout received.
*             DEF_NO,  otherwise.
*
* Caller(s)   : AppGlobal_AppInst_AuthCfg.
*
* Note(s)     : (1) This hook will be called twice for a request processed by the Authentication module:
*               (a) When the Start line of the request (with the URL) and the headers have been
*                   received and parse -> HTTPs_AUTH_STATE_REQ_URL state.
*               (b) When all the request has been completely received and parse including the body
*                   -> HTTPs_AUTH_STATE_REQ_COMPLETE state.
```

```
*********************************************************************************************
*/

CPU_BOOLEAN  AppGlobal_Auth_ParseLogoutHook (const  HTTPs_INSTANCE   *p_instance,
                                            const  HTTPs_CONN       *p_conn,
                                                   HTTPs_AUTH_STATE   state){
    CPU_BOOLEAN     is_logout = DEF_NO;
#if(HTTPs_CFG_FORM_EN == DEF_ENABLED)
    HTTPs_KEY_VAL  *p_current;
    CPU_INT16S      cmp_val;
#endif

    switch (state){/* ----------- REQUEST URL RECEIVED ---------- */
        case HTTPs_AUTH_STATE_REQ_URL:/* Check if POST logout received.            */
            cmp_val =Str_Cmp(p_conn->PathPtr, LOGOUT_PAGE_CMD);if(cmp_val ==0){
                is_logout = DEF_YES;}break;/* ---------- REQUEST BODY RECEIVED ---------- */
        case HTTPs_AUTH_STATE_REQ_COMPLETE:
#if(HTTPs_CFG_FORM_EN == DEF_ENABLED)/* Parse form fields received for logout.      */
            p_current = p_conn->FormDataListPtr;while(p_current != DEF_NULL){if(p_current->DataType == HTTPs_KEY_VAL_TYPE_PAIR){
                    cmp_val =Str_CmpIgnoreCase_N(p_current->KeyPtr,
                                                 FORM_LOGOUT_FIELD_NAME,
                                                 p_current->KeyLen);if(cmp_val ==0){
                        is_logout = DEF_YES;break;}}
                p_current = p_current->NextPtr;}
#endif
            break;

        default:break;}return(is_logout);}
```

Code Block 1 Authentication Configuration Variables

```
/*******************************************************************************************************************************

VARIABLES*********************************************************************************************************************


 HTTPs_AUTHORIZATION_CFG  HTTPs_AppGlobal_AuthInst_Cfg = {       AppGlobal_Auth_GetRequiredRightsHook};
 HTTPs_AUTH_CFG  HTTPs_AuthAppCfg = {       AppGlobal_Auth_ParseLoginHook,       AppGlobal_Auth_ParseLogoutHook};
 HTTPs_CTRL_LAYER_AUTH_INST  AppGlobal_AuthInst = {       &HTTPsAuth_CookieHooksCfg,       &AppGlobal_AuthInst_Cfg};
 HTTPs_CTRL_LAYER_APP_INST  AppGlobal_AppInst_AuthUnprotected = {       &HTTPsAuth_AppUnprotectedCookieHooksCfg,
&AppGlobal_AppInst_AuthCfg};
 HTTPs_CTRL_LAYER_APP_INST  AppGlobal_AppInst_AuthProtected = {       &HTTPsAuth_AppProtectedCookieHooksCfg,
&AppGlobal_AppInst_AuthCfg  };
```

HTTP Server REST module

- Introduction
- REST Resource
  - Hooks Configuration
  - URI Pattern Matching
    - URI Path Separator
    - URI Path Component
    - Wild Cards
    - URI Pattern and Match Examples:
- Publish REST Resources
- REST Configuration
- How to Use REST Module
- Example

Introduction

The REST Framework is built to be either used on top of the Control layer or the server core itself. It remaps the HTTP Server hooks or the applicative Control Layer hooks on a single hook per HTTP request method. This single method is called an HTTPs_REST_HOOK.

The REST Framework is designed to ease the production of simple web service applications. HTTP requests are made to perform actions on web resources. Inside your REST framework, those resources are mapped to HTTPs_REST_RESOURCE objects.

REST Resource

A resource is associated with a specific path. REST Resources are accessible through path pattern matching and specifies the method supported and the headers needed. Here's the structure defining them:

Listing - REST Resource Structure

```
typedef struct https_rest_resource {
    const  CPU_CHAR                 *PatternPtr;      /* Access path to the resource ending with an EOF char. */
    const  HTTP_HDR_FIELD            *HTTP_Hdrs;       /* List of HTTP headers to keep.                        */
    const  CPU_SIZE_T                HTTP_HdrsNbr;    /* Number of HTTP headers in the list.                  */
    const  HTTPs_REST_METHOD_HOOKS   MethodHooks;     /* Hooks Configuration for the resource.                */
} HTTPs_REST_RESOURCE;
```

Hooks Configuration

Each REST resource has its own set of hook functions. Like mentioned before, there is one hook per HTTP method, but the hook pointer can be set to DEF_NULL in the configuration if not needed for the given resource.

Listing - REST Hooks Configuration Structure

```
typedef struct https_rest_method_hooks {
    HTTPs_REST_HOOK_FNCT  Delete;
    HTTPs_REST_HOOK_FNCT  Get;
    HTTPs_REST_HOOK_FNCT  Head;
    HTTPs_REST_HOOK_FNCT  Post;
    HTTPs_REST_HOOK_FNCT  Put;
} HTTPs_REST_METHOD_HOOKS;
```

Below is the prototype for REST hook functions:

Listing - Definition of HTTPs_REST_HOOK_FNCT

```
typedef HTTPs_REST_HOOK_STATE  (*HTTPs_REST_HOOK_FNCT)  (const  HTTPs_REST_RESOURCE      *p_resource,
                                                          const  HTTPs_REST_MATCHED_URI  *p_uri,
                                                          const  HTTPs_REST_STATE         state,
                                                                 void                   **p_data,
                                                          const  HTTPs_INSTANCE          *p_instance,
                                                                 HTTPs_CONN              *p_conn,
                                                                 void                    *p_buf,
                                                          const  CPU_SIZE_T               buf_size,
                                                                 CPU_SIZE_T              *p_buf_size_used);
```

Each hook is map on multiple hooks of the server core or Control Layer, therefore the same REST hook will be called many times during an HTTP transaction processing. To differentiate at which moment the REST hook is being called, a state variable HTTPs_REST_STATE is passed to the hook function. The table below presents a mapping of the HTTPS_REST_STATE with the corresponding HTTP server hooks.

REST STATE	Hook Function in REST Module	HTTP Server Hook	Control Layer
HTTPs_REST_STATE_INIT	HTTPsREST_Authenticate()	(HTTPs_HOOK_CFG *)->OnReqHook()	(HTTPs_CTRL *)->OnReq()
HTTPs_REST_STATE_RX	HTTPsREST_RxBody()HTTPsREST_ReqRdySignal()	(HTTPs_HOOK_CFG *)->OnReqBodyRxHook() (HTTPs_HOOK_CFG *)->OnReqRdySignalHook() with argument buf_size == 0	(HTTPs_CTRL *)->OnReqBo (HTTPs_CTRL *)->OnReqSig
HTTPs_REST_STATE_TX	HTTPsREST_GetChunk()	(HTTPs_HOOK_CFG *)->OnRespChunkHook()	(HTTPs_CTRL *)->OnRespC
HTTPs_REST_STATE_CLOSE	HTTPsREST_OnTransComplete()HTTPsREST_OnConnClosed()	(HTTPs_HOOK_CFG *)->OnTransCompleteHook() (HTTPs_HOOK_CFG *)->OnConnCloseHook()	(HTTPs_CTRL *)->OnTransC (HTTPs_CTRL *)->OnConnC

URI Pattern Matching

The URI pattern matching of this framework is quite primitive and shines by its simplicity. The URI is divided into two sections: the separator and the path component. This leads to the following rules:

- The pattern string must start with a separator.
- The pattern string must end with a '\0'
- A wild card can only be used between a separator and another separator or at the end of the pattern string.
- A wild card will not match a separator char unless the wild card is the last component of the pattern string.

URI Path Separator

The separator within the pattern string is exactly the same that must be found according to rfc3986 : '/'. This character cannot be escaped and can't be used in any other context.

URI Path Component

A URI path component is the element between two path separator or between a path separator and the end of the pattern string. This element can either be a wild card or text.

Wild Cards

The wild card begins with a { and ends with a }. A wild card cannot contain another wild card. The element between the curly brackets is used as the wild card token key.

URI Pattern and Match Examples

REST pattern	Valid?	HTTP uri	Matches?	wild card(s)
/{foo}	yes	/	yes	foo=""
		/foo/bar	yes	foo="foo/bar"
/{foo/{bar}}	no			
/{foo}/bar	yes	/	no	
		/foo/bar	yes	foo="foo"
		/foo/baz/bar	no	
/{foo/baz}/bar	yes	/	no	
		/foo/bar	yes	foo/baz="foo"
		/foo/baz/bar	no	
/{foo}/{bar}	yes	/	yes	foo="" bar=""
		/foo	yes	foo="foo" bar=""
		/foo/bar	yes	foo="foo" bar="bar"
		/foo/bar/baz	yes	foo="foo" bar="bar/baz"
/{foo}/bar/{baz}	yes	/	no	
		/bar	no	
		/foo/bar	yes	foo="foo" buz=""
		/foo/bar/baz/qux	yes	foo="foo" buz="buz/qux"
/foo/bar	yes	/	no	
		/foo	no	
		/foo/bar	yes	
		/foo/bar/buz	no	
		/buz/bar	no	
\foo	no			

Publish REST Resources

Once a resource has been created, in order to make a resource accessible through the REST Framework, two steps must be followed. The first one is to call the HTTPsREST_Publish() function which will add the resource to the specified list. Then that list of resources is used in the configuration of the HTTP server.

> This function can only be called after the HTTP server initialization and before the HTTP server start. Also, the HTTPsREST_Init() function must be bound to the HTTP server or to the Control layer prior the initialization. Otherwise, the REST memory pools won't be initialized and the resource lists cannot be created.

REST Configuration

No matter which layer REST is bound to, the HTTP Server core or the Control Layer, you need to provide a HTTPs_REST_CFG structure. That structure is used to configure the REST framework behavior for that specific instance.

Listing - REST Configuration Structure

```
typedef struct https_rest_cfg {
    const CPU_INT32U  ListID;
} HTTPs_REST_CFG;
```

Listing - Server Instance Configuration Example for REST Application

```
/*
*********************************************************************************************************
*                                  REST HOOKS CONFIGURATION
*********************************************************************************************************
*/

const  HTTPs_REST_CFG HTTPs_REST_Cfg = { 0 };      /* 0 is the list ID number. The same used in  */
                                           /* the resources publishing.                */

const  HTTPs_HOOK_CFG HTTPs_REST_HookCfg = {
       HTTPsREST_Init,
       HTTPsREST_RxHeader,
       HTTPsREST_Authenticate,
       HTTPsREST_RxBody,
       HTTPsREST_ReqRdySignal,
       DEF_NULL,
       DEF_NULL,
       DEF_NULL,
       HTTPsREST_GetChunk,
       HTTPsREST_OnTransComplete,
       DEF_NULL,
       DEF_NULL,
       HTTPsREST_OnConnClosed
};

/*
*********************************************************************************************************
*                          HTTP SERVER INSTANCE CONFIGURATION
*********************************************************************************************************
*/

const  HTTPs_CFG  AppEx_HTTPsInstanceCfg = {

  ...

/*
*--------------------------------------------------------------------------------
*                                 HOOK CONFIGURATION
*--------------------------------------------------------------------------------
*/

  &HTTPs_REST_HookCfg,                    /* .HooksPtr     */

  &HTTPs_REST_Cfg,               /* .Hooks_CfgPtr */

   ...
};
```

The REST Hook Configuration variable HTTPs_REST_HookCfg and the variable HTTPs_REST_Cfg are given in a template file inside the REST module source code (http_server_rest_hook_cfg.c/h). You can use it has-is for your REST application or you can make your own if you don't need all the hooks to be defined. In the above example, the REST module is used directly on top of the HTTP server core, if you want to use it with the Control Layer, a template Control Layer hooks configuration for REST is also given inside the Control Layer add-on module source code (http_server_ctrl_layer_rest_cfg.c/h).

How to Use REST Module

1. Create your REST resource(s) in your application.
2. Define the necessary hook functions for each HTTP method of each resource and set the hook configuration inside the REST resource declaration.
3. Set the REST configuration for the server instance configuration or the Control Layer configuration.
4. Use the API function HTTPsREST_Publish() between the HTTPs_InstanceInit() and HTTPs_InstanceStart() to publish all your REST resources inside the global resources list.

Example

See the complete REST application available in the Example directory (see section HTTP Server Example Applications for more details).

HTTP Server Protocol Recognized Fields

This section regroups lists of all the HTTP fields recognized by Micrium OS HTTP Server module.

- HTTP Versions
- HTTP Methods
- HTTP Header Fields
- HTTP Content Types
- HTTP Status Codes

HTTP Versions

HTTP Version	Micrium Type
0.9	HTTP_PROTOCOL_VER_0_9
1.0	HTTP_PROTOCOL_VER_1_0
1.1	HTTP_PROTOCOL_VER_1_1

HTTP Methods

HTTP Method	Micrium Type
CONNECT	HTTP_METHOD_CONNECT
DELETE	HTTP_METHOD_DELETE
GET	HTTP_METHOD_GET
HEAD	HTTP_METHOD_HEAD
OPTIONS	HTTP_METHOD_OPTIONS
POST	HTTP_METHOD_POST
PUT	HTTP_METHOD_PUT
TRACE	HTTP_METHOD_TRACE

Those are the methods recognized by the HTTP Server and Client stacks. It doesn't mean that all those methods are supported.

HTTP Header Fields

Header Field Type	Micrium Type
Content-Type	HTTP_HDR_FIELD_CONTENT_TYPE
Content-Length	HTTP_HDR_FIELD_CONTENT_LEN
Content-Disposition	HTTP_HDR_FILED_CONTENT_DISPOSITION
Host	HTTP_HDR_FIELD_HOST
Location	HTTP_HDR_FIELD_LOCATION
Connection	HTTP_HDR_FIELD_CONN
Transfer-Encoding	HTTP_HDR_FIELD_TRANSFER_ENCODING
Accept	HTTP_HDR_FIELD_ACCEPT

Header Field Type	Micrium Type
Accept-Charset	HTTP_HDR_FIELD_ACCEPT_CHARSET
Accept-Encoding	HTTP_HDR_FIELD_ACCEPT_ENCODING
Accept-Language	HTTP_HDR_FIELD_ACCEPT_LANGUAGE
Accept-Ranges	HTTP_HDR_FIELD_ACCEPT_RANGES
Age	HTTP_HDR_FIELD_AGE
Allow	HTTP_HDR_FIELD_ALLOW
Authorization	HTTP_HDR_FIELD_AUTHORIZATION
Content-Encoding	HTTP_HDR_FIELD_CONTENT_ENCODING
Content-Language	HTTP_HDR_FIELD_CONTENT_LANGUAGE
Content-Location	HTTP_HDR_FIELD_CONTENT_LOCATION
Content-MD5	HTTP_HDR_FIELD_CONTENT_MD5
Content-Range	HTTP_HDR_FIELD_CONTENT_RANGE
Cookie	HTTP_HDR_FIELD_COOKIE
Cookie2	HTTP_HDR_FIELD_COOKIE2
Date	HTTP_HDR_FIELD_DATE
ETag	HTTP_HDR_FEILD_ETAG
Expect	HTTP_HDR_FIELD_EXPECT
Expires	HTTP_HDR_FIELD_EXPIRES
From	HTTP_HDR_FIELD_FROM
If-Modified-Since	HTTP_HDR_FIELD_IF_MODIFIED_SINCE
If-Match	HTTP_HDR_FIELD_IF_MATCH
If-None-Match	HTTP_HDR_FIELD_IF_NONE_MATCH
If-Range	HTTP_HDR_FIELD_IF_RANGE
If-Unmodified-Since	HTTP_HDR_FIELD_IF_UNMODIFIED_SINCE
Last-Modified	HTTP_HDR_FIELD_LAST_MODIFIED
Range	HTTP_HDR_FIELD_RANGE
Referrer	HTTP_HDR_FIELD_REFERER
Retry-After	HTTP_HDR_FIELD_RETRY_AFTER
Server	HTTP_HDR_FIELD_SERVER
Set-Cookie	HTTP_HDR_FIELD_SET_COOKIE
Set-Cookie2	HTTP_HDR_FIELD_SET_COOKIE2
TE	HTTP_HDR_FIELD_TE
Trailer	HTTP_HDR_FIELD_TRAILER
Upgrade	HTTP_HDR_FIELD_UPGRATE
User-Agent	HTTP_HDR_FIELD_USER_AGENT
Vary	HTTP_HDR_FIELD_VARY
Via	HTTP_HDR_FIELD_VIA
Warning	HTTP_HDR_FIELD_WARNING
WWW-Authenticate	HTTP_HDR_FIELD_WWW_AUTHENTICATE

If a header is missing for your application, contact your sales representative so that it can be included in subsequent release of Micrium OS.

HTTP Content Types

MIME Content Type	File Extension	Micrium Type
text/html	.html	HTTP_CONTENT_TYPE_HTML
application/octet-stream	.bin	HTTP_CONTENT_TYPE_OCTET_STREAM
application/pdf	.pdf	HTTP_CONTENT_TYPE_PDF
application/zip	.zip	HTTP_CONTENT_TYPE_ZIP
image/gif	.gif	HTTP_CONTENT_TYPE_GIF
image/jpeg	.jpeg, .jpg	HTTP_CONTENT_TYPE_JPEG
image/png	.png	HTTP_CONTENT_TYPE_PNG
application/javascript	.js	HTTP_CONTENT_TYPE_JS
text/plain	.txt	HTTP_CONTENT_TYPE_PLAIN
text/css	.css	HTTP_CONTENT_TYPE_CSS
application/json	.json	HTTP_CONTENT_TYPE_JSON
application/x-www-form-urlencoded	-	HTTP_CONTENT_TYPE_APP_FORM
multipart/form-data	-	HTTP_CONTENT_TYPE_MULTIPART_FORM

If a Content Type is missing for your application, contact your sales representative so that it can be included in subsequent release of Micrium OS.

HTTP Status Codes

HTTP Status Code	Micrium Type
200	HTTP_STATUS_OK
201	HTTP_STATUS_CREATED
202	HTTP_STATUS_ACCEPTED
204	HTTP_STATUS_NO_CONTENT
205	HTTP_STATUS_RESET_CONTENT
301	HTTP_STATUS_MOVED_PERMANENTLY
302	HTTP_STATUS_FOUND
303	HTTP_STATUS_SEE_OTHER
304	HTTP_STATUS_NOT_MODIFIED
305	HTTP_STATUS_USE_PROXY
307	HTTP_STATUS_TEMPORARY_REDIRECT
400	HTTP_STATUS_BAD_REQUEST
401	HTTP_STATUS_UNAUTHORIZED
403	HTTP_STATUS_FORBIDDEN
404	HTTP_STATUS_NOT_FOUND
405	HTTP_STATUS_METHOD_NOT_ALLOWED
406	HTTP_STATUS_NOT_ACCEPTABLE
408	HTTP_STATUS_REQUEST_TIMEOUT
409	HTTP_STATUS_CONFLICT
410	HTTP_STATUS_GONE

HTTP Status Code	Micrium Type
411	HTTP_STATUS_LENGTH_REQUIRED
412	HTTP_STATUS_PRECONDITION_FAILED
413	HTTP_STATUS_REQUEST_ENTITY_TOO_LARGE
414	HTTP_STATUS_REQUEST_URI_TOO_LONG
415	HTTP_STATUS_UNSUPPORTED_MEDIA_TYPE
416	HTTP_STATUS_REQUESTED_RANGE_NOT_SATISFIABLE
417	HTTP_STATUS_EXPECTATION_FAILED
500	HTTP_STATUS_INTERNAL_SERVER_ERR
501	HTTP_STATUS_NOT_IMPLEMENTED
503	HTTP_STATUS_SERVICE_UNAVAILABLE
505	HTTP_STATUS_HTTP_VERSION_NOT_SUPPORTED

If a Status Code is missing for your application, contact your sales representative so that it can be included in subsequent release of Micrium OS.

MQTT Client Module

The Micrium OS MQTT Client allows your application to access MQTT servers and perform actions on resources stored on a server.

In an embedded environment, an MQTT client is used to publish values such as sensor data to a topic, and receive messages by subscribing to topics.

- MQTT Client Overview
- MQTT Client Example Applications
- MQTT Client Configuration
- MQTT Client Programming Guide
  - MQTT Client Objects
  - MQTT Client Connection Object Setup
  - MQTT Client Message Object Setup
  - MQTT Client Callback Functions
    - MQTT Client - On Complete Callback Type
    - MQTT Client - On Error Callback Type
    - MQTT Client - On PUBLISH Received Callback Type
  - MQTT Client Message Types Supported

MQTT Client Overview

The MQTT Client module allows your application to access MQTT brokers (also know as MQTT servers). Your application can publish messages and subscribe to topics of interest so that it can receive messages from other MQTT clients.

The MQTT Client module has been designed to have a very small memory footprint and be simple to use. It provides an API that is similar to other Micrium products, allowing you to start developing quickly.

The MQTT-client supports:

- Multiple messages on same or different topics at the same time
- Multiple connections to multiple servers
- Every type of messages defined by the MQTT specifications V3.1 and V3.1.1, including the three levels of QoS for PUBLISH messages

See the section MQTT-client Message Types Supported for additional details on each feature.

Memory Usage

MQTT client applications can have dramatically different memory requirements. For example, some applications will require a connection to only one server, while others may require more; some applications will prefer to allocate objects on their own stack, while others will want to use a memory pool on the heap. So to meet all of these different requirements, the MQTT client does not have its own memory module.

As a result, all the objects required by the MQTT client *must* be passed by the application. This includes a single MQTTc_MSG object that will be used to receive any incoming PUBLISH message from the server. For any message the application wishes to send to the server, an MQTTc_MSG object must also be provided. Each MQTTc_MSG object must have its own buffer, with a size varying from a few bytes for very simple messages to a few of hundred bytes for complex messages.

> It is important that every object passed by the application to the MQTT-Client stay valid for the duration of the MQTT message (until the callback is called) or until the connection is closed for the MQTTc Connection Object using that message.

Internal Task

To support multiple simultaneous connections, the MQTT Client module has a dedicated internal task. This task also allows the queuing of multiple messages on a given connection.

Auto Select Remote IP address

When the DNS Client is present, the MQTT-Client can accept a hostname string to connect to the remote server. IP address resolution will be performed by a DNS server. If the DNS server sends back IPv4 and IPv6 addresses, the first attempt will be to use the IPv6 address to connect to the remote MQTT server. If this connection fails the IPv4 address will be used instead.

When the DNS Client is not present, you must pass an IPv4 or IPv6 remote address in string form to the MQTT-Client.

With its compile-time configuration, MQTT's code footprint can be adjusted to fit your application's requirements by enabling/disabling features.

MQTT Client Example Applications

This section describes the examples application provided for the MQTT Client module of Micrium OS.

- MQTT Client Initialization Example
  - Description
  - Configuration
    - Mandatory
  - Location
  - API
  - Notes
- MQTT Client Connect Example
  - Description
  - Configuration
    - Optional
  - Location
  - API
  - Notes
- MQTT Client Publish Example
  - Description
  - Configuration
    - Optional
  - Location
  - API
  - Notes
- MQTT Client Subscribe Example
  - Description
  - Configuration

        - Optional
    - Location
    - API
    - Notes
- MQTT Client Echo Example
    - Description
    - Configuration
        - Optional
    - Location
    - API
    - Notes

MQTT Client Initialization Example

Description

This is a generic example that shows how to initialize the MQTT client module. It accomplishes the following tasks:

- Change default task's stacks size, if specified by the example configuration
- Initialize the MQTTClient module
- Change default the task's priority, if specified by the example configuration

Configuration

Mandatory

The following #define must be added in ex_description.h to allow other examples to initialize the HTTP Server module correctly:

#define	Description
EX_MQTT_CLIENT_INIT_AVAIL	Lets the upper example layer know that the Network Initialization example is present and must be called by other examples.

Location

```
/examples/net/mqtt/ex_mqtt_client_init.c

/examples/net/mqtt/ex_mqtt_client.h
```

API

API	Description
Ex_MQTT_Client_Init()	Initialize the MQTT Client stack for the example application.

Notes

None.

MQTT Client Connect Example

Description

This is a generic example for the MQTT Client module. It shows how to connect to a broker.

Configuration

Optional

The following #define can be added to ex_description.h, as described in Example Applications section, to change default configuration value used by the example:

#define	Default value	Description
EX_MQTTc_BROKER_NAME	"mqtt.micrium.com"	Specify the broker hostname
EX_MQTTc_USERNAME	"micrium"	Specify the username
EX_MQTTc_PASSWORD	"micrium"	Specify the password
EX_MQTTc_CLIENT_ID_NAME	"micrium-example-basic"	Specify another client ID
EX_MQTTc_MSG_LEN_MAX	128	Specify the message maximum length
EX_TRACE	printf(**VA_ARGS**)	Specify where to output information about the example

Location

```
/examples/net/mqtt/ex_mqtt_client_connect.c

/examples/net/mqtt/ex_mqtt_client.h
```

API

API	Description
Ex_MQTT_Client_Connect()	Connect the client to the broker.

Notes

None.

MQTT Client Publish Example

Description

This is a generic example for the MQTT Client module. It accomplishes the following tasks:

- Connect to a broker
- Publish message on a topic

Configuration

Optional

The following #define can be added to ex_description.h, as described in Example Applications section, to change default configuration value used by the example:

#define	Default value	Description
EX_MQTTc_BROKER_NAME	"mqtt.micrium.com "	Specify the broker hostname
EX_MQTTc_USERNAME	"micrium"	Specify the username
EX_MQTTc_PASSWORD	"micrium"	Specify the password
EX_MQTTc_CLIENT_ID_NAME	"micrium-example-publish"	Specify another client ID

Location

```
/examples/net/mqtt/ex_mqtt_client_publish.c

/examples/net/mqtt/ex_mqtt_client.h
```

API

API	Description
Ex_MQTT_Client_Publish()	Open a connection and publish message from the callback

Notes

None.

MQTT Client Subscribe Example

Description

This is a generic example for the MQTT Client module. It accomplishes the following tasks:

- Connect to a broker
- Subscribe to a topic

Configuration

Optional

The following #define can be added to ex_description.h, as described in Example Applications section, to change default configuration value used by the example:

#define	Default value	Description
EX_MQTTc_BROKER_NAME	"mqtt.micrium.com "	Specify the broker hostname
EX_MQTTc_USERNAME	"micrium"	Specify the username
EX_MQTTc_PASSWORD	"micrium"	Specify the password
EX_MQTTc_CLIENT_ID_NAME	"micrium-example-sub"	Specify another client ID

Location

```
/examples/net/mqtt/ex_mqtt_client_subscribe.c

/examples/net/mqtt/ex_mqtt_client.h
```

API

API	Description
Ex_MQTT_Client_Subscribe()	Open a connection and subscribe to a topic.

Notes

None.

MQTT Client Echo Example

Description

This is a generic example for the MQTT Client module. It accomplishes the following tasks:

- Connect to a broker
- Subscribe to a topic
- Publish on the subscribed topic

Configuration

Optional

The following #define can be added to ex_description.h, as described in Example Applications section, to change default configuration value used by the example:

#define	Default value	Description
EX_MQTTc_BROKER_NAME	"mqtt.micrium.com "	Specify the broker hostname
EX_MQTTc_USERNAME	"micrium"	Specify the username
EX_MQTTc_PASSWORD	"micrium"	Specify the password
EX_MQTTc_CLIENT_ID_NAME	"micrium-example-echo"	Specify another client ID

Location

```
/examples/net/mqtt/ex_mqtt_client_echo.c

/examples/net/mqtt/ex_mqtt_client.h
```

API

API	Description
Ex_MQTT_Client_Echo()	Open a connection and publish message from the callback and receive message using subscribe.

Notes

None.

MQTT Client Configuration

This section describes the runtime configurations for the MQTT Client module.

- Initialization
- Optional Configurations
- Post-Init Configurations

Initialization

To initialize the MQTT Client module, you call the function MQTTc_Init(). This function takes no configuration argument. Unless you override an optional configuration before calling the function MQTTc_Init(), the default configurations will be used.

Optional Configurations

This section describes the optional configurations. If you do not set them in your application, the default configurations will apply.

The default values can be retrieved via the structure MQTTc_InitCfgDflt.

> Note that these configurations must be set *before* you call the function MQTTc_Init().

Table - MQTT Client Optional Configurations

Configurations	Description	Type	Function to call	Default	Field from default configuration structure
Task's stack	The MQTT client module has a dedicated task. This configuration allows you to set the stack pointer and the stack size (in quantity of elements).	CPU_STK_SIZE void *	MQTTc_ConfigureTaskStk()	A stack of 512 elements allocated on Common 's memory segment.	.StkSizeElements.StkPtr
Memory segment	The MQTT module allocates control data. You can specify the memory segment where such data should be located.	MEM_SEG *	MQTTc_ConfigureMemSeg()	General-purpose heap .	.MemSegPtr
Quantity parameters	The MQTT client task use a message queue. You can overwrite the maximum element in the message pool.	MQTTc_QTY_CFG	MQTTc_ConfigureQty()	Unlimited queue size.	.QtyCfg

Post-Init Configurations

This section describes the configurations that can be set at any time during execution *after* you called the function MQTTc_Init().

These configurations are optional. If you do not set them in your application, the default configurations will apply.

Table - HTTP Client Post-init Configurations

Configurations	Description	Type	Function to call	Default
Task priority	The MQTT client module creates a task. You can change the priority of this task at any time.	RTOS_TASK_PRIO	MQTTc_TaskPrioSet()	See Appendix A - Internal Tasks .
Task Delay	You can change the delay inside this task to allow a period of time for other tasks to run.	CPU_INT08U	MQTTc_TaskDlySet()	1u
Default Inactivity timeout	You can change the default inactivity assigned automatically to new connections.	CPU_INT16U	MQTTc_InactivityTimeoutDfltSet()	1800u

MQTT Client Programming Guide

This section includes the following sub-topics:

- MQTT Client Objects

MQTT Client Connection Object Setup
- MQTT Client Message Object Setup
- MQTT Client Callback Functions
- MQTT Client Message Types Supported

Include Files

To include MQTT functions in your application, include this file:

Include file	Description
rtos/net/include/mqtt_client.h	Functions used for MQTT API.

Configuration

Some parameters should be configured and/or optimized for your project requirements. See the section MQTT Client Configuration for further details.

API Reference

Function name	Description
MQTTc_ConnClr()	Clears an MQTT client connection before its first usage.
MQTTc_ConnSetParam()	Sets parameters related to the TCP and MQTT client connection.
MQTTc_ConnOpen()	Opens a new MQTT client connection.
MQTTc_ConnClose()	Request to close a MQTT client connection.
MQTTc_MsgClr()	Clears the Message object members.
MQTTc_MsgSetParam()	Sets parameters related to a given MQTT message.
MQTTc_Connect()	Sends a 'Connect' message to the MQTT broker.
MQTTc_Publish()	Sends a 'Publish' message to the MQTT broker.
MQTTc_Subscribe()	Sends a 'Subscribe' message to the MQTT broker.
MQTTc_SubscribeMult()	Sends a 'Subscribe' message containing multiple topics to MQTT broker.
MQTTc_Unsubscribe()	Sends an 'Unsubscribe' message to the MQTT broker.
MQTTc_UnsubscribeMult()	Sends an 'Unsubscribe' message for multiple topics to the MQTT broker.
MQTTc_PingReq()	Sends a 'PingReq' message to the MQTT broker.
MQTTc_Disconnect()	Sends a 'Disconnect' message to the MQTT broker.

MQTT Client Objects

This section describes the various structures that your application can use to create objects in order to use the MQTT Client. All MQTT Client objects must be allocated by the application and passed to the MQTT Client module.

All objects or strings passed to the MQTT Client module must stay valid and unmodified for the duration of the MQTT message (for message-oriented parameters and objects), or until the MQTT connection is closed (for Connection-oriented parameters and objects).

MQTT-Client Connection (MQTTc_CONN)

This is the main structure used by MQTT Client module. It contains all parameters related to an MQTT connection (broker port number, broker address, broker host name, etc.) and also many internal parameters for the MQTT connection and MQTT message processing.

Each configurable parameter *should* be set up with the function MQTTc_ConnSetParam(). The list of available parameters for a connection can be viewed here .

Your application should never directly modify the members of an MQTT Client Connection object at any time.

Listing - MQTTc_CONN Structure

```
struct  mqttc_conn {
    NET_SOCK_ID                SockId;               /* Connection's socket ID.                          */
    CPU_INT08U                 SockSelFlags;         /* Flags to identify which oper must be checked in Sel. */
    CPU_CHAR                  *BrokerIP_Addr;        /* MQTT broker's IP addr.                           */
    CPU_CHAR                  *BrokerNamePtr;        /* MQTT broker's name.                              */
    CPU_INT16U                 BrokerPortNbr;        /* MQTT broker's port nbr.                          */
    CPU_INT16U                 InactivityTimeout_s;  /* Inactivity timeout, in seconds.                  */
    CPU_CHAR                  *ClientID_Str;         /* Client ID str.                                   */
    CPU_CHAR                  *UsernameStr;          /* Username str.                                    */
    CPU_CHAR                  *PasswordStr;          /* Password str.                                    */
    CPU_INT16U                 KeepAliveTimerSec;    /* Keep alive timer duration, in seconds.           */
    MQTTc_WILL_CFG            *WillCfgPtr;           /* Ptr to will cfg, if any.                         */
    NET_APP_SOCK_SECURE_CFG   *SecureCfgPtr;         /* Ptr to secure will cfg, if any.                  */
                                                     /* -------------------- CALLBACKS --------------------- */
    MQTTc_CMPL_CALLBACK        OnCmpl;               /* Generic, on cmpl callback.                       */
    MQTTc_CMPL_CALLBACK        OnConnectCmpl;        /* On connect cmpl callback.                        */
    MQTTc_CMPL_CALLBACK        OnPublishCmpl;        /* On publish cmpl callback.                        */
    MQTTc_CMPL_CALLBACK        OnSubscribeCmpl;      /* On subscribe cmpl callback.                      */
    MQTTc_CMPL_CALLBACK        OnUnsubscribeCmpl;    /* On unsubscribe cmpl callback.                    */
    MQTTc_CMPL_CALLBACK        OnPingReqCmpl;        /* On ping req cmpl callback.                       */
    MQTTc_CMPL_CALLBACK        OnDisconnectCmpl;     /* On disconnect cmpl callback.                     */
    MQTTc_PUBLISH_RX_CALLBACK  OnPublishRx;          /* On publish rx'd cmpl callback.                   */
    void                      *ArgPtr;               /* Ptr to arg that will be provided to callbacks.   */
    CPU_INT32U                 TimeoutMs;            /* Timeout for 'Open' operation, in milliseconds.   */
                                                     /* ----------------- NEXT MSG VALUES ------------------ */
    CPU_INT08U                 NextMsgHeader;        /* Header of next msg to parse.                     */
    CPU_INT32U                 NextMsgRxLen;         /* Rx len of next msg.                              */
    MQTTc_MSG_TYPE             NextMsgType;          /* Next msg's type.                                 */
    CPU_INT32U                 NextMsgLen;           /* Len remaining to RX for next msg.                */
    CPU_BOOLEAN                NextMsgLenIsCmpl;     /* Flag indicating if next msg's len value is rx'd. */
    CPU_INT16U                 NextMsgMsgID;         /* ID of next msg, if any.                          */
    CPU_BOOLEAN                NextMsgMsgID_IsCmpl;  /* Flag indicating if next msg's ID has been rx'd.  */
    MQTTc_MSG                 *NextMsgPtr;           /* Ptr to next msg, if known.                       */
    MQTTc_MSG                 *PublishRxMsgPtr;      /* Ptr to msg that is used to RX publish from server. */
    MQTTc_MSG                 *TxMsgHeadPtr;         /* Ptr to head of msg needing to TX or waiting reply. */
    CPU_INT32U                 NextTxMsgTxLen;       /* Len of already xfer'd data.                      */
    MQTTc_CONN                *NextPtr;              /* Ptr to next conn.                                */
};
```

MQTT Client Message (MQTTc_MSG)

This structure contains parameters and flags related to the configuration of an MQTT message.

Each configurable parameter *should* be set up with the function MQTTc_MsgSetParam(). The list of available parameters for a connection can be viewed here .

Your application should never directly modify the members of an MQTT-Client Message object at any time.

Listing - MQTTc_MSG Structure

```
struct  mqttc_msg {
    MQTTc_CONN        *ConnPtr;                        /* Ptr to MQTTc_CONN associated.              */
    MQTTc_MSG_TYPE    Type;                            /* Msg's type.                                */
    MQTTc_MSG_STATE   State;                           /* Msg's state.                               */
    CPU_INT08U        QoS;                             /* Msg's QoS.                                 */
    CPU_INT16U        MsgID;                           /* Msg ID used by msg.                        */
    CPU_INT08U       *BufPtr;                          /* Ptr to RX/TX buf.                          */
    CPU_INT32U        BufLen;                          /* Avail buf len for msg.                     */
    CPU_INT32U        XferLen;                         /* Len of xfer.                               */
    MQTTc_ERR         Err;                             /* Err associated to processing of msg.       */
    MQTTc_MSG        *NextPtr;                         /* Ptr to next msg.                           */
};
```

MQTT Client Connection Object Setup

To configure the MQTT Client Connection object , the MQTT Client module provides the function MQTTc_ConnSetParam(). This function takes as arguments: the type of parameter, the parameter's value, and a pointer to the parameter. The following parameter types are available:

Generic Parameters

Table - MQTTc Connection's Parameters

Parameter Type	Description
MQTTc_PARAM_TYPE_BROKER_IP_ADDR	IP Address of the MQTT broker.*This field is ignored if broker name is set.*
MQTTc_PARAM_TYPE_BROKER_NAME	Name of the MQTT broker.*This method will work only if a DNS server is available**This field is ignored if the broker IP address is set.*
MQTTc_PARAM_TYPE_BROKER_PORT_NBR	Sets a specific port for the MQTT broker.*By default, the port is set to 1883.*
MQTTc_PARAM_TYPE_INACTIVITY_TIMEOUT_S	Sets the inactivity timeout of the connection*This value represents the time of inactivity allowed on a connection before it is automatically closed.*
MQTTc_PARAM_TYPE_CLIENT_ID_STR	Connection's Client ID. This ID should be unique among devices connected to the broker.
MQTTc_PARAM_TYPE_USERNAME_STR	Connection's username string.*This field is optional.*
MQTTc_PARAM_TYPE_PASSWORD_STR	Connection's password string.*This field is optional.*
MQTTc_PARAM_TYPE_KEEP_ALIVE_TMR_SEC	Connection's keep alive, in seconds.*This value represents the maximum allowed time between messages sent by the device to the broker.**If no message is sent for more than this time, the broker will close the connection.*
MQTTc_PARAM_TYPE_WILL_CFG_PTR	Pointer to last will and testament configuration.*This field is optional.*
MQTTc_PARAM_TYPE_SECURE_CFG_PTR	Pointer to secure configuration.*This field is optional.*
MQTTc_PARAM_TYPE_TIMEOUT_MS	Connection's timeout for socket 'Open', in milliseconds.
MQTTc_PARAM_TYPE_PUBLISH_RX_MSG_PTR	Pointer to MQTT-client message object that will be used to receive PUBLISH messages from the broker.*This parameter is* ***mandatory**.*

Callback options

Table - MQTTc Connection's Callbacks

MQTTc_PARAM_TYPE_CALLBACK_ON_COMPL	Sets the generic callback function to notify the application whenever an event occurs. This callback is called before every other callback and may be used in place of all the other ones.

MQTTc_PARAM_TYPE_CALLBACK_ON_CONNECT_CMPL	Sets the callback function to notify the application when a CONNECT message with the MQTT broker is completed.*This parameter is optional.*
MQTTc_PARAM_TYPE_CALLBACK_ON_PUBLISH_CMPL	Sets the callback function to notify the application when a PUBLISH message with the MQTT broker is completed.*This parameter is optional.*
MQTTc_PARAM_TYPE_CALLBACK_ON_SUBSCRIBE_CMPL	Sets the callback function to notify the application when a SUBSCRIBE message with the MQTT broker is completed.*This parameter is optional.*
MQTTc_PARAM_TYPE_CALLBACK_ON_UNSUBSCRIBE_CMPL	Sets the callback function to notify the application when an UNSUBSCRIBE message with the MQTT broker is completed.*This parameter is optional.*
MQTTc_PARAM_TYPE_CALLBACK_ON_PINGREQ_CMPL	Sets the callback function to notify the application when a PINGREQ message with the MQTT broker is completed.*This parameter is optional.*
MQTTc_PARAM_TYPE_CALLBACK_ON_DISCONNECT_CMPL	Sets the callback function to notify the application when a DISCONNECT message with the MQTT broker is completed.*This parameter is optional.*
MQTTc_PARAM_TYPE_CALLBACK_ON_ERR_CALLBACK	Sets the callback function to notify the application when an error with the socket connection occurs.*A connection can be closed by the MQTT broker, by the client when an unexpected error occurred. To be able to re-open the socket connection and re-connect the client, this callback should be provided. This parameter is optional.*
MQTTc_PARAM_TYPE_CALLBACK_ON_PUBLISH_RX	Sets the callback function to notify the application when a PUBLISH message is received from the broker.*This parameter is* **mandatory**.*
MQTTc_PARAM_TYPE_CALLBACK_ARG_PTR	Sets the argument that will be passed to every callback function.*This parameter is optional.*

Example

Listing - MQTTc Connection Configuration Example

```
CPU_BOOLEAN  App_MQTTc_ConnPrepare (MQTTc_CONN  *p_conn)
{
    RTOS_ERR  err;

                                              /* ---------- INIT NEW CONNECTION --------- */
    MQTTc_ConnClr(p_conn, &err);
    if (err.Code != RTOS_ERR_NONE) {
        return (DEF_FAIL);
    }
                                              /* Set conn parameters.                    */
    MQTTc_ConnSetParam(      &AppMQTTc_Conn,
                        MQTTc_PARAM_TYPE_BROKER_NAME,
                 (void *)APP_MQTTc_BROKER_NAME,
                       &err);
    if (err.Code != RTOS_ERR_NONE) {
        return (DEF_FAIL);
    }

    MQTTc_ConnSetParam(      &AppMQTTc_Conn,
                        MQTTc_PARAM_TYPE_CLIENT_ID_STR,
                 (void *)APP_MQTTc_CLIENT_ID_NAME,
                       &err);
    if (err.Code != RTOS_ERR_NONE) {
```

```
}MQTTc_ConnSetParam(&AppMQTTc_Conn,
                        MQTTc_PARAM_TYPE_USERNAME_STR,(void *)APP_MQTTc_USERNAME,&err);if(err.Code != RTOS_ERR_NONE)
{return(DEF_FAIL);}MQTTc_ConnSetParam(&AppMQTTc_Conn,
                        MQTTc_PARAM_TYPE_PASSWORD_STR,(void *)APP_MQTTc_PASSWORD,&err);if(err.Code != RTOS_ERR_NONE)
{return(DEF_FAIL);}MQTTc_ConnSetParam(&AppMQTTc_Conn,
                        MQTTc_PARAM_TYPE_KEEP_ALIVE_TMR_SEC,(void *)1000u,&err);if(err.Code != RTOS_ERR_NONE)
{return(DEF_FAIL);}MQTTc_ConnSetParam(&AppMQTTc_Conn,
                        MQTTc_PARAM_TYPE_CALLBACK_ON_COMPL,(void *)AppMQTTc_OnCmplCallbackFnct,&err);if(err.Code != RTOS_ERR_NONE)
{return(DEF_FAIL);}MQTTc_ConnSetParam(&AppMQTTc_Conn,
                        MQTTc_PARAM_TYPE_CALLBACK_ON_CONNECT_CMPL,(void *)AppMQTTc_OnConnectCmplCallbackFnct,&err);if(err.Code !=
RTOS_ERR_NONE){return(DEF_FAIL);}MQTTc_ConnSetParam(&AppMQTTc_Conn,
                        MQTTc_PARAM_TYPE_CALLBACK_ON_PUBLISH_CMPL,(void *)AppMQTTc_OnPublishCmplCallbackFnct,&err);if(err.Code !=
RTOS_ERR_NONE){return(DEF_FAIL);}MQTTc_ConnSetParam(&AppMQTTc_Conn,
                        MQTTc_PARAM_TYPE_CALLBACK_ON_SUBSCRIBE_CMPL,(void *)AppMQTTc_OnSubscribeCmplCallbackFnct,&err);if(err.Code !=
RTOS_ERR_NONE){return(DEF_FAIL);}MQTTc_ConnSetParam(&AppMQTTc_Conn,
                        MQTTc_PARAM_TYPE_CALLBACK_ON_ERR_CALLBACK,(void *)AppMQTTc_OnErrCallbackFnct,&err);if(err.Code !=
RTOS_ERR_NONE){return(DEF_FAIL);}MQTTc_ConnSetParam(&AppMQTTc_Conn,
                         MQTTc_PARAM_TYPE_PUBLISH_RX_MSG_PTR,(void *)&AppMQTTc_MsgPublishRx,&err);if(err.Code != RTOS_ERR_NONE)
{return(DEF_FAIL);}MQTTc_ConnSetParam(&AppMQTTc_Conn,
                        MQTTc_PARAM_TYPE_CALLBACK_ON_PUBLISH_RX,(void *)AppMQTTc_OnPublishRxCallbackFnct,&err);if(err.Code !=
RTOS_ERR_NONE){return(DEF_FAIL);}MQTTc_ConnSetParam(&AppMQTTc_Conn,
                        MQTTc_PARAM_TYPE_TIMEOUT_MS,(void *)30000u,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}return(DEF_OK);}
```

MQTT Client Message Object Setup

To set up an MQTT message, the MQTT Client provides the function MQTTc_MsgSetParam(). The function takes as arguments: the type of parameter, the pointer to the parameter, and the parameter's value. The following parameter types are available:

Table - MQTTc Message's Parameters

Parameter Type	Description
MQTTc_PARAM_TYPE_MSG_BUF_PTR	Sets a pointer to the message.
MQTTc_PARAM_TYPE_MSG_BUF_LEN	Sets the message's buffer length.

Example

Listing - MQTTc Message Configuration Example

```
static  MQTTc_MSG    AppMQTTc_Msg;
static  CPU_INT08U   AppMQTTc_MsgBuf[APP_MQTTc_MSG_LEN_MAX];

CPU_BOOLEAN  App_MQTTc_MsgPrepare (void)
{
    RTOS_ERR  err;

                                    /* ------------ INIT NEW MESSAGE ------------ */
    MQTTc_MsgClr(&AppMQTTc_Msg, &err);
    if (err.Code != RTOS_ERR_NONE) {
        return (DEF_FAIL);
    }

    MQTTc_MsgSetParam(&AppMQTTc_Msg, MQTTc_PARAM_TYPE_MSG_BUF_PTR, (void *)&AppMQTTc_MsgBuf[0u], &err);
    if (err.Code != RTOS_ERR_NONE) {
        return (DEF_FAIL);
    }

    MQTTc_MsgSetParam(&AppMQTTc_Msg, MQTTc_PARAM_TYPE_MSG_BUF_LEN, (void *)APP_MQTTc_MSG_LEN_MAX, &err);
    if (err.Code != RTOS_ERR_NONE) {
        return (DEF_FAIL);
    }

    return (DEF_OK);
}
```

MQTT Client Callback Functions

Callback functions are used by the MQTT Client module to notify your application when certain activities occur in the MQTT Client core.

Callbacks can be set up by using the MQTTc_ConnSetParam() function. The p_arg parameter that is passed in every callback can be set via MQTTc_ConnSetParam() by using MQTTc_PARAM_TYPE_CALLBACK_ARG_PTR as the type argument.

> **No blocking calls (delays, pending on an OS object, etc.) or long operations should *ever* be performed in any of these callbacks. These callbacks are executed from the MQTT Client internal task, and if this task is delayed or blocked, the entire MQTT Client module will be delayed/blocked too, preventing the reception or transmission of any message on any connection.**

On Complete Callbacks

The following callbacks are of type MQTTc_CMPL_CALLBACK. Even though they are optional, they are necessary for your application to know when a particular message has been completely sent, in order to re-use it for another message. The generic complete callback may be used to emulate all other callbacks in this category.

Table - MQTTc On Complete Callback Functions

Trigger Event	Description	type argument for MQTTc_ConnSetParam()
Any message completed (in this section, not a received PUBLISH message)	Generic notification for any event.	MQTTc_PARAM_TYPE_CALLBACK_ON_COMPL
CONNECT message completed	Notify that a CONNECT message has completed.	MQTTc_PARAM_TYPE_CALLBACK_ON_CONNECT_CMPL
PUBLISH (to broker) message completed	Notify that a PUBLISH message has completed.	MQTTc_PARAM_TYPE_CALLBACK_ON_PUBLISH_CMPL
SUBSCRIBE message completed	Notify that a SUBSCRIBE message has completed.	MQTTc_PARAM_TYPE_CALLBACK_ON_SUBSCRIBE_CMPL
UNSUBSCRIBE message completed	Notify that an UNSUBSCRIBE message has completed.	MQTTc_PARAM_TYPE_CALLBACK_ON_UNSUBSCRIBE_CMPL
PINGREQ message completed	Notify that a PINGREQ message has completed.	MQTTc_PARAM_TYPE_CALLBACK_ON_PINGREQ_CMPL
DISCONNECT message completed	Notify that a DISCONNECT message has completed.	MQTTc_PARAM_TYPE_CALLBACK_ON_DISCONNECT_CMPL

On Error Callback

This callback is of type MQTTc_ERR_CALLBACK. Even though it is optional, we recommend that your application make use of it to know if an error occurs.

Table - MQTTc On Error Callback Function

Trigger Event	Description	type argument for MQTTc_ConnSetParam()
Whenever an error occurs on the connection and is not related to a particular message	Notify that an error occurred on the connection.	MQTTc_PARAM_TYPE_CALLBACK_ON_ERR_CALLBACK

On PUBLISH Message Received Callback

This callback is of type MQTTc_PUBLISH_RX_CALLBACK. This callback is required only if the application expects to receive data from the broker. If not, it will never be called.

Table - MQTTc On PUBLISH received Callback Function

Trigger Event	Description	type argument for MQTTc_ConnSetParam()
A PUBLISH message has been completely received	Notify that a PUBLISH message has completed to be received.	MQTTc_PARAM_TYPE_CALLBACK_ON_PUBLISH_RX

MQTT Client - On Complete Callback Type

This callback function type is used by any callback that receives a notification when a message exchange with the broker has been completed, successfully or not. The parameters passed are the same in every callback and are detailed below.

Prototype

```
typedef void (*MQTTc_CMPL_CALLBACK) (MQTTc_CONN *p_conn,
                                     MQTTc_MSG  *p_msg,
                                     void       *p_arg,
                                     RTOS_ERR    err);
```

Arguments

`p_conn`

Pointer to the current MQTTc Connection Object.

`p_msg`

Pointer to the MQTTc Message Object used to send the message to the broker.

In the generic complete callback, you can use p_msg->Type to find out the type of message that just completed. The other fields of p_msg may not be relied upon.

`p_arg`

Pointer to the argument specified by your application via MQTTc_ConnSetParam() with the MQTTc_PARAM_TYPE_CALLBACK_ARG_PTR type.

`err`

Error code for this operation.

Return Values

None.

Notes / Warnings

None.

Example Template

Listing - CONNECT Complete Callback Example

```
static void AppMQTTc_OnConnectCmplCallbackFnct (MQTTc_CONN *p_conn,
                                                MQTTc_MSG  *p_msg,
                                                void       *p_arg,
                                                RTOS_ERR    err)
{
    RTOS_ERR  err_mqttc;

    (void)&p_arg;
```

```
if(err.Code != RTOS_ERR_NONE){printf("ConnectCmpl callback called with err (%i). NOT sending PUBLISH message.\n\r",
err);}else{printf("ConnectCmpl callback called. Sending PUBLISH message.\n\r");MQTTc_Publish(p_conn,
                p_msg,
                APP_MQTTc_DOMAIN_PUBLISH,
                APP_MQTTc_PUBLISH_TEST_QoS,
                DEF_YES,
                APP_MQTTc_PUBLISH_TEST_MSG,&err_mqttc);if(err_mqttc.Code != RTOS_ERR_NONE){printf("!!! APP ERROR !!! Failed to Publish test
string. Err: %i\n\r.", err_mqttc);}}}
```

Listing - Generic Complete Callback Example

```
static  void  AppMQTTc_OnCmplCallbackFnct (MQTTc_CONN  *p_conn,
                                          MQTTc_MSG   *p_msg,
                                          void        *p_arg,
                                          RTOS_ERR     err)
{
   (void)&p_conn;
   (void)&p_arg;

   if (err.Code != RTOS_ERR_NONE) {
      printf("Operation completed with err (%i). ", err);
   }

   switch (p_msg-\>Type) {
      case MQTTc_MSG_TYPE_CONNECT:                /\* Gen callback called for event type: Connect Cmpl.   \*/
         printf("Gen callback called for event type: Connect Cmpl.\n\r");
         break;
      case MQTTc_MSG_TYPE_PUBLISH:                /\* Gen callback called for event type: Publish Cmpl.   \*/
         printf("Gen callback called for event type: Publish Cmpl.\n\r");
         break;
      case MQTTc_MSG_TYPE_SUBSCRIBE:              /\* Gen callback called for event type: Subscribe Cmpl.  \*/
         printf("Gen callback called for event type: Subscribe Cmpl.\n\r");
         break;
      case MQTTc_MSG_TYPE_UNSUBSCRIBE:            /\* Gen callback called for event type: Unsubscribe Cmpl.\*/
         printf("Gen callback called for event type: Unsubscribe Cmpl.\n\r");
         break;
      case MQTTc_MSG_TYPE_PINGREQ:                /\* Gen callback called for event type: PingReq Cmpl.   \*/
         printf("Gen callback called for event type: PingReq Cmpl.\n\r");
         break;
      case MQTTc_MSG_TYPE_DISCONNECT:             /\* Gen callback called for event type: Disconnect Cmpl. \*/
         printf("Gen callback called for event type: Disconnect Cmpl.\n\r");
         break;
      default:
         printf("Gen callback called for event type: default. !!! ERROR !!! %i\n\r", p_msg-\>Type);
         break;
   }
}
```

MQTT Client - On Error Callback Type

This callback function type is used to notify your application that an error occurred within the stack that is not related to a particular message.

Prototype

```
typedef  void  (*MQTTc_ERR_CALLBACK) (MQTTc_CONN  *p_conn,
                                      void        *p_arg,
                                      RTOS_ERR     err);
```

Arguments

`p_conn`

Pointer to the current MQTTc Connection Object.

`p_arg`

Pointer to the argument specified by your application via MQTTc_ConnSetParam() with the MQTTc_PARAM_TYPE_CALLBACK_ARG_PTR type.

`err`

Error code.

Return Values

None.

Notes / Warnings

After this callback is called, your application must re-call MQTTc_ConnOpen() , and MQTTc_Connect(), since both the TCP and MQTT connections will have been closed. We recommend that you do this outside the scope of the callback, to avoid slowing the whole MQTTc stack if it has other messages to process.

Example Template

Listing - On Error Callback Example

```
static void AppMQTTc_OnErrCallbackFnct (MQTTc_CONN  *p_conn,
                                      void        *p_arg,
                                      RTOS_ERR     err)
{
    (void)&p_conn;
    (void)&p_arg;

    printf("!!! APP ERROR !!! Err detected via OnErr callback. Err = %s.\n\r", err.CodeText);
}
```

MQTT Client - On PUBLISH Received Callback Type

This callback function type is used to notify your application that a PUBLISH message has been received or that there has been an error when trying to receive it.

Prototype

```
typedef void (*MQTTc_PUBLISH_RX_CALLBACK) (      MQTTc_CONN  *p_conn,
                                          const  CPU_CHAR    *topic_name_str,
                                                 CPU_INT32U   topic_len,
                                          const  CPU_INT08U  *p_message,
                                                 CPU_INT32U   message_len,
                                                 void        *p_arg,
                                                 RTOS_ERR     err);
```

Arguments

`p_conn`

Pointer to the current MQTTc Connection Object.

`topic_name_str`

Pointer to the start of the string containing the topic name. Note: For performance reasons, this string is *not* NULL-terminated.

`topic_len`

Length, in bytes, of the topic string.

`p_message`

Pointer to the start of the buffer containing the message.

`message_len`

Length, in bytes, of the message.

`p_arg`

Pointer to the argument specified by the application via MQTTc_ConnSetParam() with the MQTTc_PARAM_TYPE_CALLBACK_ARG_PTR type.

`err`

Error code for this operation.

Return Values

None.

Notes / Warnings

None.

Example Template

Listing - On PUBLISH Received Callback Example(1) This example assumes that the message is string-based and NULL terminated

```
static void AppMQTTc_OnPublishRxCallbackFnct (      MQTTc_CONN  *p_conn,
                                             const CPU_CHAR    *topic_name_str,
                                                   CPU_INT32U   topic_len,
                                             const CPU_INT08U  *p_message,
                                                   CPU_INT32U   message_len,
                                                   void        *p_arg,
                                                   RTOS_ERR     err)
{
    (void)&p_conn;
    (void)&p_arg;

    if (err.Code != RTOS_ERR_NONE) {
        printf("!!! APP ERROR !!! Err detected when receiving a PUBLISH message (%s).\n\r", err.CodeText);
    }

    printf("Received PUBLISH message from server. Topic is %.*s.", topic_len, topic_name_str);
    printf(" Message is %s.", (CPU_CHAR *)p_message);                              /* See Note #1. */
}
```

MQTT Client Message Types Supported

MQTT Message Type	Associated Function	Associated Callback Type (set via MQTTc_ConnSetParam())
CONNECTCONNACK	MQTTc_Connect()	MQTTc_PARAM_TYPE_CALLBACK_ON_CONNECT_CMPL
PUBLISH (to server)PUBACKPUBRECPUBRELPUBCOMP	MQTTc_Publish()	MQTTc_PARAM_TYPE_CALLBACK_ON_PUBLISH_CMPL
PUBLISH (from server)PUBACKPUBRECPUBRELPUBCOMP	None, waiting on server.	MQTTc_PARAM_TYPE_CALLBACK_ON_PUBLISH_RX

MQTT Message Type	Associated Function	Associated Callback Type (set via MQTTc_ConnSetParam())
SUBSCRIBESUBACK	MQTTc_Subscribe(),MQTTc_SubscribeMult()	MQTTc_PARAM_TYPE_CALLBACK_ON_SUBSCRIBE_CMP
UNSUBSCRIBEUNSUBACK	MQTTc_Unsubscribe(),MQTTc_UnsubscribeMult()	MQTTc_PARAM_TYPE_CALLBACK_ON_UNSUBSCRIBE_C
PINGREQPINGRESP	MQTTc_PingReq()	MQTTc_PARAM_TYPE_CALLBACK_ON_PINGREQ_CMPL
DISCONNECT	MQTTc_Disconnect()	MQTTc_PARAM_TYPE_CALLBACK_ON_DISCONNECT_CM

SNTP Client Module

- SNTP Client Overview
- SNTP Client Example Applications
- SNTP Client Configuration
- SNTP Client Programming Guide

SNTP Client Overview

The Simple Network Time Protocol (SNTP) is a protocol used to synchronized computer clock times in a computer network. SNTP uses Universal Time Coordinated (UTC) to synchronize computer clocks to the millisecond, and sometimes to a fraction of a millisecond. The SNTP Client module implements the following RFC:

ftp-ftp-rfc-editor-org-in-notes-rfc4330-txt

SNTP Client Example Applications

This section describes the examples that are related to the Micrium OS SNTP Client module.

- SNTP Client Initialization Example
  - Description
  - Configuration
    - Mandatory
    - Optional
  - Location
  - API
  - Notes
- Basic SNTP Current Time Retrieve
  - Description
  - Location
  - Configuration
    - Optional
  - API
  - Notes

SNTP Client Initialization Example

Description

This is a generic example that shows how to initialize the SNTP client module.

Configuration

Mandatory

The following #define must be added in ex_description.h to allow other examples to initialize the SNTP Client module correctly:

#define	Description
EX_SNTP_CLIENT_INIT_AVAIL	Lets the upper example layer know that the SNTP Client Initialization example is present and must be called by other examples.

Optional

The following #define can be added to ex_description.h, as described in Example Applications section, to change default configuration value used by the example:

#define	Default value	Description
EX_TRACE	printf(**VA_ARGS**)	Specify the function used to output information

Location

```
/examples/net/sntp/ex_sntp_client.c

/examples/net/sntp/ex_sntp_client.h
```

API

API	Description
Ex_SNTP_Client_Init()	Initialize the Micrium OS SNTP Client module for the example application.

Notes

None.

Basic SNTP Current Time Retrieve

Description

This is a simple example that will retrieve the current time from the following NTP server: *0.pool.ntp.org*.

The example will then display the current time in the following format:

```
Day <day> of year <year>. Time: hh:mm:ss(UTC)
```

Location

```
/examples/net/sntp/ex_sntp_client.c

/examples/net/sntp/ex_sntp_client.h
```

Configuration

Optional

The following #define can be added to ex_description.h, as described in Example Applications section, to change default configuration value used by the example:

#define	Default value	Description
EX_TRACE	printf(**VA_ARGS**)	Specify the function used to output information

API

API	Description
Ex_SNTP_Client_CurTimeDisplay()	Retrieves the current time from the NTP server and displays it

Notes

None.

SNTP Client Configuration

Initialization

Initializing the SNTP Client module is done by calling the function SNTPc_Init(). This function takes no configuration argument. Unless you override an optional configuration before calling the function SNTPc_Init(), the default configurations will be used.

Optional Configurations

This section describes the configurations that are optional. If you do not set them in your application, the default configurations will apply.

You can retrieve the default values from the structure SNTPc_InitCfgDflt.

> Note that these configurations must be set *before* you call the function SNTPc_Init().

Table - SNTP Client Optional Configurations

Configurations	Description	Type	Function to call	Default	Field from default configuration structure
Memory segment	This module allocates some control data. You can specify the memory segment from where such data should be allocated.	MEM_SEG*	SNTPc_ConfigureMemSeg()	General-purpose heap .	.MemSegPtr

Post-Init Configurations

This section describes the configurations that can be set at any time during execution *after* you called the function SNTPc_Init().

These configurations are optional. If you do not set them in your application, the default configurations will apply.

Table - SNTP Client Post-init Configurations

Configurations	Description	Function to call	Default	
Default SNTP request configurations	When issuing a request to retrieve the remote time using the function `SNTPc_ReqRemoteTime()`, the port number, address family and reception timeout will be determined from default values. These default values can be overridden.	`SNTPc_DfltCfgSet()`	ServerPortNbrDflt	123
			ServerAddrFamilyDflt	None
			ReqRxTimeout_msDflt	5000ms

SNTP Client Programming Guide

This section describes the basic steps required to initialize and use the Simple Network Time Protocol (SNTP) module.

- Initializing the SNTP Client Module
- Requesting Remote Time
- Retrieving Current Time

Initializing the SNTP Client Module

First step is to initialize the SNTP client module. This is done by calling the function SNTPc_Init(). Note that the Network module MUST be initialized before the SNTP Client module.

Listing - Example of call to SNTPc_Init() in the *SNTP Client Programming Guide* page gives an example of call to SNTPc_Init().

Listing - Example of call to SNTPc_Init()

```
RTOS_ERR  err;

SNTPc_Init(&err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

Requesting Remote Time

Once you successfully initialized the SNTP Client module, you can request the remote time This is done by calling the function SNTPc_ReqRemoteTime().

Listing - Example of call to SNTPc_ReqRemoteTime() in the *SNTP Client Programming Guide* page gives an example of call to SNTPc_ReqRemoteTime() using default arguments. In this example, the remote NTP server is *0.pool.ntp.org*.

Listing - Example of call to SNTPc_ReqRemoteTime()

```
RTOS_ERR  err;
SNTP_PKT  pkt;

SNTPc_ReqRemoteTime("0.pool.ntp.org",
                    &pkt,
                    &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

Retrieving Current Time

Once you successfully requested the remote time, you can compute the current time. This is done by calling the function SNTPc_GetRemoteTime(). This function will take the pointer to the variable of type SNTP_PKT that was given to the function SNTPc_ReqRemoteTime() previously. Note that the returned time is UTC.

Listing - Example of call to SNTPc_GetRemoteTime() in the *SNTP Client Programming Guide* page gives an example of call to SNTPc_GetRemoteTime().

Listing - Example of call to SNTPc_GetRemoteTime()

```
SNTP_TS   ts;
RTOS_ERR  err;

ts = SNTPc_GetRemoteTime(&pkt, &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}

ts.Sec;          /* Represents the quantity of seconds since Jan 1, 1900.  */
ts.Frac;         /* Represents the fractions of second. 32 bit precision.  */
```

SMTP Client Module

- SMTP Client Overview
- SMTP Client Example Applications
- SMTP Client Configuration
- SMTP Client Programming Guide

SMTP Client Overview

SMTP (Simple Mail Transfer Protocol) is a protocol designed to transfer mail reliably and efficiently. When an SMTP client has a message to transmit, it establishes a two-way transmission channel to an SMTP server. The responsibility of an SMTP client is to transfer mail messages to a SMTP server, or report its failure to do so [RFC 2821].

Micrium OS SMTP module implement part of the following RFC:

ftp-ftp-rfc-editor-org-in-notes-rfc2821-txt

ftp-ftp-rfc-editor-org-in-notes-rfc2822-txt

ftp-ftp-rfc-editor-org-in-notes-rfc4616-txt

ftp-ftp-rfc-editor-org-in-notes-rfc4648-txt

ftp-ftp-rfc-editor-org-in-notes-rfc4954-txt

SMTP Client Example Applications

This section describes the examples that are related to the Micrium OS SMTP Client module.

- SMTP Client Initialization Example
  - Description
  - Configuration
    - Mandatory
    - Location
  - API
  - Notes
- SMTP Client Send Email Example
  - Description
  - Configuration
    - Mandatory
    - Optional
  - Location
  - API

SMTP Client Initialization Example

Description

This is a generic example that shows how to initialize the SMTP client module.

Configuration

Mandatory

The following #define must be added in ex_description.h to allow other examples to initialize the SNTP Client module correctly:

#define	Description
EX_SMTP_CLIENT_INIT_AVAIL	Lets the upper example layer know that the SMTP Client Initialization example is present and must be called by other examples.

Location

```
/examples/net/smtp/ex_smtp_client.c

/examples/net/smtp/ex_smtp_client.h
```

API

API	Description
Ex_SMTP_Client_Init()	Initialize the Micrium OS SMTP Client module for the example application.

Notes

None.

SMTP Client Send Email Example

Description

This is a simple example that sends an email.

Configuration

Mandatory

The following #define must be added in ex_description.h to allow other examples to initialize the SNTP Client module correctly:

#define	Description
EX_SMTPc_SERVER_ADDR	Specify the SMTP server address to use. One should be provided by your ISP or by your organization.Note that the default value is invalid ("smtp.isp.com ").
EX_SMTPc_TO_ADDR	Specify the destination email address, i.e., your email address. Note that the default value is invalid ("test_to@gmail.com ").

Optional

The following #define can be added to ex_description.h, as described in Example Applications section, to change default configuration value used by the example:

#define	Default value	Description
EX_SMTPc_FROM_NAME	"From Name"	Specify the displayed name of the sender
EX_SMTPc_FROM_ADDR	"test_from@gmail.com "	Specify the email address of the sender
EX_SMTPc_USERNAME	DEF_NULL	Specify a username, if null no authentication is applied.
EX_SMTPc_PW	DEF_NULL	Specify a password.
EX_SMTPc_MSG_SUBJECT	"Example Title"	Specify the email subject
EX_SMTPc_MSG_BODY	"Example email sent using Micrium OS"	Specify the email body
EX_TRACE	printf(**VA_ARGS**)	Specify the function used to output information

Location

```
/examples/net/smtp/ex_smtp_client.c

/examples/net/smtp/ex_smtp_client.h
```

API

API	Description
Ex_SMTP_Client_SendMail()	Send an email

SMTP Client Configuration

Initialization

Initializing the SMTP Client module is done by calling the function SMTPc_Init(). This function takes no configuration argument. Unless you override an optional configuration before calling the function SMTPc_Init(), the default configurations will be used.

Optional Configurations

This section describes the configurations that are optional. If you do not set them in your application, the default configurations will apply.

The default values can be retrieved via the structure SMTPc_InitCfgDflt.Note that these configurations must be set *before* you call the function SMTPc_Init().

Table - SMTP Client Optional Configurations

Configurations	Description	Type	Function to call	Default	Field from default configurat structure
Memory segment	This module allocates some control data. You can specify the memory segment from where such data should be allocated.	MEM_SEG*	SMTPc_ConfigureMemSeg()	General-purpose heap .	.MemSegPtr
Authentication encodes buffers	This module allocates buffers to encode the authentication data (user name and password). Two type of buffer are allocated; small buffers which will contain the username and the password and large buffers which will receive the encoded data.	CPU_INT16U	SMTPc_ConfigureAuthBufLen()	.AuthBufInputLen = 100.AuthBufOutputLen = 140	.AuthBufInputLen.AuthBufOu

Post-Init Configurations

This section describes the configurations that can be set at any time during execution *after* you called the function SMTPc_Init().

These configurations are optional. If you do not set them in your application, the default configurations will apply.

Table - SMTP Client Post-init Configurations

Configurations	Description	Function to call	Default	
Default SMTP network access timeout configurations	When performing network access, some timeout must be selected and the default value can be overridden.	`SMTPc_DfltCfgSet()`	`TimeoutConn_ms`	5000 ms
			`TimeoutTx_ms`	5000 ms
			`TimeoutRx_ms`	5000 ms
			`TimeoutClose_ms`	5000 ms

SMTP Client Programming Guide

This section describes the basic steps required to initialize and use the Simple Mail Transfer Protocol (SMTP) module.

- Initializing the SMTP Client Module
- Prepare Mail Message
- Send Mail

Initializing the SMTP Client Module

The first step is to initialize the SMTP client module. This is done by calling the function SMTPc_Init(). Note that the Network module MUST be initialized before the SMTP Client module.

Listing - Example of call to SNTPc_Init() in the *SMTP Client Programming Guide* page gives an example of a call to SMTPc_Init() .

Listing - Example of call to SNTPc_Init()

```
RTOS_ERR  err;

SMTPc_Init(&err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occured. Error handling should be added here. */
}
```

Prepare Mail Message

Once you have successfully initialized the SMTP Client module, you can prepare message to be sent using SMTP client module :

SMTPc_MsgAlloc()

SMTPc_MsgClr()

SMTPc_MsgFree()

SMTPc_MsgSetParam()

Listing - Example of how to prepare a mail message in the *SMTP Client Programming Guide* page gives an example of how to prepare a mail message:

Listing - Example of how to prepare a mail message

```
RTOS_ERR    err;
SMTPc_MSG  *p_msg;

p_msg = (SMTPc_MSG *)SMTPc_MsgAlloc(&err);
APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);

SMTPc_MsgSetParam(p_msg, SMTPc_FROM_ADDR, "test_from@mail.com", &err);
APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);

SMTPc_MsgSetParam(p_msg, SMTPc_FROM_DISPL_NAME, "From Name", &err);
APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);

SMTPc_MsgSetParam(p_msg, SMTPc_TO_ADDR, "test_to@mail.com", &err);
APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);

SMTPc_MsgSetParam(p_msg, SMTPc_MSG_SUBJECT, "This is the mail title", &err);
APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);

SMTPc_MsgSetParam(p_msg, SMTPc_MSG_BODY, "This is the mail message body", &err);
APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);

// TODO send message

SMTPc_MsgFree(p_msg, &err);
APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);
```

Send Mail

Once you have configured you mail message, you can transfer the mail using a remote SMTP server. You can send email using a high-level SMTPc_SendMail() function if you just need to send one email as shown by Listing - Example of call to SMTPc_SendMail() in the *SMTP Client Programming Guide* page.

Listing - Example of call to SMTPc_SendMail()

```
SMTPc_MSG  *p_msg;
RTOS_ERR    err;

p_msg = (SMTPc_MSG *)SMTPc_MsgAlloc(&err);
APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);

SMTPc_MsgSetParam(p_msg, SMTPc_FROM_ADDR, EX_SMTPc_FROM_ADDR, &err);
APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);

SMTPc_MsgSetParam(p_msg, SMTPc_FROM_DISPL_NAME, EX_SMTPc_FROM_NAME, &err);
APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);

SMTPc_MsgSetParam(p_msg, SMTPc_TO_ADDR, EX_SMTPc_TO_ADDR, &err);
APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);

SMTPc_MsgSetParam(p_msg, SMTPc_MSG_SUBJECT, EX_SMTPc_MSG_SUBJECT, &err);
APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);

SMTPc_MsgSetParam(p_msg, SMTPc_MSG_BODY, EX_SMTPc_MSG_BODY, &err);
APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);

SMTPc_SendMail(EX_SMTPc_SERVER_ADDR,
            DEF_NULL,
            EX_SMTPc_USERNAME,
            EX_SMTPc_PW,
            DEF_NULL,
            p_msg,
           &err);
APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);

SMTPc_MsgFree(p_msg, &err);
APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);
```

If you need to send more message at a given moment then you can use the set of low-level functions:

```
SMTPc_Connect()

SMTPc_SendMsg()

SMTPc_Disconnect()
```

FTP Client Module

- FTP Client Overview
- FTP Client Example Application
- FTP Client Configuration
- FTP Client Programming Guide

FTP Client Overview

FTP is a protocol designed to enable reliable transfer of files over a network. The FTP protocol has been implemented using TCP (Transmission Control Protocol). The FTP client implements part of the following RFCs:

RFC 959: ftp://ftp.rfc-editor.org/in-notes/rfc959.txt RFC 2389: ftp://ftp.rfc-editor.org/in-notes/rfc2389.txt Draft IETF: http://tools.ietf.org/html/draft-ietf-ftpext-mlst-16

This document describes how to configure and use the FTP Client module.

FTP Client Example Application

This section describes the examples that are related to the Micrium OS FTP Client module.

- Send Data from a buffer to a file on an FTP Server
  - Description

  - Configuration
  - Location
  - API
- Upload a file to an FTP Server
  - Description
  - Configuration
    - Mandatory
  - Location
  - API
- Receive a file into a buffer
  - Description
  - Configuration
  - Location
  - API
- Download a file from an FTP server
  - Description
  - Configuration
    - Mandatory
  - Location
  - API

Send Data from a buffer to a file on an FTP Server

Description

This is a simple example that sends the data, Ex_FTP_ClientData to the public FTP, speedtest.tele2.net , located at "upload/datamicriumos.bin".

Configuration

None

Location

```
/examples/net/ftp/ex_ftp_client.c

/examples/net/ftp/ex_ftp_client.h
```

API

API	Description
Ex_FTP_ClientSendData()	Send the data (Ex_FTP_ClientData) to a public FTP, located at "upload/datamicriumos.bin".

Upload a file to an FTP Server

Description

This is a simple example that sends a file from the File system to an FTP server.

Configuration

Mandatory

The following #define must be added in ex_description.h to allow other examples to initialize the SNTP Client module correctly:

#define	Default value	Description
EX_FTP_CLIENT_SEND_FILE_LOCAL_PATH	"local_file.txt"	Specify the file path to upload
EX_FTP_CLIENT_SEND_FILE_REMOTE_NAME	"remote_file.txt"	Specify the remote file path

Location

```
/examples/net/ftp/ex_ftp_client.c

/examples/net/ftp/ex_ftp_client.h
```

API

API	Description
Ex_FTP_ClientSendFile()	Send a file to a public FTP, speedtest.tele2.net , located at "upload/datamicriumos.bin".

Receive a file into a buffer

Description

This is a simple example that downloads a file; 1KB.zip, from the public FTP server; speedtest.tele2.net .

Configuration

None.

Location

```
/examples/net/ftp/ex_ftp_client.c

/examples/net/ftp/ex_ftp_client.h
```

API

API	Description
Ex_FTP_ClientReceiveData()	Download a file in a local buffer.

Download a file from an FTP server

Description

This is a simple example that downloads a file; 1KB.zip, from the public FTP server; speedtest.tele2.net .

Configuration

Mandatory

The following #define must be added in ex_description.h to allow other examples to initialize the SNTP Client module correctly:

#define	Default value	Description
EX_FTP_CLIENT_RECEIVE_FILE_LOCAL_PATH	"file_1KB.zip"	Specify the download file path

Location

```
/examples/net/ftp/ex_ftp_client.c

/examples/net/ftp/ex_ftp_client.h
```

API

API	Description
Ex_FTP_ClientReceiveFile()	Download a file.

FTP Client Configuration

Initialization

FTP Client module doesn't require any initialization.

Post-init configurations

This section describes the configurations that can be set for any new connection when you call the function FTPc_Open().

These configurations are optional. If you do not set them in your application, the default configurations will apply.

Table - SNTP Client Post-init Configurations

Configurations	Description	Type	Function to call	Default	
Default FTP Client network configurations	When performing network access some timeout must be selected and default value can be overwritten.	`FTPc_CFG`	`FTPc_Open()`	`CtrlConnMaxTimout_ms`	5000 ms
				`CtrlRxMaxTimout_ms`	5000 ms
				`CtrlTxMaxTimout_ms`	5000 ms
				`CtrlRxMaxDly_ms`	100 ms
				`CtrlTxMaxRetry`	3
				`CtrlTxMaxDly_ms`	100 ms
				`DTP_ConnMaxTimout_ms`	5000 ms
				`DTP_RxMaxTimout_ms`	5000 ms
				`DTP_TxMaxTimout_ms`	5000 ms
				`DTP_TxMaxRetry`	3
				`DTP_TxMaxDly_ms`	100 ms

FTP Client Programming Guide

This section describes the basic steps required to use the File Transfer Protocol (FTP) module.

- Initializing the FTP Client Module
- Open an FTP Connection
- Close an FTP Connection
- Send a Buffer to a Remote File
- Download a File to a Buffer
- Upload a File to the FTP Server
- Download a File From the FTP Server

Initializing the FTP Client Module

The FTP Client doesn't require any initialization.

Open an FTP Connection

An FTP connection must be opened before performing any command. You can open a new FTP connection using the function FTPc_Open(). Once the connection is established then you can start transferring data and file.

Listing - Example of how to open an FTP Connection in the *FTP Client Programming Guide* page gives an example of how to open a new connection:

Listing - Example of how to open an FTP Connection

```
FTPc_CONN  ftp_conn;
RTOS_ERR   err;
```

```
FTPc_Open(&ftp_conn,
          DEF_NULL,
          DEF_NULL,"speedtest.tele2.net",0,
          DEF_NULL,
          DEF_NULL,&err);APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE,;);
```

Close an FTP Connection

Once all operation on the server has been completed the FTP connection must be closed. The FTP connection can be closed using the function FTPc_Close().

Listing - Example of how to close an FTP Connection in the *FTP Client Programming Guide* page gives an example of how to open a new connection:

Listing - Example of how to close an FTP Connection

```
FTPc_CONN   ftp_conn;
RTOS_ERR    err;

FTPc_Open(&ftp_conn,
          DEF_NULL,
          DEF_NULL,
         "speedtest.tele2.net",
          0,
          DEF_NULL,
          DEF_NULL,
         &err);
APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);
FTPc_Close(&ftp_conn, &err);
APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);
```

Send a Buffer to a Remote File

Once the connection is established you can start transferring data to the FTP server

Listing - Example of how to send data to the FTP Server in the *FTP Client Programming Guide* page gives an example of how to open a new connection:

Listing - Example of how to send data to the FTP Server

```
FTPc_CONN   ftp_conn;
RTOS_ERR    err;

FTPc_Open(&ftp_conn,
          DEF_NULL,
          DEF_NULL,
         "speedtest.tele2.net",
          0,
          DEF_NULL,
          DEF_NULL,
         &err);
APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);

FTPc_SendBuf(&ftp_conn,
           "upload/data.bin",
       "This is the data",
         17,
           DEF_NO,
           &err);
APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);

FTPc_Close(&ftp_conn, &err);
APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);
```

Download a File to a Buffer

Once the connection is established you can begin to get data from a file located on an FTP server

Listing - Example of how to download a file to a buffer from the FTP Server in the *FTP Client Programming Guide* page gives an example of how to open a new connection:

Listing - Example of how to download a file to a buffer from the FTP Server

```
CPU_INT08U  buf[1024];
FTPc_CONN   ftp_conn;
RTOS_ERR    err;

FTPc_Open(&ftp_conn,
          DEF_NULL,
          DEF_NULL,
         "speedtest.tele2.net",
          0,
          DEF_NULL,
          DEF_NULL,
         &err);
APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);

FTPc_RecvBuf(&ftp_conn, "1KB.zip", buf, sizeof(buf), &file_size, &err);
APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);

FTPc_Close(&ftp_conn, &err);
APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);
```

Upload a File to the FTP Server

Once the connection is established you can start transferring file to the FTP server

Listing - Example of how to upload a file to the FTP Server in the *FTP Client Programming Guide* page gives an example of how to open a new connection:

Listing - Example of how to upload a file to the FTP Server

```
FTPc_CONN   ftp_conn;
RTOS_ERR    err;

FTPc_Open(&ftp_conn,
          DEF_NULL,
          DEF_NULL,
         "speedtest.tele2.net",
          0,
          DEF_NULL,
          DEF_NULL,
         &err);
APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);

FTPc_SendFile(&ftp_conn,
              "remote_file_path.txt"",
              "local_file_path.txt",
               DEF_NO,
              &err);

APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);

FTPc_Close(&ftp_conn, &err);
APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);
```

Download a File From the FTP Server

Once the connection is established you can start downloading files from the FTP server

Listing - Example of how to download a file from the FTP Server in the *FTP Client Programming Guide* page gives an example of how to download a file:

Listing - Example of how to download a file from the FTP Server

```
FTPc_CONN   ftp_conn;
RTOS_ERR    err;

FTPc_Open(&ftp_conn,
          DEF_NULL,
          DEF_NULL,
         "speedtest.tele2.net",
          0,
          DEF_NULL,
          DEF_NULL,
         &err);
APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);

FTPc_RecvFile(&ftp_conn, "1KB.zip", "local_1KB.zip", &err);
APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);

FTPc_Close(&ftp_conn, &err);
APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);
```

IPerf Module

IPerf Module can be used to measure network performances between two systems.

- IPerf Overview
- IPerf Example Applications
- IPerf Configuration
  - IPerf Compile-Time Configurations
  - IPerf Run-time Application Specific Configurations
- IPerf Programming Guide
  - Initialize IPerf Module

- Using IPerf Module

IPerf Overview

IPerf is a tool designed to perform performance tests and to measure various variables of a network. IPerf is a benchmarking tool for measuring performance between two systems, and it can be used as a server or as a client for both the TCP and UDP protocols.

It was originally developed in 1999 and is currently supported by National Laboratory for Applied Network Research (NLANR/DAST) at the University of Illinois at Urbana-Champaign. The original IPerf and its source code, written in C++, are in the public domain. IPerf can run on various platforms including Linux, Unix, and Windows. The original command-line based and derived GUI applications are available from many websites.

This section describes how to configure and use the IPerf module in a Micrium OS environment.

IPerf Example Applications

This section describes the examples that are related to the Micrium OS IPerf module.

- IPerf Initialization Example
  - Description
  - Configuration
    - Mandatory
    - Location
  - API
  - Notes
    - Requirements
      - Target
      - PC
    - Running the example
      - Target as TCP Server
      - Target as TCP Client
      - Target as UDP Server
      - Target as UDP Client

IPerf Initialization Example

Description

This is a generic example that shows how to initialize the IPerf module.

Configuration

Mandatory

The following #define must be added in ex_description.h to allow other examples to initialize the IPerf module correctly:

#define	Description
EX_IPERF_INIT_AVAIL	Lets the upper example layer know that the IPerf Initialization example is present and must be called by other examples.

Location

```
/examples/net/iperf/ex_iperf.c

/examples/net/iperf/ex_iperf.h
```

API

API	Description
Ex_IPerf_Init()	Initialize the Micrium OS IPerf module for the example application.

Notes

IPerf can be launched from a command line and by specifying parameters, using Telnet and Shell .

Requirements

Target

- Common-Shell. See Shell Module Programming Guide .
- Common-Auth. See Authentication Module Programming Guide .
- Telnet Server: See Telnet Server Example Applications .

PC

- IPerf application V 2.x: Can be downloaded from here. Note that IPerf version 3 or greater is not supported. Only version 2 is compatible.

Running the example

First, you must get the Telnet server running. See Telnet Server Example Applications .

Target as TCP Server

1. On Telnet client terminal enters the following line:
   iperf -s
2. On the PC command line type the following line:
   iperf -c <target_ip_address>
3. Telnet Client should display the result from the target and the PC command line whereas Iperf-PC run should display the result on the PC side.

Target as TCP Client

1. On the PC command line type the following line:
   iperf -s
2. On Telnet Client terminal enter the following line:
   iperf -c <pc_address>
3. Telnet Client should display the result from the target and the PC command line whereas Iperf-PC run should display the result on the PC side.

Target as UDP Server

1. On Telnet client terminal enters the following line:
   iperf -s -u
2. On the PC command line type the following line:
   iperf -c <target_ip_address> -u -b 1000M

Telnet Client should display the result from the target and the PC command line whereas Iperf-PC run should display the result on the PC side.

Target as UDP Client

1. On the PC command line type the following line:
   iperf -s -u
2. On Telnet Client terminal enter the following line:
   iperf -c <pc_address> -u
3. Telnet Client should display the result from the target and the PC command line whereas Iperf-PC run should display the result on the PC side.

IPerf Configuration

To properly suit your application's needs, the IPerf module must be properly configured. There are two groups of configuration parameters:

- IPerf Compile-Time Configurations
- IPerf Run-time Application Specific Configurations

IPerf Compile-Time Configurations

IPerf module is configurable at compile time via several #defines located in iperf_cfg.h file. IPerf module uses #defines when possible because they allow code and data sizes to be scaled at compile time based on enabled features. This allows the Read-Only Memory (ROM) and Random-Access Memory (RAM) footprints of the IPerf module to be adjusted based on application requirements.

It is recommended that the configuration process begins with the default configuration values which are shown in **bold** in the next sections.

Most of the #defines should be configured with the default configuration values. Another small handful of values may likely never change because there is currently only one configuration choice available. This leaves a few values that should be configured with values that may deviate from the default configuration.

Calculation Configuration

Table - IPerf Calculation Configuration

Constant	Description	Possible Values
IPERF_CFG_BANDWIDTH_CALC_EN	Determines whether the code and data space used to keep track of bandwidth is included. The performance will be better if another task is used to calculate the bandwidth.	DEF_ENABLED or DEF_DISABLED
IPERF_CFG_CPU_USAGE_MAX_CALC_EN	Determines whether the code and data space used to keep track of maximum CPU usage is included.	DEF_ENABLED or DEF_DISABLED

Server Configuration

Table - IPerf Server Configuration

Constant	Description	Possible Values
IPERF_CFG_SERVER_EN	Enables/Disables UDP & TCP server code.	DEF_ENABLED or DEF_DISABLED

Client Configuration

Table - IPerf Client Configuration

Constant	Description	Possible Values
IPERF_CFG_CLIENT_EN	Enables/Disables UDP & TCP client code.	DEF_ENABLED or DEF_DISABLED
IPERF_CFG_CLIENT_BIND_EN	Configure client to bind on same port as server.	DEF_ENABLED or DEF_DISABLED

IPerf Run-time Application Specific Configurations

This section defines the configurations related to the IPerf module, but which are specified at run-time, during the initialization process.

- Initialization
- Optional Configurations
- Post-Init Configurations
- Configuration Data Type
  - IPERF_CFG

IPERF_SERVER_CFG
- IPERF_CLIENT_CFG

Initialization

Initializing the Iperf module is done by calling the function `IPerf_Init()`. This function takes no configuration argument. Unless you override an optional configuration before calling the function `IPerf_Init()`, the default configurations will be used.

Optional Configurations

This section describes the configurations that are optional. If you do not set them in your application, the default configurations will apply.

The default values can be retrieved via the structure IPerf_CfgDflt.

> Note that these configurations must be set *before* you call the function IPerf_Init().

Table - IPerf Optional Configurations

Configurations	Description	Type	Function to call	Default	Field from default configuration structure
Task's stack	The IPerf module has a dedicated task depending. This configuration allows you to set the stack pointer and the stack size (in quantity of elements).	CPU_INT32Uvoid *	IPerf_ConfigureTaskStk()	A stack of 512 elements allocated on Common 's memory segment.	.StkSizeElements.StkPtr
Memory segment	This module allocates some control data. You can specify the memory segment from where such data should be allocated.	MEM_SEG *	IPerf_ConfigureMemSeg()	General-purpose heap .	.MemSegPtr
Configuration parameters		IPERF_CFG	IPerf_ConfigureCfg()	See IPERF_CFG	.BufLen .TestNbrMax .Server .Client

Post-Init Configurations

This section describes the configurations that can be set at any time during execution *after* you called the function IPerf_Init().

These configurations are optional. If you do not set them in your application, the default configurations will apply.

Table - HTTP Client Post-init Configurations

Configurations	Description	Type	Function to call	Default
Task priority	The IPerf module will create a task that handles the requests. You can change the priority of the created task at any time.	RTOS_TASK_PRIO	IPerf_TaskPrioSet()	See Appendix A - Internal Tasks .

Configuration Data Type

IPERF_CFG

Table - IPERF_CFG configuration structure in the *IPerf Run-time Application Specific Configurations* page describes each configuration field available in this configuration structure.

Table - IPERF_CFG configuration structure

Field	Description	Type	Default
.BufLen	Configure maximum buffer size used to send/receive.	CPU_INT32U	8192
.TestNbrMax	Configure the maximum number of tests.	CPU_INT16U	2
.Server	Configure Server parameters.	IPERF_SERVER_CFG	See IPERF_SERVER_CFG
.Client	Configure Client parameters.	IPERF_CLIENT_CFG	See IPERF_CLIENT_CFG

IPERF_SERVER_CFG

Table - IPERF_SERVER_CFG configuration structure in the *IPerf Run-time Application Specific Configurations* page describes each configuration field available in this configuration structure.

Table - IPERF_SERVER_CFG configuration structure

Field	Description	Type	Default
.AcceptMaxRetry	Configure server maximum of retries on accept.	CPU_INT16U	10
.AcceptMaxDlyMs	Configure server delay between retries on accept.	CPU_INT32U	500
.AcceptMaxTimeoutMs	Configure server maximum inactivity time on accept.	CPU_INT32U	5000
.TCP_RxMaxTimeoutMs	Configure server maximum inactivity time on TCP Rx.	CPU_INT32U	5000
.UDP_RxMaxTimeoutMs	Configure server maximum inactivity time on UDP Rx.	CPU_INT32U	5000

IPERF_CLIENT_CFG

Table - IPERF_CLIENT_CFG configuration structure in the *IPerf Run-time Application Specific Configurations* page describes each configuration field available in this configuration structure.

Table - IPERF_CLIENT_CFG configuration structure

Field	Description	Data Type	Default
.ConnMaxRetry	Configure client maximum of retries on connect.	CPU_INT16U	10
.ConnMaxDlyMs	Configure client delay between retries on connect.	CPU_INT32U	500
.ConnMaxTimeoutMs	Configure client maximum inactivity time on connect.	CPU_INT32U	5000
.TCP_TxMaxTimeoutMs	Configure client maximum inactivity time on TCP Tx.	CPU_INT32U	5000

IPerf Programming Guide

- Include Files
- Configuration
- API Reference

The following sections provide sample code describing how to use the IPerf module.

Include Files

Wherever you want to use IPerf module, you should include one or many of these files:

Include file	Description
rtos/net/include/iperf.h	Contains function prototypes, public data type, and default configuration

Configuration

Some parameters should be configured and/or optimized for your project requirements, see IPerf Configuration .

API Reference

Configuration Functions

Those functions are related to the module configuration.

Function name	Description
IPerf_ConfigureMemSeg()	Configure the memory segment that will be used to allocate internal data needed by the IPerf module instead of the default memory segment.
IPerf_ConfigureCfg()	Configure the IPerf module parameters.
IPerf_ConfigureTaskStk()	Configure IPerf's task stack size.
IPerf_TaskPrioSet()	Sets the priority of the IPerf task.

Module

Function Name	Description
IPerf_Init()	Initialize the IPerf module.

Using IPerf from the Code

These function can be used when IPerf must be launched from the code instead of the command line

Function Name	Description
IPerf_TestStart()	Validates and schedules a new IPerf test as it is done from the command line.
IPerf_TestRelease()	Removes the test in ring array holding.
IPerf_TestGetStatus()	Gets test's status.
IPerf_TestGetResults()	Gets test's result values.
IPerf_Reporter()	Prints IPerf test results before, during, and after the performance test. This function can be called once the IPerf_TestStart() returns a valid test ID.Note that Test is released automatically once completed.

Initializing IPerf module

The first step is to initialize the IPerf module. This is done by calling the function IPerf_Init(). Note that the Network module MUST be initialized before the IPerf module.

[Listing - Example of call to IPerf_Init()](listing---example-of-call-to-iperf_init() in the *initialize iperf module* page gives an example of a call to iperf_init().

Listing - Example of call to IPerf_Init()

```
RTOS_ERR  err;

IPerf_Init(&err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occured. Error handling should be added here. */
}
```

Using IPerf Module

Communication Link

A communication link is recommended between the Target and the PC to allow to synchronize tools launch between both hosts. For this, the Telnet Server module or a serial link can be used.

Shell Command

When Common-Shell module is present in Micrium OS, IPerf adds a command that allows launching from a command line. If Common-Shell is not present the function to launch IPerf specific function must be called with proper arguments, see Using IPerf from the code).

Command reference

The IPerf Shell module includes only one command which is *"iperf"*. The following table show the multiple options associated with this command.

Option	Description	Type	Usage example
-s	Create and launch an IPerf test as a server.	Server only	iperf -s
-c	Create and launch an IPerf test as a client. In client mode, the command must specify the IPerf test server address to connect to.	Client only	iperf -c 192.168.0.2
-h	Print the available argument to *"iperf"* command	General	iperf -h
-v	Print the version of the IPerf module	General	iperf -v
-f	Specified the format of the data saved.	General	iperf -s -f b
-t	The time in seconds to transmit for. Iperf normally works by repeatedly sending an array of *len* bytes for *time* seconds. The default is 10 seconds. See also the -l and -n options.	Client only	iperf -c 192.168.0.2 -t 10
-n	The number of buffers to transmit. Normally, Iperf sends for 10 seconds. The -n option overrides this and sends an array of *len* bytes *num* times, no matter how long that takes. See also the -l and -t options.	Client only	iperf -c 192.168.0.2 -n 200000
-l	The length of buffers to read or write. Iperf works by writing an array of *len* bytes a number of times. The default is 8 KB for TCP, 1470 bytes for UDP. Note for UDP, this is the datagram size and needs to be lowered when using IPv6 addressing to 1450 or less to avoid fragmentation. See also the -n and -t options.	General	iperf -s -l 4096
-p	The server port for the server to listen on and the client to connect to. This should be the same in both client and server. The default is 5001, the same as TCP.	General	iperf -s -p 5001
-u	Use UDP rather than TCP. See also the -b option.	General	iperf -s -u
-w	Sets the socket buffer sizes to the specified value. For TCP, this sets the TCP window size. For UDP it is just the buffer which datagrams are received in, and so limits the largest receivable datagram size.	General	iperf -s -w 4096
-D	Run the server as a daemon (Unix platforms) On Win32 platforms where services are available, Iperf will start running as a service.	Server only	iperf -s -D
-V	Bind to an IPv6 address	Client only	iperf -c fe80::1234:5678 -V
-i	Sets the interval time in seconds between periodic bandwidth, jitter, and loss reports. If non-zero, a report is made every *interval* seconds of the bandwidth since the last report. If zero, no periodic reports are printed. The default is zero.	General	iperf -s -i 1

Telnet Server Module

- Telnet Server Overview
- Telnet Server Example Applications
- Telnet Server Configuration
  - Telnet Server Run-time Application Specific Configuration
  - Telnet Server Instance Configuration

- Telnet Server Programming Guide
  - Initialize and Create a Telnet Server Instance

Telnet Server Overview

Telnet is a protocol allowing access to a remote host on the network. Once connected to a telnet server, the client host can issue commands that will be executed on the remote machine, as if they were entered directly on the server host. The Telnet server module implements parts of the following RFCs:

RFC 854 ftp-ftp-rfc-editor-org-in-notes-rfc854-txt RFC 857 ftp-ftp-rfc-editor-org-in-notes-rfc857-txt RFC 858 ftp-ftp-rfc-editor-org-in-notes-rfc858-txt

This section describes how to configure and use the Telnet server module.

Telnet Server Example Applications

This section describes the examples that are related to the Micrium OS Telnet server module.

- Telnet Server Initialization Example
  - Description
  - Configuration
    - Mandatory
    - Location
  - API
  - Notes
- Telnet Server Instance Example
  - Description
  - Configuration
    - Optional
  - Location
  - API
  - Notes
    - Requirements
      - Target
      - PC
    - Running the example

Telnet Server Initialization Example

Description

This is a generic example that shows how to initialize the Telnet Server module.

Configuration

Mandatory

The following #define must be added in ex_description.h to allow other examples to initialize the IPerf module correctly:

#define	Description
EX_TELNET_SERVER_INIT_AVAIL	Lets the upper example layer know that the Telnet Server example is present and must be called by other examples.

Location

```
/examples/net/telnet/ex_telnet_server.c

/examples/net/telnet/ex_telnet_server.h
```

API

API	Description
Ex_TELNET_Server_Init()	Initialize the Micrium OS Telnet Server module for the example application.

Notes

None.

Telnet Server Instance Example

Description

This is a basic Telnet server examples that show how to start a Telnet server instance and execute commands from a remote client.

The example starts a Telnet server with default configuration and can execute commands loaded in Shell module .

It accomplishes the following tasks:

- Create user
- Initialize the telnet server instance
- Start the telnet server instance

Configuration

Optional

The following #define can be added to ex_description.h, as described in Example Applications section, to change default configuration value used by the example:

#define	Default value	Description
EX_TELNET_USERNAME	"DUT"	Specify the username to access the telnet server
EX_TELNET_PASSWORD	"micrium"	Specify the password to access the telnet server
EX_TELNET_SERVER_SRV_TASK_STK_SIZE	TELNET_SERVER_TASK_CFG_STK_SIZE_ELEMENTS_DFLT	Specify a stack size for the telnet service task
EX_TELNET_SERVER_SRV_TASK_PRIO	TELNET_SERVER_SRV_TASK_PRIO_DFLT	Specify a priority for the telnet service task
EX_TELNET_SERVER_SESSION_TASK_STK_SIZE	TELNET_SERVER_SESSION_TASK_CFG_STK_SIZE_ELEMENTS_DFLT	Specify a stack size for the telnet session task

#define	Default value	Description
EX_TELNET_SERVER_SESSION_TASK_PRIO	TELNET_SERVER_SESSION_TASK_PRIO_DFLT	Specify a priority for the telnet session task

Location

```
/examples/net/telnet/ex_telnet_server.c

/examples/net/telnet/ex_telnet_server.h
```

API

API	Description
Ex_TELNET_Server_InstanceCreate()	Initializes and starts a default telnet server instance.

Notes

None.

Requirements

Target

- Authentication module: See Authentication Module Programming Guide
  - Must create a user which will be used to log on Telnet server
- Shell Module: See Shell Module Programming Guide
  - You can create your own function to be executed from telnet or can use a build in command provided by a module such as "ifconfig" provided by the Network core Module.

PC

- Your preferred Telnet client

Running the example

1. Start your preferred telnet client and connect to the target using the IP address assigned to the target. See Starting a Network Interface for further information about how to retrieve target's IP address.
2. Once the connection is accepted, the telnet server will ask for a user and a password. Use the credential created as described in the Requirements .
3. When successfully logged in, you will see the welcome message appear following by the command prompt. You can then execute any command loaded in the Shell module .

Telnet Server Configuration

In order to address your application's needs, Telnet server module must be properly configured. There are two groups of configuration parameters:

- Telnet Server Run-time Application Specific Configuration
- Telnet Server Instance Configuration

Telnet Server Run-time Application Specific Configuration

This section defines the configurations related to OS Micrium Telnet Server but that are specified at run-time, during the initialization process.

- Initialization
- Optional configurations

Initialization

Initializing the Telnet Server module is done by calling the function TELNETs_Init(). This function takes no configuration argument. Unless you override an optional configuration before calling the function TELNETs_Init(), the default configurations will be used.

Optional configurations

This section describes the configurations that are optional. If you do not set them in your application, the default configurations will apply.

The default values can be retrieved via the structure TELNETs_InitCfgDflt.

> Note that these configurations must be set *before* you call the function TELNETs_Init().

Table - HTTP Server Optional Configurations

Configurations	Description	Type	Function to call	Default	Field from default configuration structure
Memory segment	This module allocates some control data. You can specify the memory segment from where such data should be allocated.	MEM_SEG*	TELNETs_ConfigureMemSeg()	General-purpose heap .	.MemSegPtr

Telnet Server Instance Configuration

This section defines the configurations related to Telnet Server module that are specified at run-time and that are specific to each added Telnet server instance.

To add a Telnet server instance, call the TELNETs_InstanceInit() function. This function requires two configurations arguments (described below).

- Server instance Configuration: p_cfg
  - Server
  - Connection
  - Authentication
  - File System
  - Echo Feature
- Tasks Configuration
  - Server task: p_task_srv_cfg
  - Session Task: p_task_session_cfg
  - Guidelines on How to Properly Set the Priority and Stack Size

Server instance Configuration: p_cfg

`p_cfg` is a pointer to a configuration structure of the type TELNETs_CFG. Telnet Server instance configuration is based on a structure that contains different configuration sections with parameter settings. This section should provide you with an in-depth understanding of all instance configuration parameters. You will also discover which settings to modify in order to enhance the functionalities and the performances. Refer to the configuration field description section for further details. The TELNETs_CFG object must be passed to the TELNETs_InstanceInit() API function during the initialization of the Telnet server instance.

Server

Table - Task Configuration

Structure Field	Type	Description	Possible Values
.Port	CPU_INT16U	Configure instance server port number.1. Default Telnet port number used by all web browser is 23. The default port number is defined by the following value: TELNET_SERVER_CFG_SERVER_PORT_DFLT	TELNET_SERVER_CFG_SERVER_PORT_DFLT
.IP_Type	TELNETs_IP_TYPE	Configure socket type. Select which kind of IP addresses can be accepted by the web server instance.When only one IP type is selected, only one socket and TCP connection will be reserved to listen for incoming connections.When the two IP types are selected, two sockets and TCP connections will be reserved for listening.	TELNETs_IP_TYPE_IPv4TELNETs_IP_TYPE_IPv6T
.SecureCfgPtr	TELNETs_SECURE_CFG	Configure instance secure (SSL/TLS) configuration structure.'Secure' field is used to enabled or disable SSL/TLS:1. DEF_NULL, the web server instance is not secure and doesn't use SSL/TLS.2. Point to a secure configuration structure, the web server is secure and use SSL/TLS.The secure server can be enabled *only* if the application project contains a secure module supported by the TCP/IP Module such as:1. NanoSSL provided by Mocana.2. CyaSSL provided by YaSSL.	- DEF_NULL for a non-secure server- Pointer to to be used.

Connection

Table - Task Configuration

Structure Field	Type	Description	Possible Values
.ConnInactivityTimeout_s	CPU_INT32U	Configure connection maximum inactivity timeout in integer seconds.For each connection when the inactivity timeout occurs the connection is automatically closed whatever what the state of the connection was.	**3600**
.TxTriesMaxNbr	TELNETs_IP_TYPE	The maximum number of tries the server attempt when sending data following a transmission error.	*Must* be >= 1**3u**
.RxBufLen	CPU_INT16U	Configure buffer length used by the server task.Each connection has a buffer to receive. If the memory is limited the buffer size can be reduced, but the performance could be impacted.	*Must* be >= 120**512u**
.NVT_BufLen	CPU_INT16U	Configure buffer length used by the session task.	*Must* be >= 120**512u**

Authentication

Table - Task Configuration

Structure Field	Type	Description	Possible Values
.UsernameStrLenMax	CPU_INT16U	These values determine the maximum lengths for the username used by the authentication mechanism.	*Must* be >= 132u
.PasswordStrLenMax	CPU_INT16U	These values determine the maximum lengths for the password used by the authentication mechanism.	*Must* be >= 132u
.LoginTriesMaxNbr	CPU_INT08U	The maximum number of tries the server gives to the connecting client in the authentication process.	*Must* be >= 13u
.WelcomeMsgStr	CPU_CHAR	Welcome message sent to the client upon successful client authentication.	Const StringTELNET_SERVER_CFG_WELCOME_MSG_STR_DFLT

File System

Table - Task Configuration

Structure Field	Type	Description	Possible Values
.FS_PathLenMax	CPU_INT16U	These values determine the maximum lengths for the password used by the authentication mechanism.	*Must* be >= 1128u

Echo Feature

Table - Task Configuration

Structure Field	Type	Description	Possible Values
.EchoEn	CPU_BOOLEAN	Configure if the echo mode is whether or not enabled.	DEF_ENABLED or DEF_DISABLED

Tasks Configuration

Telnet Instance uses a task to listen on the sockets and accept new incoming client connections. This task is called the *server* or *service task*. From there, a session task is created when a new connection is established. This is the task that is used to process the user commands.

Server task: p_task_srv_cfg

Each Telnet server instance has a kernel task associated. p_task_srv_cfg is a pointer to a configuration structure of the type RTOS_TASK_CFG that allows configuring the priority, stack base pointer, and stack size for that task.

Session Task: p_task_session_cfg

Each Telnet connection has a kernel task associated. p_task_session_cfg is a pointer to a configuration structure of the type RTOS_TASK_CFG that allows configuring the priority, stack base pointer, and stack size for that task.

Guidelines on How to Properly Set the Priority and Stack Size

Task Priorities

The priority of the Telnet server's task greatly depends on the requirements of your application. For some applications, it might be better to set it at a high priority, especially if your application requires a lot of tasks and is CPU intensive.

Task Stack Sizes

The arbitrary stack size of **1024** is a good starting point for most applications.

The only guaranteed method of determining the required task stack sizes is to calculate the maximum stack usage for each task. Obviously, the maximum stack usage for a task is the total stack usage along the task's most-stack-greedy function path. Note that the most-stack-greedy function path is not necessarily the longest or deepest function path.

The easiest and best method for calculating the maximum stack usage for any task/function should be performed statically by the compiler or by a static analysis tool since these can calculate function/task maximum stack usage based on the compiler's actual code generation and optimization settings. So for optimal task stack configuration, we recommend investing in a task stack calculator tool compatible with your build toolchain.

Telnet Server Programming Guide

- Include Files
- Configuration
- API Reference

The following sections provide sample code describing how to use the Telnet Server module.

Include Files

Wherever you want to use Telnet server, you should include one or many of these files:

Include file	Description
rtos/net/include/telnet_server.h	Contains function prototypes, public data type and default configuration

Configuration

Some parameters should be configured and/or optimized for your project requirements.

API Reference

Configuration Functions

Those functions are related to the module configuration.

Function name	Description
TELNETs_ConfigureMemSeg()	Configure the memory segment that will be used to allocate internal data needed by the Telnet server module instead of the default memory segment.

Module

Function Name	Description
TELNETs_Init()	Initialize the Telnet server module.

Instance

These functions are associated to instance initialization and creation

Function Name	Description
TELNETs_InstanceInit()	Initializes a Telnet server instance.
TELNETs_InstanceStart()	Starts a specific Telnet server instance which had been previously initialized.

Initialize and Create a Telnet Server Instance

This section describes the basic steps required to initialize and use the Telnet Server (TELNETs) module.

- Initializing Telnet Server Module
- Create Telnet Server Instance
- Authentication

Initializing Telnet Server Module

The first step is to initialize the Telnet server module. This is done by calling the function TELNETs_Init(). Note that the Network module *must* be initialized before the Telnet Server module.

Listing - Example of call to TELNETs_Init() in the *Initialize and Create a Telnet Server Instance* page gives an example of a call to TELNETs_Init().

Listing - Example of call to TELNETs_Init()

```
RTOS_ERR  err;

TELNETs_Init(&err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

Create Telnet Server Instance

Once you have successfully initialized the Telnet Server module, you can create a server instance. This is done by calling the function TELNETs_InstanceInit() and TELNETs_InstanceStart().

Listing - Example of Telnet server instance creation in the *Initialize and Create a Telnet Server Instance* page gives an example of how to create a Telnet server instance with the default configuration.

Listing - Example of Telnet server instance creation

```
TELNETs_INSTANCE  *p_instance;
RTOS_ERR          err;

p_instance = TELNETs_InstanceInit(DEF_NULL, DEF_NULL, DEF_NULL, &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}

TELNETs_InstanceStart(p_instance, &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

Authentication

Users and passwords are managed by the Common Authentication module. For further information about how to create user credentials, see Authentication Module Programming Guide .

TFTP Client Module

- TFTP Client Overview
- TFTP Client Example Application
- TFTP Client Configuration
- TFTP Client Programming Guide

TFTP Client Overview

TFTP is a simple file transfer protocol that uses UDP as its transport protocol. This protocol was designed to be small and easy to implement, so it lacks most of the features of regular FTP. The only thing it can do is read and write files from and to a remote host.

TFTP Client Module implements part of the TFTP protocol: ftp-ftp-rfc-editor-org-in-notes-rfc1350-txt

This section describes how to configure and use the TFTP client module. Note that TFTP requires a file system such as Micrium OS File System .

TFTP Client Example Application

This section describes the examples that are related to the Micrium OS TFTP Client module.

- Download and Upload a File to a TFTP Server
  - Description
  - Location
  - Modification Required
  - API

Download and Upload a File to a TFTP Server

Description

This is a simple example that downloads (GET) a file from a TFTP server and then uploads (PUT) the same file back to the TFTP server.

Location

The example implementation is located in */examples/net/tftp/ex_tftp_client.c*.

Modification Required

EX_TFTP_CLIENT_SERVER_HOSTNAME must be modified to specify the TFTP server address or hostname on your network.

EX_TFTP_CLIENT_FILE_LOCAL_PATH must be modified to specify the local file name and path to where the downloaded file will be saved and from where the uploaded file will be taken. Note that the file path must also contain the File System volume name in order to have a complete absolute path that will be understood by the File System.

EX_TFTP_CLIENT_FILE_REMOTE_PATH must be modified to specify the name and path of the file on the remote TFTP server that you wish to download with the GET method.

EX_TFTP_CLIENT_FILE_REMOTE_UPLOAD_PATH must be modified to specify the name and path of the file that will be uploaded back on the TFTP server with the PUT method.

API

The function Ex_TFTP_ClientGetAndPut() downloads and uploads a file to a TFTP server of your choice.

TFTP Client Run-time Application Specific Configurations

Initialization

TFTP Client module doesn't require any initialization.

Post-Init Configurations

This section describes the configurations that can be set for any new request when you call the function TFTPc_Get() or TFTPc_Put().

These configurations are optional. If you do not set them in your application, the default configurations will apply.

Table - SNTP Client Post-init Configurations

Configurations	Description	Type	Function to call	Default	
Default TFTP Client network configurations	When performing network access some timeout must be selected and default value can me overwriten.	TFTPc_CFG	TFTPc_Get() TFTPc_Put()	ServerPortNbr	69
				RxInactivityTimeout_ms	5000 ms
				TxInactivityTimeout_ms	5000 ms

TFTP Client Programming Guide

This section describes the basic steps required to use the Trivial File Transfer Protocol (TFTP) client module.

- Initializing the TFTP Client Module
- Upload a File to the TFTP Server
- Download a File from the TFTP Server

Initializing the TFTP Client Module

The TFTP Client doesn't require any initialization.

Upload a File to the TFTP Server

You can send a file using the function TFTPc_Put() as shown in Listing - Example of how to upload a file to a TFTP Server in the *TFTP Client Programming Guide* page:

Listing - Example of how to upload a file to a TFTP Server

```
RTOS_ERR    err;

TFTPc_Put("server_hostname",
          DEF_NULL,
          "local_file_path.txt",
          "remote_file_path.txt",
          TFTPc_MODE_OCTET,
         &err);
APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);
```

Download a File from the TFTP Server

You can download a file using the function TFTPc_Get() as shown in Listing - Example of how to download a file from a TFTP Server in the *TFTP Client Programming Guide* page:

Listing - Example of how to download a file from a TFTP Server

```
RTOS_ERR    err;

TFTPc_Get("server_hostname",
          DEF_NULL,
          "local_file_path.txt",
          "remote_file_path.txt",
          TFTPc_MODE_OCTET,
         &err);
APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE, ;);
```

TFTP Server Module

- TFTP Server Overview

- TFTP Server Example Application
- TFTP Server Configuration
- TFTP Server Programming Guide

TFTP Server Overview

TFTP is a simple file transfer protocol that uses UDP as its transport protocol. This protocol was designed to be small and easy to implement, so it lacks most of the features of regular FTP. The only thing it can do is read and write files from and to a remote host.

TFTP Server module implements part of the TFTP protocol: ftp-ftp-rfc-editor-org-in-notes-rfc1350-txt

This section describes how to configure and use the TFTP Server module. Note that TFTP requires a file system such as Micrium OS File System .

TFTP Server Example Application

This section describes the examples that are related to the Micrium OS TFTP server module.

- TFTP Server Initialization Example
  - Description
  - Configuration
  - API
  - Notes

TFTP Server Initialization Example

Description

This is a generic example that shows how to initialize the TFTP Server module.

Configuration

The example will set up a RamDisk media and configure a volume to use as the TFTP server root directory. This will be used to store the files.

EX_TFTP_SERVER_RAMDISK_SEC_SIZE can be modified to change the sector size of the RamDisk.

EX_TFTP_SERVER_RAMDISK_SEC_NBR can be modified to change the number of sectors for the RamDisk.

EX_TFTP_SERVER_FILE_VOL can be modified to change the name of the File System volume used by the TFTP server.

API

API	Description
Ex_TFTP_ServerInit()	Initialize the Micrium OS TFTP Server module for the example application.

Notes

Microsoft Windows provides a TFTP client application that you can used to test the TFTP server on your platform.

Open a Windows command prompt and enter "tftp -h". You should see a message from the TFTP service. If you receive an error message, it means that the service is not activated on your PC, and you should enable it in your Windows Services.

Once you have the TFTP client service running, you can upload files to your TFTP server with the PUT command and download files with the GET command.

TFTP Server Configuration

In order to address your application's needs, the TFTP server module must be properly configured. There is only one set of configuration parameters:

- TFTP Server Run-time Application Specific Configurations

TFTP Server Run-time Application Specific Configurations

This section defines the configurations related to TFTP server that are specified at run-time, during the initialization process.

- Initialization
- Optional Configurations
- Post-Init Configurations

Initialization

To initialize the TFTP Server module, you must call the function TFTPs_Init(). This function takes only one mandatory configuration argument. Unless you override a configuration before calling the function TFTPs_Init(), the default configurations will be used.

Optional Configurations

This section describes the configurations that are optional. If you do not set them in your application, the default configurations will apply.

The default values can be retrieved via the structure TFTPs_InitCfgDflt.

> Note that these configurations must be set *before* you call the function TFTPs_Init().

Table - TFTP Server Optional Configurations

Configurations	Description	Type	Function to call	Default	Field from default configuration structure
Task's stack	This configuration allows you to set the stack pointer and the stack size (in the number of elements) for the TFTP server.	CPU_INT32Uvoid *	TFTPs_ConfigureTaskStk()	A stack of 512 elements allocated on Common 's memory segment.	.StkSizeElements .StkPtr
Memory segment	This module allocates control data. You can specify the memory segment from where such data should be allocated.	MEM_SEG *	TFTPs_ConfigureMemSeg()	General-purpose heap .	.MemSegPtr
Connection parameters	The TFTP server allows to configure the connection parameters.	TFTPs_CONN_CFG	TFTPs_ConfigureConnParam()	Socket mode: TFTPs_SOCK_SEL_IPv4Port number : 69Reception timeout: 5000 ms	.ConnCfg

Post-Init Configurations

This section describes the configurations that can be set at any time during execution *after* you called the function TFTPs_Init().

These configurations are optional. If you do not set them in your application, the default configurations will apply.

Table - TFTP Server Post-Init Configurations

Configurations	Description	Type	Function to call	Default
Task priority	The TFTP server module will create a task that handles the TFTP requests. You can change the priority of the created task at any time.	RTOS_TASK_PRIO	TFTPs_TaskPrioSet()	See Appendix A - Internal Tasks .

TFTP Server Connection Configurations

Table - TFTPs_CONN_CFG Configuration Structure in the *TFTP Server Connection Configurations* page describes each configuration field available in this configuration structure.

Table - TFTPs_CONN_CFG Configuration Structure

Field	Description
.SockSel	Type of Socket for the TFTP server:- TFTPs_SOCK_SEL_IPv4- TFTPs_SOCK_SEL_IPv6- TFTPs_SOCK_SEL_IPv4_IPv6
.Port	Port number for the TFTP server
.RxTimeoutMax	Maximum timeout for reception

TFTP Server Programming Guide

This section describes the basic steps required to use the Trivial File Transfer Protocol (TFTP) server module.

- Include Files
- Configuration
- Initialization

Include Files

To include TFTP server functions in your application, include this file:

Include file	Description
rtos/net/include/tftp_server.h	Contains function prototypes, public data type, and default configuration

Configuration

Some parameters should be configured and/or optimized for your project requirements. See the section TFTP Server Configuration for further details.

Initialization

To start the TFTP server for the first time, you must call the TFTPs_Init() function.

Listing - TFTP Server Initialization Function

```
RTOS_ERR  err;
```

```
&err);APP_RTOS_ASSERT_CRITICAL(err.Code == RTOS_ERR_NONE,;);
```

USB Device

USB Device

NOTE: This documentation refers to a deprecated software component that will no longer be supported and removed in an future release. Consider using the latest Silicon Labs USB stack instead. For more information, see USB Device.

USB is one of the most successful communication interfaces in the history of computer systems, and is the de facto standard for connecting computer peripherals. Micrium OS USB Device is a USB device module designed specifically for embedded systems. Built from the ground up with Micrium's quality, scalability, and reliability, it has gone through a rigorous validation process to comply with the USB 2.0 specification. This manual describes how to initialize, start, and use Micrium OS USB Device. It explains the various configuration values and their uses, as well as providing a porting guide for your hardware. This manual also includes an overview of the technology, types of configuration possibilities, implementation procedures, and examples of typical usage for every available class.

To help you understand the USB concepts quickly, the documentation features many examples of USB with basic functions. These examples will provide you with a framework that allows you to build devices quickly. These examples include:

- Microphone, speaker or headset (Audio Class)
- USB-to-serial adapter (Communications Device Class)
- Ethernet-emulated devices (CDC EEM)
- Mouse or keyboard (Human Interface Device Class)
- Removable storage device (Mass Storage Class)
- Custom device (Vendor Class)

You can find more information about the basic concepts and theory behind the USB protocol in the USB section of the Technologies Overview Manual.

USB Device Overview

USB Device Overview

- Specifications
- Features
- Limitations

Specifications

- Complies with the "Universal Serial Bus specification revision 2.0"
- Implements the "Interface Association Descriptor Engineering Change Notice (ECN)"
- Transfer types
  - Control
  - Bulk
  - Interrupt
  - Isochronous
- USB classes
  - Audio 1.0
  - Communication Device Class (CDC)
  - Abstract Control Model (ACM)
  - Ethernet Emulation Model (EEM)
  - Human Interface Device (HID)
  - Mass Storage Class (MSC)
  - Vendor-specific class framework

Features

- Scalable to include only required features to minimize memory footprint
- Support Full-speed (12 Mbit/s) and High-speed (480 Mbit/s)
- Supports multiple devices
- Supports composite (multi-function) devices
- Supports multi-configuration devices
- Supports USB power-saving functionalities (device suspend and resume)
- Support for Microsoft OS Descriptors
- Complete integration of Mass Storage Class into Micrium OS File System module
- Complete integration of CDC EEM class into Micrium OS Network module

Limitations

- Some drivers don't support Isochronous transfers.

Integrating USB Device Into Your Project

Integrating USB Device Into Your Project

Micrium OS USB Device is composed of several components, each of which is a set of files that implement specific functions. The USB Device module consists of one component for the core part named "USB Device". This component is mandatory and must always be part of your project. It also consists of one component per class and driver, and at least one of each of these is also mandatory. To use USB Device, you must add these files to your project and populate your RTOS description file .

Starting the USB Device Module Quickly

Micrium offers a set of example applications that demonstrate some of the features of Micrium OS USB Device and help you start the development of your application. We highly recommended that you start from one of these examples.

The following sections describe each example applications available.

- USB Device Core Example Applications
- USB Device CDC ACM Class Example Applications
- USB Device CDC EEM Class Example Applications
- USB Device HID Class Example Applications
- USB Device MSC Class Example Applications
- USB Device Vendor Class Example Applications

USB Device Core Example Applications

USB Device Core Example Applications

This section describes the examples that are related to the core component of Micrium OS USB Device.

Simple Initialization Example

This is a generic example for the USB device module. It accomplishes the following tasks:

- Initialize the USB device core module
- Add one USB device controller
- Add a Full-Speed configuration to the device
- Add a High-Speed configuration to the device (only if the device is High-Speed capable)
- Initialize all the available class application example(s)
- Start the device

This example is mandatory if you add any class application examples. By default, this example will assume the presence of a USB controller named "usb0" and will attempt to add it. If this device is not present on your platform or you prefer to use another one, modify the define EX_USBD_CTRLR_NAME accordingly.

Location

The example implementation is located in /examples/usb/device/ex_usbd.h and /examples/usb/device/all/ex_usbd_all.c.

API

This application offers two API functions named Ex_USBD_Init() and Ex_USBD_Start(). The function Ex_USBD_Init() must be called by your application first. Once your application is ready to connect to the USB host, you can then call the function Ex_USBD_Start().

USB Device Configuration

USB Device Configuration

This section discusses how to configure Micrium OS USB Device. There are three groups of configuration parameters:

- USB Device Compile-Time Configuration
- USB Device Run-Time Application-Specific Configuration
- USB Device and Device Controller Driver Configuration

USB Device Compile-Time Configuration

Micrium OS USB Device can be configured at compile time via approximately a set of #defines located in the usbd_cfg.h file. USB Device uses #defines when possible because they allow code and data sizes to be scaled at compile time based on which features are enabled. This allows the read-only memory (ROM) and random-access memory (RAM) footprints of Micrium OS USB Device to be adjusted based on your application's requirements.

Recommended: Start the configuration process with the default values (highlighted in **bold**).

The sections below are organized based on the order in the template configuration file, usbd_cfg.h.

- Core Configuration
- Classes Configuration

Core Configuration

Table - Generic Configuration Constants

Constant	Description	Possible values
USBD_CFG_OPTIMIZE_SPD	Optimizes for either better performance or for smallest code size. Enabling this define will optimize Micrium OS USB Device code for better performance, and disabling this define will lead to a smaller code size.	DEF_ENABLED or DEF_DISABLED
USBD_CFG_STR_EN	Enables or disables the USB strings. Disabling this will cause the device to not store any USB description strings passed from the application. This means the host will be unable to retrieve the description strings (such as manufacturer and product name).	DEF_ENABLED or DEF_DISABLED
USBD_CFG_HS_EN	Enables or disables supports for High-Speed devices. If your USB device controller(s) do not support High-Speed, or you configure them to be Full-Speed only, this should be set to DEF_DISABLED to reduce RAM/ROM usage.	DEF_ENABLED or DEF_DISABLED
USBD_CFG_EP_ISOC_EN	Enables or disables support for Isochronous endpoints. This feature is currently unavailable. As a result, it should be set to DEF_DISABLED.	DEF_ENABLED or DEF_DISABLED

Constant	Description	Possible values
USBD_CFG_URB_EXTRA_EN	Enables or disables support for allocation of more than one USB Request Block (URB) per endpoint. This feature is currently unavailable. As a result, it should be set to DEF_DISABLED.	DEF_ENABLED or DEF_DISABLED
USBD_CFG_MS_OS_DESC_EN	Enables or disables support for Microsoft OS descriptors. Enabling this feature will cause the device to respond to Microsoft OS string descriptor requests and Microsoft OS-specific descriptors. This feature is useful if you are using the Vendor class and you don't want to provide an .inf file for your Microsoft Windows users. For more information on Microsoft OS descriptors, refer to the Microsoft Hardware Dev Center .	DEF_ENABLED or DEF_DISABLED

Classes Configuration

Some classes may have specific compile-time configurations. Refer to USB Device Classes for more information.

USB Device Run-Time Application-Specific Configuration

- Core Initialization
  - p_qty_cfg
- Optional Configurations
- Post-Init Configurations

Core Initialization

You must initialize Micrium OS USB Device by calling the function USBD_Init(). This function takes one configuration argument that is described below.

p_qty_cfg

p_qty_cfg is a pointer to a configuration structure of type USBD_QTY_CFG. Its purpose is to inform the USB device module on how many USB objects to allocate.

The number of objects to allocate depends mostly on the class functions you will add to your device(s). For more information, refer to the section "Resource needs from core" of each class function you will add.

Table - USBD_CFG configuration structure in the *USB Device Run-Time Application-Specific Configuration* page describes each configuration field available in this configuration structure.

Table - USBD_CFG configuration structure

Field	Description
.DevQty	Number of devices that will be added via the USBD_DevAdd() function.
.ConfigQty	The total number of configurations (for all of your USB devices) that will be added via the USBD_ConfigAdd() function. Remember that if you have a High-speed device, you must add both a Full-speed and a High-speed configuration to your device in order to comply with the Universal Serial Bus 2.0 specification.
.IF_Qty	The total number of USB interfaces to be added for all your devices and configurations. This greatly depends on the class(es) used. For more information on how many interfaces a class requires, refer to the section "Resources required from the core" of your class(es).
.IF_AltQty	The total number of USB alternate interfaces to be added for all your devices and configurations. This greatly depends on the class(es) used. This value must always be equal or greater than .IF_Qty. For more information on how many alternate interfaces a class requires, refer to the section "Resources required from the core" of your class(es).

Field	Description
.IF_GrpQty	The total number of USB interface groups that will be added for all your devices and configurations. This greatly depends on the class(es) used. For more information on how many interface groups requires a class, refer to the section "Resources required from the core" of your class(es).
.EP_DescQty	The total number of Endpoint descriptors that will be added for all your devices and configurations. This greatly depends on the class(es) used. For more information on how many endpoint descriptors a class requires, refer to the section "Resources required from the core" of your class(es).
.URB_ExtraQty	The total number of extra URBs. This greatly depends on the class(es) used. For more information on how many extra URBs a class requires, refer to the section "Resources required from the core" of your class(es).This configuration has no impact if the configuration USBD_CFG_URB_EXTRA_EN is set to DEF_DISABLED.
.StrQty	The total number of USB strings per device.This configuration has no impact if the configuration USBD_CFG_STR_EN is set to DEF_DISABLED.
.EP_OpenQty	The total number of opened endpoints per device. A device requires at least two opened endpoints for control transfers, but you must also add the endpoints of the class(es) used. For more information on how many opened endpoints a class requires, refer to the section "Resources required from the core" of your class(es).

Optional Configurations

This section describes the configurations that are optional. If you do not set them in your application, the default configurations will apply.

The default values can be retrieved via the structure USBD_InitCfgDflt.

> Note that these configurations must be set BEFORE you call the function USBD_Init().

Table - USB Device Core Optional Configurations

Configurations	Description	Type	Function to call	Default	Field from default configuration structure
Buffer alignment	The USB Device module allocates buffers used for data transfers with the host. You can use this function to set the address alignment for those buffers. Note: If you have more than one USB device controller you must set the alignment to the highest value.	CPU_SIZE_T	USBD_ConfigureBufAlignOctets()	Size of cache line, or CPU alignment, if no cache.	.BufAlignOctets

Configurations	Description	Type	Function to call	Default	Field from default configuration structure
Memory segments	The USB Device module allocates control data and buffers used for data transfers with the host. It has the ability to use a different memory segment for the control data and for the data buffers.	MEM_SEG*	USBD_ConfigureMemSeg()	General-purpose heap .	.MemSegPtr
Memory segments	The USB Device module allocates control data and buffers used for data transfers with the host. It has the ability to use a different memory segment for the control data and for the data buffers.	MEM_SEG*	USBD_ConfigureMemSeg()	General-purpose heap .	.MemSegBufPtr

Post-Init Configurations

This section describes the configurations that can be set at any time during execution *after* you called the function USBD_Init().

These configurations are optional. If you do not set them in your application, the default configurations will apply.

Table - USB Device Core Optional Configurations

Configurations	Description	Type	Function to call	Default
Standard requests timeout	Timeout, in milliseconds, for the standard requests executed by the core module.	CPU_INT32U	USBD_StdReqTimeoutSet()	5000
Device task priority	Each USB device added via the function USBD_DevAdd() will create a task. You can change the priority of the created task at any time.	RTOS_TASK_PRIO	USBD_DevTaskPrioSet()	N/A

USB Device and Device Controller Driver Configuration

- p_task_cfg
  - Guidelines on How to Properly Set the Priority and Stack Size
- p_dev_cfg
- p_dev_drv_cfg
- p_bus _fnct

This section describes USB device driver configuration. Each newly-added device has an individual configuration, which is specified at run-time.

To add a USB device, you must call the USBD_DevAdd() function. This call requires four configurations arguments, which are described below.

p_task_cfg

Each USB device has a kernel task associated with it. p_task_cfg is a pointer to a configuration structure of type RTOS_TASK_CFG that allows you to configure the priority, stack base pointer, and stack size (in quantity of stack elements) for that task.

Guidelines on How to Properly Set the Priority and Stack Size

Task Priorities

The priority of a USB device's task greatly depends on the requirements of your application. If your application requires a lot of tasks and is CPU intensive, it may be best to set the device task to a high priority; otherwise, the task might not be able to process the bus and control requests on time. On the other hand, if you plan on using asynchronous communication, and data throughput is not a priority, you might want to give the core task a low priority.

Task Stack Sizes

A stack size of **1024** is a good starting point for most applications.

Refer to this page for more information on how to properly set the stack size.

p_dev_cfg

p_dev_cfg is a pointer to a configuration structure of type USBD_DEV_CFG. Its purpose is to give the USB device module basic information regarding your device, such as Vendor/Product ID, device strings, etc.

Table - USBD_DEV_CFG configuration structure in the *USB Device and Device Controller Driver Configuration* page describes each configuration field available in this configuration structure.

Table - USBD_DEV_CFG configuration structure

Field	Description
.VendorID	Your vendor identification number as delivered by the USB Implementers Forum. For more information on how you can obtain a vendor ID, see http://www.usb.org/developers/vendor/ .
.ProductID	Your product identification number.
.DeviceBCD	Release number of your device.
.ManufacturerStrPtr	Pointer to a string describing the manufacturer of your device. This configuration is ignored when the configuration USBD_CFG_STR_EN is set to DEF_DISABLED .
.ProductStrPtr	Pointer to a string describing your product. This configuration is ignored when the configuration USBD_CFG_STR_EN is set to DEF_DISABLED.
.SerialNbrStrPtr	Pointer to a string containing the serial number of your device. This configuration is ignored when the configuration USBD_CFG_STR_EN is set to DEF_DISABLED.
.LangId	Identification number of the language of your device's strings. Possible values are:
	- USBD_LANG_ID_ARABIC_SAUDI_ARABIA
	- USBD_LANG_ID_CHINESE_TAIWAN
	- USBD_LANG_ID_ENGLISH_US
	- USBD_LANG_ID_ENGLISH_UK
	- USBD_LANG_ID_FRENCH
	- USBD_LANG_ID_GERMAN
	- USBD_LANG_ID_GREEK
	- USBD_LANG_ID_ITALIAN
	- USBD_LANG_ID_PORTUGUESE
	- USBD_LANG_ID_SANSKRIT
	This configuration is ignored when the configuration USBD_CFG_STR_EN is set to DEF_DISABLED.

p_dev_drv_cfg

p_dev_drv_cfg is a pointer to a configuration structure of type USBD_DEV_DRV_CFG. Its purpose is to give the USB device controller driver associated with your device basic information regarding the number of resources to allocate.

Table - USBD_DEV_DRV_CFG configuration structure in the *USB Device and Device Controller Driver Configuration* page describes each configuration field available in this configuration structure.

Table - USBD_DEV_DRV_CFG configuration structure

Field	Description
.EP_OpenQty	Maximum number of endpoints that can be opened at any time on this device.
.URB_ExtraQty	Number of extra URB to allocate for this device.This configuration is ignored when the configuration USBD_CFG_URB_EXTRA_EN is set to DEF_DISABLED.

p_bus_fnct

p_bus_cfg is a pointer to a configuration structure of type USBD_BUS_FNCTS. Its purpose is to give the USB device module a set of optional callback functions to be called when a bus event occurs.

A null pointer (DEF_NULL) can be passed to this argument if no callbacks are needed.

Table - USBD_BUS_FNCTS configuration structure in the *USB Device and Device Controller Driver Configuration* page describes each configuration field available in this configuration structure.

Table - USBD_BUS_FNCTS configuration structure

Field	Description	Function signature
.Reset	Function called when a reset signal is received from the host.	void Reset (CPU_INT08U dev_nbr)
.Suspend	Function called when the device is suspended by the host.	void Suspend(CPU_INT08U dev_nbr)
.Resume	Function called when a resume signal is received from the host.	void Resume (CPU_INT08U dev_nbr)
.CfgSet	Function called when a configuration has been set by the host.	void CfgSet (CPU_INT08U dev_nbr, CPU_INT08U cfg_val)
.CfgClr	Function called when a configuration has been cleared by the host.	void CfgClr (CPU_INT08U dev_nbr, CPU_INT08U cfg_val)
.Conn	Function called when the device has been connected.	void Conn (CPU_INT08U dev_nbr)
.Disconn	Function called when the device has been disconnected.	void Disconn(CPU_INT08U dev_nbr)

USB Device Programming Guide

USB Device Programming Guide

This section explains how to use the USB Device module.

- Initial Setup of USB Device Module

Initial Setup of USB Device Module

This section describes the basic steps required to initialize the USB Device module and to add, prepare, and start a device.

- Initializing the USB Device Module
  - Initializing the USB Device Core
  - Initializing the Class(es)
- Adding Your USB Device
- Building Your USB Device
  - Adding Configuration(s)
  - Adding USB Function(s)
- Starting Your USB Device

Initializing the USB Device Module

Initializing the USB Device Core

Begin by initializing the USB device module core by calling the function USBD_Init().

Listing - Example of call to USBD_Init() in the *Initial Setup of USB Device Module* page shows an example of call to USBD_Init(). For more information on the configuration arguments to pass to USBD_Init(), see USB Device Run-Time Application-Specific Configuration .

Listing - Example of call to USBD Init()

```
RTOS_ERR      err;
USBD_QTY_CFG  qty_cfg;

qty_cfg.DevQty       = 1u;
qty_cfg.ConfigQty    = 2u;
qty_cfg.IF_Qty       = 2u;
qty_cfg.IF_AltQty    = 2u;
qty_cfg.IF_GrpQty    = 0u;
qty_cfg.EP_DescQty   = 2u;
qty_cfg.URB_ExtraQty = 0u;
qty_cfg.StrQty       = 5u;
qty_cfg.EP_OpenQty   = 4u;

USBD_Init(&qty_cfg,
          &err);
if (err.Code != RTOS_ERR_NONE) {
  /* An error occurred. Error handling should be added here. */
}
```

Initializing the Class(es)

Once the USB device module core has been initialized, you must initialize each class you intend to use. See section "Programming Guide" of your class(es) for more information.

Adding Your USB Device

Once you successfully initialize the USB device module, you can start adding your device(s) by calling the function USBD_DevAdd(). This function must be called for each device you want to create/add.

Listing - Example of call to USBD_DevAdd() in the *Initial Setup of USB Device Module* page gives an example of call to USBD_DevAdd() using default arguments. This example adds a USB device controller called "usb0". For more information on configuring arguments to pass to USBD_DevAdd(), see USB Device and Device Controller Driver Configuration .

Listing - Example of call to USBD_DevAdd()

```
RTOS_ERR         err;
CPU_INT08U       dev_nbr;
RTOS_TASK_CFG    dev_task_cfg;
USBD_DEV_CFG     dev_cfg;
USBD_DEV_DRV_CFG dev_drv_cfg;

dev_task_cfg.Prio            = 8u;
dev_task_cfg.StkSizeElements = 1024u;
dev_task_cfg.StkPtr          = DEF_NULL; /* Allocate task stack from general purpose heap.   */

dev_cfg.VendorID           = 0xFFFEu;
dev_cfg.ProductID          = 0x0100u;
dev_cfg.DeviceBCD          = 0x0100u;
dev_cfg.ManufacturerStrPtr = "Micrium inc";
dev_cfg.ProductStrPtr      = "USB Demo Device";
dev_cfg.SerialNbrStrPtr    = "0123456789ABCDEF";
dev_cfg.LangId             = USBD_LANG_ID_ENGLISH_US;

dev_drv_cfg.EP_OpenQty   = 5u;
dev_drv_cfg.URB_ExtraQty = 0u;

dev_nbr = USBD_DevAdd("usb0",
                      &dev_task_cfg,
                      &dev_cfg,
                      &dev_drv_cfg,
                       DEF_NULL,
                      &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

Building Your USB Device

Adding Configuration(s)

Once you successfully add your device, you can start adding the USB functions to it, starting with a new configuration. A device must have at least one configuration, and for a High-Speed device, you must add a configuration for each speed. To add configuration(s), call the function USBD_ConfigAdd() function. This function must be called for each configuration you want to add.

If you have a High-speed device, and you create a High-Speed and a Full-Speed configuration, and both have the same functionalities, you must link them together. This is necessary to comply with the Universal Serial Bus specification, revision 2.0. To link two configurations, call the function USBD_ConfigOtherSpeed() .

Listing - Adding configuration(s) to your device in the *Initial Setup of USB Device Module* page gives an example of how to add a Full-speed and a High-speed configuration to your device.If your USB device supports Full-Speed only, you don't need to make the second call to USBD_ConfigAdd().

Listing - Adding configuration(s) to your device

```
RTOS_ERR        err;
CPU_INT08U      config_nbr_fs;
CPU_INT08U      config_nbr_hs;

                                        /* Adding a full-speed configuration to the device.   */
config_nbr_fs = USBD_ConfigAdd(dev_nbr,              /* Device number returned by USBD_DevAdd().        */
                               DEF_BIT_NONE,          /* No special attributes to the configuration.      */
                               100u,                  /* Max power consumption: 100mA.                    */
                               USBD_DEV_SPD_FULL,     /* Full-Speed configuration.                        */
                              "Config Add Example Full-Speed config",
                              &err);
if (err.Code != RTOS_ERR_NONE) {
   /* An error occured. Error handling should be added here. */
}

                                        /* Adding a high-speed configuration to the device.   */
config_nbr_hs = USBD_ConfigAdd(dev_nbr,              /* Device number returned by USBD_DevAdd().        */
                               DEF_BIT_NONE,          /* No special attributes to the configuration.      */
                               100u,                  /* Max power consumption: 100mA.                    */
                               USBD_DEV_SPD_HIGH,     /* High-Speed configuration.                        */
                              "Config Add Example High-Speed config",
                              &err);
if (err.Code != RTOS_ERR_NONE) {
   /* An error occured. Error handling should be added here. */
}

                                    /* Associate Full and High-speed configurations together ... */
                                    /* ... since they will provide the same functionalities.    */
USBD_ConfigOtherSpeed(dev_nbr,                /* Device number returned by USBD_DevAdd().            */
                      config_nbr_fs,
                      config_nbr_hs,
                     &err);
if (err.Code != RTOS_ERR_NONE) {
   /* An error occurred. Error handling should be added here. */
}
```

Adding USB Function(s)

Once you have successfully added at least one configuration to your device, you can add the interfaces and endpoints to your device. Each USB class has its own needs in terms of interface and endpoints type, quantity, and other parameters. Micrium OS USB Device adds interfaces and endpoints in the classes it offers.

From your application, you can instantiate a USB class and add it to a configuration. For more information on the concept of USB device class instances, see USB Device Classes . Note that you can instantiate and add many different class instances to a configuration to create a multi-function (composite) device.

Listing - Adding a class instance to your device in the *Initial Setup of USB Device Module* page gives a generic example of how to create a class instance and add it to a configuration.

Listing - Adding a class instance to your device

```
RTOS_ERR    err;
CPU_INT08U  class_nbr;

                                        /* Create an instance of the class you want to use.*/
class_nbr = USBD_<class>_Add(&err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}

                                        /* Add the class instance to the Full-Speed configuration.*/
USBD_<class>_ConfigAdd(class_nbr,              /* Class number returned by USBD_<class>_Add().          */
                       dev_nbr,                /* Device number returned by USBD_DevAdd().              */
                       config_nbr_fs,          /* Configuration number returned by USBD_ConfigAdd().    */
                      &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}

                                        /* Add the class instance to the High-Speed configuration.*/
                                        /* Only if you created a High-Speed configuration.        */
USBD_<class>_ConfigAdd(class_nbr,              /* Class number returned by USBD_<class>_Add().          */
                       dev_nbr,                /* Device number returned by USBD_DevAdd().              */
                       config_nbr_hs,          /* Configuration number returned by USBD_ConfigAdd().    */
                      &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

Starting Your USB Device

At this point, the device is still not detectable by the host if it is already connected. Once you have built/prepared your device, you can start it and make it visible to the USB host. This is done by calling the function USBD_DevStart().You have to start each device you added individually.

Listing - Starting your device in the *Initial Setup of USB Device Module* page gives an example of how to start your device using the USBD_DevStart() function.

Listing - Starting your device

```
RTOS_ERR  err;

USBD_DevStart(dev_nbr,                         /* Device number returned by USBD_DevAdd().              */
             &err)
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

USB Device Hardware Porting Guide

USB Device Hardware Porting Guide

The Micrium OS USB Device module uses a hardware driver that can be customized for any USB device controller using a *Board Support Package* (BSP). The BSP has two purposes:

- It initializes and configures any resources needed by the USB controller but which are provided by an external module.
- It provides hardware information to the USB device driver.

Micrium provides example BSPs for some popular platforms. If one is available for your platform, we recommend that you use it as a starting point. However, if no example BSP is available for your platform, the information in this section will help you understand how to correctly port the USB device module to your platform.

Note that each USB device controller (that you are planning to use) will require a BSP and all the steps described in this section will have to be performed for each of them.

- USB Device BSP Functions Guide
- USB Device Hardware Information
- USB Device Controller Registration to the Platform Manager

USB Device BSP Functions Guide

The Board Support Package contains a set of functions that support your hardware platform on Micrium OS. These functions are called by the USB device driver to perform hardware configuration or initialization of IO pins, interrupts, etc. It is your responsibility to either create these functions from scratch or to modify example code to fit your needs and the specifics of your hardware.

Each of these functions is described below.

- Initialize
- Connect
- Disconnect
- ISR Handling

Initialize

The first function you need to implement is a generic initialization function which is called by the driver only once at initialization. Listing - Init function signature in the *USB Device BSP Functions Guide* page shows the signature of this function.

Listing - Init function signature

```
void  BSP_USBD_EFM32_STK_Init (USBD_DRV  *p_drv);
```

Its purpose is to initialize any resources that will be necessary to the USB device controller, such as IO pins, power, ISR registration, clock, etc.

This function also receives an argument called p_drv. This argument will be useful when handling the USB device controller-related interrupts. It is then important to either save it as a global variable in your BSP or, if your interrupt controller supports this feature, pass it to your interrupt vector directly.

Connect

The second function to implement is a function that is called by the driver when the initialization is done, and the controller is started via the function USBD_DevStart(). Listing - Conn function signature in the *USB Device BSP Functions Guide* page shows the signature of the function.

Listing - Conn function signature

```
void  BSP_USBD_EFM32_STK_Conn (void);
```

Its purpose is to set the pull-up pin to signal the device connection to the host. Note that this may be handled directly by the USB controller, which means that this function may remain empty.

Disconnect

The third function to implement is called by the driver when the controller is stopped via the USBD_DevStop() function. Listing - Disconn function signature in the *USB Device BSP Functions Guide* page shows the signature of the function.

Listing - Disconn function signature

```
void  BSP_USBD_EFM32_STK_Disconn (void);
```

Its purpose is to clear the pull-up pin to signal the device disconnection to the host. Note that this may be handled directly by the USB controller, which means that this function may remain empty.

ISR Handling

Each USB Device driver has an ISR handler function that must be called each time a USB device interrupt is triggered. However, for most platforms, it will be necessary to implement an intermediate ISR handler in the BSP. This BSP ISR will then call the driver's ISR handler. This is necessary, as some interrupt controllers may require the interrupt status to be cleared each time it is triggered. It is also necessary if your interrupt controller does not support passing an argument to the interrupt vector, as the driver's ISR handler takes the p_drv received in the Init() function of the BSP as an argument.

Note that your BSP ISR handler must be registered to the interrupt controller in the Init() function.

Listing - Example of BSP ISR implementation in the *USB Device BSP Functions Guide* page shows an example of an ISR handler implemented in the BSP (that follows the CMSIS naming standard).

Listing - Example of BSP ISR implementation

```
void USB_IRQHandler (void)
{
    /* TODO: Clear interrupt status, if needed. */

    OSIntEnter();
    BSP_USBD_DrvPtr->API_Ptr->ISR_Handler(BSP_USBD_DrvPtr);              (1)
    OSIntExit();
}
```

(1) Here, the driver's ISR is called. Note that the global variable BSP_USBD_DrvPtr is a copy of the p_drv argument received from the BSP Init() function.

USB Device Hardware Information

- Endpoint Information Table
- Hardware Driver Information
- BSP API Structure
- Device Hardware Information

Endpoint Information Table

Your USB device controller has a set of pipes (used as endpoints) that are available to the USB device software. Micrium OS USB Device must be provided with information about these pipes. The way to provide this information is by using a

table of elements of type USBD_DRV_EP_INFO. Note that the last element of the table *must* be a null entry.

The information regarding the pipes of your USB device controller can be found in the manual for your MCU.

Table - USBD_DRV_EP_INFO structure in the *USB Device Hardware Information* page describes each configuration field available in this structure.

Table - USBD_DRV_EP_INFO structure

Field	Description	
.Attrib	This is a bitmap that represents the different endpoint types and directions that this pipe can support. Possible values of the bitmap are:	
	Value	Description
	USBD_EP_INFO_TYPE_CTRL	Pipe supports transfers of type Control.
	USBD_EP_INFO_TYPE_ISOC	Pipe supports transfers of type Isochronous.
	USBD_EP_INFO_TYPE_BULK	Pipe supports transfers of type Bulk.
	USBD_EP_INFO_TYPE_INTR	Pipe supports transfers of type Interrupt.
	USBD_EP_INFO_DIR_OUT	Pipe supports OUT (host-to-device) transfers.
	USBD_EP_INFO_DIR_IN	Pipe supports IN (device-to-host) transfers.
.Nbr	Endpoint number.	
.MaxPktSize	Maximum buffer size available for this pipe.	

Listing - Example of endpoint description table in the *USB Device Hardware Information* page gives an example of an endpoint information table. In this case, the controller has four bi-directional endpoints. Three are capable of all transfer types, one is the default control pipe. Also, note the null entry at the end of the table.

Listing - Example of endpoint description table

```
static USBD_DRV_EP_INFO BSP_USBD_EFM32_EP_InfoTbl[] = {
 { USBD_EP_INFO_TYPE_CTRL                                                                 | USBD_EP_INFO_DIR_OUT, 0u, 64u },
 { USBD_EP_INFO_TYPE_CTRL                                                                 | USBD_EP_INFO_DIR_IN, 0u, 64u },
 { USBD_EP_INFO_TYPE_CTRL | USBD_EP_INFO_TYPE_ISOC | USBD_EP_INFO_TYPE_BULK | USBD_EP_INFO_TYPE_INTR | USBD_EP_INFO_DIR_OUT, 1u,
64u },
 { USBD_EP_INFO_TYPE_CTRL | USBD_EP_INFO_TYPE_ISOC | USBD_EP_INFO_TYPE_BULK | USBD_EP_INFO_TYPE_INTR | USBD_EP_INFO_DIR_IN, 1u, 64u
},
 { USBD_EP_INFO_TYPE_CTRL | USBD_EP_INFO_TYPE_ISOC | USBD_EP_INFO_TYPE_BULK | USBD_EP_INFO_TYPE_INTR | USBD_EP_INFO_DIR_OUT, 2u,
64u },
 { USBD_EP_INFO_TYPE_CTRL | USBD_EP_INFO_TYPE_ISOC | USBD_EP_INFO_TYPE_BULK | USBD_EP_INFO_TYPE_INTR | USBD_EP_INFO_DIR_IN, 2u, 64u
},
 { USBD_EP_INFO_TYPE_CTRL | USBD_EP_INFO_TYPE_ISOC | USBD_EP_INFO_TYPE_BULK | USBD_EP_INFO_TYPE_INTR | USBD_EP_INFO_DIR_OUT, 3u,
64u },
 { USBD_EP_INFO_TYPE_CTRL | USBD_EP_INFO_TYPE_ISOC | USBD_EP_INFO_TYPE_BULK | USBD_EP_INFO_TYPE_INTR | USBD_EP_INFO_DIR_IN, 3u, 64u
},
 { USBD_EP_INFO_TYPE_CTRL | USBD_EP_INFO_TYPE_ISOC | USBD_EP_INFO_TYPE_BULK | USBD_EP_INFO_TYPE_INTR | USBD_EP_INFO_DIR_OUT, 4u,
64u },
 { USBD_EP_INFO_TYPE_CTRL | USBD_EP_INFO_TYPE_ISOC | USBD_EP_INFO_TYPE_BULK | USBD_EP_INFO_TYPE_INTR | USBD_EP_INFO_DIR_IN, 4u, 64u
},
 { USBD_EP_INFO_TYPE_CTRL | USBD_EP_INFO_TYPE_ISOC | USBD_EP_INFO_TYPE_BULK | USBD_EP_INFO_TYPE_INTR | USBD_EP_INFO_DIR_OUT, 5u,
64u },
 { USBD_EP_INFO_TYPE_CTRL | USBD_EP_INFO_TYPE_ISOC | USBD_EP_INFO_TYPE_BULK | USBD_EP_INFO_TYPE_INTR | USBD_EP_INFO_DIR_IN, 5u, 64u
},
 { DEF_BIT_NONE, 0u, 0u }
};
```

Hardware Driver Information

The USB device driver requires information about the USB device controller on your MCU, which you provide using a structure of type USBD_DRV_INFO. This information can be found in the manual of your MCU.

Table - USBD_DRV_INFO structure in the *USB Device Hardware Information* page describes each configuration field available in this structure.

Table - USBD DRV INFO structure

Field	Description
.BaseAddr	Base address of the USB device controller registers set. This corresponds to the address of the first register.
.MemAddr	Address of the dedicated memory to be used. Most of the time, this field is not used and can be set to 0.
.MemSize	Size of the dedicated memory to be used. Most of the time, this field is not used and can be set to 0.
.Spd	Maximum desired USB speed for this USB device controller. Possible values:
	- USBD_DEV_SPD_FULL
	- USBD_DEV_SPD_HIGH (if USB device controller supports it)
.EP_InfoTbl	Pointer to the endpoint information table created at step endpoint information table.

Listing - Example of driver information structure in the *USB Device Hardware Information* page shows an example of a driver hardware information structure.

Listing - Example of driver information structure

```
static USBD_DRV_INFO BSP_USBD_EFM32_DrvInfoPtr = {
#if defined(USBC_MEM_BASE)
 .BaseAddr = USBC_MEM_BASE,
#else
 .BaseAddr = USB_MEM_BASE,
#endif
 .MemAddr = 0x00000000u,
 .MemSize = 0u,
 .Spd = USBD_DEV_SPD_FULL,
 .EP_InfoTbl = BSP_USBD_EFM32_EP_InfoTbl
};
```

BSP API Structure

In order to provide a pointer to the BSP functions for the USB device controller driver, you have to create a structure of type USBD_DRV_BSP_API.

Table - USBD_DRV_BSP_API structure in the *USB Device Hardware Information* page describes each field available in this structure.

Table - USBD_DRV_BSP_API structure

Field	Description
.Init	Pointer to the BSP initialization function.
.Conn	Pointer to the BSP connect function.
.Disconn	Pointer to the BSP disconnect function.

Listing - Example of BSP API structure in the *USB Device Hardware Information* page shows an example of a BSP API structure.

Listing - Example of BSP API structure

```
static USBD_DRV_BSP_API BSP_USBD_USBHS_BSP_API_Ptr = {
    .Init    = BSP_USBD_EFM32_STK_Init,
    .Conn    = BSP_USBD_EFM32_STK_Conn,
    .Disconn = BSP_USBD_EFM32_STK_Disconn
};
```

Device Hardware Information

The last step is to create the main device hardware information structure. This structure contains only pointers to other structures.

Table - USBD_DEV_HW_INFO structure in the *USB Device Hardware Information* page describes each configuration field available in this structure.

Table - USBD_DEV_HW_INFO structure

Field	Description
.DrvAPI_Ptr	Pointer to the driver API structure you are using with your driver. Some drivers may provide more than one API structure.
.DrvInfoPtr	Pointer to the structure you created at step hardware driver information .
.BSP_API_Ptr	Pointer to the BSP API structure you created at step BSP API Structure .

USB Device Controller Registration to the Platform Manager

Once the hardware information structure for your USB device controller is ready, it must be registered with the Platform Manager . This should normally be done using the BSP_OS_Init() function that is located in the file bsp_os.c.

There are two different macros located in the file usb_ctrlr.h that you can call to register a USB Device controller. Table - USB Device Controller Register Macros in the *USB Device Controller Registration to the Platform Manager* page describes these different macros.

Table - USB Device Controller Register Macros

Macro	Description
USB_CTRLR_HW_INFO_REG()	Registers hardware information for a USB controller that has both Device and Host functionalities. Refer to USB Host Hardware Porting Guide for more information on the host side.
USB_CTRLR_HW_INFO_DEV_ONLY_REG()	Registers hardware information for a USB device controller.

Listing - Example of USB Controller Registration in the *USB Device Controller Registration to the Platform Manager* page shows an example of how to register a USB Device controller.

Listing - Example of USB Controller Registration

```
#include  <rtos_description.h>
#include  <rtos/usb/include/usb_ctrlr.h>

#if defined(RTOS_MODULE_USB_HOST_AVAIL)                        (1)
BSP_HW_INFO_EXT(const  USBH_HC_HCD_HW_INFO, BSP_USBH_EFM32_PBHCI_HwInfo);
#endif

#if defined(RTOS_MODULE_USB_DEV_AVAIL)
BSP_HW_INFO_EXT(const  USBD_DEV_CTRLR_HW_INFO, BSP_USBD_EFM32_HwInfo);
#endif

void  BSP_OS_Init (void)
{

    /* ... */
```

```
/* ------------- REGISTER USB CONTROLLERS ------------- */
#ifdefined(RTOS_MODULE_USB_DEV_AVAIL)USB_CTRLR_HW_INFO_DEV_ONLY_REG("usb0",(2)&BSP_USBD_EFM32_HwInfo);
#endif

#ifdefined(RTOS_MODULE_USB_DEV_AVAIL)||defined(RTOS_MODULE_USB_HOST_AVAIL)USB_CTRLR_HW_INFO_REG("usb1",
(3)&BSP_USBD_EFM32_HwInfo,&BSP_USBH_EFM32_PBHCI_HW_Info);
#endif
}
```

(1) Since the hardware information global variables are declared in another file, you must declare them as external in your bsp_os.c file. Always use the BSP_HW_INFO_EXT() macro.

(2) Registering a USB device-only controller with the tag "usb0". The tag will be used later on when calling the USBD_DevAdd() function.

(3) Registering a USB controller that implements both host and device functionalities. The controller name is "usb1". The tag will be used later on when calling the USBD_DevAdd() and/or USBH_HC_Add() function.

USB Device Classes

USB Device Classes

The USB classes available in Micrium OS USB Device share some common characteristics. This section explains these characteristics and their interactions with the core layer.

About Class Instances

The USB classes available in USB Device implement the concept of class instances. A class instance represents one function within a device. The function can be described by one interface or by a group of interfaces and belongs to a specific class.

Each USB class implementation has some configurations and functions in common, based on the concept of class instance. The common configurations and functions are presented in Table - Constants and Functions Related to the Concept of Multiple Class Instances in the *USB Device Classes* page. In the column title 'Constants or Function', the placeholder XXXX can be replaced by the name of the class: CDC, HID, MSC, CDC_EEM or VENDOR (Vendor for function names).

Table - Constants and Functions Related to the Concept of Multiple Class Instances

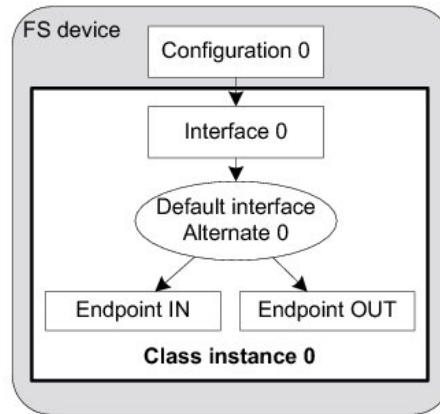
Constant or Function	Description
.ClassInstanceQty (in structure USBD_XXXX_QTY_CFG)	Configures the maximum number of class instances.
.ConfigQty (in structure USBD_XXXX_QTY_CFG)	Configures the maximum number of configurations per device. During the class initialization, a created class instance will be added to one or more configurations.
USBD_XXXX_Add()	Creates a new class instance.
USBD_XXXX_CfgAdd()	Adds an existing class instance to the specified device configuration.

In terms of code implementation, the class will declare a local global table that contains a class control structure. The size of the table is determined by the constant USBD_XXXX_CFG_MAX_NBR_DEV. This class control structure is associated with one class instance and will contain specific information to manage the class instance.

The following illustrations present several case scenarios. Each illustration includes a code listing that corresponds to the case scenario.

Figure 1 Multiple Class Instances - FS Device (1 Configuration with 1 Interface) in the *USB Device Classes* page represents a typical USB device. The device is Full-Speed (FS) and contains a single configuration. The function of the device is described by one interface composed of a pair of endpoints for data communication. One class instance is created, and it will allow you to manage the entire interface with its associated endpoint.

Figure - Multiple Class Instances - FS Device (1 Configuration with 1 Interface)



The code corresponding to Figure 1 Multiple Class Instances - FS Device (1 Configuration with 1 Interface) in the *USB Device Classes* page is shown in Listing - Multiple Class Instances - FS Device (1 Configuration with 1 Interface) in the *USB Device Classes* page.

Listing - Multiple Class Instances - FS Device (1 Configuration with 1 Interface)

```
RTOS_ERR          err;
CPU_INT08U        class_0;
USBD_XXXX_QTY_CFG  qty_cfg;

qty_cfg.ClassInstanceQty = 1u;
qty_cfg.ConfigQty        = 1u;

USBD_XXXX_Init(&qty_cfg, &err);                                  (1)
if (err.Code != RTOS_ERR_NONE) {
    /* $$$$ Handle the error. */
}

class_0 = USBD_XXXX_Add(&err);                                   (2)
if (err.Code != RTOS_ERR_NONE) {
    /* $$$$ Handle the error. */
}

USBD_XXXX_ConfigAdd(class_0, dev_nbr, config_0, &err);                    (3)
if (err.Code != RTOS_ERR_NONE) {
    /* $$$$ Handle the error. */
}
```

(1) Initialize the class. All internal variables, structures, and class ports will be initialized. Note that the Init() function in some classes may take other arguments.

(2) Create the class instance, which is class_0. The function USBD_XXXX_Add() allocates a class control structure associated with class_0. Depending on the class, USBD_XXXX_Add() may have additional parameters aside from the error code that represent class-specific information stored in the class control structure.

(3) Add the class instance, class_0, to the specified configuration number, config_0, on the device referenced by dev_nbr. USBD_XXXX_ConfigAdd() will create the interface 0 and its associated IN and OUT endpoints. As a result, the class instance encompasses the interface 0 and its endpoints. Any communication done on the interface 0 will use the class instance number, class_0.

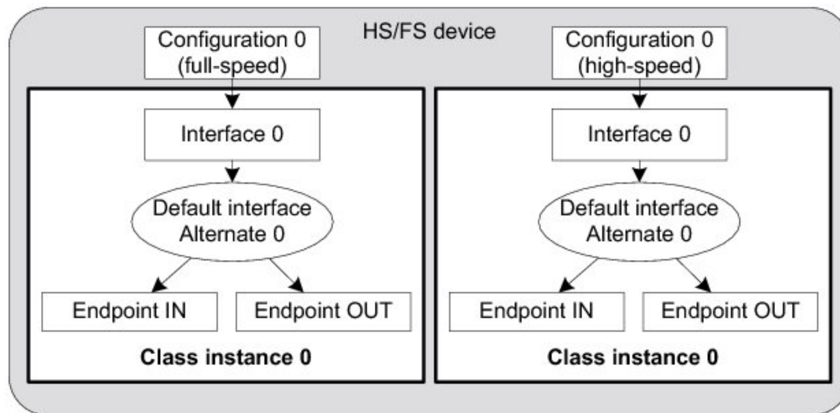
Figure - Multiple Class Instances - HS/FS Device (2 Configurations and 1 Single Interface) in the *USB Device Classes* page represents an example of a high-speed capable device. The device can support High-Speed (HS) and Full-Speed (FS). The device will contain two configurations:

- One valid if the device operates at full-speed
- Another if it operates at high-speed

In each configuration, interface 0 is the same, but its associated endpoints are different. The difference will be the endpoint maximum packet size, which varies according to the speed. If a high-speed host enumerates this device, then by default, the device will work in high-speed mode and therefore the high-speed configuration will be active. The host can learn about the full-speed capabilities by getting a *Device_Qualifier* descriptor followed by an *Other_Speed_Configuration* descriptor. These two descriptors describe a configuration of a high-speed capable device if it were operating at its other possible speed (refer to Universal Serial Bus 2.0 Specification revision 2.0, section 9.6, for more details about these descriptors).

In our example, the host may want to reset and enumerate the device again in full-speed mode. In this case, the full-speed configuration is active. But whatever the active configuration, the same class instance is used. Indeed, the same class instance can be added to different configurations, although a class instance cannot be added multiple times to the same configuration.

Figure - Multiple Class Instances - HS/FS Device (2 Configurations and 1 Single Interface)



The code corresponding to Figure - Multiple Class Instances - HS/FS Device (2 Configurations and 1 Single Interface) in the *USB Device Classes* page is shown in Listing - Multiple Class Instances - HS/FS Device (2 Configurations and 1 Single Interface) in the *USB Device Classes* page.

Listing - Multiple Class Instances - HS/FS Device (2 Configurations and 1 Single Interface)

```
RTOS_ERR          err;
CPU_INT08U        class_0;
USBD_XXXX_QTY_CFG  qty_cfg;

qty_cfg.ClassInstanceQty = 1u;
qty_cfg.ConfigQty        = 2u;

USBD_XXXX_Init(&qty_cfg, &err);                                     (1)
if (err.Code != RTOS_ERR_NONE) {
   /* $$$$ Handle the error. */
}

class_0 = USBD_XXXX_Add(&err);                                      (2)
if (err.Code != RTOS_ERR_NONE) {
   /* $$$$ Handle the error. */
}

USBD_XXXX_ConfigAdd(class_0, dev_nbr, config_0_fs, &err);           (3)
if (err.Code != RTOS_ERR_NONE) {
   /* $$$$ Handle the error. */
}

USBD_XXXX_ConfigAdd(class_0, dev_nbr, config_0_hs, &err);           (4)
if (err.Code != RTOS_ERR_NONE) {
   /* $$$$ Handle the error. */
}
```

(1) Initialize the class. Any internal variables, structures, and class ports will be initialized.

(2) Create the class instance, class_0. The function USBD_XXXX_Add() allocates a class control structure associated to class_0. Depending on the class, besides the parameter for an error code, USBD_XXXX_Add() may have additional parameters representing class-specific information stored in the class control structure.

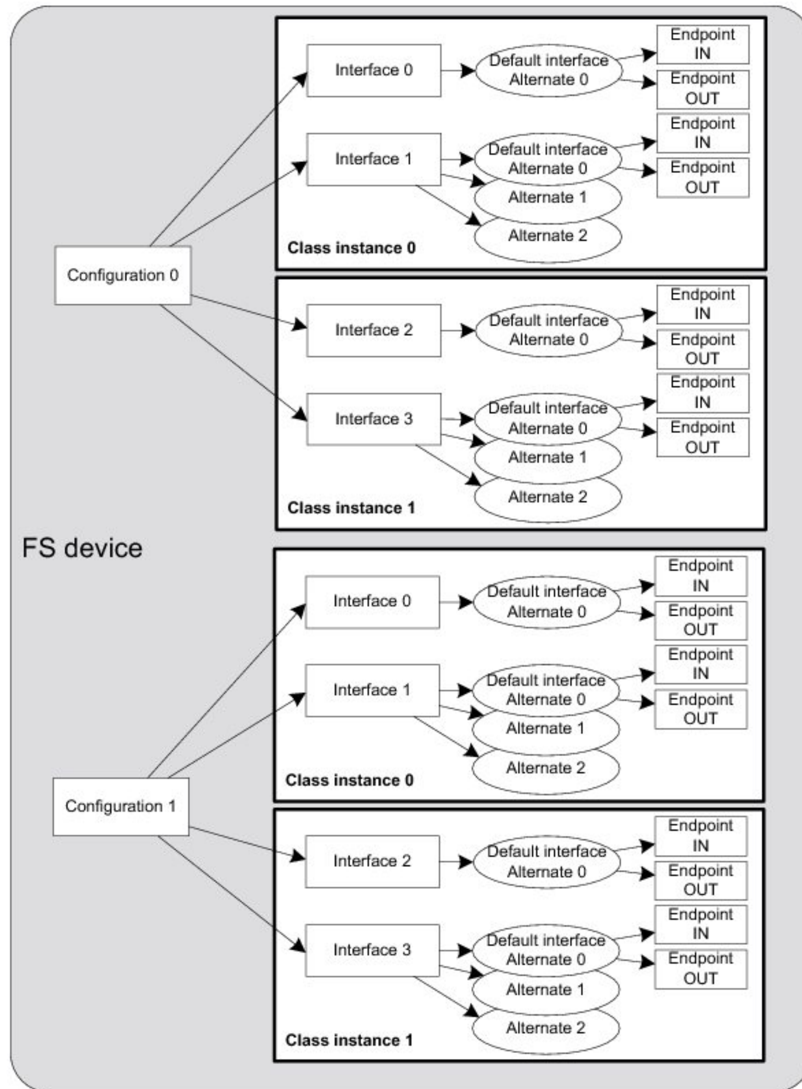
(3) Add the class instance, class_0, to the full-speed configuration, config_0_fs. USBD_XXXX_ConfigAdd() will create the interface 0 and its associated IN and OUT endpoints. If the full-speed configuration is active, any communication done on the interface 0 will use the class instance number, class_0.

(4) Add the class instance, class_0, to the high-speed configuration, config_0_hs.

Figure - Multiple Class Instances - FS Device (2 Configurations and Multiple Interfaces) in the *USB Device Classes* page represents a more complex example. A full-speed device is composed of two configurations. The device has two functions which belong to the same class, but each function is described by two interfaces and has a pair of bidirectional endpoints.

In this example, two class instances are created. Each class instance is associated with a group of interfaces as opposed to Figure - Multiple Class Instances - FS Device (1 Configuration with 1 Interface) in the *USB Device Classes* page and Figure - Multiple Class Instances - FS Device (2 Configurations and Multiple Interfaces) in the *USB Device Classes* page where the class instance was associated with a single interface.

Figure - Multiple Class Instances - FS Device (2 Configurations and Multiple Interfaces)



The code corresponding to Figure - Multiple Class Instances - FS Device (2 Configurations and Multiple Interfaces) in the *USB Device Classes* page is shown in Listing - Multiple Class Instances - FS Device (2 Configurations and Multiple Interfaces) in the *USB Device Classes* page. The error handling is omitted for clarity.

Listing - Multiple Class Instances - FS Device (2 Configurations and Multiple Interfaces)

```
RTOS_ERR          err;
CPU_INT08U        class_0;
CPU_INT08U        class_1;
USBD_XXXX_QTY_CFG  qty_cfg;

qty_cfg.ClassInstanceQty = 2u;
qty_cfg.ConfigQty        = 2u;

USBD_XXXX_Init(&qty_cfg, &err);                                    (1)

class_0 = USBD_XXXX_Add(&err);                                     (2)
class_1 = USBD_XXXX_Add(&err);                                     (3)

USBD_XXXX_ConfigAdd(class_0, dev_nbr, cfg_0, &err);                (4)
USBD_XXXX_ConfigAdd(class_1, dev_nbr, cfg_0, &err);                (5)

USBD_XXXX_ConfigAdd(class_0, dev_nbr, cfg_1, &err);                (6)
USBD_XXXX_ConfigAdd(class_1, dev_nbr, cfg_1, &err);                (6)
```

(1) Initialize the class. Any internal variables, structures, and class ports will be initialized.

(2) Create the class instance, class_0. The function USBD_XXXX_Add() allocates a class control structure associated with class_0.

(3) Create the class instance, class_1. The function USBD_XXXX_Add() allocates another class control structure associated with class_1.

(4) Add the class instance, class_0, to the configuration, cfg_0. USBD_XXXX_ConfigAdd() will create the interface 0, interface 1, alternate interfaces, and the associated IN and OUT endpoints. The class instance number, class_0, will be used for any data communication on interface 0 or interface 1.

(5) Add the class instance, class_1, to the configuration, cfg_0. USBD_XXXX_ConfigAdd() will create the interface 2, interface 3 and their associated IN and OUT endpoints. The class instance number, class_1, will be used for any data communication on interface 2 or interface 3.

(6) Add the same class instances, class_0 and class_1, to the other configuration, cfg_1.

USB Device CDC ACM Class

This section describes the Communications Device Class (CDC) class and the associated CDC subclass supported by Micrium's USB Device stack. Micrium OS USB-Device currently supports the Abstract Control Model (ACM) subclass, which is commonly used for serial emulation.

CDC includes various telecommunication and networking devices. Telecommunication devices encompass analog modems, analog and digital telephones, ISDN terminal adapters, etc. For example, networking devices contain ADSL and cable modems, Ethernet adapters, and hubs. CDC defines a framework to encapsulate existing communication services standards, such as V.250 (for modems over telephone network) and Ethernet (for local area network devices), using a USB link. A communication device is in charge of device management, call management when needed, and data transmission.

CDC defines seven major groups of devices. Each group belongs to a model of communication which may include several subclasses. Each group of devices has its own specification document besides the CDC base class. The seven groups are:

- Public Switched Telephone Network (PSTN), devices including voiceband modems, telephones, and serial emulation devices.
- Integrated Services Digital Network (ISDN) devices, including terminal adaptors and telephones.
- Ethernet Control Model (ECM) devices, including devices supporting the IEEE 802 family (ex.: cable and ADSL modems, WiFi adapters).
- Asynchronous Transfer Mode (ATM) devices, including ADSL modems and other devices connected to ATM networks (workstations, routers, LAN switches).
- Wireless Mobile Communications (WMC) devices, including multi-function communications handset devices used to manage voice and data communications.
- Ethernet Emulation Model (EEM) devices which exchange Ethernet-framed data.

- Network Control Model (NCM) devices, including high-speed network devices (High Speed Packet Access modems, Line Terminal Equipment)

The CDC and the associated subclass implementation complies with the following specifications:

- *Universal Serial Bus, Class Definitions for Communications Devices, Revision 1.2*, November 3 2010.
- *Universal Serial Bus, Communications, Subclass for PSTN Devices, Revision 1.2*, February 9 2007.

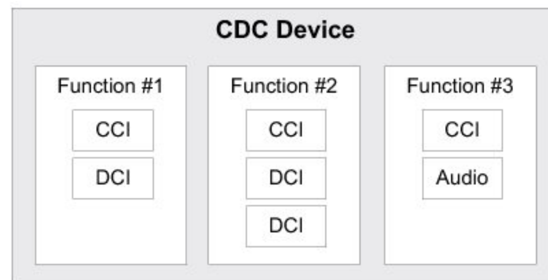
USB Device CDC Base Class Overview

A CDC device is composed of the following interfaces to implement communication capability:

- **Communications Class Interface (CCI)**: responsible for the device management and optionally the call management. The device management enables the general configuration and control of the device and the notification of events to the host. The call management enables calls establishment and termination. Call management might be multiplexed through a DCI. A CCI is mandatory for all CDC devices. It identifies the CDC function by specifying the communication model supported by the CDC device. The interface(s) following the CCI can be any defined USB class interface, such as Audio or a vendor-specific interface. The vendor-specific interface is represented specifically by a DCI.
- **Data Class Interface (DCI)**: responsible for data transmission. The data transmitted and/or received do not follow a specific format. Data could be raw data from a communication line, data following a proprietary format, etc. All the DCIs following the CCI can be seen as subordinate interfaces.

A CDC device must have at least one CCI and zero or more DCIs. One CCI and any subordinate DCI together provide a feature to the host. This capability is also referred to as a function. In a CDC composite device, you could have several functions. Therefore, the device would be composed of several sets of CCI and DCI(s) as shown in Figure - CDC Composite Device in the *USB Device CDC Base Class Overview* page.

Figure - CDC Composite Device



A CDC device is likely to use the following combination of endpoints:

- A pair of control IN and OUT endpoints called the default endpoint.
- An optional bulk or interrupt IN endpoint.
- A pair of bulk or isochronous IN and OUT endpoints.

Table - CDC Endpoint Usage in the *USB Device CDC Base Class Overview* page indicates the usage of the different endpoints and by which interface of the CDC they are used:

Table - CDC Endpoint Usage

Endpoint	Direction	Interface	Usage
Control IN	Device-to-host	CCI	Standard requests for enumeration, class-specific requests, device management, and optionally call management.
Control OUT	Host-to-device	CCI	Standard requests for enumeration, class-specific requests, device management, and optionally call management.
Interrupt or bulk IN	Device-to-host	CCI	Events notification, such as ring detect, serial line status, network status.

Endpoint	Direction	Interface	Usage
Bulk or isochronous IN	Device-to-host	DCI	Raw or formatted data communication.
Bulk or isochronous OUT	Host-to-device	DCI	Raw or formatted data communication.

Most communication devices use an interrupt endpoint to notify the host of events. Isochronous endpoints should not be used for data transmission when a proprietary protocol relies on data retransmission in case of USB protocol errors. Isochronous communication can inherently loose data since it has no retry mechanisms.

The seven major models of communication encompass several subclasses. A subclass describes the way the device should use the CCI to handle the device management and call management. Table - CDC Subclasses in the *USB Device CDC Base Class Overview* page shows all the possible subclasses and the communication model they belong to.

Table - CDC Subclasses

Subclass	Communication model	Example of devices using this subclass
Direct Line Control Model	PSTN	Modem devices directly controlled by the USB host
Abstract Control Model	PSTN	Serial emulation devices, modem devices controlled through a serial command set
Telephone Control Model	PSTN	Voice telephony devices
Multi-Channel Control Model	ISDN	Basic rate terminal adaptors, primary rate terminal adaptors, telephones
CAPI Control Model	ISDN	Basic rate terminal adaptors, primary rate terminal adaptors, telephones
Ethernet Networking Control Model	ECM	DOC-SIS cable modems, ADSL modems that support PPPoE emulation, Wi-Fi adaptors (IEEE 802.11-family), IEEE 802.3 adaptors
ATM Networking Control Model	ATM	ADSL modems
Wireless Handset Control Model	WMC	Mobile terminal equipment connecting to wireless devices
Device Management	WMC	Mobile terminal equipment connecting to wireless devices
Mobile Direct Line Model	WMC	Mobile terminal equipment connecting to wireless devices
OBEX	WMC	Mobile terminal equipment connecting to wireless devices
Ethernet Emulation Model	EEM	Devices using Ethernet frames as the next layer of transport. Not intended for routing and Internet connectivity devices
Network Control Model	NCM	IEEE 802.3 adaptors carrying high-speed data bandwidth on network

USB Device CDC ACM Class Resource Needs from Core

Each time you add a CDC ACM class instance to a USB configuration via a call to the function USBD_ACM_SerialConfigAdd(), the following resources will be allocated from the core.

Resource	Quantity
Interfaces	2
Alternate interfaces	2
Endpoint descriptors	3
Interface groups	1

USB Device CDC ACM Class Example Applications

- CDC ACM Class Terminal Example
- Location
- Running the Demo Application
- API

CDC ACM Class Terminal Example

This example emulates a USB-to-Serial adapter that displays a simple menu on a serial terminal. The menu allows you to send single or multiple characters to the adapter. The data is then sent back by the adapter, thereby creating a loopback.

Location

The example implementation is located in /examples/usb/device/all/ex_usbd_cdc_acm_terminal.c.

To execute it, you will also need some files on the host side. The files can be downloaded from the Micrium web site .

To install the Windows driver for the device, use the .inf file from the Windows application files located in /micrium_usb_dev_host_app/OS/Windows/CDC/INF.

Running the Demo Application

In this section, we will assume Windows is the host operating system. Upon connection of your CDC ACM device, Windows will enumerate your device and load the native driver usbser.sys to handle the device communication. The first time you connect your device to the host, you must indicate to Windows which driver to load using an INF file (refer to the About INF Files section for more details about INF files). The INF file tells Windows to load the usbser.sys driver. Indicating the INF file to Windows has to be done only once. Windows will then automatically recognize the CDC ACM device and load the proper driver for any new connection. The process of indicating the INF file may vary according to the Windows operating system version:

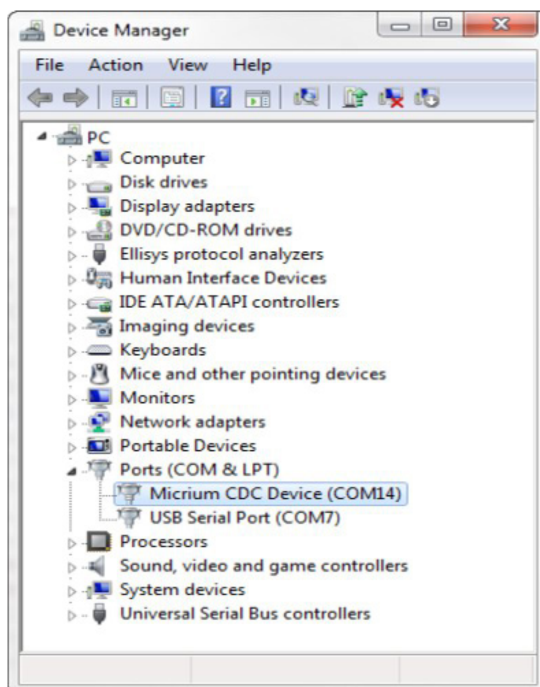
- Windows XP directly opens the Found New Hardware Wizard. Follow the different steps of the wizard until you reach the page where you can indicate the path of the INF file.
- Windows Vista and later won't open a "Found New Hardware Wizard". They will only indicate that no driver was found for the vendor device. You have to manually open the wizard. When you open the Device Manager, your CDC ACM device should appear with a yellow icon. Right-click on your device and choose 'Update Driver Software...' to open the wizard. Follow the different steps of the wizard until you reach the page where you can indicate the path of the INF file.

The INF file is located in:

/micrium_usb_dev_host_app/OS/Windows/CDC/INF

Refer to the About INF Files section for more details about how to edit the INF file to match your Vendor ID (VID) and Product ID (PID). By default, the provided INF files define 0xFFFE for VID and 0x1234 for PID. Once the driver is loaded, Windows creates a virtual COM port as shown in Figure - Windows Device Manager and Created Virtual COM Port in the *USB Device CDC ACM Class Example Applications* page.

Figure - Windows Device Manager and Created Virtual COM Port



The usbser.sys driver is already digitally signed by Windows. Micrium provides only an INF file, usbser.inf, telling Windows that your device uses that driver. Under Windows 7, providing this INF file was sufficient to load the driver usbser.sys for managing the CDC device. Windows 7 would display a warning message saying that the publisher of the driver can't be verified but it was possible to continue the driver loading. Since Windows 8.x, Microsoft has enforced by default the loading of digitally signed driver packages. This is called Driver Signature Enforcement. Windows 8.x won't let you load the CDC driver if the driver package is not fully signed. The Micrium CDC driver package is composed of usbser.inf (unsigned) and usbser.sys (signed). Basically, Windows 8.x requires a digitally signed INF file.

For development purposes, it is possible to disable the Driver Signature Enforcement. You can follow the instructions described on this page and you will be able to load the CDC driver and communicate with your USB device.

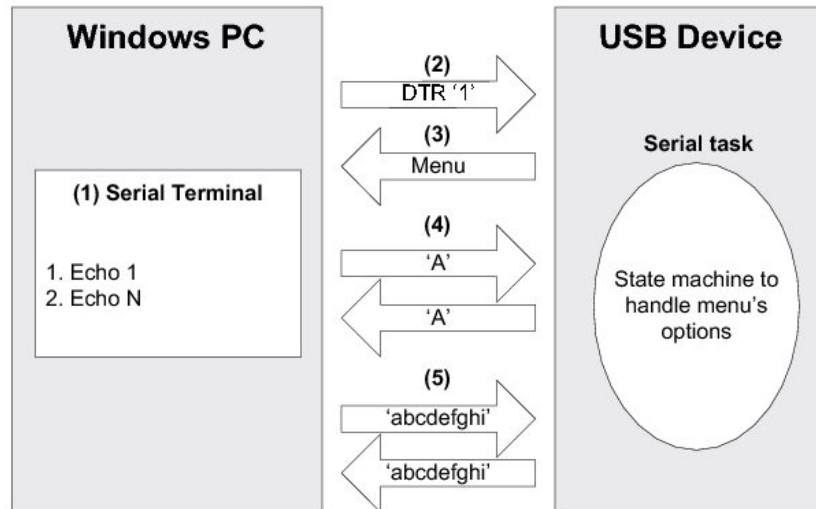
For your USB product release, you will have to follow the official procedure from Microsoft to sign the driver package (INF file + usbser.sys). The procedure is called the *Release-Signing* and is described here .

Micrium cannot provide an already-signed CDC driver package that would avoid disabling the Driver Signature Enforcement feature because the INF file contains a Vendor and Product IDs specific to the USB device manufacturer. For your USB product, the INF file must contain an official Vendor ID assigned to your company by the USB Implementer Forum. Micrium does not possess an official USB vendor ID. It is the customer's responsibility to go through the official signing process as you are the USB device manufacturer.

Note that this restriction does not exist anymore under Windows 10 and later as the INF file usbser.sys is now natively digitally signed by Microsoft.

Figure - Serial Demo in the *USB Device CDC ACM Class Example Applications* page presents the steps to follow to use the serial demo.

Figure - Serial Demo



(1) Open a serial terminal (for instance, HyperTerminal). Open the COM port matching to your CDC ACM device with the serial settings (baud rate, stop bits, parity and data bits) you want. This operation will send a series of CDC ACM class-specific requests (GET_LINE_CODING, SET_LINE_CODING, SET_CONTROL_LINE_STATE) to your device. Note that Windows Vista and later don't provide HyperTerminal anymore. You may use other free serial terminals such *TeraTerm* (http://ttssh2.sourceforge.jp/), *Hercules* (http://www.hw-group.com/products/hercules/index_en.html), *RealTerm* (http://realterm.sourceforge.net/), etc.

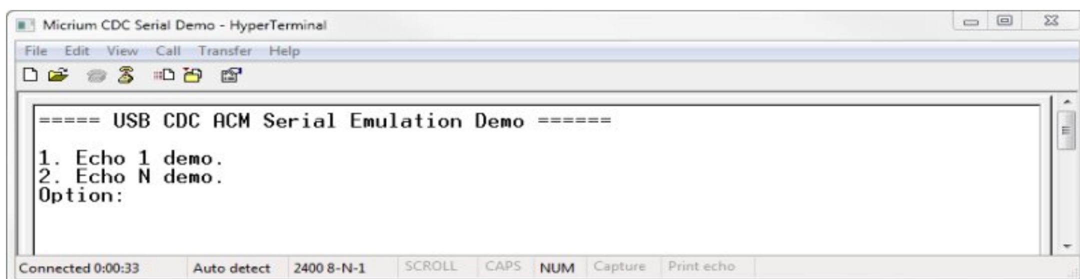
(2) In order to start the communication with the serial task on the device side, the Data Terminal Ready (DTR) signal must be set and sent to the device. The DTR signal prevents the serial task from sending characters if the terminal is not ready to receive data. Sending the DTR signal may vary depending on your serial terminal. For example, *HyperTerminal* sends a properly set DTR signal automatically upon opening of the COM port. *Hercules* terminal allows you to set and clear the DTR signal from the graphical user interface (GUI) with a checkbox. Other terminals do not permit to set/clear DTR or the DTR set/clear's functionality is difficult to find and to use.

(3) Once the serial task receives the DTR signal, the task sends a menu to the serial terminal with two options as presented in Figure - CDC Serial Demo Menu in HyperTerminal in the *USB Device CDC ACM Class Example Applications* page.

(4) The menu option #1 is the *Echo 1 demo*. It allows you to send one unique character to the device. This character is received by the serial task and sent back to the host.

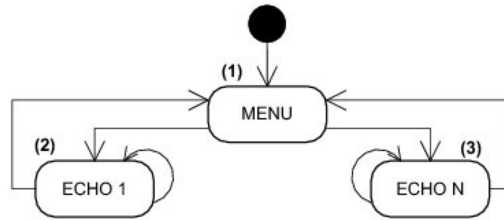
(5) The menu option #2 is the *Echo N demo*. It allows you to send several characters to the device. All the characters are received by the serial task and sent back to the host. The serial task can receive a maximum of 512 characters.

Figure - CDC Serial Demo Menu in HyperTerminal



To support the two demos, the serial task implements a state machine as shown in Figure - Serial Demo State Machine in the *USB Device CDC ACM Class Example Applications* page. Basically, the state machine has two paths corresponding to the user choice in the serial terminal menu.

Figure - Serial Demo State Machine



(1) Once the DTR signal has been received, the serial task is in the MENU state.

(2) If you choose the menu option #1, the serial task will echo back any single character sent by the serial terminal as long as “Ctrl+C” is not pressed.

(3) If you choose the menu option #2, the serial task will echo all the received characters sent by the serial terminal as long as “Ctrl+C” is not pressed.

Table - Serial Terminals and CDC Serial Demo in the *USB Device CDC ACM Class Example Applications* page shows four possible serial terminals which you may use to test the CDC ACM class.

Table - Serial Terminals and CDC Serial Demo

Terminal	DTR set/clear	Menu option(s) usable
HyperTerminal	Yes (properly set DTR signal automatically sent upon COM port opening)	1 and 2
Hercules	Yes (a checkbox in the GUI allows you to set/clear DTR)	1 and 2
RealTerm	Yes (Set/Clear DTR buttons in the GUI)	1 and 2
TeraTerm	Yes (DTR can be set using a macro. GUI does NOT allows you to set/clear DTR easily)	1 and 2

API

This example offers only one API named Ex_USBD_ACM_SerialInit(). This function is normally called from a USB device core example.

USB Device CDC ACM Subclass Overview

The CDC base class is composed of a Communications Class Interface (CCI) and Data Class Interface (DCI), and this is discussed in detail in the section USB Device CDC Base Class Overview . This section discusses a CCI of type ACM. It consists of a default endpoint for the management element and an interrupt endpoint for the notification element. A pair of bulk endpoints is used to carry unspecified data over the DCI.

The ACM subclass is used by two types of communication devices:

- Devices supporting AT commands (for instance, voiceband modems).
- Serial emulation devices which are also called Virtual COM port devices.

There are several subclass-specific requests for the ACM subclass. They allow you to control and configure the device. The complete list and description of all ACM requests can be found in the specification “*Universal Serial Bus, Communications, Subclass for PSTN Devices, revision 1.2, February 9, 2007*”, section 6.2.2.

From this list, Micrium's ACM subclass supports the following:

Table - ACM Requests Supported by Micrium

Subclass request	Description
SetCommFeature	The host sends this request to control the settings for a given communications feature. Not used for serial emulation.
GetCommFeature	The host sends this request to get the current settings for a given communications feature. Not used for serial emulation.
ClearCommFeature	The host sends this request to clear the settings for a given communications feature. Not used for serial emulation.
SetLineCoding	The host sends this request to configure the ACM device settings: baud rate, number of stop bits, parity type and number of data bits. For a serial emulation, this request is sent automatically by a serial terminal each time you configure the serial settings for an open virtual COM port.
GetLineCoding	The host sends this request to get the current ACM settings (baud rate, stop bits, parity, data bits). For a serial emulation, serial terminals send this request automatically during virtual COM port opening.
SetControlLineState	The host sends this request to control the carrier for half-duplex modems and indicate that Data Terminal Equipment (DTE) is ready or not. In the serial emulation case, the DTE is a serial terminal. For a serial emulation, certain serial terminals allow you to send this request with the controls set.
SetBreak	The host sends this request to generate an RS-232 style break. For a serial emulation, certain serial terminals allow you to send this request.

Micrium's ACM subclass uses the interrupt IN endpoint to notify the host about the current *serial line state*. The serial line state is a bitmap informing the host about:

- Data discarded because of overrun
- Parity error
- Framing error
- State of the ring signal detection
- State of break detection mechanism
- State of transmission carrier
- State of receiver carrier detection

Micrium's ACM subclass implementation complies with the following specification:

- *Universal Serial Bus, Communications, Subclass for PSTN Devices, revision 1.2, February 9, 2007.*

USB Device CDC ACM Class Configuration

This section discusses how to configure the CDC ACM Class (Communication Device Class, Abstract Control Model). There are two groups of configuration parameters:

- USB Device CDC ACM Class Run-Time Application Specific Configurations
- USB Device CDC ACM Class Instance Configurations

USB Device CDC ACM Class Run-Time Application Specific Configurations

- Initialization
  - CDC Base Class
    - p_qty_cfg
  - ACM Subclass
    - subclass_instance_qty
- Optional Configurations
  - CDC Base Class
    - Memory segment
  - ACM Subclass
    - Buffer Alignment
    - Memory Segments

Initialization

CDC Base Class

Before initializing the USB Device CDC ACM class module, you must first initialize the CDC base class by calling the function USBD_CDC_Init(). This function uses one configuration argument, as follows:

####### p_qty_cfg

p_cfg is a pointer to a configuration structure of type USBD_CDC_QTY_CFG. Its purpose is to inform the USB device module about how many USB CDC objects to allocate.

Table - USBD CDC QTY CFG configuration structure in the *USB Device CDC ACM Class Run-Time Application Specific Configurations* page describes each configuration field available in this configuration structure.

Table - USBD_CDC_QTY_CFG configuration structure

Field	Description
.ClassInstanceQty	Number of class instances you will allocate via a call to the function USBD_ACM_SerialAdd().
.ConfigQty	Number of configurations. ACM class instances can be added to one or more configurations via a call to the USBD_ACM_SerialConfigAdd().
.DataIF_Qty	Total number of data interfaces (DCI) for all the CDC functions. Each CDC ACM function added via a call to the function USBD_ACM_SerialAdd() will add a data interface.

ACM Subclass

Once the CDC base class is initialized, the ACM subclass can be initialized by calling the function USBD_ACM_SerialInit(). This function takes one configuration argument that is described as follows.

subclass_instance_qty

subclass_instance_qty configures the number of subclass instances you will allocate via a call to the function USBD_ACM_SerialAdd().

Optional Configurations

This section describes the configurations that are optional. If you do not set them in your application, the default configurations will apply.

CDC Base Class

The default values can be retrieved via the structure USBD_CDC_InitCfgDflt.

Note that these configurations must be set *before* you call the function USBD_CDC_Init().

Memory segment

This module allocates some control data. It has the ability to allocate this data from a specified memory segment.

Type	Function to call	Default	Field from default configuration structure
MEM_SEG*	USBD_CDC_ConfigureMemSeg()	General-purpose heap .	.MemSegPtr

ACM Subclass

The default values can be retrieved via the structure USBD_CDC_ACM_SerialInitCfgDflt.

Note: these configurations must be set *before* you call the function USBD_ACM_SerialInit().

Buffer Alignment

This module allocates buffers used for data transfers with the host. You may need address alignment for these buffers, depending on your USB controller. If you use more than one USB controller, you must set the alignment to the largest value.

Type	Function to call	Default	Field from default configuration structure
CPU_SIZE_T	USBD_ACM_SerialConfigureBufAlignOctets()	Size of cache line, or CPU alignment, if no cache.	.BufAlignOctets

Memory Segments

The USB Device module allocates control data and buffers used for data transfers with the host. It has the ability to use a different memory segment for the control data and for the data buffers.

Type	Function to call	Default	Field from default configuration structure
MEM_SEG*	USBD_ACM_SerialConfigureMemSeg()	General-purpose heap .	.MemSegPtr
MEM_SEG*	USBD_ACM_SerialConfigureMemSeg()	General-purpose heap .	.MemSegBufPtr

USB Device CDC ACM Class Instance Configurations

This section defines the configurations related to the CDC ACM serial class instances.

- Class Instance Creation
  - line state interval
  - call mgmt capabilities

Class Instance Creation

Creating a CDC ACM serial class instance is done by calling the function USBD_ACM_SerialAdd(). This function takes two configuration arguments that are described here.

line_state_interval

This is the interval (in milliseconds) that your CDC ACM serial class instance will report the line state notifications to the host. This value must be a power of two (1, 2, 4, 8, 16, etc).

call_mgmt_capabilities

Call Management Capabilities bitmap. Possible values of the bitmap are:

Value (bit)	Description
USBD_ACM_SERIAL_CALL_MGMT_DEV	Device handles call management itself.
USBD_ACM_SERIAL_CALL_MGMT_DATA_CCI_DCI	Device can send/receive call management information over a Data Class interface.

USB Device CDC ACM Class Programming Guide

This section explains how to use the CDC Abstract Control Model class.

- Initializing the USB Device CDC ACM Class
- Adding a USB Device CDC ACM Class Instance to your Device
- Communicating using the CDC ACM Class

Initializing the USB Device CDC ACM Class

To add CDC ACM class functionality to your device, you must first initialize the CDC base class and the ACM subclass by calling the functions USBD_CDC_Init() and USBD_ACM_SerialInit().

Listing - Example of initialization of CDC ACM class in the *Initializing the USB Device CDC ACM Class* page gives an example of a call to USBD_CDC_Init() and USBD_ACM_SerialInit() using default arguments. For more information about the configuration arguments to pass to these functions, see USB Device CDC ACM Class Run-Time Application Specific Configurations .

Listing - Example of initialization of CDC ACM class

```
RTOS_ERR         err;
USBD_CDC_QTY_CFG  cdc_qty_cfg;

cdc_qty_cfg.ClassInstanceQty = 1u;
cdc_qty_cfg.ConfigQty        = 2u;
cdc_qty_cfg.DataIF_Qty       = 1u;

USBD_CDC_Init(&cdc_qty_cfg,
              &err);
if (err.Code != RTOS_ERR_NONE) {
   /* An error occurred. Error handling should be added here. */
}

USBD_ACM_SerialInit(1u
                    &err);
if (err.Code != RTOS_ERR_NONE) {
   /* An error occurred. Error handling should be added here. */
}
```

Adding a USB Device CDC ACM Class Instance to your Device

To add CDC ACM class functionality to your device, you must create an instance, then add it to your device's configuration(s).

- Creating a CDC ACM Class Instance
- Adding the CDC ACM Class Instance to Your Device's Configuration(s)
- Registering Event Notification Callbacks

Creating a CDC ACM Class Instance

Create a CDC ACM class instance by calling the function USBD_ACM_SerialAdd().

Listing - Creating a CDC ACM function via USBD ACM SerialAdd() in the *Adding a USB Device CDC ACM Class Instance to your Device* page shows an example of how to create a CDC EEM class instance via USBD_CDC_EEM_Add().

Listing - Creating a CDC ACM function via USBD ACM SerialAdd()

```
CPU_INT08U  class_nbr;
RTOS_ERR    err;

class_nbr = USBD_ACM_SerialAdd(64u,
                               USBD_ACM_SERIAL_CALL_MGMT_DATA_CCI_DCI | USBD_ACM_SERIAL_CALL_MGMT_DEV
                               &err);
if (err.Code != RTOS_ERR_NONE) {
   /* An error occurred. Error handling should be added here. */
}
```

Adding the CDC ACM Class Instance to Your Device's Configuration(s)

Once you have created a CDC ACM class instance, you can add it to a configuration by calling the function USBD_ACM_SerialConfigAdd().

Listing - Example of call to USBD ACM SerialConfigAdd() in the *Adding a USB Device CDC ACM Class Instance to your Device* page shows an example of a call to USBD_ACM_SerialConfigAdd().

Listing - Example of call to USBD ACM SerialConfigAdd()

```
RTOS_ERR  err;

USBD_ACM_SerialConfigAdd(class_nbr,                    (1)
                         dev_nbr,                      (2)
                         config_nbr_fs,                (3)
                        &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}

USBD_ACM_SerialConfigAdd(class_nbr,                    (4)
                         dev_nbr,
                         config_nbr_hs,
                        &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

(1) Class number to add to the configuration returned by USBD_ACM_SerialAdd().

(2) Device number returned by USBD_DevAdd().

(3) Configuration number (here adding it to a Full-Speed configuration).

(4) Adding the same class instance to the High-Speed configuration. This should be done only if you have a High-Speed device.

Registering Event Notification Callbacks

The CDC ACM Serial class can notify your application of any changes in line control or coding via two notification callback functions: USBD ACM SerialLineCtrlReg() and USBD_ACM_SerialLineCodingReg(). Note that this step is optional. Listing - CDC ACM callbacks registration in the *Adding a USB Device CDC ACM Class Instance to your Device* page illustrates the use of the callback registration functions. Listing - CDC ACM callbacks implementation in the *Adding a USB Device CDC ACM Class Instance to your Device* page shows an example of implementation of the callback functions.

Listing - CDC ACM callbacks registration

```
USBD_ACM_SerialLineCodingReg(class_nbr,
                             App_USBD_ACM_SerialLinceCodingChngd,
                             DEF_NULL,
                            &err);

USBD_ACM_SerialLineCtrlReg(class_nbr,
                           App_USBD_ACM_SerialLineCtrlChngd,
                           DEF_NULL,
                          &err);
```

Listing - CDC ACM callbacks implementation

```
CPU_BOOLEAN  App_USBD_ACM_SerialLinceCodingChngd (CPU_INT08U                subclass_nbr,
                                                  USBD_ACM_SERIAL_LINE_CODING  *p_line_coding,
                                                  void                         *p_arg)
{
    CPU_INT32U  baudrate_new;
    CPU_INT08U  parity_new;
    CPU_INT08U  stop_bits_new;
    CPU_INT08U  data_bits_new;

    /* TODO: Apply new line coding.*/
    baudrate_new  = p_line_coding->BaudRate;
    parity_new    = p_line_coding->Parity;
    stop_bits_new = p_line_coding->StopBits;
    data_bits_new = p_line_coding->DataBits;

    return (DEF_OK);                                   (1)
}

void  App_USBD_ACM_SerialLineCtrlChngd (CPU_INT08U   subclass_nbr,
                                        CPU_INT08U   event,
                                        CPU_INT08U   event_chngd,
                                        void        *p_arg)
{
    CPU_BOOLEAN  rts_state;
    CPU_BOOLEAN  rts_state_chngd;
    CPU_BOOLEAN  dtr_state;
    CPU_BOOLEAN  dtr_state_chngd;
    CPU_BOOLEAN  brk_state;
    CPU_BOOLEAN  brk_state_chngd;

    /* TODO: Apply new line control.  */
    rts_state       = DEF_BIT_IS_SET(event,       USBD_ACM_SERIAL_CTRL_RTS);
    rts_state_chngd = DEF_BIT_IS_SET(event_chngd, USBD_ACM_SERIAL_CTRL_RTS);
    dtr_state       = DEF_BIT_IS_SET(event,       USBD_ACM_SERIAL_CTRL_DTR);
    dtr_state_chngd = DEF_BIT_IS_SET(event_chngd, USBD_ACM_SERIAL_CTRL_DTR);
    brk_state       = DEF_BIT_IS_SET(event,       USBD_ACM_SERIAL_CTRL_BREAK);
    brk_state_chngd = DEF_BIT_IS_SET(event_chngd, USBD_ACM_SERIAL_CTRL_BREAK);
}
```

(1) It is important to return DEF_FAIL to this function if the line coding applying failed. Otherwise, return DEF_OK.

Communicating using the CDC ACM Class

- Serial Status
  - Line Coding
  - Line Control
  - Line State
- Subclass Instance Communication

Serial Status

Line Coding

The USB host controls the line coding (baud rate, parity, etc) of the CDC ACM device. When necessary, the application is responsible for setting the line coding.

There are three functions provided to retrieve and set the current line coding. They are described in Table - CDC ACM Line Coding Functions in the *Communicating using the CDC ACM Class* page.

Table - CDC ACM Line Coding Functions

Function	Description
USBD_ACM_SerialLineCodingGet()	Your application can get the current line coding settings set either from the host with SetLineCoding requests or with the function USBD_ACM_SerialLineCodingSet().
USBD_ACM_SerialLineCodingSet()	Your application can set the line coding. The host can retrieve the settings with the GetLineCoding request.
USBD_ACM_SerialLineCodingReg()	Your application registers a callback called by the ACM subclass upon reception of the SetLineCoding request. Your application can perform any specific operations. For more information, see USB Device CDC ACM Class Instance Configurations .

Line Control

The USB host controls the line control (RTS and DTR pins, break signal, ...) of the CDC ACM device. W hen necessary, your application is responsible for applying the line controls.

There are two functions provided to retrieve and set the current line controls. They are described in Table - CDC ACM Line Control Functions in the *Communicating using the CDC ACM Class* page.

Table - CDC ACM Line Control Functions

Function	
USBD_ACM_SerialLineCtrlGet()	Your application can get the current control line state set by the host with the SetControlLineState request.
USBD_ACM_SerialLineCtrlReg()	Your application registers a callback called by the ACM subclass upon reception of the SendBreak or SetControlLineState requests. Your application can perform any specific operations. See USB Device CDC ACM Class Instance Configurations for more information.

Line State

The USB host retrieves the line state at a regular interval. Your application must update the line state each time it changes. W hen necessary, your application is responsible for setting the line state.

There are two functions provided to retrieve and set the current line controls. They are described in Table - CDC ACM Line State Functions in the *Communicating using the CDC ACM Class* page.

Table - CDC ACM Line State Functions

Function	
USBD_ACM_SerialLineStateSet()	Your application can set any line state event(s). While setting the line state, an interrupt IN transfer is sent to the host to inform about it a change in the serial line state.
USBD_ACM_SerialLineStateClr()	Application can clear two events of the line state: transmission carrier and receiver carrier detection. All the other events are self-cleared by the ACM serial emulation subclass.

Subclass Instance Communication

Micrium's ACM subclass offers the following functions to communicate with the host. For more details about the functions' parameters, see the CDC ACM Subclass Functions reference.

Function name	Operation
USBD_ACM_SerialRx()	Receives data from host through a bulk OUT endpoint. This function is blocking.
USBD_ACM_SerialTx()	Sends data to host through a bulk IN endpoint. This function is blocking.

Table - CDC ACM Communication API Summary

USBD_ACM_SerialRx() and USBD_ACM_SerialTx() provide synchronous communication, which means that the transfer is blocking. This means that, upon calling the function, the application blocks until the transfer is complete with or without an error. A timeout can be specified to avoid waiting forever. Listing - Serial Read and Write Example in the *Communicating using the CDC ACM Class* page shows a read and write example that receives data from the host using the bulk OUT endpoint and sends data to the host using the bulk IN endpoint.

Listing - Serial Read and Write Example

```
CPU_INT08U  rx_buf[2];
CPU_INT08U  tx_buf[2];
RTOS_ERR    err;

(void)USBD_ACM_SerialRx(subclass_nbr,                          (1)
                        rx_buf,                  (2)
                        2u,
                        0u,                      (3)
                       &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}

(void)USBD_ACM_SerialTx(subclass_nbr,                          (1)
                        tx_buf,                  (4)
                        2u,
                        0u,                      (3)
                       &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

(1) The class instance number created with USBD_ACM_SerialAdd() provides an internal reference to the ACM subclass to route the transfer to the proper bulk OUT or IN endpoint.

(2) Your application must ensure that the buffer provided to the function is large enough to accommodate all the data. Otherwise, synchronization issues might happen.

(3) To avoid an infinite blocking situation, specify a timeout expressed in milliseconds. A value of '0' makes the application task wait forever.

(4) The application provides the initialized transmit buffer.

USB Device CDC EEM Class

This section describes the Communication Device Class Ethernet Emulation Model subclass (CDC EEM) supported by Micrium OS USB Device. CDC EEM is a protocol that allows the usage of the USB as an Ethernet link. The device is seen by the host as a device on an Ethernet network, so all typical applications can be run on the device (FTP, HTTP, DHCP, etc.).

Microsoft Windows and Apple Mac OS do not provide any driver for CDC EEM devices, but commercial drivers can easily be found. Linux supports CDC EEM devices since kernel version 2.6.34.

CDC EEM represents the physical layer in the OSI model. It requires a network stack to implement higher layers. Micrium OS Network module offers an Ethernet driver for Micrium OS USB Device's CDC EEM subclass.

The CDC EEM implementation offered by Micrium OS USB Device is in compliance with the following specification:

- *Universal Serial Bus Communications Class Subclass Specification for Ethernet Emulation Model Devices*, Revision 1.0 February 2, 2005.

USB Device CDC EEM Class Overview

This page presents an overview of the CDC EEM protocol.

- Overview
- CDC EEM Messages
  - CDC EEM Message Format
  - CDC EEM Header Format
  - CDC EEM Data Header Format
  - CDC EEM Command Header Format

Overview

A CDC EEM device is composed of the following endpoints:

- A pair of Bulk IN and OUT endpoints.

Table - CDC EEM Subclass Endpoints Usage in the *USB Device CDC EEM Class Overview* page describes the usage of the different endpoints:

Table - CDC EEM Subclass Endpoints Usage

Endpoint	Direction	Usage
Bulk In	Device-to-host	Send Ethernet frames and commands to host.
Bulk OUT	Host-to-device	Receive Ethernet frames and commands from host.

CDC EEM Messages

CDC EEM defines a header that is prepended to each message sent to / received from the host. The CDC EEM header has a size of two bytes. A USB transfer on the Bulk endpoints can contain multiple EEM messages. An EEM message can also span on multiple USB transfers.

CDC EEM Message Format

Table - CDC EEM Message Format in the *USB Device CDC EEM Class Overview* page describes the content of a CDC EEM message.

Table - CDC EEM Message Format

Bytes	0..1	2..N
Content	Header	Payload (optional)

CDC EEM Header Format

Table - CDC EEM Header Format in the *USB Device CDC EEM Class Overview* page describes the content of a CDC EEM header.

Table - CDC EEM Header Format

Bit	15	14 .. 0
Content	bmType	Depends on bmType value

- bmType represents the message type, either a regular Ethernet frame or a CDC EEM specific command.

CDC EEM Data Header Format

Table - CDC EEM Data Header Format in the *USB Device CDC EEM Class Overview* page describes the content of a CDC EEM data message header.

Table - CDC EEM Data Header Format

Bit	15	14	13 .. 0
Content	bmType (0)	bmCRC	Length of Ethernet frame

- bmCRC indicates if the CRC was calculated on the Ethernet frame. If not, CRC is set to 0xDEADBEEF.

CDC EEM Command Header Format

Table - CDC EEM Command Header Format in the *USB Device CDC EEM Class Overview* page describes the content of a CDC EEM command message header. Note that the EEM commands provide USB local link management. This management is opaque to the network stack.

Table - CDC EEM Command Header Format

Bit	15	14	13 .. 11	10 .. 0
Content	bmType (1)	bmReserved (0)	bmEEMCmd	bmEEMCmdParam

- bmEEMCmd represents the command code to execute.
- bmEEMCmdParam contains command data. The format depends on the bmEEMCmd.

For more information on CDC EEM messages format, see "Universal Serial Bus Communications Class Subclass Specification for Ethernet Emulation Model Devices" revision 1.0. February 2, 2005, section 5.1.

USB Device CDC EEM Class Resource Needs from Core

Each time you add a CDC EEM class instance to a USB configuration (via a call to the function USBD_CDC_EEM_ConfigAdd()), the following resources will be allocated from the core.

Resource	Quantity
Interfaces	1
Alternate interfaces	1
Endpoint descriptors	2
Interface groups	0

USB Device CDC EEM Class Example Applications

This example creates and opens a Micrium OS Network Ethernet interface that uses the CDC EEM USB device class as a transport layer.

This example will allow you to accomplish the following tasks:

- Initialize the CDC EEM class
- Create and register a CDC EEM class instance
- Add the CDC EEM class instance to the Full-Speed configuration
- Add the CDC EEM class instance to the High-Speed configuration (if available)
- Add the new Ethernet Interface created from the CDC EEM class instance to Micrium OS Network
- Start the Ethernet interface

Location

The example implementation is located in /examples/usb/device/all/ex_usbd_cdc_eem_net.c.

Running the Demo Application

Currently, only the Linux operating system has built-in support for CDC EEM devices. Note that some third-party commercial drivers can be found for Microsoft Windows and Apple macOS.

Once connected to a Linux host, the operating system will add a new network interface called "usb*x*" (where *x* is a number starting from 0). The command ifconfig can be used from a terminal prompt to see the different network interfaces

available, and to set a static IP address to the "usb*x*" interface. This will be necessary if you don't have a DHCP server on your target.

Once done, you should be ready to perform ping commands from a terminal prompt and use any other application implemented on your device (HTTP, FTP, etc).

API

This example offers only one API named Ex_USBD_CDC_EEM_Init(). This function is normally called from a USB device core example.

USB Device CDC EEM Class Configuration

To configure the CDC EEM class, use the following configuration parameters:

- USB Device CDC EEM Class Run-time Application Specific Configurations

USB Device CDC EEM Class Run-time Application Specific Configurations

- Class Initialization
  - p_qty_cfg
- Optional Configurations
  - Buffer Alignment
  - Receive Buffers
  - Echo Buffer Length
  - Memory Segments

Class Initialization

To initialize the Micrium OS USB Device CDC EEM class, you call the function USBD_CDC_EEM_Init(). This function takes one configuration argument, which is described below.

p_qty_cfg

p_qty_cfgis a pointer to a configuration structure of type USBD_CDC_EEM_QTY_CFG. Its purpose is to inform the USB device module on how many USB CDC EEM objects to allocate.

Table - USBD_CDC_EEM_QTY_CFG configuration structure in the *USB Device CDC EEM Class Run-time Application Specific Configurations* page describes each configuration field available in this configuration structure.

Table - USBD CDC EEM QTY CFG configuration structure

Field	Description	Recommended value
.ClassInstanceQty	Number of class instances you will allocate via a call to the function USBD_CDC_EEM_Add().	1
.ConfigQty	Number of configurations. EEM class instances can be added to one or more configuration via a call to the function USBD_CDC_EEM_ConfigAdd().	2

Optional Configurations

This section describes the configurations that are optional. If you do not set them in your application, the default configurations will apply.

The default values can be retrieved via the structure USBD_CDC_EEM_InitCfgDflt.

Note: these configurations must be set *before* you call the function USBD_CDC_EEM_Init().

Buffer Alignment

This module allocates some buffers used for data transfers with the host. You may need a specific address alignment for these buffers depending on your USB controller. If you use more than one USB controller, you must set the alignment to the largest value.

Type	Function to call	Default	Field from default configuration structure
CPU_SIZE_T	USBD_CDC_EEM_ConfigureBufAlignOctets()	Size of cache line, or CPU alignment, if no cache.	.BufAlignOctets

Receive Buffers

Configures the quantity and the length, in octets, of the receive buffers for each CDC EEM function. The receive buffers must have a length of at least 1520 octets.

Unless your USB driver supports URB queuing, you must set the receive buffer quantity to 1.

Type	Function to call	Default	Field from default configuration structure
CPU_INT08U	USBD_CDC_EEM_ConfigureRxBuf()	1 buffer of 1520 octets.	.RxBufQty
CPU_INT32U	USBD_CDC_EEM_ConfigureRxBuf()	1 buffer of 1520 octets.	.RxBufLen

Echo Buffer Length

Configures the length, in octets, of the buffer used to handle the echo requests.

Type	Function to call	Default	Field from default configuration structure
CPU_INT16U	USBD_CDC_EEM_ConfigureEchoBufLen()	64 octets.	.EchoBufLen

Memory Segments

This module allocates some control data and buffers used for data transfers with the host. It has the ability to use a different memory segment for the control data and for the data buffers.

Type	Function to call	Default	Field from default configuration structure
MEM_SEG*	USBD_CDC_EEM_ConfigureMemSeg()	General-purpose heap .	.MemSegPtr
MEM_SEG*	USBD_CDC_EEM_ConfigureMemSeg()	General-purpose heap .	.MemSegBufPtr

USB Device CDC EEM Class Programming Guide

This section explains how to use the CDC EEM class.

- Initializing the USB Device CDC EEM Class
- Adding a USB Device CDC EEM Class Instance to your Device

Initializing the USB Device CDC EEM Class

To add CDC EEM class functionality to your device, you must first initialize the class by calling the function USBD_CDC_EEM_Init().

Listing - Example of call to USBD_CDC_EEM_Init() in the *Initializing the USB Device CDC EEM Class* page shows an example of a call to USBD_CDC_EEM_Init() using default arguments. For more information about the configuration arguments to pass to USBD_CDC_EEM_Init(), see USB Device CDC EEM Class Run-time Application Specific Configurations .

Listing - Example of call to USBD CDC EEM Init()

```
RTOS_ERR            err;
USBD_CDC_EEM_QTY_CFG  eem_qty_cfg;

eem_qty_cfg.ClassInstanceQty = 1u;
```

```
eem_qty_cfg.ClassInstanceQty =1u;
eem_qty_cfg.ConfigQty      =2u;USBD_CDC_EEM_Init(&eem_qty_cfg,&err);if(err.Code != RTOS_ERR_NONE){/* An error occurred. Error handling should be added here. */}
```

Adding a USB Device CDC EEM Class Instance to your Device

To add CDC EEM class functionality to your device, you must create an instance, then add it to your device's configuration(s).

- Creating a CDC EEM Class Instance
- Adding the CDC EEM Class Instance to Your Device's Configuration(s)

Creating a CDC EEM Class Instance

There are two ways to create a CDC EEM class instance. This can be done by either calling the function USBD_CDC_EEM_Add() or USBD_CDC_EEM_NetIF_Reg(). The function USBD_CDC_EEM_Add() will simply create a CDC EEM class instance and provide the class instance number.

On the other hand, the function USBD_CDC_EEM_NetIF_Reg() will create the CDC EEM class instance, add the network interface to the Platform Manager , and provide the interface name to the caller. The name allows you to add the network interface to Micrium OS Network (using the function NetIF_Ether_Add()) afterward.

The recommended method is to add a CDC EEM class instance (the explanation follows below). You must have the Micrium OS Network Ethernet module in your project to enable this function.

Listing - Creating a CDC EEM function via USBD_CDC_EEM_NetIF_Reg() in the *Adding a USB Device CDC EEM Class Instance to your Device* page shows an example of how to create a CDC EEM class instance via USBD_CDC_EEM_NetIF_Reg().

Listing - Creating a CDC EEM function via USBD CDC EEM NetIF Reg()

```
CPU_INT08U                  class_nbr;
CPU_CHAR                   *net_if_name;
USBD_CDC_EEM_NET_IF_ETHER_CFG   net_if_ether_cfg;
RTOS_ERR                    err;

net_if_ether_cfg.RxBufQty      = 5u;
net_if_ether_cfg.TxBufQty      = 5u;
net_if_ether_cfg.TxBufSmallLen = 128u;
net_if_ether_cfg.TxBufSmallQty = 2u;

Mem_Copy(net_if_ether_cfg.HW_AddrStr,
         "12-34-56-78-9A-BC",
         NET_IF_802x_ADDR_SIZE_STR);

class_nbr = USBD_CDC_EEM_NetIF_Reg(&net_if_name,
                                   &net_if_ether_cfg,
                                   &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

Adding the CDC EEM Class Instance to Your Device's Configuration(s)

Once you have created a CDC EEM class instance, you can add it to a configuration by calling the function USBD_CDC_EEM_ConfigAdd().

Listing - Example of call to USBD_CDC_EEM_ConfigAdd() in the *Adding a USB Device CDC EEM Class Instance to your Device* page shows an example of call to USBD_CDC_EEM_ConfigAdd().

Listing - Example of call to USBD CDC EEM ConfigAdd()

```
RTOS_ERR  err;

USBD_CDC_EEM_ConfigAdd(class_nbr,                    (1)
                      dev_nbr,                      (2)
                      config_nbr_fs,                (3)
                     "CDC EEM Full-Speed Interface",
                     &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}

USBD_CDC_EEM_ConfigAdd(class_nbr,                    (4)
                      dev_nbr,
                      config_nbr_hs,
                     "CDC EEM High-Speed Interface",
                     &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

(1) Class number to add to the configuration returned by USBD_CDC_EEM_Add().

(2) Device number returned by USBD_DevAdd().

(3) Configuration number (here adding it to a Full-Speed configuration).

(4) Adding the same class instance to the High-Speed configuration. This should be done only if you have a High-Speed device.

USB Device HID Class

This section describes the Human Interface Device (HID) class supported by Micrium OS USB Device.

The HID class encompasses devices used by humans to control computer operations: for example, keyboards, mice, pointing devices, and game devices.

The HID class can also be used in a composite device that contains controls such as knobs, switches, buttons, and sliders. For instance, mute and volume controls in an audio headset are controlled by the HID function of the headset. HID class can exchange data for any purpose using only control and interrupt transfers.

The HID class is one of the oldest and most widely-used USB classes. All the major host operating systems provide a native driver to manage HID devices, which is why a variety of vendor-specific devices work with the HID class. This class also includes various types of output items such as LEDs, audio, tactile feedback, etc.

The HID implementation complies with the following specifications:

- *Device Class Definition for Human Interface Devices (HID), 6/27/01, Version 1.11.*
- *Universal Serial Bus HID Usage Tables, 10/28/2004, Version 1.12.*

USB Device HID Class Overview

Overview

A HID device is composed of the following endpoints:

- A pair of control IN and OUT endpoints called the default endpoint.
- An interrupt IN endpoint.
- An optional interrupt OUT endpoint.

Table - HID Class Endpoints Usage in the *USB Device HID Class Overview* page describes the usage of the different endpoints:

Table - HID Class Endpoints Usage

Endpoint	Direction	Usage
Control IN	Device-to-host	Standard requests for enumeration, class-specific requests, and data communication (Input, Feature reports sent to the host with GET_REPORT request).
Control OUT	Host-to-device	Standard requests for enumeration, class-specific requests and data communication (Output, Feature reports received from the host with SET_REPORT request).
Interrupt IN	Device-to-host	Data communication (Input and Feature reports).
Interrupt OUT	Host-to-device	Data communication (Output and Feature reports).

Report

A host and a HID device exchange data using reports. A report contains formatted data giving information about controls and other physical entities of the HID device. A control is manipulable by the user and operates an aspect of the device. For instance, a control can be a button on a mouse or a keyboard, a switch, etc. Other entities inform the user about the state of certain device's features. For instance, LEDs on a keyboard notify the user about the caps lock on, the numeric keypad active, etc.

The format and the use of a report data is understood by the host by analyzing the content of a *Report descriptor*. Analyzing the content is done by a parser. The Report descriptor describes the data provided by each control in a device. It is composed of *items* which are pieces of information about the device and consist of a 1-byte prefix and variable-length data. For more details about the item format, refer to "*Device Class Definition for Human Interface Devices (HID) Version 1.11*", section 5.6 and 6.2.2.

There are three principal types of items:

- *Main item* defines or groups certain types of data fields.
- *Global item* describes data characteristics of a control.
- *Local item* describes data characteristics of a control.

Each item type is defined by different functions. An item function can also be called a tag. An item function can be seen as a sub-item that belongs to one of the three principal item types. Table - Item's Function Description for each Item Type in the *USB Device HID Class Overview* page gives a brief overview of the item's functions in each item type. For a complete description of the items in each category, refer to "Device Class Definition for Human Interface Devices (HID) Version 1.11", section 6.2.2.

Table - Item's Function Description for each Item Type

Item type	Item function	Description
Main	Input	Describes information about the data provided by one or more physical controls.
Main	Output	Describes data sent to the device.
Main	Feature	Describes device configuration information sent to or received from the device which influences the overall behavior of the device or one of its components.
Main	Collection	Group related items (Input, Output or Feature).
Main	End of Collection	Closes a collection.

Item type	Item function	Description
Global	Usage Page	Identifies a function available within the device.
Global	Logical Minimum	Defines the lower limit of the reported values in logical units.
Global	Logical Maximum	Defines the upper limit of the reported values in logical units.
Global	Physical Minimum	Defines the lower limit of the reported values in physical units, that is the Logical Minimum expressed in physical units.
Global	Physical Maximum	Defines the upper limit of the reported values in physical units, that is the Logical Maximum expressed in physical units.
Global	Unit Exponent	Indicates the unit exponent in base 10. The exponent ranges from -8 to +7.
Global	Unit	Indicates the unit of the reported values. For instance, length, mass, temperature units, etc.
Global	Report Size	Indicates the size of the report fields in bits.
Global	Report ID	Indicates the prefix added to a particular report.
Global	Report Count	Indicates the number of data fields for an item.
Global	Push	Places a copy of the global item state table on the CPU stack.
Global	Pop	Replaces the item state table with the last structure from the stack.
Local	Usage	Represents an index to designate a specific Usage within a Usage Page. It indicates the vendor's suggested use for a specific control or group of controls. A usage supplies information to an application developer about what a control is actually measuring.
Local	Usage Minimum	Defines the starting usage associated with an array or bitmap.
Local	Usage Maximum	Defines the ending usage associated with an array or bitmap.
Local	Designator Index	Determines the body part used for a control. Index points to a designator in the Physical descriptor.
Local	Designator Minimum	Defines the index of the starting designator associated with an array or bitmap.
Local	Designator Maximum	Defines the index of the ending designator associated with an array or bitmap.
Local	String Index	String index for a String descriptor. It allows a string to be associated with a particular item or control.
Local	String Minimum	Specifies the first string index when assigning a group of sequential strings to controls in an array or bitmap.
Local	String Maximum	Specifies the last string index when assigning a group of sequential strings to controls in an array or bitmap.
Local	Delimiter	Defines the beginning or end of a set of local items.

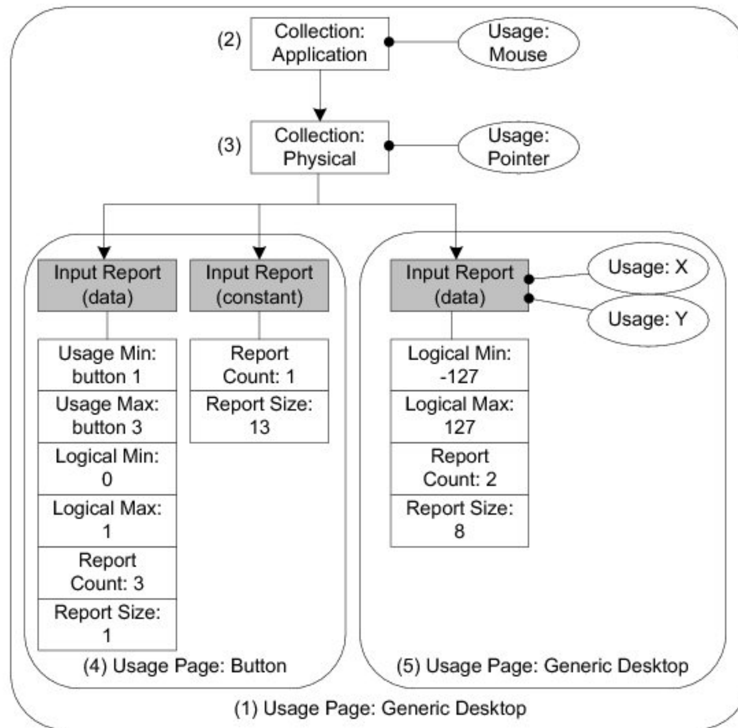
A control's data must define at least the following items:

- Input, Output or Feature Main items.
- Usage Local item.
- Usage Page Global item.
- Logical Minimum Global item.
- Logical Maximum Global item.
- Report Size Global item.

Report Count Global item.

Table - HID Class Endpoints Usage in the *USB Device HID Class Overview* page shows the representation of a Mouse Report descriptor content from a host HID parser perspective. The mouse has three buttons (left, right and wheel). The code presented in Listing - Mouse Report Descriptor Example in the *USB Device HID Class Configuration* page is an example of code implementation corresponding to this mouse Report descriptor representation.

Figure - Report Descriptor Content from a Host HID Parser View



(1) The *Usage Page* item function specifies the general function of the device. In this example, the HID device belongs to a generic desktop control.

(2) The *Collection Application* groups Main items that have a common purpose and may be familiar to applications. In the diagram, the group is composed of three Input Main items. For this collection, the suggested use for the controls is a mouse as indicated by the *Usage* item.

(3) Nested collections may be used to give more details about the use of a single control or group of controls to applications. In this example, the Collection Physical, nested into the Collection Application, is composed of the same three Input items forming the Collection Application. The *Collection Physical* is used for a set of data items that represent data points collected at one geometric point. In the example, the suggested use is a pointer as indicated by the Usage item. Here the pointer usage refers to the mouse position coordinates and the system software will translate the mouse coordinates in movement of the screen cursor.

(4) Nested usage pages are also possible and give more details about a certain aspect within the general function of the device. In this case, two Inputs items are grouped and correspond to the buttons of the mouse. One Input item defines the three buttons of the mouse (right, left and wheel) in terms of number of data fields for the item (*Report Count* item), size of a data field (*Report Size* item) and possible values for each data field (*Usage Minimum* and *Maximum*, *Logical Minimum* and *Maximum* items). The other Input item is a 13-bit constant allowing the Input report data to be aligned on a byte boundary. This Input item is used only for padding purpose.

(5) Another nested usage page referring to a generic desktop control is defined for the mouse position coordinates. For this usage page, the Input item describes the data fields corresponding to the x- and y-axis as specified by the two Usage items.

After analyzing the previous mouse Report descriptor content, the host's HID parser is able to interpret the Input report data sent by the device with an interrupt IN transfer or in response to a GET_REPORT request. The Input report data

corresponding to the mouse Report descriptor shown in Figure - Report Descriptor Content from a Host HID Parser View in the *USB Device HID Class Overview* page is presented in Table - Input Report Sent to Host and Corresponding to the State of a 3-Buttons Mouse in the *USB Device HID Class Overview* page. The total size of the report data is 4 bytes. Different types of reports may be sent over the same endpoint. For the purpose of distinguishing the different types of reports, a 1-byte report ID prefix is added to the data report. If a report ID was used in the example of the mouse report, the total size of the report data would be 5 bytes.

Table - Input Report Sent to Host and Corresponding to the State of a 3-Buttons Mouse

Bit offset	Bit count	Description
0	1	Button 1 (left button).
1	1	Button 2 (right button).
2	1	Button 3 (wheel button).
3	13	Not used.
16	8	Position on axis X.
24	8	Position on axis Y.

A Physical descriptor indicates the part or parts of the body intended to activate a control or controls. An application may use this information to assign a functionality to the control of a device. A Physical descriptor is an optional class-specific descriptor and most devices have little gain for using it. Refer to “Device Class Definition for Human Interface Devices (HID) Version 1.11” section 6.2.3 for more details about this descriptor.

USB Device HID Class Resource Needs from Core

Each time you add a HID class instance to a USB configuration via a call to the function USBD_HID_ConfigAdd(), the following resources will be allocated from the core.

Resource	Quantity
Interfaces	1
Alternate interfaces	1
Endpoint descriptors	1 (2 if interrupt OUT endpoint is enabled)
Interface groups	0

USB Device HID Class Example Applications

HID Class Mouse Example

This example emulates a mouse moving back and forth.

This example will allow you to accomplish the following tasks:

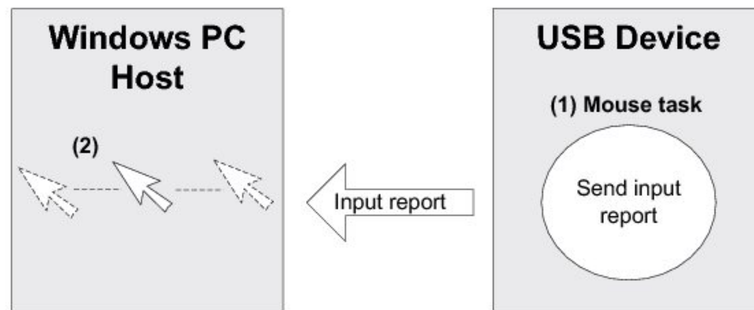
- Initialize the HID class
- Create a HID class instance
- Add the HID class instance to the Full-Speed configuration
- Add the HID class instance to the High-Speed configuration (if available)
- Create a kernel task that emulates the mouse movement

Running the Demo Application

The mouse demo does not require anything extra on the PC. You just need to plug the HID device running the mouse demo to the PC and see the screen cursor moving.

Figure - HID Mouse Demo in the *USB Device HID Class Example Applications* page illustrates the mouse demo with the host and device interactions:

Figure - HID Mouse Demo



(1) On the device side, the task Ex_USBD_HID_MouseTask() simulates a mouse movement by setting the coordinates X and Y to a certain value and by sending an Input report that contains these coordinates. The Input report is sent by calling the USBD_HID_Wr() function through the interrupt IN endpoint. The mouse demo does not simulate any button clicks; only mouse movement.

(2) The host PC polls the HID device periodically following the polling interval of the interrupt IN endpoint. The polling interval is specified in the Endpoint descriptor matching to the interrupt IN endpoint. The host receives and interprets the Input report content. The simulated mouse movement is translated into a movement of the screen cursor. While the device side application is running, the screen cursor moves endlessly.

API

This example makes use of only one API call named Ex_USBD_HID_Init(). This function is normally called from a USB device core example.

USB Device HID Class Configuration

To configure the HID class, there are two groups of configuration parameters:

- USB Device HID Class Run-Time Application Specific Configurations
- USB Device HID Class Instance Configurations

USB Device HID Class Run-Time Application Specific Configurations

- Class Initialization
  - p_qty_cfg
- Optional Configurations
  - Buffer alignment
  - Quantity of report ID
  - Quantity of Push/Pop items
  - Timer task stack
  - Memory segments
- Post-Init Configurations
  - Timer task priority

Class Initialization

To initialize the Micrium OS USB Device HID class module, you call the USBD_HID_Init() function. This function takes one configuration argument which is described below.

p_qty_cfg

p_qty_cfg is a pointer to a configuration structure of type USBD_HID_QTY_CFG. Its purpose is to inform the USB device module about how many USB HID objects to allocate.

Table - USBD_HID_QTY_CFG configuration structure in the *USB Device HID Class Run-Time Application Specific Configurations* page describes each configuration field available in this configuration structure.

Table - USBD_HID_QTY_CFG configuration structure

Field	Description	Recommended Value
Field	Description	Recommended Value
.ClassInstanceQty	Number of class instances you will allocate via a call to the function USBD_HID_Add().	1
.ConfigQty	Number of configurations. HID class instances can be added to one or more configurations via a call to the function USBD_HID_ConfigAdd() .	2

Optional Configurations

This section describes the configurations that are optional. If you do not set them in your application, the default configurations will apply.

The default values can be retrieved via the structure USBD_HID_InitCfgDflt.

> Note that these configurations must be set *before* you call the function USBD_HID_Init().

Buffer alignment

This module allocates buffers used for data transfers with the host. You may need a specific address alignment for these buffers depending on your USB controller. If you use more than one USB controller, you must set the alignment to the highest value.

Type	Function to Call	Default	Field from Default Configuration Structure
CPU_SIZE_T	USBD_HID_ConfigureBufAlignOctets()	Size of cache line, or CPU alignment, if no cache.	.BufAlignOctets

Quantity of report ID

Configures the total number of report IDs to allocate.

Type	Function to Call	Default	Field from Default Configuration Structure
CPU_INT08U	USBD_HID_ConfigureReportID_Qty()	2	.ReportID_Qty

Quantity of Push/Pop items

Configures the total number of Push/Pop items to allocate.

Type	Function to Call	Default	Field from Default Configuration Structure
CPU_INT08U	USBD_HID_ConfigurePushPopItemsQty()	0	.PushPopItemsQty

Timer task stack

The timer task handles all the timer-based HID operations. This configuration allows you to set the stack pointer and the stack size (in number of elements).

Type	Function to Call	Default	Field from Default Configuration Structure
CPU_INT32U *	USBD_HID_ConfigureTmrTaskStk()	A stack of 512 elements allocated on Common 's memory segment.	.TmrTaskStkSizeElements
void *	USBD_HID_ConfigureTmrTaskStk()	A stack of 512 elements allocated on Common 's memory segment.	.TmrTaskStkPtr

Memory segments

This module allocates control data and buffers used for data transfers with the host. It has the ability to use a different memory segment for the control data and for the data buffers.

Type	Function to Call	Default	Field from Default Configuration Structure
MEM_SEG*	USBD_HID_ConfigureMemSeg()	General-purpose heap .	.MemSegPtr
MEM_SEG*	USBD_HID_ConfigureMemSeg()	General-purpose heap .	.MemSegBufPtr

Post-Init Configurations

This section describes the configurations that can be set at any time during execution *after* you have called the function USBD_HID_Init().

These configurations are optional. If you do not set them in your application, the default configurations will apply.

Timer task priority

The timer task is created when the HID class is initialized. You can change the priority of the task at any time.

Type	Function to call	Default
RTOS_TASK_PRIO	USBD_HID_TmrTaskPrioSet()	See Appendix A - Internal Tasks .

USB Device HID Class Instance Configurations

- Class Instance Creation
  - subclass
  - protocol
  - country_code
  - p_report_desc_and_report_desc_len
  - p_phy_desc and phy_desc_len
  - interval_in and interval_out
  - p_hid_callback

This section defines the configurations related to the HID class instances.

Class Instance Creation

Creating a HID class instance is done by calling the function USBD_HID_Add() . This function takes several configuration arguments that are described below.

subclass

Code of the HID subclass. Possible values are:

- USBD_HID_SUBCLASS_NONE
- USBD_HID_SUBCLASS_BOOT

A HID device that uses the boot subclass must use standard report formats. For more information on the subclass codes, see section 4.2 of HID specification revision 1.11.

protocol

Protocol used by the HID device. Possible values are:

- USBD_HID_PROTOCOL_NONE
- USBD_HID_PROTOCOL_KBD
- USBD_HID_PROTOCOL_MOUSE

If your HID function is a mouse, the protocol should be set to USBD_HID_PROTOCOL_MOUSE. If it is a keyboard, it should be set to USBD_HID_PROTOCOL_KBD . Otherwise, the protocol should be set to USBD_HID_PROTOCOL_NONE. For more

information on the subclass codes, see section 4.3 of HID specification revision 1.11.

country_code

ID of the country code. Possible values are:

- USBD_HID_COUNTRY_CODE_NOT_SUPPORTED
- USBD_HID_COUNTRY_CODE_ARABIC
- USBD_HID_COUNTRY_CODE_BELGIAN
- USBD_HID_COUNTRY_CODE_CANADIAN_BILINGUAL
- USBD_HID_COUNTRY_CODE_CANADIAN_FRENCH
- USBD_HID_COUNTRY_CODE_CZECH_REPUBLIC
- USBD_HID_COUNTRY_CODE_DANISH
- USBD_HID_COUNTRY_CODE_FINNISH
- USBD_HID_COUNTRY_CODE_FRENCH
- USBD_HID_COUNTRY_CODE_GERMAN
- USBD_HID_COUNTRY_CODE_GREEK
- USBD_HID_COUNTRY_CODE_HEBREW
- USBD_HID_COUNTRY_CODE_HUNGARY
- USBD_HID_COUNTRY_CODE_INTERNATIONAL
- USBD_HID_COUNTRY_CODE_ITALIAN
- USBD_HID_COUNTRY_CODE_JAPAN_KATAKANA
- USBD_HID_COUNTRY_CODE_KOREAN
- USBD_HID_COUNTRY_CODE_LATIN_AMERICAN
- USBD_HID_COUNTRY_CODE_NETHERLANDS_DUTCH
- USBD_HID_COUNTRY_CODE_NORWEGIAN
- USBD_HID_COUNTRY_CODE_PERSIAN_FARSI
- USBD_HID_COUNTRY_CODE_POLAND
- USBD_HID_COUNTRY_CODE_PORTUGUESE
- USBD_HID_COUNTRY_CODE_RUSSIA
- USBD_HID_COUNTRY_CODE_SLOVAKIA
- USBD_HID_COUNTRY_CODE_SPANISH
- USBD_HID_COUNTRY_CODE_SWEDISH
- USBD_HID_COUNTRY_CODE_SWISS_FRENCH
- USBD_HID_COUNTRY_CODE_SWISS_GERMAN
- USBD_HID_COUNTRY_CODE_SWITZERLAND
- USBD_HID_COUNTRY_CODE_TAIWAN
- USBD_HID_COUNTRY_CODE_TURKISH_Q
- USBD_HID_COUNTRY_CODE_UK
- USBD_HID_COUNTRY_CODE_US
- USBD_HID_COUNTRY_CODE_YUGOSLAVIA
- USBD_HID_COUNTRY_CODE_TURKISH_F

The country code identifies which country the hardware is localized for. Most hardware is not localized and therefore this value would be USBD_HID_COUNTRY_CODE_NOT_SUPPORTED (0). However, keyboards may use the field to indicate the language of the key caps.

For more information on the country codes, see section 6.2.1 of HID specification revision 1.11.

p_report_desc and report_desc_len

p_report_desc is a pointer to a buffer that contains the report descriptor and report_desc_len is the length of the report descriptor buffer, in octets.

For each of your HID functions, you must provide a report descriptor. The report descriptor indicates to the host how the periodic report that will be sent by the device should be parsed. Writing your own report descriptor can be challenging, and that is why there are some resources to help.

HID Class Example

Micrium's HID class application example provides an example of a report descriptor for a simple mouse. Listing - Mouse Report Descriptor Example in the *USB Device HID Class Instance Configurations* page shows the example mouse report

descriptor.

Listing - Mouse Report Descriptor Example

```
static  CPU_INT08U  App_USBD_HID_ReportDesc[] = {                    (1) (2)
 USBD_HID_GLOBAL_USAGE_PAGE       + 1, USBD_HID_USAGE_PAGE_GENERIC_DESKTOP_CONTROLS,
 USBD_HID_LOCAL_USAGE            + 1, USBD_HID_CA_MOUSE,              (3)
 USBD_HID_MAIN_COLLECTION         + 1, USBD_HID_COLLECTION_APPLICATION,    (4)
  USBD_HID_LOCAL_USAGE           + 1, USBD_HID_CP_POINTER,            (5)
  USBD_HID_MAIN_COLLECTION        + 1, USBD_HID_COLLECTION_PHYSICAL,      (6)
   USBD_HID_GLOBAL_USAGE_PAGE     + 1, USBD_HID_USAGE_PAGE_BUTTON,         (7)
   USBD_HID_LOCAL_USAGE_MIN       + 1, 0x01,
   USBD_HID_LOCAL_USAGE_MAX       + 1, 0x03,
   USBD_HID_GLOBAL_LOG_MIN        + 1, 0x00,
   USBD_HID_GLOBAL_LOG_MAX        + 1, 0x01,
   USBD_HID_GLOBAL_REPORT_COUNT   + 1, 0x03,
   USBD_HID_GLOBAL_REPORT_SIZE    + 1, 0x01,
   USBD_HID_MAIN_INPUT            + 1, USBD_HID_MAIN_DATA     |
                                  USBD_HID_MAIN_VARIABLE |
                                  USBD_HID_MAIN_ABSOLUTE,
   USBD_HID_GLOBAL_REPORT_COUNT   + 1, 0x01,                          (8)
   USBD_HID_GLOBAL_REPORT_SIZE    + 1, 0x0D,
   USBD_HID_MAIN_INPUT            + 1, USBD_HID_MAIN_CONSTANT,
                                                   (9)
   USBD_HID_GLOBAL_USAGE_PAGE     + 1, USBD_HID_USAGE_PAGE_GENERIC_DESKTOP_CONTROLS,
   USBD_HID_LOCAL_USAGE           + 1, USBD_HID_DV_X,
   USBD_HID_LOCAL_USAGE           + 1, USBD_HID_DV_Y,
   USBD_HID_GLOBAL_LOG_MIN        + 1, 0x81,
   USBD_HID_GLOBAL_LOG_MAX        + 1, 0x7F,
   USBD_HID_GLOBAL_REPORT_SIZE    + 1, 0x08,
   USBD_HID_GLOBAL_REPORT_COUNT   + 1, 0x02,
   USBD_HID_MAIN_INPUT            + 1, USBD_HID_MAIN_DATA     |
                                  USBD_HID_MAIN_VARIABLE |
                                  USBD_HID_MAIN_RELATIVE,
  USBD_HID_MAIN_ENDCOLLECTION,                                   (10)
 USBD_HID_MAIN_ENDCOLLECTION                                     (11)
};
```

(1) The table representing a mouse report descriptor is initialized in such way that each line corresponds to a short item. The latter is formed from a 1-byte prefix and a 1-byte data. Refer to “Device Class Definition for Human Interface Devices (HID) Version 1.11”, sections 5.3 and 6.2.2.2 for more details about short items format. This table content corresponds to the mouse report descriptor content viewed by a host HID parser in Figure - Report Descriptor Content from a Host HID Parser View in the *USB Device HID Class Overview* page.

(2) The Generic Desktop Usage Page is used.

(3) Within the Generic Desktop Usage Page, the usage tag suggests that the group of controls is for controlling a mouse. A mouse collection typically consists of two axes (X and Y) and one, two, or three buttons.

(4) The mouse collection is started.

(5) Within the mouse collection, a usage tag suggests more specifically that the mouse controls belong to the pointer collection. A pointer collection is a collection of axes that generates a value to direct, indicate, or point user intentions to an application.

(6) The pointer collection is started.

(7) The Buttons Usage Page defines an Input item composed of three 1-bit fields. Each 1-bit field represents the mouse’s button 1, 2 and 3 respectively and can return a value of 0 or 1.

(8) The Input Item for the Buttons Usage Page is padded with 13 other bits.

(9) Another Generic Desktop Usage Page is indicated for describing the mouse position with the axes X and Y. The Input item is composed of two 8-bit fields whose value can be between -127 and 127.

(10) The pointer collection is closed.

(11) The mouse collection is closed.

USB.org HID page

The USB Implementers Forum (USB-IF) provides a tool called "HID Descriptor Tool" along with other information on the report descriptor format. Seehttp://www.usb.org/developers/hidpage/ for more information.

p_phy_desc and phy_desc_len

p_phy_desc is a pointer to a buffer that contains the physical descriptor and phy_desc_len is the length of the physical descriptor buffer, in octets.

The physical descriptor is a descriptor that provides information about the specific part or parts of the human body that are activating a control or controls. For more information on physical descriptors, see section 6.2.3 of HID specification revision 1.11.

The physical descriptor is optional and most of the time ignored. The buffer passed here can be set to DEF_NULL and the length set to 0.

interval_in and interval_out

interval_in and interval_out represent the polling interval of the IN interrupt endpoint and the OUT interrupt endpoint.

This represents the polling interval of the endpoint, in milliseconds. Setting this value depends on how frequently your device is susceptible to generate a new report for the host. For instance, if a report is generated every 16 milliseconds, the interval should be 16 or less.

The value must be a power of 2 (1, 2, 4, 8, 16, etc.).

interval_out value is ignored if ctrl_rd_en is set to DEF_ENABLED.

p_hid_callback

p_hid_callback is a pointer to a structure of type USBD_HID_CALLBACK . Its purpose is to give the HID Class a set of optional callback functions to be called when a HID event occurs.

A null pointer (DEF_NULL) can be passed to this argument if no callbacks are needed.

Table Table - USBD_HID_CALLBACK configuration structure in the *USB Device HID Class Instance Configurations* page describes each configuration field available in this configuration structure.

Table - USBD_HID_CALLBACK configuration structure

Field	Description	Function signature
.FeatureReportGet	Called when the host requests a feature report as described in your report descriptor.	`CPU_BOOLEAN FeatureReportGet(CPU_INT08U class_nbr, CPU_INT08U report_id, CPU_INT08U *p_report_buf, CPU_INT16U report_len);`
.FeatureReportSet	Called when the host sets a feature report as described in your report descriptor.	`CPU_BOOLEAN FeatureReportSet(CPU_INT08U class_nbr, CPU_INT08U report_id, CPU_INT08U *p_report_buf, CPU_INT16U report_len);`
.ProtocolGet	Retrieves current active protocol.	`CPU_INT08U ProtocolGet(CPU_INT08U class_nbr, RTOS_ERR *p_err);`
.ProtocolSet	Sets current active protocol.	`void ProtocolSet(CPU_INT08U class_nbr, CPU_INT08U protocol, RTOS_ERR *p_err);`
.ReportSet	Called when the host sets a report as described in your report descriptor (when it sends a report).	`void ReportSet(CPU_INT08U class_nbr, CPU_INT08U report_id, CPU_INT08U *p_report_buf, CPU_INT16U report_len);`

USB Device HID Class Programming Guide

This section explains how to use the HID class.

- Initializing the USB Device HID Class
- Adding a USB Device HID Class Instance to your Device
- Communicating using the USB Device HID Class

Initializing the USB Device HID Class

To add HID Class functionality to your device, you must first initialize the class by calling the function USBD_HID_Init().

Listing - Example of call to USBD_HID_Init() in the *Initializing the USB Device HID Class* page shows an example of a call to USBD_HID_Init() using default arguments. For more information on the configuration arguments to pass to USBD_HID_Init(), see USB Device HID Class Run-Time Application Specific Configurations .

Listing - Example of call to USBD_HID_Init()

```
RTOS_ERR         err;
USBD_HID_QTY_CFG  hid_qty_cfg;

hid_qty_cfg.ClassInstanceQty = 1u;
hid_qty_cfg.ConfigQty        = 2u;

USBD_HID_Init(&hid_qty_cfg,
              &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occured. Error handling should be added here. */
}
```

Adding a USB Device HID Class Instance to your Device

To add HID class functionality to your device, you must create an instance, then add it to your device's configuration(s).

Creating a HID class Instance

Create a HID class instance by calling the function USBD_HID_Add().

Listing - Adding a mouse function via USBD_HID_Add() in the *Adding a USB Device HID Class Instance to your Device* page shows an example of how to create a simple mouse function via USBD_HID_Add() using default arguments. For more information on the configuration arguments to pass to USBD_HID_Add(), see USB Device HID Class Instance Configurations .

Listing - Adding a mouse function via USBD_HID_Add()

```
/* Global constants.  */
static  const  CPU_INT08U  Ex_USBD_HID_Mouse_ReportDesc[] = {
   USBD_HID_GLOBAL_USAGE_PAGE           + 1, USBD_HID_USAGE_PAGE_GENERIC_DESKTOP_CONTROLS,
   USBD_HID_LOCAL_USAGE                 + 1, USBD_HID_CA_MOUSE,
   USBD_HID_MAIN_COLLECTION             + 1, USBD_HID_COLLECTION_APPLICATION,
      USBD_HID_LOCAL_USAGE              + 1, USBD_HID_CP_POINTER,
      USBD_HID_MAIN_COLLECTION          + 1, USBD_HID_COLLECTION_PHYSICAL,
         USBD_HID_GLOBAL_USAGE_PAGE   + 1, USBD_HID_USAGE_PAGE_BUTTON,
         USBD_HID_LOCAL_USAGE_MIN     + 1, 0x01,
         USBD_HID_LOCAL_USAGE_MAX     + 1, 0x03,
         USBD_HID_GLOBAL_LOG_MIN      + 1, 0x00,
         USBD_HID_GLOBAL_LOG_MAX      + 1, 0x01,
         USBD_HID_GLOBAL_REPORT_COUNT + 1, 0x03,
         USBD_HID_GLOBAL_REPORT_SIZE  + 1, 0x01,
         USBD_HID_MAIN_INPUT          + 1, USBD_HID_MAIN_DATA | USBD_HID_MAIN_VARIABLE | USBD_HID_MAIN_ABSOLUTE,
         USBD_HID_GLOBAL_REPORT_COUNT + 1, 0x01,
         USBD_HID_GLOBAL_REPORT_SIZE  + 1, 0x0D,
         USBD_HID_MAIN_INPUT          + 1, USBD_HID_MAIN_CONSTANT,
         USBD_HID_GLOBAL_USAGE_PAGE   + 1, USBD_HID_USAGE_PAGE_GENERIC_DESKTOP_CONTROLS,
         USBD_HID_LOCAL_USAGE         + 1, USBD_HID_DV_X,
         USBD_HID_LOCAL_USAGE         + 1, USBD_HID_DV_Y,
         USBD_HID_GLOBAL_LOG_MIN      + 1, 0x81,
         USBD_HID_GLOBAL_LOG_MAX      + 1, 0x7F,
         USBD_HID_GLOBAL_REPORT_SIZE  + 1, 0x08,
         USBD_HID_GLOBAL_REPORT_COUNT + 1, 0x02,
         USBD_HID_MAIN_INPUT          + 1, USBD_HID_MAIN_DATA | USBD_HID_MAIN_VARIABLE | USBD_HID_MAIN_RELATIVE,
      USBD_HID_MAIN_ENDCOLLECTION,
   USBD_HID_MAIN_ENDCOLLECTION
};

/* Local variables.*/
CPU_INT08U  class_nbr;
RTOS_ERR    err;

class_nbr = USBD_HID_Add(USBD_HID_SUBCLASS_BOOT,
                         USBD_HID_PROTOCOL_MOUSE,
                         USBD_HID_COUNTRY_CODE_NOT_SUPPORTED,
                         Ex_USBD_HID_Mouse_ReportDesc,
                         sizeof(Ex_USBD_HID_Mouse_ReportDesc),
                         DEF_NULL,
                         0u,
                         2u,
                         2u,
                         DEF_YES,
                         DEF_NULL,
                        &err);
if (err.Code != RTOS_ERR_NONE) {
   /* An error occurred. Error handling should be added here. */
}
```

Adding the HID Class Instance to Your Device's Configuration(s)

Once you have created a HID class instance, you can add it to a configuration by calling the function USBD_HID_ConfigAdd().

Listing - Example of call to USBD_HID_ConfigAdd() in the *Adding a USB Device HID Class Instance to your Device* page shows an example of a call to USBD_HID_ConfigAdd().

Listing - Example of call to USBD HID ConfigAdd()

```
RTOS_ERR  err;

USBD_HID_ConfigAdd(class_nbr,                    (1)
                   dev_nbr,                      (2)
                   config_nbr_fs,                (3)
                  &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}

USBD_HID_ConfigAdd(class_nbr,                    (4)
                   dev_nbr,
                   config_nbr_hs,
                  &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

(1) Class number to add to the configuration returned by USBD_HID_Add().

(2) Device number returned by USBD_DevAdd().

(3) Configuration number (here adding it to a Full-Speed configuration).

(4) Adding the same class instance to the High-Speed configuration. This should be done only if you have a High-Speed device.

Communicating using the USB Device HID Class

- Class Instance Communication
- Synchronous Communication

Class Instance Communication

The HID class offers the following functions to communicate with the host.

Table - HID Communication API Summary

Function name	Operation
USBD_HID_Rd()	Receives data from the host through interrupt OUT endpoint. This function is blocking.
USBD_HID_Wr()	Sends data to the host through interrupt IN endpoint. This function is blocking.

Synchronous Communication

Synchronous communication means that the transfer is blocking. Upon the function call, the applications blocks until the transfer is completed with or without an error. A timeout can be specified to avoid waiting forever.

Listing - Synchronous HID Read and Write Example in the *Communicating using the USB Device HID Class* page shows a read and write example that receives data from the host using the interrupt OUT endpoint and sends data to the host using the interrupt IN endpoint.

Listing - Synchronous HID Read and Write Example

```
CPU_INT08U  rx_buf[2];
CPU_INT08U  tx_buf[2];
RTOS_ERR    err;

(void)USBD_HID_Rd(         class_nbr,                          (1)
                  (void *)rx_buf,                              (2)
                          2u,
                          0u,                                  (3)
                         &err);
```

```
if(err.Code != RTOS_ERR_NONE){/* An error occurred. Error handling should be added here. */}(void)USBD_HID_Wr(       class_nbr,(1)(void *)tx_buf,
(4)2u,0u,(3)&err);if(err.Code != RTOS_ERR_NONE){/* An error occurred. Error handling should be added here. */}
```

(1) The class instance number created from USBD_HID_Add() provides an internal reference for the HID class to route the transfer to the proper interrupt OUT or IN endpoint.

(2) The application must ensure that the buffer provided to the function is large enough to accommodate all the data. Otherwise, synchronization issues might happen. Internally, the read operation is done either with the control endpoint or with the interrupt endpoint, depending on the control read flag set when calling USBD_HID_Add().

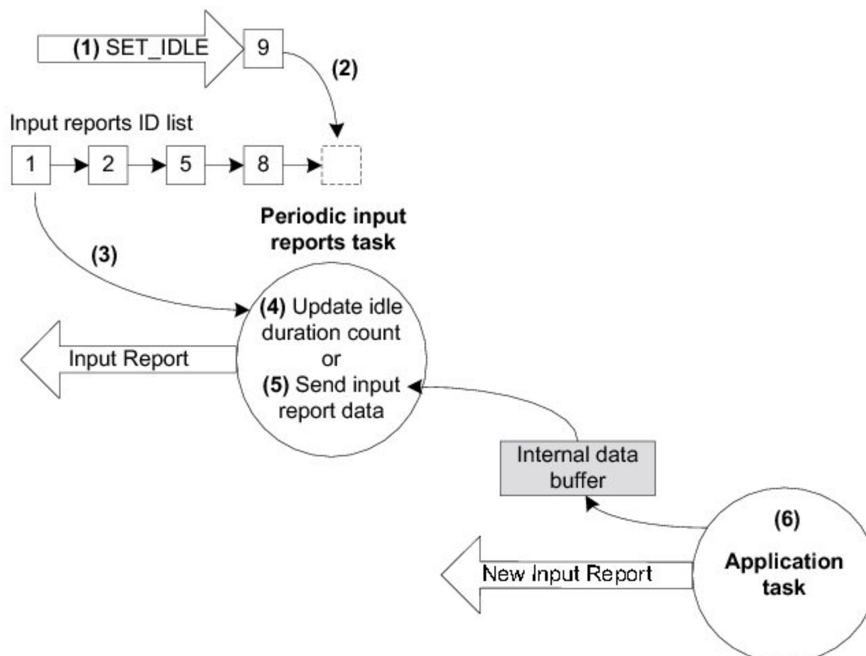
(3) To avoid an infinite blocking situation, a timeout expressed in milliseconds can be specified. A value of ‘0’ makes the application task wait forever.

(4) The application provides the initialized transmit buffer.

HID Periodic Input Reports Task

To save bandwidth, the host has the ability to silence reports from an interrupt IN endpoint by limiting the reporting frequency. To do so, the host must send the SET_IDLE request. The HID class implemented by Micrium contains an internal task that respects the reporting frequency limitation that you can apply to one or several input reports. Figure - Periodic Input Reports Task in the *HID Periodic Input Reports Task* page shows the functioning of the periodic input reports tasks.

Figure - Periodic Input Reports Task



(1) The device receives a SET_IDLE request. This request specifies an idle duration for a given report ID. For more details about the SET_IDLE request, refer to “Device Class Definition for Human Interface Devices (HID) Version 1.11”, section 7.2.4. A report ID allows you to distinguish among the different types of reports sent from the same endpoint.

(2) A report ID structure (allocated during the HID class initialization phase) is updated with the idle duration. An idle duration counter is initialized with the idle duration value. The report ID structure is inserted at the end of a linked list containing input reports ID structures. The idle duration value is expressed in 4-ms unit which gives a range of 4 to 1020 ms. If the idle duration is less than the polling interval of the interrupt IN endpoint, the reports are generated at the polling interval.

(3) Every 4 ms, the periodic input report task browses the input reports ID list. For each input report ID, the task performs one of two possible operations. The duration of the task period matches the 4-ms unit used for the idle duration. If no

SET_IDLE requests have been sent by the host, the input reports ID list is empty and the task has nothing to process. The task processes only report IDs that are different from 0 and with an idle duration greater than 0.

(4) For a given input report ID, the task verifies if the idle duration has elapsed. If the idle duration has not elapsed, the counter is decremented and no input report is sent to the host.

(5) If the idle duration has elapsed (that is, the idle duration counter has reached zero), an input report is sent to the host by calling the USBD_HID_Wr() function via the interrupt IN endpoint.

(6) The input report data sent by the task comes from an internal data buffer allocated for each input report described in the Report descriptor. An application task can call the USBD_HID_Wr() function to send an input report. After sending the input report data, USBD_HID_Wr() updates the internal buffer associated to an input report ID with the data just sent. Then, the periodic input reports task always sends the same input report data after each idle duration elapsed and until the application task updates the data in the internal buffer. There is some locking mechanism to avoid corruption of the input report ID data in the event of a modification happening at the exact time of transmission done by the periodic input report task.

USB Device MSC Class

This section describes the mass storage device class (MSC) supported by Micrium OS USB Device. MSC is a protocol that enables the transfer of information between a USB device and a host. The information being transferred is anything that can be stored electronically: executable programs, source code, documents, images, configuration data, or other text or numeric data. The USB device appears as an external storage medium to the host, enabling the transfer of files via drag and drop.

A file system defines how the files are organized in the storage media. The USB mass storage class specification does not require any particular file system to be used on conforming devices. Instead, it provides a simple interface to read and write sectors of data using the Small Computer System Interface (SCSI) transparent command set. As such, operating systems may treat the USB drive like a hard drive, and can format it with any file system they like.

The USB mass storage device class supports two transport protocols:

- Bulk-Only Transport (BOT)
- Control/Bulk/Interrupt (CBI) Transport (used only for floppy disk drives)

The mass storage device class implements the SCSI transparent command set using the BOT protocol only, which signifies that only bulk endpoints will be used to transmit data and status information. The MSC implementation supports multiple logical units.

The MSC implementation is in compliance with the following specifications:

- *Universal Serial Bus Mass Storage Class Specification Overview*, Revision 1.3 Sept. 5, 2008.
- *Universal Serial Bus Mass Storage Class Bulk-Only Transport*, Revision 1.0 Sept. 31, 1999.

USB Device MSC Class Overview

- Protocol
- Endpoints
- Class Requests
- Small Computer System Interface (SCSI)

Protocol

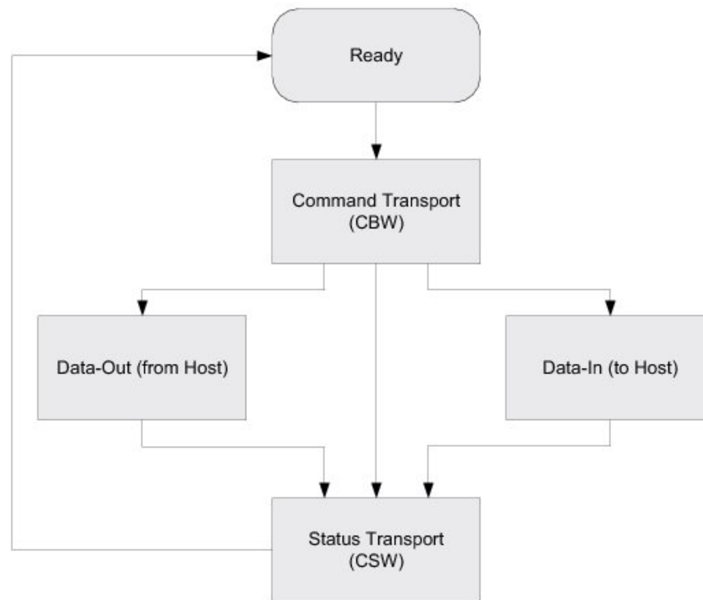
In this section, we will discuss the Bulk-Only Transport (BOT) protocol of the Mass Storage Class. The Bulk-Only Transport protocol has three stages:

- The Command Transport
- The Data Transport
- The Status Transport

Mass storage commands are sent by the host through a structure called the Command Block Wrapper (CBW). For commands requiring a data transport stage, the host will attempt to send or receive the exact number of bytes from the device as specified by the length and flag fields of the CBW. After the data transport stage, the host attempts to receive a Command Status Wrapper (CSW) from the device that details the status of the command as well as any data residue (if

any). For commands that do not include a data transport stage, the host attempts to receive the CSW directly after CBW is sent. The protocol is detailed in Figure - MSC Protocol in the *USB Device MSC Class Overview* page.

Figure - MSC Protocol



Endpoints

On the device side, in compliance with the BOT specification, the MSC is composed of the following endpoints:

- A pair of control IN and OUT endpoints called default endpoint.
- A pair of bulk IN and OUT endpoints.

Table - MSC Endpoint Usage in the *USB Device MSC Class Overview* page indicates the different usages of the endpoints.

Table - MSC Endpoint Usage

Endpoint	Direction	Usage
Control IN	Device to Host	Enumeration and MSC class-specific requests
Control OUT	Host to Device	Enumeration and MSC class-specific requests
Bulk IN	Device to Host	Send CSW and data
Bulk OUT	Host to Device	Receive CBW and data

Class Requests

There are two defined control requests for the MSC BOT protocol. These requests and their descriptions are detailed in Table - Mass Storage Class Requests in the *USB Device MSC Class Overview* page.

Table - Mass Storage Class Requests

Class Requests	Description
Bulk-Only Mass Storage Reset	This request is used to reset the mass storage device and its associated interface. This request readies the device to receive the next command block.

Class Requests	Description
Get Max LUN	This request is used to return the highest logical unit number (LUN) supported by the device. For example, a device with LUN 0 and LUN 1 will return a value of 1. A device with a single logical unit will return 0 or stall the request. The maximum value that can be returned is 15.

Small Computer System Interface (SCSI)

SCSI is a set of standards for handling communication between computers and peripheral devices. These standards include commands, protocols, electrical interfaces and optical interfaces. Storage devices that use other hardware interfaces, such as USB, use SCSI commands for obtaining device/host information and controlling the device's operation and transferring blocks of data in the storage media.

SCSI commands cover a vast range of device types and functions and as such, devices need a subset of these commands. In general, the following commands are necessary for basic communication:

- INQUIRY
- READ CAPACITY(10)
- READ(10)
- REQUEST SENSE
- TEST UNIT READY
- WRITE(10)

USB Device MSC Class Resource Needs from Core

Each time you add an MSC class instance to a USB configuration via the function USBD_MSC_ConfigAdd(), the following resources will be allocated from the core.

Resource	Quantity
Interfaces	1
Alternate interfaces	1
Endpoint descriptors	2
Interface groups	0

USB Device MSC Class Example Applications

- MSC Class Ramdisk LUN Example
  - Location
  - Running the Demo Application
  - API
- MSC Class Ramdisk Shared Example
  - Location
  - Running the Demo Application
  - API

MSC Class Ramdisk LUN Example

This example creates a simple MSC function with a single Logical Unit using a RAM disk as a storage medium. It requires the presence of Micrium OS FS in your project, and assumes that the Block Device module is initialized .

This example will allow you to accomplish the following tasks:

- Create a RAM disk media named "ramusb"
- Initialize the MSC class
- Create a MSC class instance
- Add a logical unit to the MSC class instance
- Add the MSC class instance to the Full-Speed configuration
- Add the MSC class instance to the High-Speed configuration (if available)
- Attach the "ramusb" storage media to the logical unit

Location

The example implementation is located in /examples/usb/device/all/ex_usbd_msc_ramdisk_lun.c.

Running the Demo Application

Once you connect the device to a Windows host PC (and once the drivers are installed), you should see a new drive available that represents your USB flash drive. Since it uses a RAM disk, you will have to format the drive each time the target is power cycled.

You can then create files/folders, read them, etc.

API

This example offers only one API named Ex_USBD_MSC_Init(). This function is normally called from a USB device core example.

MSC Class Ramdisk Shared Example

This example creates a simple MSC function with a single Logical Unit using a RAM disk as a storage medium. Its purpose is to demonstrate how a storage media can be shared between a USB host and your embedded application. It requires the presence of Micrium OS FS in your project and assumes that the Block Device module is initialized .

This example will allow you to accomplish the following tasks:

- Create and configure a RAMDisk media named "ramusb"
- Initialize the MSC class
- Create a MSC class instance
- Add a logical unit to the MSC class instance
- Add the MSC class instance to the Full-Speed configuration
- Add the MSC class instance to the High-Speed configuration (if available)
- Create a kernel task that will handle the application that shares the media with the host.

Location

The example implementation is located in /examples/usb/device/all/ex_usbd_msc_ramdisk_shared.c.

Running the Demo Application

This application is executed within the task body implemented in the function Ex_USBD_MSC_Task(). Within a loop, the application executes the following steps:

- Open the volume present on the RAM disk storage device using the Micrium OS FS API
- Create/Open a file named "Embedded.txt"
- Add the string "Iteration number <X>" to the file content (where <X> represents the number of the iteration)
- Close the file, volume and device
- Attach the RAM disk device to the MSC class instance logical unit. From this point, the media is visible from the host side
- Wait five seconds
- Detach the RAM disk device from the MSC class instance logical unit. From this point, the media appears as unavailable from the host side

The loop executes for 10 iterations. After the 10 iterations, the RAMDisk device remains attached to the logical unit.

API

This example offers only one API named Ex_USBD_MSC_Init(). This function is normally called from a USB device core example.

USB Device MSC Class Configuration

To configure the MSC class, there are three groups of configuration parameters:

- USB Device MSC Class Compile-Time Configurations
- USB Device MSC Class Run-time Application Specific Configurations
- USB Device MSC Class Logical Unit Configuration

USB Device MSC Class Compile-Time Configurations

Micrium OS USB Device MSC class is configurable at compile time via one #define located in the usbd_cfg.h file.

We recommend that you begin the configuration process with the default configuration values, which are shown in **bold**.

Table - Generic Configuration Constants

Constant	Description	Possible values
USBD_SCSI_CFG_64_BIT_LBA_EN	Enables or disables support for Logical Block Address (LBA) of 64 bits.	DEF_ENABLED or DEF_DISABLED

USB Device MSC Class Run-time Application Specific Configurations

- Class Initialization
  - p_qty_cfg
- Optional Configurations
- Post-Init Configurations
- Class Instance Creation
  - p_msc_task_cfg

Class Initialization

Initializing the USB Device MSC class module of Micrium OS is done by calling the USBD_MSC_Init() function. This function takes one configuration argument which is described below.

p_qty_cfg

p_qty_cfg is a pointer to a configuration structure of type USBD_MSC_QTY_CFG. Its purpose is to inform the USB device MSC module on how many USB MSC objects to allocate.

Table - USBD_MSC_QTY_CFG configuration structure in the *USB Device MSC Class Run-time Application Specific Configurations* page describes each configuration field available in this configuration structure.

Table - USBD_MSC_QTY_CFG configuration structure

Field	Description	Recommended value
.ClassInstanceQty	Number of class instances you will allocate via a call to the function USBD_MSC_Add().	1
.ConfigQty	Number of configuration to which a class instance can be added via a call to the function USBD_MSC_ConfigAdd().	2
.LUN_Qty	Number of logical units per class instance that you will add via a call to the function USBD_MSC_SCSI_LunAdd().	1

Optional Configurations

This section describes the configurations that are optional. If you do not set them in your application, the default configurations will apply.

The default values can be retrieved via the structure USBD_MSC_InitCfgDflt.

> Note: These configurations must be set *before* you call the function USBD_MSC_Init().

Configurations	Description	Type	Function to call	Default	Field from default configuration structure
Buffer alignment	This module allocates some buffers used for data transfers with the host. You may need a specific address alignment for these buffers depending on your USB controller. If you use more than one USB controller, you must set the alignment to the highest value.	CPU_SIZE_T	USBD_MSC_ConfigureBufAlignOctets()	Size of cache line, or CPU alignment, if no cache.	.BufAlignOctets
Data buffer length	Configures the length of the MSC data buffer, in octets.	CPU_INT32U	USBD_MSC_ConfigureDataBufLen()	512 octets	.DataBufLen
Memory segments	This module allocates some control data and buffers used for data transfers with the host. It has the ability to use a different memory segment for the control data and for the data buffers.	MEM_SEG*	USBD_MSC_ConfigureMemSeg()	General-purpose heap .	.MemSegPtr
Memory segments	This module allocates some control data and buffers used for data transfers with the host. It has the ability to use a different memory segment for the control data and for the data buffers.	MEM_SEG*	USBD_MSC_ConfigureMemSeg()	General-purpose heap .	.MemSegBufPtr

Table - MSC Class Optional Configurations

Post-Init Configurations

This section describes the configurations that can be set at any time during execution *after* you called the function USBD_MSC_Init().

These configurations are optional. If you do not set them in your application, the default configurations will apply.

Configurations	Description	Type	Function to call	Default
MSC instance task priority	Each MSC class instance created via the function USBD_MSC_Add() will create a task. You can change the priority of the created task at any time.	RTOS_TASK_PRIO	USBD_MSC_TaskPrioSet()	N/A

Table - MSC Class Post-init Configurations

Class Instance Creation

Creating a USB Device MSC class instance is done by calling the USBD_MSC_Add() function. This function takes one configuration argument that is described below.

p_msc_task_cfg

p_msc_task_cfg is a pointer to a configuration structure of type RTOS_TASK_CFG that allows you to configure the priority, stack base pointer and stack size for the MSC instance task. This task handles all the MSC request and data transfers.

USB Device MSC Class Logical Unit Configuration

Adding a logical unit to an MSC class instance is done by calling the function USBD_MSC_SCSI_LunAdd() . This function takes two configuration arguments that are described below.

p_lu_fncts

p_lu_fncts is a pointer to a structure of type USBD_SCSI_LU_FNCTS. Its purpose is to provide the MSC class with a set of optional callback functions that are called when an event occurs on the logical unit.

A null pointer (DEF_NULL) can be passed to this argument if no callbacks are needed.

Table - USBD_SCSI_LU_FNCTS configuration structure in the *USB Device MSC Class Logical Unit Configuration* page describes each configuration field available in this configuration structure.

Table - USBD_SCSI_LU_FNCTS configuration structure

Field	Description	Function signature
.HostEjectEvent	Function called when a logical unit is ejected from the host.	void HostEjectEvent(CPU_INT08U lu_nbr);

p_lu_info

p_lu_info is a pointer to a structure of type USBD_SCSI_LU_INFO. Its purpose is to provide the information on the logical unit to the MSC class.

Table - USBD_SCSI_LU_INFO configuration structure in the *USB Device MSC Class Logical Unit Configuration* page describes each configuration field available in this configuration structure.

Table - USBD_SCSI_LU_INFO configuration structure

Field	Description
.SCSI_StorageAPI_Ptr	Pointer to the media driver API that will handle this logical unit. See USB Device MSC Class Storage Drivers for more information on storage drivers.
.VendIdPtr	Pointer to a string that contains the vendor identification of the logical unit. The maximum length of the string is 8 characters.
.ProdIdPtr	Pointer to a string that contains the product identification of the logical unit. The maximum length of the string is 16 characters.
.ProdRevLevel	Product revision level.

Field	Description
.IsRdOnly	Flag that indicates if the logical unit should be seen as read only from the point of view of the host (DEF_YES) or not (DEF_NO).

USB Device MSC Class Programming Guide

This section explains how to use the MSC class.

- Initializing the USB Device MSC Class
- Adding a USB Device MSC Class Instance to your Device
- USB Device MSC Class Logical Unit Handling

Initializing the USB Device MSC Class

To add MSC Class functionality to your device, you must first initialize the class by calling the function USBD_MSC_Init().

Listing - Example of call to USBD_MSC_Init() in the *Initializing the USB Device MSC Class* page shows an example of call to USBD_MSC_Init() using default arguments. For more information on the configuration arguments to pass to USBD_MSC_Init(), see USB Device MSC Class Run-time Application Specific Configurations .

Listing - Example of call to USBD_MSC_Init()

```
RTOS_ERR         err;
USBD_MSC_QTY_CFG  msc_qty_cfg;

msc_qty_cfg.ClassInstanceQty = 1u;
msc_qty_cfg.ConfigQty        = 2u;
msc_qty_cfg.LUN_Qty          = 1u;

USBD_MSC_Init(&msc_qty_cfg,
          &err);
if (err.Code != RTOS_ERR_NONE) {
   /* An error occurred. Error handling should be added here. */
}
```

Adding a USB Device MSC Class Instance to your Device

To add MSC class functionality to your device, you must first create an instance, then add it to your device's configuration(s). You must add at least one logical unit to your instance.

Creating an MSC Class Instance

Create a MSC class instance by calling the function USBD_MSC_Add().

Listing - Example of call to USBD_MSC_Add() in the *Adding a USB Device MSC Class Instance to your Device* page gives an example of call to USBD_MSC_Add() using default arguments. For more information on the configuration arguments to pass to USBD_MSC_Add(), see USB Device MSC Class Run-time Application Specific Configurations .

Listing - Example of call to USBD_MSC_Add()

```
CPU_INT08U     class_nbr;
RTOS_TASK_CFG  msc_task_cfg;
RTOS_ERR       err;

msc_task_cfg.Prio            = 20u;
msc_task_cfg.StkSizeElements = 512u;
msc_task_cfg.StkPtr          = DEF_NULL;

class_nbr = USBD_MSC_Add(&msc_task_cfg
                   &err);
if (err.Code != RTOS_ERR_NONE) {
```

```
/* An error occurred. Error handling should be added here. */}
```

Adding the MSC Class Instance to Your Device's Configuration(s)

Once you have created an MSC class instance, you can add it to a configuration by calling the function USBD_MSC_ConfigAdd().

Listing - Example of call to USBD_MSC_ConfigAdd() in the *Adding a USB Device MSC Class Instance to your Device* page shows an example of call to USBD_MSC_ConfigAdd() using default arguments.

Listing - Example of call to USBD_MSC_ConfigAdd()

```
RTOS_ERR  err;

USBD_MSC_ConfigAdd(class_nbr,                    (1)
                   dev_nbr,                      (2)
                   config_nbr_fs,                (3)
                  &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}

USBD_MSC_ConfigAdd(class_nbr,                    (4)
                   dev_nbr,
                   config_nbr_hs,
                  &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

(1) Class number to add to the configuration returned by USBD_MSC_Add().

(2) Device number returned by USBD_DevAdd().

(3) Configuration number (here adding it to a Full-Speed configuration).

(4) Adding the same class instance to the High-Speed configuration. This should be done only if you have a High-Speed device.

USB Device MSC Class Logical Unit Handling

- Adding a Logical Unit
- Attaching/Detaching a Storage Medium

Adding a Logical Unit

When adding a logical unit to your MSC class instance, it must be bound to a storage medium (RAMDisk, SD card, Flash memory, etc). The MSC class uses a storage driver to communicate with storage media.

Even though you could create your own storage media driver, Micrium recommends the use of the universal block device driver. This driver allows you to use any type of storage media (RAMDisk, SD, NOR, NAND, etc) that are supported by the block device API of Micrium OS FS.

This driver also allows you to use the driver with or without Micrium OS FS available in your project.

Listing - Adding a logical unit via USBD_MSC_SCSI_LunAdd() in the *USB Device MSC Class Logical Unit Handling* page shows an example of adding a logical unit via USBD_MSC_SCSI_LunAdd().

Listing - Adding a logical unit via USBD_MSC_SCSI_LunAdd()

```
CPU_INT08U        lu_nbr;
USBD_SCSI_LU_INFO  lu_info;
RTOS_ERR          err;

lu_info.SCSI_StorageAPI_Ptr = &USBD_SCSI_StorageBlkDevAPI;      (1)
lu_info.VendIdPtr          = "Micrium";
lu_info.ProdIdPtr          = "FS example";
lu_info.ProdRevLevel       =  0x1000u;
lu_info.IsRdOnly           =  DEF_NO;

lu_nbr = USBD_MSC_SCSI_LunAdd(class_nbr,
                              &lu_info,
                               DEF_NULL,
                              &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

Attaching/Detaching a Storage Medium

Once the logical unit has been added, a storage medium must be attached to it in order to be available from the host side.

The MSC class offers two functions to control the storage media association to the logical unit: USBD_MSC_SCSI_LunAttach() and USBD_MSC_SCSI_LunDetach(). These functions allow you to emulate the removal of a storage device in order to re-gain access from the embedded application if necessary.

Listing - Example of Media Attach/Detach in the *USB Device MSC Class Logical Unit Handling* page shows an example of how to use the function USBD_MSC_SCSI_LunAttach() and USBD_MSC_SCSI_LunDetach().

Listing - Example of Media Attach/Detach

```
RTOS_ERR  err;

USBD_MSC_SCSI_LunAttach(class_nbr,
                        lu_nbr,
                        media_name,                        (1)
                       &err)
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}

...                                                        (2)

USBD_MSC_SCSI_LunDetach(class_nbr,
                        lu_nbr,
                       &err)
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}

...                                                        (3)

USBD_MSC_SCSI_LunAttach(class_nbr,
                        lu_nbr,
                        media_name,                        (4)
                       &err)
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}

...                                                        (5)
```

(1) Here we pass a string to the block device media that we want to use as storage. This string id is associated to a storage device that must have been registered to the platform manager previously.

(2) From this moment, if the MSC device is connected to a host, the storage media is accessible.

(3) If the MSC device is connected to a host, the media will now appear as unavailable. At this moment, operations can be performed on the media from the embedded application using Micrium OS FS API.

(4) Again, we pass a string to the block device media that we want to use as storage. It can technically be another storage medium than in (1), even though it is the same logical unit.

(5) Again, if the MSC device is connected to the host, the storage media will appear as connected.

USB Device MSC Class Storage Drivers

The USB Device MSC Class needs a storage driver to communicate with a storage medium. For the moment, Micrium offers only one driver.

Universal Block Device Driver

The universal block device driver allows you to use any storage media (RAMDisk, SD card, NAND, NOR, etc.) that are supported by the Block Device module.

This driver also allows the same media to be accessed from your embedded application via the Micrium OS FS API.

Table - Universal Block Device Driver Description in the *USB Device MSC Class Storage Drivers* page provides a description of this driver.

Table - Universal Block Device Driver Description

Name	Universal Block Device
Storage API structure name	USBD_SCSI_StorageBlkDevAPI located in usbd_scsi_storage_blk_dev.h

USB Device Vendor Class

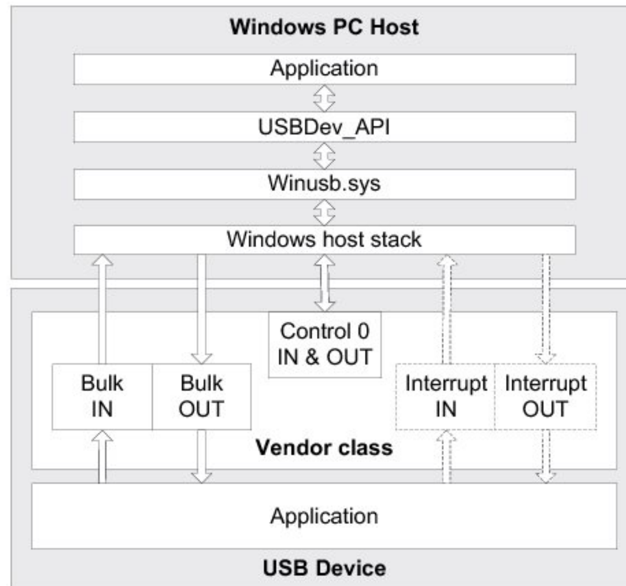
The Vendor class allows you to build vendor-specific devices that can implement a proprietary protocol. It relies on a pair of bulk endpoints to transfer data between the host and the device. Bulk transfers are convenient for transferring large amounts of unstructured data and provide a reliable exchange of data by using an error detection and retry mechanism.

In addition to bulk endpoints, the Vendor class can also use an optional pair of interrupt endpoints. Any operating system (OS) can work with the Vendor class provided that the OS has a driver to handle the Vendor class. Depending on the OS, the driver can be native or vendor-specific. For instance, under Microsoft Windows®, your application interacts with the WinUSB driver provided by Microsoft to communicate with the vendor device.

USB Device Vendor Class Overview

Figure - General Architecture Between Windows Host and Vendor Class in the *USB Device Vendor Class Overview* page shows the general architecture between the host and the device using the Vendor class. In this example, the host operating system is MS Windows.

Figure - General Architecture Between MS Windows Host and Vendor Class



On the MS Windows side, the application communicates with the vendor device by interacting with the USBDev_API library. This library, provided by Micrium, offers an API to manage a device and its associated pipes, and to communicate with the device through control, bulk and interrupt endpoints. USBDev_API is a wrapper that allows the use of the WinUSB functions exposed by Winusb.dll.

On the device side, the Vendor class is composed of the following endpoints:

- A pair of control IN and OUT endpoints called the default endpoint.
- A pair of bulk IN and OUT endpoints.
- A pair of interrupt IN and OUT endpoints. This pair is optional.

Table - Vendor Class Endpoints Usage in the *USB Device Vendor Class Overview* page indicates the usage of the different endpoints:

Table - Vendor Class Endpoints Usage

Endpoint	Direction	Usage
Control IN	Device-to-host	Standard requests for enumeration and vendor-specific requests.
Control OUT	>Host-to-device	Standard requests for enumeration and vendor-specific requests.
Bulk IN	Device-to-host	Raw data communication. Data can be structured according to a proprietary protocol.
Bulk OUT	Host-to-device	Raw data communication. Data can be structured according to a proprietary protocol.
Interrupt IN	Device-to-host	Raw data communication or notification. Data can be structured according to a proprietary protocol.
Interrupt OUT	Host-to-device	Raw data communication or notification. Data can be structured according to a proprietary protocol.

The device application can use bulk and interrupt endpoints to send or receive data to or from the host. It can only use the default endpoint to decode vendor-specific requests sent by the host. The standard requests are managed internally by the Core layer of Micrium OS USB Device.

USB Device Vendor Class Resource Needs from Core

Each time you add a vendor class instance to a configuration via the function USBD_Vendor_ConfigAdd(), the following resources will be allocated from the core.

Resource	Quantity
Interfaces	1
Alternate interfaces	1
Endpoint descriptors	2 (4 if you enabled interrupt endpoints)
Interface groups	0

USB Device Vendor Class Example Applications

- Vendor Class Loopback Example
- Location
- Configuration
- Running the Demo Application
  - GUID
- API

Vendor Class Loopback Example

This simple example receives data from a host and re-transmits it, hence creating a loopback. It offers two implementations: one that uses the synchronous API, and another one that uses the asynchronous API. Both implementations can be enabled at the same time, therefore creating a composite device.

This example will allow you to accomplish the following tasks:

- Initialize the Vendor class
- Create a Vendor class instance
- Add the Vendor class instance to the Full-Speed configuration
- Add the Vendor class instance to the High-Speed configuration (if available)
- Set the Microsoft Extended Property to specify the GUID
- Create a kernel task that implements the loopback (one task for the synchronous implementation and/or one task for the asynchronous implementation)

Location

The example implementation is located in /examples/usb/device/all/ex_usbd_vendor_loopback.c.

To execute it, you will also need some files on the host side. The files can be downloaded from the Micrium web site .

To install the Windows driver for the device (only if you do not use the Microsoft OS Descriptors), use the .inf file from the Windows application files located in micrium_usb_dev_host_app/OS/Windows/Vendor/INF.

To execute the example application, you will also require a custom executable application on the host side. These files are located in micrium_usb_dev_host_app/OS/Windows/Vendor/Visual Studio 2010/exe/x86.

Configuration

There are two #defines available in the file ex_usbd_vendor_loopback.c, which are used to select the synchronous and/or asynchronous implementation:

- EX_USBD_VENDOR_LOOPBACK_CFG_SYNC_EN
- EX_USBD_VENDOR_LOOPBACK_CFG_SYNC_EN

They must be set to either DEF_ENABLED to enable the implementation or DEF_DISABLED to disable the implementation. At least one of the #defines should be enabled at any time. Both can be enabled at the same time.

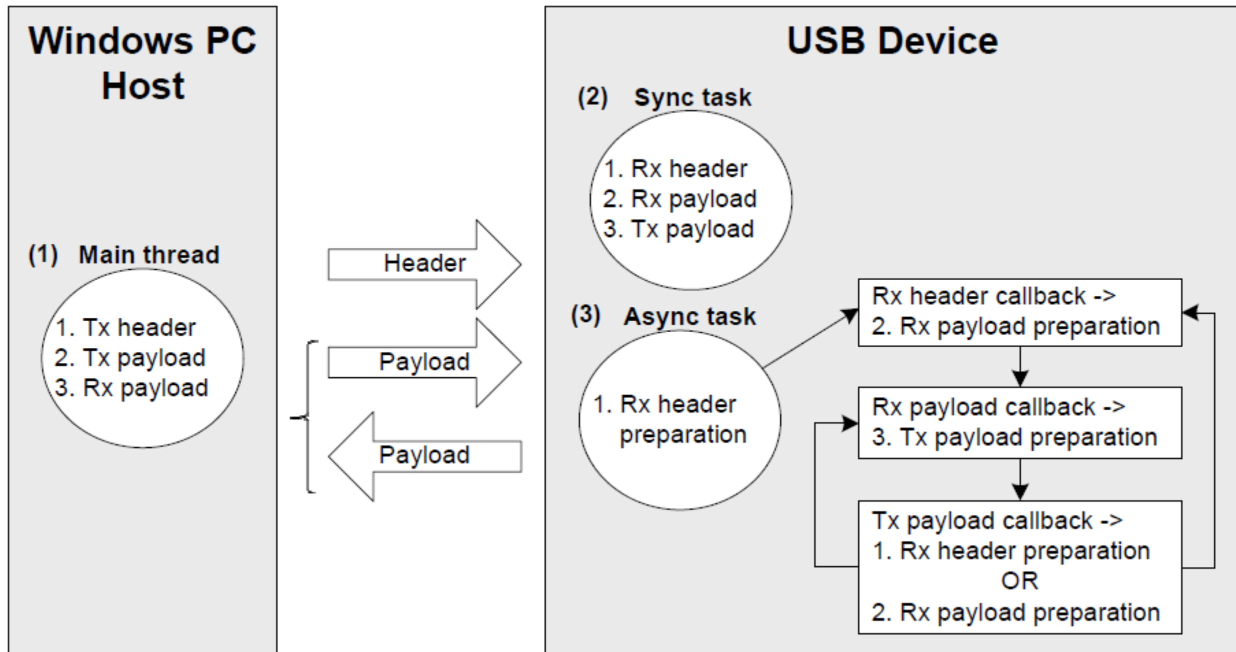
Running the Demo Application

The application used to demonstrate Vendor class communication between the host and the device is called the *Echo* demo. This demo implements a simple protocol allowing the device to echo the data sent by the host.

There are two varieties of the Echo demo. The Echo Sync application demonstrates the *synchronous* communication API, and the Echo Async application demonstrates the *asynchronous* communication API. These are both discussed in the section Communicating using the USB Device Vendor Class .

Figure - Echo Demo in the *USB Device Vendor Class Example Applications* page illustrates the Echo demo with host and device interactions:

Figure - Echo Demo



(1) The Windows application executes a simple protocol, which consists of sending a header indicating the total payload size, sending the data payload to the device, and receiving the same data payload back from the device. The data payload is split into small chunks of write and read operations of 512 bytes. The write operation is done using a bulk OUT endpoint, and the read uses a bulk IN endpoint.

(2) On the device, the Echo Sync application performs a series of tasks that execute in lockstep with the Windows PC, and the read and write operations performed on the device mirror those performed on the host. So, a read operation on the device implies a bulk OUT endpoint on the host, while a write on the device implies a bulk IN endpoint on the host. Both the device and host wait until the other has performed its task.

(3) The Echo Async application performs the same steps as the Sync task, but using the asynchronous API. The Async task starts the first asynchronous OUT transfer to receive the header. Then, a callback associated with the header reception is called by the device stack. It prepares the next asynchronous OUT transfer to receive the payload. The read payload callback sends the payload back to the host via an asynchronous IN transfer. The write payload callback is called and either prepares the next header reception if the entire payload has been sent to the host, or prepares a next OUT transfer to receive a new chunk of data payload. A side note: this task is also used in case of error during the protocol communication.

When the vendor device is first connected, Windows enumerates the device by retrieving the standard descriptors. Microsoft does not provide any specific driver for the Vendor class, so if you don't use the Microsoft OS descriptors, you must indicate to Windows which driver to load using an INF file (refer to About INF files for more details about INF). The INF file tells Windows to load the WinUSB generic driver (provided by Microsoft). Indicating the INF file to Windows needs to be done only once. Windows will then automatically recognize the vendor device and load the proper driver for any new connection. The process of indicating the INF file may vary according to the Windows operating system version:

- Windows XP directly opens the “Found New Hardware Wizard”. Follow the different steps of the wizard until the page where you can indicate the path of the INF file.
- Windows Vista and later won’t open a “Found New Hardware Wizard”. It will just indicate that no driver was found for the vendor device. You have to manually open the wizard. Open the Device Manager, the vendor device connected appears under the category ‘Other Devices’ with a yellow icon. Right-click on your device and choose ‘Update Driver Software…’ to open the wizard. Follow the different steps of the wizard until the page where you can indicate the path of the INF file.

Once the driver is successfully loaded, the Windows host application is ready to be launched. There are two executables:

- EchoSync.exe for the Windows application with the synchronous communication API of USBDev_API
- EchoAsync.exe for the Windows application with the asynchronous IN API of USBDev_API

The Windows application interacts with WinUSB driver via USBDev_API, which is a wrapper for the WinUSB driver. USBDev_API is provided by Micrium. Refer to USBDev_API for more details about USBDev_API and the WinUSB driver. The Echo Sync or Async demo will first determine the number of vendor devices connected to the PC. For each detected device, the demo will open a bulk IN and a bulk OUT pipe. Then the demo is ready to send/receive data to/from the device. You will have to enter the maximum number of transfers you want as shown in Figure - Demo Application at Startup in the *USB Device Vendor Class Example Applications* page.

Figure - Demo Application at Startup

In the example Figure - Demo Application at Startup in the *USB Device Vendor Class Example Applications* page, the demo handles 10 transfers. Each transfer is sent after the header, following the simple protocol described in Figure - Echo Demo in the *USB Device Vendor Class Example Applications* page. The first transfer will have a data payload of 1 byte. Then, subsequent transfers will have their size incremented by 1 byte until the last transfer. In our example, the last transfer will have 10 bytes.

Figure - Demo Application Execution (Single Device) in the *USB Device Vendor Class Example Applications* page shows the execution.

Figure - Demo Application Execution (Single Device)

```
C:\Micrium\Software\uC-USB-Device-V4\App\Host\OS\Windows\Vendor\Visual Studio 2010\exe\x...
Number of devices attached = 1
>> Device #1 open.
>> Device #1: WinUSB internal alternate setting set.
>> Device #1: 1 interface(s) = default IF + 0 associated interface(s).
>> Device #1 attached is FULL-SPEED.
>> Device #1 (IF0): Bulk IN pipe open.
>> Device #1 (IF0): Bulk OUT pipe open.
------------------------------------------------------------
-  Communication with default interface of USB Device #1  -
------------------------------------------------------------
----> Specify the number of transfers: 10


Sending/Receiving [1] bytes...OK
Sending/Receiving [2] bytes...OK
Sending/Receiving [3] bytes...OK
Sending/Receiving [4] bytes...OK
Sending/Receiving [5] bytes...OK
Sending/Receiving [6] bytes...OK
Sending/Receiving [7] bytes...OK
Sending/Receiving [8] bytes...OK
Sending/Receiving [9] bytes...OK
Sending/Receiving [10] bytes...OK
----> Device #1 (IF0): communication successful!

Do you want to continue ? (YES = y or Y, NO = n or N) y

------------------------------------------------------------
-  Communication with default interface of USB Device #1  -
------------------------------------------------------------
----> Specify the number of transfers: 2545
```

The demo will prompt you to perform another transfer. Figure - Demo Application Execution (Single Device) in the *USB Device Vendor Class Example Applications* page shows the example of a single device with one vendor interface. For composite devices, the demo is able to communicate with each vendor interface, and it will open bulk IN and OUT pipes for each one. You will be asked the maximum number of transfers for each interface in the device. Figure - Demo Application Execution (Composite Device) in the *USB Device Vendor Class Example Applications* page shows an example of a composite device.

Figure - Demo Application Execution (Composite Device)

```
C:\Micrium\Software\uC-USB-Device-V4\App\Host\OS\Windows\Vendor\Visual Studio 2010\exe\x...
Number of devices attached = 1
>> Device #1 open.
>> Device #1: WinUSB internal alternate setting set.
>> Device #1: 2 interface(s) = default IF + 1 associated interface(s).
>> Device #1 attached is FULL-SPEED.
>> Device #1 (IF0): Bulk IN pipe open.
>> Device #1 (IF0): Bulk OUT pipe open.
>> Device #1 (IF1): Bulk OUT pipe open.
>> Device #1 (IF1): Bulk OUT pipe open.
------------------------------------------------------------
-  Communication with default interface of USB Device #1  -
------------------------------------------------------------
----> Specify the number of transfers: 5


Sending/Receiving [1] bytes...OK
Sending/Receiving [2] bytes...OK
Sending/Receiving [3] bytes...OK
Sending/Receiving [4] bytes...OK
Sending/Receiving [5] bytes...OK
----> Device #1 (IF0): communication successful!

------------------------------------------------------------------
-  Communication with associated interface #1 of USB Device #1  -
------------------------------------------------------------------
----> Specify the number of transfers: 5

Sending/Receiving [1] bytes...OK
Sending/Receiving [2] bytes...OK
Sending/Receiving [3] bytes...OK
Sending/Receiving [4] bytes...OK
Sending/Receiving [5] bytes...OK
----> Device #1 (IF1): communication successful!

Do you want to continue ? (YES = y or Y, NO = n or N) _
```

Figure - Demo Application Execution (Composite Device)

GUID

A Globally Unique IDentifier (GUID) is a 128-bit value that uniquely identifies a class or other entity. Windows uses GUIDs for identifying two types of devices classes:

- Device setup class
- Device interface class

Windows assigns a *device setup GUID* to devices that are installed in all the same way, and which all use the same class installer and co-installers. Class installers and co-installers are DLLs that provide functions related to the installation of the device.

A *device interface GUID* provides the mechanism that applications use to communicate with the driver that has been assigned to all devices in a class. Refer to Using GUID for more details about the GUID.

The device setup class GUID is used in WinUSB_single.inf and WinUSB_composite.inf.These INF files define a new device setup class that will be added to the Windows registry under HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Class when a vendor device is first connected. Listing - New Device Setup Class in the *USB Device Vendor Class Example Applications* page shows entries in the INF file that define the new device setup class.

Listing - New Device Setup Class

```
Class = MyDeviceClass ; Name of the device setup class.
ClassGuid = {11111111-2222-3333-4444-555555555555} ; Device setup class GUID
```

The INF file allows Windows to register all the information necessary to associate the driver Winusb.sys with the connected vendor device. The Windows Echo application is able to retrieve the attached vendor device thanks to the device interface class GUID. WinUSB_single.inf and WinUSB_composite.inf define the following device interface class GUID: {143f20bd-7bd2-4ca6-9465-8882f2156bd6}. The Echo application includes a header file called usbdev_guid.h. This header file defines the variable shown in Listing - GUID Global Variable in the *USB Device Vendor Class Example Applications* page.

Listing - GUID Global Variable

```
GUID USBDev_GUID = {0x143f20bd,0x7bd2,0x4ca6,{0x94,0x65,0x88,0x82,0xf2,0x15,0x6b,0xd6}};
```

USBDev_GUID is a structure whose fields represent the device interface class GUID that is defined in WinUSB_single.inf and WinUSB_composite.inf. The USBDev_GUID variable will be passed as a parameter to the function USBDev_Open(). A handle will be returned by USBDev_Open(), and the application uses this handle to access the device.

API

This example uses only one API named Ex_USBD_Vendor_Init(). This function is normally called from a USB device core example.

USB Device Vendor Class Configuration

To configure the Vendor class, there are two groups of configuration parameters:

- USB Device Vendor Class Run-Time Application Specific Configurations
- USB Device Vendor Class Instance Configurations

USB Device Vendor Class Run-Time Application Specific Configurations

- Class Initialization
  - p_qty_cfg
- Optional Configurations

Class Initialization

To initialize the Micrium OS USB Device Vendor class, you call the function USBD_Vendor_Init(). This function takes one configuration argument which is described below.

p_qty_cfg

p_qty_cfg is a pointer to a configuration structure of type USBD_VENDOR_QTY_CFG. Its purpose is to inform the USB device module about how many USB vendor objects to allocate.

Table - USBD_VENDOR_QTY_CFG configuration structure in the *USB Device Vendor Class Run-Time Application Specific Configurations* page describes each configuration field available in this configuration structure.

Table - USBD_VENDOR_QTY_CFG configuration structure

Field	Description	Recommended Value
.ClassInstanceQty	Number of class instances you will allocate via a call to the function USBD_Vendor_Add().	1
.ConfigQty	Number of configurations. Vendor class instances can be added to one or more configurations via a call to the function USBD_Vendor_ConfigAdd().	2

Optional Configurations

This section describes the configurations that are optional. If you do not set them in your application, the default configurations will apply.

The default values can be retrieved via the structure USBD_Vendor_InitCfgDflt.

> Note: These configurations must be set *before* you call the function USBD_Vendor_Init().

Table - Vendor Class Optional Configurations

Configurations	Description	Type	Function to Call	Default	Field Confi Struc
Microsoft extended properties quantity	Configures the number of Microsoft Extended properties that will be added per vendor class instance via a call to the function USBD_Vendor_MS_ExtPropertyAdd(). **Note**: This configuration is ignored when USBD_CFG_MS_OS_DESC_EN is set to DEF_DISABLED.	CPU_INT08U	USBD_Vendor_ConfigureMsExtPropertiesQty()	1	.MsEx
Memory segments	This module allocates some control data. It has the ability to use a specified memory segment.	MEM_SEG*	USBD_Vendor_ConfigureMemSeg()	General-purpose heap .	.Mem

USB Device Vendor Class Instance Configurations

This section defines the configurations related to the Vendor class instances.

- Class Instance Creation
  - intr_en
  - interval
  - req_callback

Class Instance Creation

Creating a Vendor class instance is done by calling the function USBD_Vendor_Add(). This function takes three configuration arguments that are described below.

intr_en

Boolean that indicates if a pair of interrupt endpoints should be added or not.

Value	Description
DEF_ENABLED	A pair of IN/OUT endpoints will be added and made available to the embedded application.
DEF_DISABLED	No interrupt endpoint will be added. Only a pair of Bulk IN/OUT endpoint will be available.

interval

- If you set intr_en to DEF_ENABLED, you can specify the interrupt endpoints polling interval (in milliseconds).
- If you set intr_en to DEF_DISABLED, you can set interval to 0 as it will be ignored by the class.

req_callback

req_callback is a function that you can specify to handle the class specific control requests. If you don't use any class specific requests, you can set this to DEF_NULL.

Listing - Signature of Class Specific Request Function in the *USB Device Vendor Class Instance Configurations* page provides the expected signature of your class specific requests handler.

Listing - Signature of Class Specific Request Function

```
CPU_BOOLEAN  App_USBD_VendorReqHandle(      CPU_INT08U      class_nbr,      (1)
                                    CPU_INT08U      dev_nbr,        (2)
                                const  USBD_SETUP_REQ  *p_setup_req);   (3)
```

(1) Vendor class instance number.

(2) Device number.

(3) Pointer to received setup request from host.

USB Device Vendor Class Programming Guide

This section explains how to use the Vendor class.

- Initializing the USB Device Vendor Class
- Adding a USB Device Vendor Class Instance to your Device
- Communicating using the USB Device Vendor Class

Initializing the USB Device Vendor Class

To add a Vendor Class functionality to your device, you must first initialize the class by calling the function USBD_Vendor_Init().

Listing - Example of call to USBD_Vendor_Init() in the *Initializing the USB Device Vendor Class* page shows an example of a call to USBD_Vendor_Init() using default arguments. For more information about the configuration arguments that are passed to USBD_Vendor_Init(), see USB Device Vendor Class Run-Time Application Specific Configurations .

Listing - Example of call to USBD_Vendor_Init()

```
RTOS_ERR            err;
USBD_VENDOR_QTY_CFG  vendor_qty_cfg;

vendor_qty_cfg.ClassInstanceQty = 1u;
vendor_qty_cfg.ConfigQty        = 2u;
```

```
&err);if(err.Code != RTOS_ERR_NONE){/* An error occurred. Error handling should be added here. */}
```

Adding a USB Device Vendor Class Instance to your Device

To add vendor class functionality to your device, you must first create an instance, then add it to your device's configuration(s). You can also optionally add one or more Microsoft Extended Properties to your class instance.

- Creating a Vendor Class Instance
- Adding the Vendor Class Instance to Your Device's Configuration(s)
- Adding Microsoft Extended Properties to your Class Instance (Optional)

Creating a Vendor Class Instance

Create a Vendor class instance by calling the function USBD_Vendor_Add().

Listing - Example of call to USBD_Vendor_Add() in the *Adding a USB Device Vendor Class Instance to your Device* page shows an example of a call to USBD_Vendor_Add() using default arguments. For more information about the configuration arguments to pass to USBD_Vendor_Add(), see USB Device Vendor Class Instance Configurations .

Listing - Example of call to USBD_Vendor_Add()

```
CPU_INT08U  class_nbr;
RTOS_ERR    err;

class_nbr = USBD_Vendor_Add(DEF_DISABLED,                (1)
                            0u,                          (2)
                            App_USBD_Vendor_VendorReq,   (3)
                            &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

(1) No Interrupt endpoints with this class instance.

(2) Interval is ignored since Interrupt endpoints are disabled.

(3) Callback function that is part of your application that handles vendor-specific class requests. See Communicating using the USB Device Vendor Class for more information.

Adding the Vendor Class Instance to Your Device's Configuration(s)

Once you have created a vendor class instance, you can add it to a configuration by calling the function USBD_Vendor_ConfigAdd().

Listing - Example of call to USBD_Vendor_ConfigAdd() in the *Adding a USB Device Vendor Class Instance to your Device* page shows an example of a call to USBD_Vendor_ConfigAdd() using default arguments.

Listing - Example of call to USBD_Vendor_ConfigAdd()

```
RTOS_ERR  err;

USBD_Vendor_ConfigAdd(class_nbr,                    (1)
                      dev_nbr,                     (2)
                      config_nbr_fs,               (3)
                     &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}

USBD_Vendor_ConfigAdd(class_nbr,                    (4)
                      dev_nbr,
                      config_nbr_hs,
                     &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

(1) Class number to add to the configuration returned by USBD_Vendor_Add().

(2) Device number returned by USBD_DevAdd().

(3) Configuration number (here adding it to a Full-Speed configuration).

(4) Adding the same class instance to the High-Speed configuration. This should be done only if you have a High-Speed device.

Adding Microsoft Extended Properties to your Class Instance (Optional)

Once you have created a vendor class instance, you can add one or multiple Microsoft extended properties. Adding a Microsoft extended property is particularly useful so you can avoid having to provide an .inf file when connecting your device to a PC running Microsoft Windows OS.

You can add different types of extended properties, but the most common one is a property that will provide the GUID to Windows. For more information on GUID and on how Windows normally loads a driver for a USB device, see Host Operating Systems: Microsoft Windows . Adding a Microsoft Extended property is done by calling the function USBD_Vendor_MS_ExtPropertyAdd() .

> To add a Microsoft extended property to your vendor class instance, you must set the configuration USBD_CFG_MS_OS_DESC_EN to DEF_ENABLED. For more information on this configuration, see USB Device Compile-Time Configuration .

Listing - Adding a Microsoft GUID in the *Adding a USB Device Vendor Class Instance to your Device* page gives an example of how to add a property *DeviceInterfaceGUID* with a value of *{143F20BD-7BD2-4CA6-9465-8882F2156BD6}* to your Vendor class instance via the Microsoft Extended Properties.

Listing - Adding a Microsoft GUID

```
static  const  CPU_INT08U  App_USBD_Vendor_PropertyNameGUID[] = {
    'D', 0u, 'e', 0u, 'v', 0u, 'i', 0u, 'c', 0u, 'e', 0u,
    'I', 0u, 'n', 0u, 't', 0u, 'e', 0u, 'r', 0u, 'f', 0u, 'a', 0u, 'c', 0u, 'e', 0u,
    'G', 0u, 'U', 0u, 'I', 0u, 'D', 0u, 0u,  0u
};

static  const  CPU_INT08U  App_USBD_Vendor_MS_GUID[] = {
    '{', 0u, '1', 0u, '4', 0u, '3', 0u, 'F', 0u, '2', 0u, '0', 0u, 'B', 0u, 'D', 0u,
    '-', 0u, '7', 0u, 'B', 0u, 'D', 0u, '2', 0u, '-', 0u, '4', 0u, 'C', 0u, 'A', 0u, '6', 0u,
    '-', 0u, '9', 0u, '4', 0u, '6', 0u, '5', 0u,
    '-', 0u, '8', 0u, '8', 0u, '8', 0u, '2', 0u, 'F', 0u, '2', 0u, '1', 0u, '5', 0u, '6', 0u, 'B', 0u, 'D', 0u, '6', 0u,  '}', 0u, 0u, 0u
};

RTOS_ERR  err;

USBD_Vendor_MS_ExtPropertyAdd(class_nbr,
```

```
                USBD_MS_OS_PROPERTY_TYPE_REG_SZ,
                App_USBD_Vendor_PropertyNameGUID,sizeof(App_USBD_Vendor_PropertyNameGUID),
                App_USBD_Vendor_MS_GUID,sizeof(App_USBD_Vendor_MS_GUID),&err);if(err.Code != RTOS_ERR_NONE){/* An error occurred.
Error handling should be added here. */}
```

Communicating using the USB Device Vendor Class

- General
- Synchronous Communication
- Asynchronous Communication
- Vendor Request

General

The Vendor class offers the following functions to communicate with the host. For more details about the parameters of the function, see USB Device Vendor API .

Table - Vendor Communication API Summary

Function name	Operation
USBD_Vendor_Rd()	Receives data from host through bulk OUT endpoint. This function is blocking.
USBD_Vendor_Wr()	Sends data to host through bulk IN endpoint. This function is blocking.
USBD_Vendor_RdAsync()	Receives data from host through bulk OUT endpoint. This function is non-blocking.
USBD_Vendor_WrAsync()	Sends data to host through bulk IN endpoint. This function is non-blocking.
USBD_Vendor_IntrRd()	Receives data from host through interrupt OUT endpoint. This function is blocking.
USBD_Vendor_IntrWr()	Sends data to host through interrupt IN endpoint. This function is blocking.
USBD_Vendor_IntrRdAsync()	Receives data from host through interrupt OUT endpoint. This function is non-blocking.
USBD_Vendor_IntrWrAsync()	Sends data to host through interrupt IN endpoint. This function is non-blocking.

The vendor requests are also another way to communicate with the host. When managing vendor requests sent by the host, the application can receive or send data from or to the host using the control endpoint; you will need to provide an application callback passed as a parameter of USBD_Vendor_Add().

Synchronous Communication

Synchronous communication means that the transfer is blocking. When a function is called, the application blocks until the transfer completes with or without an error. A timeout can be specified to avoid waiting forever.

Listing - Synchronous Bulk Read and Write Example in the *Communicating using the USB Device Vendor Class* page shows a read and write example that receives data from the host using the bulk OUT endpoint and sends data to the host using the bulk IN endpoint.

Listing - Synchronous Bulk Read and Write Example

```
CPU_INT08U  rx_buf[2];
CPU_INT08U  tx_buf[2];
RTOS_ERR    err;

(void)USBD_Vendor_Rd(        class_nbr,                          (1)
                (void *)&rx_buf[0],                          (2)
                     2u,
                     0u,                                (3)
                    &err);
if (err.Code != USBD_ERR_NONE) {
    /* $$$$ Handle the error. */
}
```

```
(void *)&tx_buf[0],(4)2u,0u,(3)
                              DEF_FALSE,(5)&err);if(err.Code != USBD_ERR_NONE){/* $$$$ Handle the error. */}
```

(1) The class instance number created with USBD_Vendor_Add() provides an internal reference to the Vendor class to route the transfer to the proper bulk OUT or IN endpoint.

(2) The application must ensure that the buffer provided to the function is large enough to accommodate all the data. Otherwise, synchronization issues might happen.

(3) In order to avoid an infinite blocking situation, a timeout expressed in milliseconds can be specified. A value of '0' makes the application task wait forever.

(4) The application provides the initialized transmit buffer.

(5) If this flag is set to DEF_TRUE, and the transfer length is multiple of the endpoint maximum packet size, the device stack will send a zero-length packet to the host to signal the end of the transfer.

The use of interrupt endpoint communication functions, USBD_Vendor_IntrRd() and USBD_Vendor_IntrWr(), is similar to bulk endpoint communication functions presented in Listing - Synchronous Bulk Read and Write Example in the *Communicating using the USB Device Vendor Class* page.

Asynchronous Communication

Asynchronous communication means that the transfer is non-blocking. When a function is called, the application passes the transfer information to the device stack and does not block. Other application processing can be done while the transfer is in progress over the USB bus. Once the transfer has completed, a callback function is called by the device stack to inform the application about the transfer completion. Listing - Asynchronous Bulk Read and Write Example in the *Communicating using the USB Device Vendor Class* page shows an example of asynchronous read and write.

Listing - Asynchronous Bulk Read and Write Example

```
void App_USBD_Vendor_Comm (CPU_INT08U  class_nbr)
{
    CPU_INT08U  rx_buf[2];
    CPU_INT08U  tx_buf[2];
    RTOS_ERR    err;

    USBD_Vendor_RdAsync(         class_nbr,                            (1)
                        (void *)&rx_buf[0],                    (2)
                                 2u,
                                 App_USBD_Vendor_RxCmpl,                (3)
                                 DEF_NULL,                     (4)
                                &err);
    if (err.Code != USBD_ERR_NONE) {
        /* $$$$ Handle the error. */
    }
    USBD_Vendor_WrAsync(         class_nbr,                            (1)
                        (void *)&tx_buf[0],                    (5)
                                 2u,
                                 App_USBD_Vendor_TxCmpl,                (3)
                                 DEF_NULL,                     (4)
                                 DEF_FALSE,                    (6)
                                &err);

    if (err.Code != USBD_ERR_NONE) {
        /* $$$$ Handle the error. */
    }
}

static  void  App_USBD_Vendor_RxCmpl (CPU_INT08U   class_nbr,                (3)
                                      void        *p_buf,
                                      CPU_INT32U   buf_len,
                                      CPU_INT32U   xfer_len,
                                      void        *p_callback_arg,
                                      USBD_ERR     err)
{
    if (err.Code == USBD_ERR_NONE) {
        /* $$$$ Do some processing. */
    } else {
        /* $$$$ Handle the error. */
    }
}

static  void  App_USBD_Vendor_TxCmpl (CPU_INT08U   class_nbr,                (3)
                                      void        *p_buf,
                                      CPU_INT32U   buf_len,
                                      CPU_INT32U   xfer_len,
                                      void        *p_callback_arg,
                                      USBD_ERR     err)
{
    if (err.Code == USBD_ERR_NONE) {
        /* $$$$ Do some processing. */
    } else {
        /* $$$$ Handle the error. */
    }
}
```

(1) The class instance number provides an internal reference to the Vendor class to route the transfer to the proper bulk OUT or IN endpoint.

(2) The application must ensure that the buffer provided is large enough to accommodate all the data. Otherwise, there may be synchronization issues.

(3) The application provides a callback function pointer passed as a parameter. Upon completion of the transfer, the device stack calls this callback function so that the application can finalize the transfer by analyzing the transfer result. For

instance, on completion of a read operation, the application might perform processing on the received data. Upon write completion, the application can indicate if the write was successful and how many bytes were sent.

(4) An argument associated with the callback can be also passed. Then in the callback context, some private information can be retrieved.

(5) The application provides the initialized transmit buffer.

(6) If this flag is set to DEF_TRUE, and the transfer length is a multiple of the endpoint maximum packet size, the device stack will send a zero-length packet to the host to signal the end of transfer.

The use of interrupt endpoint communication functions, USBD_Vendor_IntrRdAsync() and USBD_Vendor_IntrWrAsync(), is similar to the bulk endpoint communication functions presented in Listing - Asynchronous Bulk Read and Write Example in the *Communicating using the USB Device Vendor Class* page.

Vendor Request

The USB 2.0 specification defines three types of requests: standard, class, and vendor. All standard requests are handled directly by the core layer, while class requests are managed by the proper associated class.

Vendor requests can be processed by the vendor class. To process vendor requests, you must provide an application callback as a parameter of USBD_Vendor_Add() . Once a vendor request is received by the USB device, it must be decoded properly. Listing - Example of Vendor Request Decoding in the *Communicating using the USB Device Vendor Class* page shows an example of vendor request decoding.

Certain requests may be required to receive from or send to the host during the data stage of a control transfer. If no data stage is present, you only have to decode the Setup packet. This example shows the three types of data stage management: no data, data OUT and data IN.

Listing - Example of Vendor Request Decoding

```
#define  APP_VENDOR_REQ_NO_DATA                  0x01u
#define  APP_VENDOR_REQ_RECEIVE_DATA_FROM_HOST      0x02u
#define  APP_VENDOR_REQ_SEND_DATA_TO_HOST          0x03u

#define  APP_VENDOR_REQ_DATA_BUF_SIZE              50u

static  CPU_INT08U   AppVendorReqBuf[APP_VENDOR_REQ_DATA_BUF_SIZE];

static  CPU_BOOLEAN  App_USBD_Vendor_VendorReq (       CPU_INT08U       class_nbr,
                                               CPU_INT08U       dev_nbr,
                                         const  USBD_SETUP_REQ  *p_setup_req)    (1)
{
   CPU_BOOLEAN  valid;
   RTOS_ERR     err_usb;
   CPU_INT16U   req_len;

   (void)&class_nbr;

   switch(p_setup_req->bRequest) {                                      (2)
      case APP_VENDOR_REQ_NO_DATA:                                      (3)
          APP_TRACE_DBG(("Vendor request [0x%X]:\r\n", p_setup_req->bRequest));
          APP_TRACE_DBG(("wIndex  = %d\r\n",          p_setup_req->wIndex));
          APP_TRACE_DBG(("wLength = %d\r\n",          p_setup_req->wLength));
          APP_TRACE_DBG(("wValue  = %d\r\n",          p_setup_req->wValue));
          valid = DEF_OK;
          break;

      case APP_VENDOR_REQ_RECEIVE_DATA_FROM_HOST:                       (4)
          req_len = p_setup_req->wLength;
          if (req_len > APP_VENDOR_REQ_DATA_BUF_SIZE) {
             return (DEF_FAIL);                       /* Not enough room to receive data.          */
          }
          APP_TRACE_DBG(("Vendor request [0x%X]:\r\n", p_setup_req->bRequest));
                                                   /* Receive data via Control OUT EP.          */
          (void)USBD_CtrlRx(         dev_nbr,
                            (void *)&AppVendorReqBuf[0u],
                                    req_len,
                                    0u,                  /* Wait transfer completion forever.         */
                                   &err_usb);
          if (err_usb.Code != USBD_ERR_NONE) {
             APP_TRACE_DBG(("Error receiving data from host: %d\r\n", err_usb));
             valid = DEF_FAIL;
          } else {
             APP_TRACE_DBG(("wIndex  = %d\r\n", p_setup_req->wIndex));
             APP_TRACE_DBG(("wLength = %d\r\n", p_setup_req->wLength));
             APP_TRACE_DBG(("wValue  = %d\r\n", p_setup_req->wValue));
             APP_TRACE_DBG(("Received %d octets from host via Control EP OUT\r\n", req_len));
             valid = DEF_OK;
          }
          break;

      case APP_VENDOR_REQ_SEND_DATA_TO_HOST:                            (5)
          APP_TRACE_DBG(("Vendor request [0x%X]:\r\n", p_setup_req->bRequest));
          req_len = APP_VENDOR_REQ_DATA_BUF_SIZE;
          Mem_Set((void *)&AppVendorReqBuf[0u],            /* Fill buf with a pattern.                  */
                  'A',
                  req_len);
                                                   /* Send data via Control IN EP.              */
          (void)USBD_CtrlTx(         dev_nbr,
                            (void *)&AppVendorReqBuf[0u],
                                    req_len,
                                    0u,                  /* Wait transfer completion forever.         */
                                    DEF_NO,
                                   &err_usb);
          if (err_usb.Code != USBD_ERR_NONE) {
             APP_TRACE_DBG(("Error sending data to host: %d\r\n", err_usb));
```

```
}else{APP_TRACE_DBG(("wIndex  = %d\r\n", p_setup_req->wIndex));APP_TRACE_DBG(("wLength = %d\r\n", p_setup_req->wLength));APP_TRACE_DBG(("wValue  = %d\r\n", p_setup_req->wValue));APP_TRACE_DBG(("Sent %d octets to host via Control EP IN\r\n", req_len));
             valid = DEF_OK;}break;

       default:(6)
           valid = DEF_FAIL;/* Request is not supported.               */break;}return(valid);}
```

(1) The core will pass the Setup packet content to your application. The structure USBD_SETUP_REQ contains the same fields as defined by the USB 2.0 specification (refer to section "9.3 USB Device Requests" of the specification for more details):

```
typedef  struct  usbd_setup_req {
   CPU_INT08U  bmRequestType;  /* Characteristics of request.                  */
   CPU_INT08U  bRequest;      /* Specific request.                            */
   CPU_INT16U  wValue;        /* Varies according to request.                 */
   CPU_INT16U  wIndex;        /* Varies according to request; typically used as index.*/
   CPU_INT16U  wLength;       /* Transfer length if data stage present.       */
} USBD_SETUP_REQ;
```

(2) Determine the request. You may use a switch statement if you are using different requests. In this example, there are three different requests corresponding to the three types of the data stage: APP_VENDOR_REQ_NO_DATA, APP_VENDOR_REQ_RECEIVE_DATA_FROM_HOST, and APP_VENDOR_REQ_SEND_DATA_TO_HOST.

(3) If no data stage is present, you only need to decode the other fields. The presence of a data stage or not is indicated by the field wLength being non-null or null.

(4) If the host sends data to the device, you must call the function USBD_CtrlRx() . The buffer provided should be able to contain up to wLength bytes. If any error occurs, return DEF_FAIL to the core that will stall the status stage of the control transfer, indicating to the host that the request cannot be processed. DEF_OK is returned in case of success.

(5) If the host receives data from the device, you must call the function USBD_CtrlTx(). If any error occurs, return DEF_FAIL to the core that will stall the status stage of the control transfer, indicating to the host that the request cannot be processed. DEF_OK is returned in case of success.

(6) In this example, all requests not recognized are marked by returning DEF_FAIL to the core. This one will stall the data or status stage of the control transfer indicating to the host that the request is not supported.

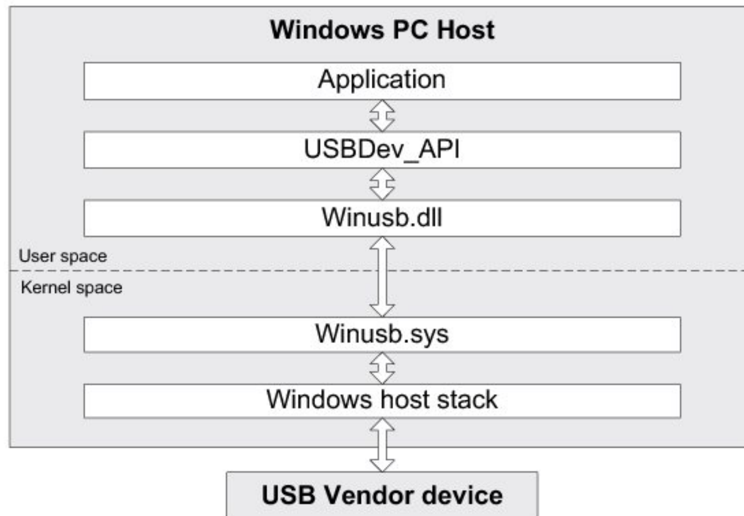
The host sends vendor requests using the function USBDev_CtrlReq(). Refer to the page USBDev_API for more details about how to send vendor requests on the host side.

USBDev_API

- Management
- Communication
  - Synchronous
  - Asynchronous
- Control Transfer

The Windows host application communicates with a vendor device through *USBDev_API*. The latter is a wrapper developed by Micrium allowing the application to access the WinUSB functionalities to manage a USB device. Windows USB (WinUSB) is a generic driver for USB devices. The WinUSB architecture consists of a kernel-mode driver (winusb.sys) and a user-mode dynamic link library (winusb.dll) that exposes WinUSB functions. USBDev_API eases the use of WinUSB by providing a comprehensive API (refer to the USBDev_API Functions Reference for the complete list). Figure - USBDev_API and WinUSB in the *USBDev_API* page shows the USBDev_API library and WinUSB.

Figure - USBDev_API and WinUSB



For more about WinUSB architecture, refer to Microsoft's MSDN online documentation at: http://msdn.microsoft.com/en-us/library/ff540207(v=VS.85).aspx

Management

USBDev_API offers the following functions to manage a device and its function's pipes.

Function name	Operation
USBDev_DevQtyGet	Gets number of devices belonging to a specified Globally Unique IDentifier (GUID) and connected to the host.
USBDev_Open()	Opens a device.
USBDev_Close	Closes a device.
USBDev_BulkIn_Open	Opens a bulk IN pipe.
USBDev_BulkOut_Open	Opens a bulk OUT pipe.
USBDev_IntrIn_Open	Opens an interrupt IN pipe.
USBDev_IntrOut_Open	Opens an interrupt OUT pipe.
USBDev_PipeClose	Closes a pipe.

Table - USBDev_API Device and Pipe Management API

Listing - USBDev_API Device and Pipe Management Example in the *USBDev_API* page shows an example of device and pipe management. The steps to manage a device typically consist in:

- Opening the vendor device connected to the host.
- Opening required pipes for this device.
- Communicating with the device via the open pipes.
- Closing pipes.
- Closing the device.

Listing - USBDev_API Device and Pipe Management Example

```
HANDLE  dev_handle;
HANDLE  bulk_in_handle;
HANDLE  bulk_out_handle;
DWORD   err;
DWORD   nbr_dev;
```

```
nbr_dev =USBDev_DevQtyGet(USBDev_GUID,&err);(1)if(err != ERROR_SUCCESS){/* $$$$ Handle the error. */}

dev_handle =USBDev_Open(USBDev_GUID,1,&err);(2)if(dev_handle == INVALID_HANDLE_VALUE){/* $$$$ Handle the error. */}

bulk_in_handle =USBDev_BulkIn_Open(dev_handle,0,0,&err);(3)if(bulk_in_handle == INVALID_HANDLE_VALUE){/* $$$$ Handle the error. */}

bulk_out_handle =USBDev_BulkOut_Open(dev_handle,0,0,&err);(3)if(bulk_out_handle == INVALID_HANDLE_VALUE){/* $$$$ Handle the error. */}/* Communicate with the device. */(4)USBDev_PipeClose(bulk_in_handle,&err);(5)if(err != ERROR_SUCCESS){/* $$$$ Handle the error. */}USBDev_PipeClose(bulk_out_handle,&err);if(err != ERROR_SUCCESS){/* $$$$ Handle the error. */}USBDev_Close(dev_handle,&err);(6)if(err != ERROR_SUCCESS){/* $$$$ Handle the error. */}
```

(1) Get the number of devices connected to the host under the specified GUID. A GUID provides a mechanism for applications to communicate with a driver assigned to devices in a class. The number of devices could be used in a loop to open at once all the devices. In this example, one device is assumed.

(2) Open the device by retrieving a general device handle. This handle will be used for pipe management and communication.

(3) Open a bulk pipe by retrieving a pipe handle. In the example, a bulk IN and a bulk OUT pipes are open. If the pipe does not exist for this device, an error is returned. When opening a pipe, the interface number and alternate setting number are specified. In the example, bulk IN and OUT pipes are part of the default interface. Opening an interrupt IN and OUT pipes with USBDev_IntIn_Open() or USBDev_IntOut_Open() is similar to bulk IN and OUT pipes.

(4) Transferring data on the open pipes can take place now. The pipe communication is described in the Communication section.

(5) Close a pipe by passing the associated handle. The closing operation aborts any transfer in progress for the pipe and frees any allocated resources.

(6) Close the device by passing the associated handle. The operation frees any allocated resources for this device. If a pipe has not been closed by the application, this function will close any forgotten open pipes.

Communication

Synchronous

Synchronous communication means that the transfer is blocking. Upon function call, the application blocks until the end of transfer is completed with or without an error. A timeout can be specified to avoid waiting forever. Listing - USBDev_API Synchronous Read and Write Example in the *USBDev_API* page presents a read and write example using a bulk IN pipe and a bulk OUT pipe.

Listing - USBDev_API Synchronous Read and Write Example

```
UCHAR  rx_buf[2];
UCHAR  tx_buf[2];
DWORD  err;

(void)USBDev_PipeRd(bulk_in_handle,                          (1)
                &rx_buf[0],                            (2)
                 2u,
                 5000u,                               (3)
                &err);
if (err != ERROR_SUCCESS) {
    /* $$$$ Handle the error. */
}

(void)USBDev_PipeWr(bulk_out_handle,                          (1)
                &tx_buf[0],                            (4)
                 2u,
                 5000u,                               (3)
                &err);
if (err != ERROR_SUCCESS) {
    /* $$$$ Handle the error. */
}
```

(1) The pipe handle gotten with USBDev_BulkIn_Open() or USBDev_BulkOut_Open() is passed to the function to schedule the transfer for the desired pipe.

(2) The application provides a receive buffer to store the data sent by the device.

(3) To avoid an infinite blocking situation, a timeout expressed in milliseconds can be specified. A value of '0' makes the application thread wait forever. In the example, a timeout of 5 seconds is set.

(4) The application provides the transmit buffer that contains the data for the device.

Asynchronous

Asynchronous communication means that the transfer is non-blocking. Upon function call, the application passes the transfer information to the device stack and does not block. Other application processing can be done while the transfer is in progress over the USB bus. Once the transfer has completed, a callback is called by USBDev_API to inform the application about the transfer completion.

Listing - USBDev_API Asynchronous Read Example in the *USBDev_API* page presents a read example. The asynchronous write is not offered by USBDev_API.

Listing - USBDev_API Asynchronous Read Example

```
UCHAR  rx_buf[2];
DWORD  err;

USBDev_PipeRdAsync(       bulk_in_handle,                          (1)
                   &rx_buf[0],                          (2)
                    2u,
                    App_PipeRdAsyncComplete,                    (3)
              (void *)0u,                          (4)
                   &err);
if (err != ERROR_SUCCESS) {
    /* $$$$ Handle the error. */
}

static  void  App_PipeRdAsyncComplete(void   *p_buf,                    (3)
                             DWORD   buf_len,
                             DWORD   xfer_len,
                             void   *p_callback_arg,
                             DWORD   err)
{
    (void)p_buf;
    (void)buf_len;
    (void)xfer_len;
    (void)p_callback_arg;                                (4)

    if (err == ERROR_SUCCESS) {
        /* $$$$ Process the received data. */
    } else {
        /* $$$$ Handle the error. */
    }
}
```

(1) The pipe handle gotten with USBDev_BulkIn_Open() is passed to the function to schedule the transfer for the desired pipe.

(2) The application provides a receive buffer to store the data sent by the device.

(3) The application provides a callback passed as a parameter. Upon completion of the transfer, USBDev_API calls this callback so that the application can finalize the transfer by analyzing the transfer result. For instance, upon read operation completion, the application may do a certain processing with the received data.

(4) An argument associated to the callback can also be passed. Then, in the callback context, some private information can be retrieved.

Control Transfer

You can communicate with the device through the default control endpoint by using the function USBDev_CtrlReq(). You will be able to define the three types of requests (standard, class or vendor) and to use the data stage or not of a control transfer to move data. More details about control transfers can be found in “Universal Serial Bus Specification, Revision 2.0, April 27, 2000”, section 5.5 and 9.3.

USB Device Troubleshooting

USB Device Troubleshooting

Te following are some common issues that you might encounter while developing an application using Micrium OS USB Device.

- My device is not detected by Windows.
- My device used to work properly, but now that I added/removed/modified a class instance, it behaves weirdly.
- My CDC or Vendor device is not working properly on Windows 8.
- My device has a low data transfer rate.
- My device works on Windows but not on Mac OS and/or Linux.
- My composite device (multi-function device) is not recognized by Windows.
- When I create a class instance or add a class instance to a configuration, I keep getting an error RTOS_ERR<rsrc>_ALLOC. What do I do?

My device is not detected by Windows.

There can be many causes for this issue. The first thing to verify is that you have properly called the functions USBD_DevAdd() and USBD_DevStart() for your USB controller, and that both functions return with the error code RTOS_ERR_NONE. Also make sure that you have added at least one configuration to your device with one class.

If you have written your own BSP, make sure the USB controller clock, power and I/O pins are properly configured, and check that the USB ISR is triggered when you connect the device to the host.

My device used to work properly, but now that I added/removed/modified a class instance, it behaves weirdly.

Once Windows associates one or more drivers with a given device, the association cannot be changed. So, if for instance, you connect a device with an HID class instance, and you modify it to be an MSC device and reconnect it, Windows will assume it is still an HID device. In order to force Windows to re-associate its drivers, you can either uninstall the device completely using the device manager or set a new Product ID in your USBD_DEV_CFG structure.

My CDC or Vendor device is not working properly on Windows 8.

Windows 8 requires that all the provided .inf files must be certified. For a device using the Vendor class, the Microsoft OS descriptors can be enabled using the configuration USBD_CFG_MS_OS_DESC_EN. For a CDC device, refer to USB Device CDC ACM Class Example Applications .

My device has a low data transfer rate.

There can be many causes for this issue. Here are some tips to increase the throughput:

- Transmit the largest buffer possible
- Determine if there is any special action that can be taken in your driver to improve the data throughput, such as DMA transfers, double buffering for bulk transfers, etc.

Keep in mind that if your device operates at Full-Speed, you will never get better throughput than ~850 KBytes/sec.

My device works on Windows but not on Mac OS and/or Linux.

The USB Device module has been successfully tested on both Linux and Mac OS. However, some classes may require certain specific configurations to operate correctly under Mac OS and/or Linux. Refer to the user manual of the class you use for more information.

Note that for our Vendor class, we provide a helper library (USBDev API) that simplifies the development of host applications. This library is compatible only with Windows.

My composite device (multi-function device) is not recognized by Windows.

If at least one of your device's function requires a .inf file to load a driver (such as CDC ACM and Vendor), our .inf example file is unlikely to work as-is. You will probably have to modify it. Refer to the About inf files section for more information.

When I create a class instance or add a class instance to a configuration, I keep getting an error RTOS_ERR<rsrc>_ALLOC. What do I do?

Micrium OS USB Device allocates a pool of each type of resources at initialization. Your application must specify the quantity of resources of each type to allocate. If you get a resource allocation error, you will have to increase the quantity of the given resource to allocate. For more information, refer to USB Device Run-Time Application-Specific Configuration.

Migration Guide to Silicon Labs USB Device stack

Migration Guide to Silicon Labs USB Device Stack

Overview

The following are main differences between the Silicon Labs USB Device stack and Micrium OS USB Device stack:

- Silicon Labs USB Device stack only supports one USB device.
- Silicon Labs USB Device stack does not support High Speed.
- Silicon Labs USB Device stack does not support the Audio class and the CDC EEM subclass.
- Silicon Labs USB Device stack does not support Isochronous endpoints.
- Silicon Labs USB Device stack uses compile-time #defines for configuration.
- Silicon Labs USB Device stack does not use dynamic allocation. Required data is allocated statically.
- Silicon Labs USB Device stack uses the CMSIS-RTOS2 abstraction layer and can therefore work with different OSes.
- Silicon Labs USB Device stack has a better hardware integration; No BSP and Platform Manager integration are required unlike with MicriumOS.
- Silicon Labs APIs return a `sl_status_t` integer code instead of the MicriumOS `RTOS_ERR` struct.

Initialization

Unlike Micrium OS, Silicon Labs USB Device stack does not use dynamic allocation to initialize the different pool objects needed. As a result, notice that you don't need to pass any `USBD_XXXX_QTY_CFG` type argument with initialization functions. Instead, set a compile-time #define-s in the *sl_usbd_core_config.h* file. The quantity fields in the different `USBD_XXXX_QTY_CFG` structures should all have an associated #define, except if the feature is no longer supported by Silicon Labs stack.

As mentioned, Silicon Labs stack only handles one USB Device. The driver for the USB device controller is integrated with the stack. As a result, you don't need a BSP or the Platform Manager. The only required hardware configuration is the GPIO setup when the VBUS SENSE signal is used, which is configurable inside the *sl_usbd_driver_config.h* file.

Note that if you are using Silicon Labs Simplicity Studio and the Project Configuration Tools, all your USB Device initialization code can be generated and integrated to the system initialization code of your project automatically. All you need to do is set up your project file accordingly. For examples about how to use Project Configuration Tools, see USB Device sample apps in Simplicity Studio.

See API function mapping tables at section API Names Equivalence to help migrate your code to Silicon Labs USB Device stack. Examples below also show the main differences during the initialization process between Micrium OS-USBD and Silicon Labs USB Device.

Example - Core Initialization with Micrium OS shows how the USB Device core initialization was done with Micrium OS-USBD stack and Example - Core Initialization with Silicon Labs shows how to perform the same initialization with the Silicon Labs stack.

Example - Core Initialization with Micrium OS

```
CPU_INT08U      config_nbr_fs;
CPU_INT08U      dev_nbr;
USBD_QTY_CFG    cfg_qty_usbd;
USBD_DEV_CFG    cfg_usbd_dev;
USBD_DEV_DRV_CFG cfg_usbd_dev_drv;
USBD_DEV_CFG    cfg_usbd_dev;
USBD_DEV_DRV_CFG cfg_usbd_dev_drv;
RTOS_ERR        err;

// Initializing USB device module                    (1)
cfg_qty_usbd.DevQty = 1u;
cfg_qty_usbd.ConfigQty = 2u;
cfg_qty_usbd.IF_Qty = 10u;
cfg_qty_usbd.IF_AltQty = 10u;
cfg_qty_usbd.IF_GrpQty = 2u;
cfg_qty_usbd.EP_DescQty = 10u;
cfg_qty_usbd.URB_ExtraQty = 10u;
cfg_qty_usbd.StrQty = 15u;
cfg_qty_usbd.EP_OpenQty = 10u;

USBD_Init(&cfg_qty_usbd, &err);
if (err.Code != RTOS_ERR_NONE) {
  // An error occurred. Error handling should be added here.
}

// Adding Device with controller info                  (2)
cfg_usbd_dev.ProductID = 0x1234u;
cfg_usbd_dev.ProductID = 0x1001u;
cfg_usbd_dev.ProductID = 0x0001u;
cfg_usbd_dev.VendorID = 0xFFFEu;
cfg_usbd_dev.DeviceBCD = 0x0100u;
cfg_usbd_dev.ManufacturerStrPtr = "Micrium inc";
cfg_usbd_dev.ProductStrPtr = "Product";
cfg_usbd_dev.SerialNbrStrPtr = "1234567890ABCDEF";
cfg_usbd_dev.LangId = USBD_LANG_ID_ENGLISH_US;

cfg_usbd_dev_drv.EP_OpenQty = 10u;
cfg_usbd_dev_drv.URB_ExtraQty = 0u;

dev_nbr = USBD_DevAdd(EX_USBD_CTRLR_NAME,
                      DEF_NULL,
                      &cfg_usbd_dev,
                      &cfg_usbd_dev_drv,
                      DEF_NULL,
                      &err);
if (err.Code != RTOS_ERR_NONE) {
  // An error occurred. Error handling should be added here.
}

// Adding full-speed configuration                     (3)
config_nbr_fs = USBD_ConfigAdd(dev_nbr,
                               DEF_BIT_NONE,
                               100u,
                               USBD_DEV_SPD_FULL,
                               "Full-Speed config",
                               &err);
if (err.Code != RTOS_ERR_NONE) {
  // An error occurred. Error handling should be added here.
}
```

(1) Instead of calling `USBD_Init()` , call `sl_usbd_core_init()` . The `USBD_QTY_CFG` argument is not needed. Instead, set the pool quantities in the *sl_usbd_core_config.h* file.

(2) Because Silicon Labs USB Device stack supports only one USB Device, you don't need to call any API function to add a USB device. Instead of the `USBD_DEV_CFG` argument, the device information can be configured in the

sl_usbd_device_config.h file. Because the driver is integrated to the stack, no `USBD_DEV_DRV_CFG` argument needs to be passed to the USB device stack.

(3) Instead of calling `USBD_ConfigAdd()`, call `sl_usbd_core_add_configuration()` to add a new configuration. This new function does not take a *device_number* argument since only one device can be handled by the stack.

Example - Core Initialization with Silicon Labs

```
sl_status_t  status;
uint8_t      config_nbr_fs;

// Initializing USB device module
status = sl_usbd_core_init();
if (status != SL_STATUS_OK) {
  // An error occurred. Error handling should be added here.
}


// Adding a full-speed configuration to the device
status = sl_usbd_core_add_configuration(0,
                                        100u,
                                        SL_USBD_DEVICE_SPEED_FULL,
                                        "Full-Speed config",
                                        &config_nbr_fs);
if (status != SL_STATUS_OK) {
  // An error occured. Error handling should be added here.
}
```

Classes are also initialized differently. The following examples show CDC-ACM and highlight differences. Example - CDC ACM Class Initialization with Micrium OS shows how the CDC ACM Class was initialized with Micrium OS-USBD stack and Example - CDC ACM Class Initialization with Silicon Labs shows how the same is now done with Silicon Labs USB Device stack. Note that the migration process will be very similar for the other supported classes and therefore only the CDC ACM example is shown.

Example - CDC ACM Class Initialization with Micrium OS

```
CPU_INT08U       cdc_acm_nbr;
USBD_CDC_QTY_CFG cfg_qty_cdc;
RTOS_ERR         err;

// Initializing CDC base class                          (1)
cfg_qty_cdc.ClassInstanceQty = 1u;
cfg_qty_cdc.ConfigQty = 2u;
cfg_qty_cdc.DataIF_Qty = 2u;

USBD_CDC_Init(&cfg_qty_cdc, &err);
if (err.Code != RTOS_ERR_NONE) {
  // An error occurred. Error handling should be added here.
}

// Initializing ACM subclass                            (2)
USBD_ACM_SerialInit(1u, &err);
if (err.Code != RTOS_ERR_NONE) {
  // An error occurred. Error handling should be added here.
}

// Create CD ACM class instance                         (3)
cdc_acm_nbr = USBD_ACM_SerialAdd(64u,
                  (USBD_ACM_SERIAL_CALL_MGMT_DATA_CCI_DCI | USBD_ACM_SERIAL_CALL_MGMT_DEV),
                  &err);
if (err.Code != RTOS_ERR_NONE) {
  // An error occurred. Error handling should be added here.
}

// Add class instance to config                         (4)
USBD_ACM_SerialConfigAdd(cdc_acm_nbr,
             dev_nbr,
             config_nbr_fs,
             &err);
if (err.Code != RTOS_ERR_NONE) {
  // An error occurred. Error handling should be added here.
}

// Register line coding and ctrl line change callbacks.          (5)
// These 2 calls are optinnal.
USBD_ACM_SerialLineCodingReg(cdc_acm_nbr,
               App_USBD_CDC_ACM_TerminalLineCoding,
               DEF_NULL,
               &err);
if (err.Code != RTOS_ERR_NONE) {
  // An error occurred. Error handling should be added here.
}

USBD_ACM_SerialLineCtrlReg(cdc_acm_nbr,
              App_USBD_CDC_ACM_TerminalLineCtrl,
              DEF_NULL,
              &err);
if (err.Code != RTOS_ERR_NONE) {
  // An error occurred. Error handling should be added here.
}
```

(1) Instead of calling `USBD_CDC_Init()`, call `sl_usbd_cdc_init()`. Instead of passing a configuration structure of type `USBD_QTY_CFG`, set the pool quantities in the *sl_usbd_core_config.h* file

(2) Instead of calling `USBD_ACM_SerialInit()`, call `sl_usbd_cdc_acm_init()`. Instead of passing a `subclass_instance_qty` argument, set the pool quantity for CDC ACM in the *sl_usbd_core_config.h* file.

(3) Instead of calling `USBD_ACM_SerialAdd()`, call `sl_usbd_cdc_acm_create_instance()`.

(4) Instead of calling `USBD_ACM_SerialConfigAdd()`, call `sl_usbd_cdc_acm_add_to_configuration()`. Not need for `dev_nbr` argument.

(5) Instead of using functions `USBD_ACM_SerialLineCodingReg()` and `USBD_ACM_SerialLineCtrlReg()` to register callback functions, pass an `sl_usbd_cdc_acm_callbacks_t` argument to `sl_usbd_cdc_acm_create_instance()`.

Example - CDC ACM Class Initialization with Silicon Labs

```
sl_status_t  status;
uint8_t      subclass_nbr;


// Callback functions
sl_usbd_cdc_acm_callbacks_t app_usbd_cdc_acm_callbacks = {
  app_usbd_cdc_acm_connect,
  app_usbd_cdc_acm_disconnect,
  app_usbd_cdc_acm_line_control_changed,
  app_usbd_cdc_acm_line_coding_changed,
};

// Initializing CDC base class
status = sl_usbd_cdc_init();
if (status != SL_STATUS_OK) {
  // An error occurred. Error handling should be added here.
}

// Initializing ACM subclass
status = sl_usbd_cdc_acm_init();
if (status != SL_STATUS_OK) {
  // An error occurred. Error handling should be added here.
}

// Create CD ACM class instance
status = sl_usbd_cdc_acm_create_instance(64u,
                                         SL_USBD_ACM_SERIAL_CALL_MGMT_DATA_CCI_DCI | SL_USBD_ACM_SERIAL_CALL_MGMT_DEV,
                                         app_usbd_cdc_acm_callbacks,
                                         &subclass_nbr);
if (status != SL_STATUS_OK) {
  // An error occurred. Error handling should be added here.
}

// Add class instance to config
status = sl_usbd_cdc_acm_add_to_configuration(subclass_nbr,
                                              config_nbr_fs);
if (status != SL_STATUS_OK) {
  // An error occurred. Error handling should be added here.
}
```

API Names Equivalence

Tables below show the API mapping between the deprecated MicriumOS-USBD stack and the new Silicon Labs USB Device stack.

- Core API Functions
- CDC ACM Class API Functions
- HID Class API Functions
- MSC Class API Functions
- Vendor Class API Functions

Core API Functions

MicriumOS USBD	Silicon Labs USB Device	Information
USBD_ConfigureBufAlignOctets()	NA	The internal buffers are already aligned to match the USB Controller requirement.
USBD_ConfigureMemSeg()	NA	Silicon Labs USB device stack does not use dynamic allocation. Required data is allocated statically.
USBD_Init()	sl_usbd_core_init()	
USBD_DevAdd()	NA	Silicon Labs USB Device Stack supports only one device that is automatically added when initializing the stack.
USBD_DevTaskPrioSet()	-	You can configure the USB Device core task priority at compile-time with `SL_USBD_TASK_PRIORITY` configuration.
USBD_DevNbrGetFromName()	NA	Only one device is supported with Silicon Labs USB device stack.
USBD_DevStart()	sl_usbd_core_start_device()	With Silicon Labs USB Device stack, `SL_USBD_AUTO_START_USB_DEVICE` configuration allows you to automatically start the device when enabled. In that case, no need to call `sl_usbd_core_start_device()` .
USBD_DevStop()	sl_usbd_core_stop_device()	
USBD_DevStateGet()	sl_usbd_core_get_device_state()	
USBD_DevSpdGet()	sl_usbd_core_get_device_speed()	
USBD_DevSelfPwrSet()	sl_usbd_core_set_device_self_power()	
USBD_DevSetMS_VendorCode()	NA	Microsoft OS descriptors are not supported by Silicon Labs USB Device stack.
USBD_DevGetCfg()	sl_usbd_core_get_device_configuration()	
USBD_DevFrameNbrGet()	sl_usbd_core_get_device_frame_number()	
USBD_ConfigAdd()	sl_usbd_core_add_configuration()	
USBD_ConfigOtherSpeed()	NA	Silicon Labs USB device stack does not support High Speed mode.

MicriumOS USBD	Silicon Labs USB Device	Information
USBD_IF_Add()	sl_usbd_core_add_interface()	
USBD_IF_AltAdd()	sl_usbd_core_add_alt_interface()	
USBD_IF_Grp()	sl_usbd_core_add_interface_group()	
USBD_DescDevGet()	sl_usbd_core_get_device_descriptor()	
USBD_DescConfigGet()	sl_usbd_core_get_configuration_descriptor()	
USBD_DescStrGet()	sl_usbd_core_get_string_descriptor()	
USBD_StrAdd()	sl_usbd_core_add_string()	
USBD_StrIxGet()	sl_usbd_core_get_string_index()	
USBD_DescWr08()	sl_usbd_core_write_08b_to_descriptor_buf()	
USBD_DescWr16()	sl_usbd_core_write_16b_to_descriptor_buf()	
USBD_DescWr24()	sl_usbd_core_write_24b_to_descriptor_buf()	
USBD_DescWr32()	sl_usbd_core_write_32b_to_descriptor_buf()	
USBD_DescWr()	sl_usbd_core_write_buf_to_descriptor_buf()	
USBD_CtrlRx()	sl_usbd_core_write_control_sync()	
USBD_CtrlTx()	sl_usbd_core_read_control_sync()	
USBD_BulkAdd()	sl_usbd_core_add_bulk_endpoint()	
USBD_BulkRx()	sl_usbd_core_read_bulk_sync()	
USBD_BulkRxAsync()	sl_usbd_core_read_bulk_async()	
USBD_BulkTx()	sl_usbd_core_write_bulk_sync()	
USBD_BulkTxAsync()	sl_usbd_core_write_bulk_async()	
USBD_IntrAdd()	sl_usbd_core_add_interrupt_endpoint()	
USBD_IntrRx()	sl_usbd_core_read_interrupt_sync()	
USBD_IntrRxAsync()	sl_usbd_core_read_interrupt_async()	
USBD_IntrTx()	sl_usbd_core_write_interrupt_sync()	
USBD_IntrTxAsync()	sl_usbd_core_write_interrupt_async()	
USBD_IsocAdd()	NA	Isochronous endpoints are not supported.
USBD_IsocSyncRefreshSet()	NA	Isochronous endpoints are not supported.
USBD_IsocSyncAddrSet()	NA	Isochronous endpoints are not supported.
USBD_IsocRxAsync()	NA	Isochronous endpoints are not supported.
USBD_IsocTxAsync()	NA	Isochronous endpoints are not supported.
USBD_EP_TxZLP()	sl_usbd_core_endpoint_write_zlp()	
USBD_EP_RxZLP()	sl_usbd_core_endpoint_read_zlp()	
USBD_EP_Abort()	sl_usbd_core_abort_endpoint()	
USBD_EP_Stall()	sl_usbd_core_stall_endpoint()	

MicriumOS USBD	Silicon Labs USB Device	Information
USBD_EP_IsStalled()	sl_usbd_core_is_endpoint_stalled()	
USBD_EP_MaxPktSizeGet()	sl_usbd_core_get_max_endpoint_packet_size()	
USBD_EP_MaxPhyNbrGet()	sl_usbd_core_get_max_phy_endpoint_number()	
USBD_EP_MaxNbrOpenGet()	sl_usbd_core_get_max_open_endpoint_number()	

CDC ACM Class API Functions

MicriumOS USBD	Silicon Labs USB Device	Information
CDC API Functions		
USBD_CDC_ConfigureMemSeg()	NA	Silicon Labs USB device stack does not use dynamic allocation. Required data is allocated statically.
USBD_CDC_Init()	sl_usbd_cdc_init()	
USBD_CDC_Add()	sl_usbd_cdc_create_instance()	
USBD_CDC_ConfigAdd()	sl_usbd_cdc_add_to_configuration()	
USBD_CDC_IsConn()	sl_usbd_cdc_is_enabled()	
USBD_CDC_DataIF_Add()	sl_usbd_cdc_add_data_interface()	
USBD_CDC_DataRx()	sl_usbd_cdc_read_data()	
USBD_CDC_DataTx()	sl_usbd_cdc_write_data()	
USBD_CDC_Notify()	sl_usbd_cdc_notify_host()	
CDC ACM API Functions		
USBD_ACM_SerialConfigureBufAlignOctets()	NA	The internal buffers are already aligned to match the USB Controller requirement.
USBD_ACM_SerialConfigureMemSeg()	NA	Silicon Labs USB device stack does not use dynamic allocation. Required data is allocated statically.
USBD_ACM_SerialInit()	sl_usbd_cdc_acm_init()	
USBD_ACM_SerialAdd()	sl_usbd_cdc_acm_create_instance()	
USBD_ACM_SerialConfigAdd()	sl_usbd_cdc_acm_add_to_configuration()	
USBD_ACM_SerialIsConn()	sl_usbd_cdc_acm_is_enabled()	
USBD_ACM_SerialRx()	sl_usbd_cdc_acm_read()	
USBD_ACM_SerialTx()	sl_usbd_cdc_acm_write()	
USBD_ACM_SerialLineCtrlGet()	sl_usbd_cdc_acm_get_line_control_state()	
USBD_ACM_SerialLineCodingGet()	sl_usbd_cdc_acm_get_line_coding()	

MicriumOS USBD	Silicon Labs USB Device	Information
USBD_ACM_SerialLineCodingSet()	sl_usbd_cdc_acm_set_line_coding()	
USBD_ACM_SerialLineStateSet()	sl_usbd_cdc_acm_set_line_state_event()	
USBD_ACM_SerialLineStateClr()	sl_usbd_cdc_acm_clear_line_state_event()	
USBD_ACM_SerialLineCtrlReg()	-	Use the p_acm_callbacks argument of sl_usbd_cdc_acm_create_instance() to register callback functions. In the sl_usbd_cdc_acm_callbacks_t struct, you have the line_control_changed field to register a callback function to notify you of Line Control changes.
USBD_ACM_SerialLineCodingReg()	-	Use the p_acm_callbacks argument of sl_usbd_cdc_acm_create_instance() to register callback functions. In the sl_usbd_cdc_acm_callbacks_t struct, you have the line_coding_changed field to register a callback function to notify you of Line Coding changes.

HID Class API Functions

MicriumOS USBD	Silicon Labs USB Device	Information
USBD_HID_ConfigureBufAlignOctets()	NA	The internal buffers are already aligned to match the USB Controller requirement.
USBD_HID_ConfigureReportID_Qty()	-	You can configure the number of HID report IDs at compile-time with SL_USBD_HID_REPORT_ID_QUANTITY configuration.
USBD_HID_ConfigurePushPopItemsQty()	-	You can configure the number of HID push/pop items at compile-time with SL_USBD_HID_PUSH_POP_ITEM_QUANTITY configuration.
USBD_HID_ConfigureMemSeg()	NA	Silicon Labs USB device stack does not use dynamic allocation. Required data is allocated statically.
USBD_HID_ConfigureTmrTaskStk()	-	You can configure the HID Timer task priority at compile-time with SL_USBD_TASK_PRIORITY configuration.
USBD_HID_Init()	sl_usbd_hid_init()	
USBD_HID_TmrTaskPrioSet()	-	You can configure the HID Timer task stack size at compile-time with SL_USBD_HID_TIMER_TASK_STACK_SIZE configuration.
USBD_HID_Add()	sl_usbd_hid_create_instance()	
USBD_HID_ConfigAdd()	sl_usbd_hid_add_to_configuration()	
USBD_HID_IsConn()	sl_usbd_hid_is_enabled()	

MicriumOS USBD	Silicon Labs USB Device	Information
USBD_HID_Rd()	sl_usbd_hid_read_sync()	
USBD_HID_Wr()	sl_usbd_hid_write_sync()	
-	sl_usbd_hid_write_async()	
-	sl_usbd_hid_read_async()	

MSC Class API Functions

MicriumOS USBD	Silicon Labs USB Device	Information
USBD_MSC_ConfigureBufAlignOctets()	NA	The internal buffers are already aligned to match the USB Controller requirement.
USBD_MSC_ConfigureDataBufLen()	-	You can configure the MSC buffer size at compile-time with SL_USBD_MSC_DATA_BUFFER_SIZE configuration.
USBD_MSC_ConfigureMemSeg()	NA	Silicon Labs USB device stack does not use dynamic allocation. Required data is allocated statically.
USBD_MSC_Init()	sl_usbd_msc_init() and sl_usbd_msc_scsi_init()	Silicon Labs USB Device splits the MSC class and SCSI subclass functions. Both init functions need to be call when SCSI subclass used.
USBD_MSC_Add()	sl_usbd_msc_create_instance() and sl_usbd_msc_scsi_create_instance()	Silicon Labs USB Device splits the MSC class and SCSI subclass functions. Only the SCSI function needs to be call when SCSI subclass is used.
USBD_MSC_TaskPrioSet()	-	You can configure each MSC instance task priority at compile-time with SL_USBD_MSC_SCSI_<INSTANCE>_TASK_PRIORITY configuration.
USBD_MSC_ConfigAdd()	sl_usbd_msc_add_to_configuration() and sl_usbd_msc_scsi_add_to_configuration()	Silicon Labs USB Device splits the MSC class and SCSI subclass functions. Only the SCSI function needs to be call when SCSI subclass is used.
USBD_MSC_IsConn()	sl_usbd_msc_is_enabled() and sl_usbd_msc_scsi_is_enabled()	
USBD_MSC_SCSI_LunAdd()	sl_usbd_msc_lun_add() and sl_usbd_msc_scsi_lun_add()	
USBD_MSC_SCSI_LunAttach()	sl_usbd_msc_scsi_lun_attach()	
USBD_MSC_SCSI_LunDetach()	sl_usbd_msc_scsi_lun_detach()	
-	sl_usbd_msc_scsi_lun_get_capacity()	

Vendor Class API Functions

MicriumOS USBD	Silicon Labs USB Device	Information
USBD_Vendor_ConfigureMsExtPropertiesQty()	NA	Microsoft OS descriptors are not supported by Silicon Labs USB Device stack.
USBD_Vendor_ConfigureMemSeg()	NA	Silicon Labs USB device stack does not use dynamic allocation. Required data is allocated statically.
USBD_Vendor_Init()	sl_usbd_vendor_init()	
USBD_Vendor_Add()	sl_usbd_vendor_create_instance()	
USBD_Vendor_ConfigAdd()	sl_usbd_vendor_add_to_configuration()	
USBD_Vendor_IsConn()	sl_usbd_vendor_is_enabled()	
USBD_Vendor_IntrRd()	sl_usbd_vendor_read_interrupt_sync()	
USBD_Vendor_IntrRdAsync()	sl_usbd_vendor_read_interrupt_async()	
USBD_Vendor_IntrWr()	sl_usbd_vendor_write_interrupt_sync()	
USBD_Vendor_IntrWrAsync()	sl_usbd_vendor_write_interrupt_async()	
USBD_Vendor_MS_ExtPropertyAdd()	NA	Microsoft OS descriptors are not supported by Silicon Labs USB Device stack.
USBD_Vendor_Rd()	sl_usbd_vendor_read_bulk_sync()	
USBD_Vendor_RdAsync()	sl_usbd_vendor_read_bulk_async()	
USBD_Vendor_Wr()	sl_usbd_vendor_write_bulk_sync()	
USBD_Vendor_WrAsync()	sl_usbd_vendor_write_bulk_async()	

USB-Host

USB Host

NOTE: This documentation refers to a deprecated software component that will no longer be supported and removed in an future release. Consider using the latest Silicon Labs USB stack instead. For more information, see USB Device.

USB is one of the most successful communication interface standards in the history of computer systems, and is the de-facto standard for connecting computer peripherals. Micrium OS USB Host is a USB host module designed specifically for embedded systems. Built from the ground up with Micrium's quality, scalability and reliability, it has gone through a rigorous validation process to comply with the USB 2.0 specification. This section describes how to initialize, start, and use Micrium OS USB Host. It provides details about the various configuration values and their uses, and a porting guide for your hardware. It also provides information such as overview, configuration possibilities, implementation details, and examples of typical usage for every available class. The examples featured in this documentation allow you to understand the USB concepts, and at the same time they provide a framework to quickly support devices connection of several types:

- Android accessories (AOAP)
- Mouse or keyboard (Human Interface Device Class)
- Removable storage device (Mass Storage Class)
- USB-to-serial adapter (FTDI, SiLabs, Prolific, ...) (USB2SER, CDC ACM)
- USB modem (CDC-ACM)
- External hubs

USB Host Overview

USB Host Overview

- Specifications
- Features

Specifications

- Complies with the "Universal Serial Bus specification revision 2.0"
- Implements the "Interface Association Descriptor Engineering Change Notice (ECN)"
- Transfer types
  - Control
  - Bulk
  - Interrupt
- USB classes
  - Android Open Accessory Protocol (AOAP)
  - Communication Device Class (CDC)
  - Abstract Control Model (ACM)
  - Human Interface Device (HID)
  - Mass Storage Class (MSC)
  - USB-to-serial (FTDI, Silicon Labs and Prolific adapters)

Features

- Scalable to contain only required features and minimize memory footprint
- Supports Low-speed (1.5Mbit/s), Full-speed (12 Mbit/s) and High-speed (480 Mbit/s)
- Supports multiple devices/external hubs
- Supports composite (multi-function) devices
- Supports multi-configuration devices
- Supports USB power saving functionalities (device suspend and resume)
- Complete integration of Mass Storage Class to Micrium OS File System module
- Extensive control on devices (reset, disconnect)
- Extensive control on memory allocations.
- Convenient "lsusb" command via Common's Shell module

Integrating USB Host Into Your Project

Integrating USB Host Into Your Project

Micrium OS USB Host is composed of several components, each of which is a set of files that implement specific functions. The USB Host module consists of one mandatory component for the core part named "USB Host" and several optional components for the various classes and drivers. At least one of each of these is mandatory. To use USB Host, you must add these files to your project and populate your RTOS description file .

Starting the USB Host Module Quickly

Micrium offers a set of example applications that demonstrate some of the features of Micrium OS USB Host and help you start the development of your application. Start from one of these examples.

The following sections describe each example applications available.

- USB Host Core Example Applications
- USB Host Android Accessory Class Example Applications
- USB Host CDC ACM Class Example Applications
- USB Host HID Class Example Applications
- USB Host MSC Class Example Applications
- USB Host USB-to-Serial Class Example Applications

USB Host Core Example Applications

USB Host Core Example Applications

This section describes the examples that are related to the core component of Micrium OS USB Host.

- USB Host Module Simple Initialization Example
  - Location
  - API
- Simple Device Port Operation Example
  - Description
  - Location
  - API

USB Host Module Simple Initialization Example

This is a generic example for the USB host module. It accomplishes the following tasks:

- Initialize the USB host core module with one host
- Initialize the PBHCI module, if it is part of your project
- Add one USB host controller
- Initialize all the available class application example(s)
- Start the USB host controller

This example is mandatory if you add any class application examples.

> By default, this example will assume the presence of a USB controller named "usb1" and will attempt to add it. If this device is not present on your platform or you prefer to use another one, modify the define EX_USBH_CTRLR_NAME accordingly.

Location

The example implementation is located in /examples/usb/host/ex_usbh.c and /examples/usb/host/ex_usbh.h.

API

This application offers two API named Ex_USBH_Init() and Ex_USBH_Start(). The function Ex_USBH_Init() must be called first from your application. Once your application is ready to accept device connections, you can call the function Ex_USBH_Start().

Simple Device Port Operation Example

Description

This is a simple example application that shows you how to perform basic operations on USB devices such as suspend, resume, and reset.

Location

The example implementation is located in /examples/usb/host/ex_usbh_dev_port_oper.c and /examples/usb/host/ex_usbh_dev_port_oper.h.

API

This application offers three API named Ex_USBH_DevSuspendExec() and Ex_USBH_DevResumeExec() and Ex_USBH_DevResetExec(). These functions can be called at any time after a device has been connected.

USB Host Configuration

USB Host Configuration

To configure Micrium OS USB Host, there are three groups of configuration parameters:

- USB Host Compile-Time Configurations
- USB Host Run-Time Application Specific Configurations
- USB Host and Host Controller Driver Configuration

USB Host Memory Schemes

- USBH_CFG_INIT_ALLOC_EN
- USBH_CFG_OPTIMIZE_SPD_EN

The USB Host module allocates internal objects (devices, functions, interface, endpoints, etc.) to manage the devices that are connected. Determining the number of objects that need to be allocated is complicated as it greatly depends on your application and on the devices that will be connected. Unfortunately, devices that appear to have the same functionalities — for example, two simple mouses with a few buttons — may use different approaches to present themselves to the host, and may require a different number of objects to be allocated. That is why, by default, the USB host module is configured to allocate all its objects dynamically. It uses dynamic memory pools provided by Common's Lib module, and allocates the objects from the memory segment(s) you passed during the initialization. For more information on the memory segments and the dynamic memory pools, see the Memory Segments and LIB Heap page.

This means that objects are allocated as the devices are connected. This does not create memory leaks; once a device is disconnected, the objects that were allocated for one device can then be used for another one.

There are two compile-time configurations that are related to memory allocations: USBH_CFG_OPTIMIZE_SPD_EN and USBH_CFG_INIT_ALLOC_EN. Both can be set to either DEF_ENABLED or DEF_DISABLED, which makes four possibilities. In the next sections, the four possibilities will be explained.

The following sections explain these two configurations.

USBH_CFG_INIT_ALLOC_EN

This configuration enables or disables the allocation of USB objects at initialization. When disabled, objects are allocated as the need arises (when devices are connected). The USB host module uses memory pools for its USB objects (devices, functions, interfaces, endpoints, etc.). These memory pools are global, which means that all the host controllers, devices, etc. will allocate their objects inside the same pools.

When USBH_CFG_INIT_ALLOC_EN is set to DEF_DISABLED, the global memory pools are created with an initial quantity of objects of 0 and with no maximum.

When USBH_CFG_INIT_ALLOC_EN is set to DEF_ENABLED, the global memory pools are created with an initial and maximum quantity of objects as given in the CFG_INIT_ALLOC structures.

Table - Benefits and downsides of allocation at initialization feature disabled in the *USB Host Memory Schemes* page shows the benefits and downsides of not allocating the USB objects at initialization.

Table - Benefits and downsides of allocation at initalization feature disabled

Benefits	Downsides
• No hard limit on any kind of USB objects. The only limitation is the size of the memory segment. • Memory allocation will always represent the worst case of a specific use case. • Much simpler configuration. • No need to determine the number of objects to create.	• No determinism in memory allocation. • Less efficient spatial location of memory.

USBH_CFG_OPTIMIZE_SPD_EN

This configuration controls the way the objects are retrieved internally. The objects in the stack are represented as a tree. So, when an object at the extremity of the tree must be retrieved, the USB host module must start at the root the tree. Each object maintains a list of their child objects, and this list can either be a simple linked list, or an array of pointers to these objects. Selecting which of methods to use is done by using the USBH_CFG_OPTIMIZE_SPD_EN configuration.

When USBH_CFG_OPTIMIZE_SPD_EN is set to DEF_DISABLED, a linked list is used to keep a reference to the child objects.

When USBH_CFG_OPTIMIZE_SPD_EN is set to DEF_ENABLED, an array of pointers is used. The size of these arrays is given in the CFG_OPTIMIZE_SPD structures.

Table - Benefits and downsides of optimization for speed feature disabled in the *USB Host Memory Schemes* page shows the benefits and downsides of not optimizing the USB host module for speed.

Table - Benefits and downsides of optimization for speed feature disabled

Benefits	Downsides
• No hard limit on tree element sizes (no maximum quantity of children per element). • Much simpler configuration. • No need to determine the maximum quantity of children per object. • Consumes less RAM in most cases.	• No determinism in execution time (need to browse a list to find element). • Slightly slower execution time (in average).

USB Host Compile-Time Configurations

- Core Configuration
- Class Drivers Configuration

Micrium OS USB Host is configurable at compile time via approximately a set of #defines located in usbh_cfg.h file. USB Host uses #defines when possible because they allow code and data sizes to be scaled at compile time based on enabled features. This allows the Read-Only Memory (ROM) and Random-Access Memory (RAM) footprints of Micrium OS USB Host to be adjusted based on application requirements.

We recommend that you begin the configuration process with the default configuration values, which are shown in **bold** in the next sections.

These sections are organized according to the order in Micrium OS USB Host's template configuration file, usbh_cfg.h.

Core Configuration

Table - Generic Configuration Constants

Constant	Description	Possible values
USBH_CFG_OPTIMIZE_SPD_EN	Optimizes for either better performance or for smallest code size. Enabling this define will optimize USB Host code for better performance, and disabling this define will lead to smaller code size. Enabling this configuration also causes the CFG_EXT configuration structures to become mandatory. See USB Host Memory Schemes for more information on this configuration.	DEF_ENABLED or **DEF_DISABLED**
USBH_CFG_INIT_ALLOC_EN	Allocate all USB resources at initialization or as they are needed. Enabling this ensures that all the USB resources (Device, Interfaces, Endpoints, etc) are allocated from the memory segment(s) during the initialization of the USB Host module. This requires you to provide extra configurations. Disabling this will cause the USB Host module to allocate the resources as devices get connected. Enabling this configuration also causes the CFG_EXT configuration structures to become mandatory. See USB Host Memory Schemes for more information on this configuration.	DEF_ENABLED or **DEF_DISABLED**
USBH_CFG_ALT_IF_EN	Enables or disables support for alternate USB interfaces.	DEF_ENABLED or **DEF_DISABLED**
USBH_CFG_STR_EN	Enables or disables support for USB string. Enabling this configuration allows you to retrieve the USB strings from the devices.	**DEF_ENABLED** or DEF_DISABLED
USBH_CFG_UNINIT_EN	Enables or disables support for the uninitialization of the USB host module.	DEF_ENABLED or **DEF_DISABLED**
USBH_CFG_FIELD_EN_MASK	The USB host module queries information on all the devices. But some information may be of no use to the core module or your application, and so does not need to be stored. This configuration allows you to determine which fields should be stored. This configuration is a bitmap and multiple values can be or'ed. Note that some class drivers may require some of these fields.	• USBH_CFG_FIELD_EN_DEV_SPEC_NBR • USBH_CFG_FIELD_EN_DEV_SUBCLASS • USBH_CFG_FIELD_EN_DEV_PROTOCOL • USBH_CFG_FIELD_EN_DEV_VENDOR_ID • USBH_CFG_FIELD_EN_DEV_PRODUCT_ID • USBH_CFG_FIELD_EN_DEV_REL_NBR • USBH_CFG_FIELD_EN_CONFIG_MAX_PWR • USBH_CFG_FIELD_EN_CONFIG_ATTR • USBH_CFG_FIELD_EN_FNCT_SUBCLASS • USBH_CFG_FIELD_EN_FNCT_PROTOCOL • USBH_CFG_FIELD_EN_IF_CLASS • USBH_CFG_FIELD_EN_IF_SUBCLASS • USBH_CFG_FIELD_EN_IF_PROTOCOL • USBH_CFG_FIELD_EN_ALL • USBH_CFG_FIELD_EN_NONE
USBH_CFG_PERIODIC_XFER_EN	Enables or disables support for periodic (interrupt and isochronous) transfers. This can be disabled if you don't use external hubs and you only use class drivers that don't use periodic transfers (such as MSC or AOAP, for instance). USBH_HUB_CFG_EXT_HUB_EN *must* be set to DEF_DISABLED to disable this feature.	**DEF_ENABLED** or DEF_DISABLED
USBH_HUB_CFG_EXT_HUB_EN	Enables or disables support for external hubs. If you plan to connect devices only to the root hub, you can disable this feature.	**DEF_ENABLED** or DEF_DISABLED

Class Drivers Configuration

Some class drivers may have specific compile-time configurations. Refer to the class driver manual section for more information.

USB Host Run-Time Application Specific Configurations

- Core Initialization
  - host_qty
- Optional Configurations
  - Buffer Alignment
  - Maximum Descriptor Length
  - Event Functions
  - Memory Segments
  - Asynchronous Task's Stack
  - Hub Task's Stack
  - Optimize Speed Quantities
  - Allocation at Initialization Quantities
- Post-Init Configurations
  - Standard Requests Timeout
  - Asynchronous Task Priority
  - Hub Task Priority
  - Preferred String Language ID

This section defines the configurations related to Micrium OS USB Host but which are specified during the initialization process. You may find that determining the appropriate values for some of these configurations is challenging, so we recommend that you consult this guide: USB Host Device Resource Needs .

Core Initialization

To initialize Micrium OS USB Host, you call the function USBH_Init(). This function takes one configuration argument that is described here.

host_qty

Determines the number of USB hosts you want to have. Each USB host has a unique set of device addresses. You should have one host for each host controller that you plan to use on your system.

Note that if a USB host controller uses companion controller(s), they must use the same host.

Optional Configurations

This section describes the configurations that are optional. If you do not set them in your application, the default configurations will apply.

The default values can be retrieved via the structure USBH_InitCfgDflt.Note that these configurations must be set *before* you call the function USBH_Init().

Buffer Alignment

This module allocates buffers used for data transfers with the devices. You may have a specific need for address alignment for these buffers depending on your USB controller. If you use more than one USB controller, you must set the alignment to the largest value.

Type	Function to call	Default	Field from default configuration structure
CPU_SIZE_T	USBH_ConfigureBufAlignOctets()	Size of cache line, or CPU alignment, if no cache.	.BufAlignOctets

Maximum Descriptor Length

The USB Host core module uses a single buffer to retrieve the different descriptors from the devices.

Type	Function to call	Default	Field from default configuration structure
CPU_INT16U	USBH_ConfigureMaxDescLen()	128	.MaxDescLen

Event Functions

Configures a set of application callbacks to be called on certain USB events (device connection, disconnection, etc.).

Type	Function to call	Default	Field from default configuration structure
USBH_EVENT_FNCTS	USBH_ConfigureEventFncts()	None.	.EventFnctsPtr

Memory Segments

This module allocates control data and buffers used for data transfers with the devices. It has the ability to use a different memory segment for the control data and for the data buffers.

Type	Function to call	Default	Field from default configuration structure
MEM_SEG*	USBH_ConfigureMemSeg()	General-purpose heap .	.MemSegPtr .MemSegBufPtr

Asynchronous Task's Stack

The asynchronous task handles all the USB transfer completions. This configuration allows you to set the stack pointer and the stack size (in quantity of elements).

Type	Function to call	Default	Field from default configuration structure
CPU_INT32Uvoid *	USBH_ConfigureAsyncTaskStk()	A stack of 512 elements allocated on Common 's memory segment.	.AsyncTaskStkSizeElements .AsyncTaskStkPtr

Hub Task's Stack

The hub task handles all the hub-related events such as device connection, disconnection, etc. This configuration allows you to set the stack pointer and the stack size (in quantity of elements).

Type	Function to call	Default	Field from default configuration structure
CPU_INT32Uvoid *	USBH_ConfigureHubTaskStk()	A stack of 768 elements allocated on Common 's memory segment.	.HubTaskStkSizeElements .HubTaskStkPtr

Optimize Speed Quantities

This configuration is mandatory *if* you set USBH_CFG_OPTIMIZE_SPD_EN to DEF_ENABLED and therefore USBH_ConfigureOptimizeSpdCfg() *must* be called from your application.

When the optimize speed mode is enabled, you must provide further information. When this mode is enabled, the module will use indexes and arrays to retrieve its internal objects from your handles instead of browsing through a list. This configuration specifies the size of each of these tables.This configuration is available only when USBH_CFG_OPTIMIZE_SPD_EN is set to DEF_ENABLED.

Type	Function to call	Default	Field from default configuration structure
USBH_CFG_OPTIMIZE_SPD	USBH_ConfigureOptimizeSpdCfg()	No default value.	.OptimizeSpd

Allocation at Initialization Quantities

This configuration is mandatory *if* you set USBH_CFG_INIT_ALLOC_EN to DEF_ENABLED and therefore USBH_ConfigureInitAllocCfg() *must* be called from your application.

When this module allocates all its objects at initialization, it must know how many objects of each type to allocate. This configuration specifies the number of objects to allocate.This configuration is available only when USBH_CFG_INIT_ALLOC_EN is set to DEF_ENABLED.

Type	Function to call	Default	Field from default configuration structure
USBH_CFG_INIT_ALLOC	USBH_ConfigureInitAllocCfg()	No default value.	.InitAlloc

Post-Init Configurations

This section describes the configurations that can be set at any time during execution *after* you called the function USBH_Init().

These configurations are optional. If you do not set them in your application, the default configurations will apply.

Standard Requests Timeout

Timeout, in milliseconds, for the standard requests executed by this module.

Type	Function to call	Default
CPU_INT32U	USBH_StdReqTimeoutSet()	5000

Asynchronous Task Priority

The USB host module will create a task that handles the data transfers completion. You can change the priority of the created task at any time.

Type	Function to call	Default
RTOS_TASK_PRIO	USBH_AsyncTaskPrioSet()	See Appendix A - Internal Tasks .

Hub Task Priority

The USB host module will create a task that handles the hub-related events. You can change the priority of the created task at any time.The Hub task *must* have a higher priority than any other USB host task or any other application tasks that perform USB transfers.

Type	Function to call	Default
RTOS_TASK_PRIO	USBH_HUB_TaskPrioSet()	See Appendix A - Internal Tasks .

Preferred String Language ID

Preferred language ID to use when retrieving USB strings from a device. Possible values can be found in the file usbh_core_langid.h.This configuration is available only when USBH_CFG_STR_EN is set to DEF_ENABLED.

Type	Function to call	Default
CPU_INT16U	USBH_PreferredStrLangID_Set()	USBH_LANGID_ENGLISH_UNITEDSTATES

USB Host Run-time Application Specific Configurations Event Functions

USBH_EVENT_FNCTS is a structure that contains application event functions. Its purpose is to provide a set of application callbacks that are called on certain USB events (device connection, disconnection, etc.).

For more information on these callbacks, see USB Host Event Callbacks Usage .

Table - USBH_EVENT_FNCTS callback structure in the *USB Host Run-time Application Specific Configurations Event Functions* page describes each configuration field available in this configuration structure.

Table - USBH_EVENT_FNCTS callback structure

Field	Description	Function signature
.DevConnAccept	A device was connected. Your application can decline the device connection by returning DEF_NO. Otherwise, DEF_YES must be returned.	CPU_BOOLEAN App_DevConnAccept (void)
.DevConfigAccept	The configuration of the newly connected device was retrieved. Your application can decline the configuration by returning DEF_NO. Otherwise, DEF_YES must be returned.	CPU_BOOLEAN App_DevConfigAccept(void)
.DevConfig	The connected device is now in the configured state. It is ready for communication.	void App_DevConfig (USBH_DEV_HANDLE dev_handle, CPU_INT08U cfg_nbr, RTOS_ERR err);
.FnctConnFail	A function connection of your newly connected device has failed. Refer to the error code (err) passed to this function for more information.	void App_FnctConnFail (USBH_FNCT_HANDLE fnct_handle, RTOS_ERR err);
.DevConnFail	The connection of your newly connected device has failed. Refer to the error code (err) passed to this function for more information.	void App_DevConnFail (CPU_INT08U hub_addr, CPU_INT08U port_nbr, RTOS_ERR err);
.DevDisconn	A device was disconnected.	void App_DevDisconn (USBH_DEV_HANDLE dev_handle)

USB Host Run-Time Application-Specific Configurations for Speed Optimization

The structure USBH_CFG_OPTIMIZE_SPD allows you to configure the number of references per object.

Table - USBH_CFG_OPTIMIZE_SPD configuration structure in the *USB Host Run-Time Application-Specific Configurations for Speed Optimization* page describes each configuration field available in this configuration structure.

Table - USBH_CFG_OPTIMIZE_SPD configuration structure

Field	Description
.HC_PerHostQty	Maximum number of host controllers per host. Unless you have a USB host controller that uses companion controllers, this should be set to 1.
.DevPerHostQty	Maximum number of connected devices per host that you want to support.
.FnctPerConfigQty	Maximum number of functions per configuration that you want to support. If you don't plan to use composite devices (multi-functions), this can be set to 1.
.IF_PerFnctQty	Maximum number of interfaces per function. For most of the devices, 1 will work here. If you use class drivers such as CDC ACM, 2 or more may be needed here, depending on your device.

USB Host Run-time Application Specific Configurations for Allocation at Initialization

The structure USBH_CFG_INIT_ALLOC allows you to configure the number of USB objects to allocate.

Table - USBH_CFG_INIT_ALLOC configuration structure in the *USB Host Run-time Application Specific Configurations for Allocation at Initialization* page describes each configuration field available in this configuration structure.

Table - USBH_CFG_INIT_ALLOC configuration structure

Field	Description
.DevQtyTot	Total number of devices.
.FnctQtyTot	Total number of USB functions.
.IF_QtyTot	Total number of interfaces.
.EP_QtyTot	Total number of endpoints.
.EP_QtyOpenTot	Total number of simultaneously opened endpoints.
.URB_QtyTot	Total number of USB request blocks.
.HubFnctQtyTot	Total number of Hub functions (excluding root hub(s)).
.HubEventQtyTot	Total number of Hub events.

USB Host and Host Controller Driver Configuration

- p_hc_cfg_ext
- USBH_HC_CFG_DEDICATED_MEM
- USBH_HC_CFG_OPTIMIZE_SPD
- USBH_HC_CFG_INIT_ALLOC

This section describes the configurations for the Micrium OS USB Host module that are specified at run-time, and which are specific to each added host controller.

To add a USB Host Controller, you call the USBD_HC_Add() function. This function takes one configurations argument that is described here.

p_hc_cfg_ext

p_hc_cfg_ext is a pointer to a configuration structure of type USBH_HC_CFG_EXT. It is an optional configuration structure that is necessary only if you enable the configurations USBH_CFG_OPTIMIZE_SPD_EN and/or USBH_CFG_INIT_ALLOC_EN. Otherwise, a null pointer (DEF_NULL) can be passed to this argument.

Table - USBH_HC_CFG_EXT configuration structure in the *USB Host and Host Controller Driver Configuration* page describes each configuration field available in this configuration structure.

Table - USBH_HC_CFG_EXT configuration structure

Field	Description	
.MaxPeriodicInterval	Maximum periodic interval for interrupt and isochronous endpoints. Any larger intervals will be reduced to this interval. This is an advanced configuration, so we recommend keeping the default value. Possible values are: • USBH_PERIODIC_XFER_MAX_INTERVAL_DFLT • USBH_PERIODIC_XFER_MAX_INTERVAL_001 • USBH_PERIODIC_XFER_MAX_INTERVAL_002 • USBH_PERIODIC_XFER_MAX_INTERVAL_004 • USBH_PERIODIC_XFER_MAX_INTERVAL_008 • USBH_PERIODIC_XFER_MAX_INTERVAL_016 • USBH_PERIODIC_XFER_MAX_INTERVAL_032 • USBH_PERIODIC_XFER_MAX_INTERVAL_064 • USBH_PERIODIC_XFER_MAX_INTERVAL_128 • USBH_PERIODIC_XFER_MAX_INTERVAL_256 • USBH_PERIODIC_XFER_MAX_INTERVAL_512 • USBH_PERIODIC_XFER_MAX_INTERVAL_1024 • USBH_PERIODIC_XFER_MAX_INTERVAL_2048 • USBH_PERIODIC_XFER_MAX_INTERVAL_4096 This configuration can be useful to reduce the RAM usage of your driver. However, depending on your host controller and driver, not all these values may be supported.	USBH_PERIODIC_XFER_MAX_INTERVAL_DFLT
.DedicatedMemCfgPtr	Pointer to a configuration structure of type USBH_HC_CFG_DEDICATED_MEM. Used to inform the driver on the size and number of data buffers to allocate in the dedicated memory. This is needed *only* if your USB controller requires dedicated memory for the data buffers. A null pointer (DEF_NULL) *must* be passed otherwise.	DEF_NULL
.CfgOptimizeSpdPtr	Pointer to a structure of type USBH_HC_CFG_OPTIMIZE_SPD. This is needed *only* if USBH_CFG_OPTIMIZE_SPD_EN is set to DEF_ENABLED. Should be a null pointer (DEF_NULL) otherwise.	DEF_NULL
.CfgInitAllocPtr	Pointer to a structure of type USBH_HC_CFG_INIT_ALLOC. This is needed *only* if USBH_CFG_INIT_ALLOC_EN is set to DEF_ENABLED. Should be a null pointer (DEF_NULL) otherwise.	DEF_NULL

USBH_HC_CFG_DEDICATED_MEM

If your USB controller is configured to use dedicated memory for the data buffers, you will have to provide additional information to the USB host controller driver.

Table - USBH HC CFG DEDICATED MEM configuration structure in the *USB Host and Host Controller Driver Configuration* page describes each configuration field available in this configuration structure.

Table - USBH_HC_CFG_DEDICATED_MEM configuration structure

Field	Description
.DataBufQty	Number of data buffers that should be allocated by the host controller in the dedicated memory. Any provided buffer that is not already inside the dedicated memory will first be copied into one of these buffers. The more buffers you allocate, the more transfers you will be able to queue. Keep in mind that if you allocate a buffer from the dedicated memory segment in your application, the driver won't copy it to one of these buffers, therefore improving the throughput.
.DataBufLen	Length in octets of each data buffer allocated in the dedicated memory.

USBH_HC_CFG_OPTIMIZE_SPD

If you set the configuration USBH_CFG_OPTIMIZE_SPD_EN to DEF_ENABLED, you will have to provide further information to the USB host controller driver. In this case, the USB host controller driver will use indexes and arrays to retrieve its internal objects instead of browsing through a list. So it is necessary to know the size of these tables.

Table - USBH_HC_CFG_OPTIMIZE_SPD configuration structure in the *USB Host and Host Controller Driver Configuration* page describes each configuration field available in this configuration structure.

Table - USBH_HC_CFG_OPTIMIZE_SPD configuration structure

Field	Description
.XferDescQty	Number of transfer descriptors.
.XferDescIsocQty	Number of isochronous transfer descriptors. Not supported for the moment, always set to 0.

USBH_HC_CFG_INIT_ALLOC

If you set the configuration USBH_CFG_INIT_ALLOC_EN to DEF_ENABLED, you must provide additional information to the USB host controller driver. In this case, the USB host controller driver will allocate all its USB objects at initialization and will be unable to allocate more objects during execution. So, you must provide the number of each object type to allocate. This greatly depends on your needs and on the USB devices you plan to use.

Table - USBH_HC_CFG_INIT_ALLOC configuration structure in the *USB Host and Host Controller Driver Configuration* page describes each configuration field available in this configuration structure.

Table - USBH_HC_CFG_INIT_ALLOC configuration structure

Field	Description
.EP_DescQty	Number of endpoint descriptors.
.EP_DescIsocQty	Number of isochronous endpoint descriptors. Not supported for the moment, always set to 0.
.XferDescQty	Number of transfer descriptors.
.XferDescIsocQty	Number of isochronous transfer descriptors. Not supported for the moment, always set to 0.

Note that if you enabled both USBH_CFG_OPTIMIZE_SPD_EN and USBH_CFG_INIT_ALLOC_EN, the values of the fields .XferDescQty and.XferDescIsocQty from both configuration structures *must* match.

USB Host Programming Guide

USB Host Programming Guide

This section explains how to use the USB Host module.

- Initial Setup of USB Host Module
- USB Host Event Callbacks Usage

Initial Setup of USB Host Module

This section describes the basic steps required to initialize the USB Host module and to add and start a host controller.

- Initializing the USB Host Module
  - Initializing the USB Host Core
  - Initializing the USB Host PBHCI Module (not always necessary)
  - Initializing the Class(es)
- Adding Your Host Controller(s)
- Starting Your Host Controller(s)
- A Note on the Usage of Companion Controllers
  - Add
  - Start

Initializing the USB Host Module

Initializing the USB Host Core

The first step is to initialize the USB host module core. This is done by calling the function USBH_Init().

Listing - Example of call to USBH_Init() in the *Initial Setup of USB Host Module* page shows an example of a call to USBH_Init() that will create a single host. For more information on the configuration arguments to pass to USBH_Init(), see USB Host Run-Time Application Specific Configurations .

Listing - Example of call to USBH_Init()

```
RTOS_ERR  err;

USBH_Init(1u,
       &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

Initializing the USB Host PBHCI Module (not always necessary)

If at least one of the drivers you use is of type PBHCD (Pipe-Based Host Controller Driver), you *must* initialize the Pipe-Based Host Controller Interface (PBHCI). This is done by calling the function USBH_PBHCI_Init().

For more information on the PBHCI, and to learn how to determine the type of your driver(s), refer to USB Host Hardware Porting Guide .

Listing - Example of call to USBH_PBHCI_Init() in the *Initial Setup of USB Host Module* page gives an example of a call to USBH_PBHCI_Init().

Listing - Example of call to USBH_PBHCI_Init()

```
RTOS_ERR  err;

USBH_PBHCI_Init(&err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

Initializing the Class(es)

Once the USB host module core has been initialized, you must initialize each class you intend to use. See the programming guide of your class(es) for more information.

Adding Your Host Controller(s)

Once you successfully initialized the USB host module, you can start adding your host controller(s). This is done by calling the function USBH_HC_Add(). This function must be called for each host controller you want to add.

Listing - Example of call to USBH_HC_Add() in the *Initial Setup of USB Host Module* page shows an example of a call to USBH_HC_Add() using default arguments. In this example, USB host controller "usb0" is added to the USB host module. For more information about the configuration arguments to pass to USBH_HC_Add(), see USB Host Controller Driver Configuration . For more information on how to register a USB controller (if you have to write your own BSP), see USB Host Hardware Porting Guide .

Listing - Example of call to USBH_HC_Add()

```
RTOS_ERR        err;
USBH_HC_HANDLE  hc_handle;

hc_handle = USBH_HC_Add("usb0",
                        DEF_NULL,
                        &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

If your platform uses companion controller(s) (for example, an OHCI controller that is a companion of an EHCI controller), the main controller *must* always be added *before* the companion controller(s).

Starting Your Host Controller(s)

Once you successfully added your host controller(s), you should start it/them. This is done by calling the function USBH_HC_Start(). This function must be called for each host controller you added.

Listing - Example of call to USBH_HC_Start() in the *Initial Setup of USB Host Module* page gives an example of call to USBH_HC_Start().

Listing - Example of call to USBH_HC_Start()

```
RTOS_ERR  err;

USBH_HC_Start(hc_handle,
              &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

If your platform uses companion controller(s) (for example, an OHCI controller that is a companion of an EHCI controller), the main controller *must* always be started *after* the companion controller(s).

A Note on the Usage of Companion Controllers

When a USB host controller uses one or more companion controller(s), each controller must be added separately. However, the order is very important for the add and start procedures.

Add

When adding the USB controllers (via the function USBH_HC_Add()), you must always add the main controller *first*. Then you can add the companion controller(s) in no particular order.

Start

When starting the USB controllers (via the function USBH_HC_Start()), you must always start the companion controller(s) *first*. Then you can start the main controller once its companion controller(s) are started.

USB Host Event Callbacks Usage

- DevConnAccept
- DevConfigAccept
- DevConfig
- FnctConnFail
- DevConnFail
- DevDisconn

The USB Host module offers a set of callbacks that are called when certain events occur. These callbacks are given to the USB host module via a configuration structure of type USBH_EVENT_FNCTS. Note that these callbacks are optional and you can define only the ones you actually need. Any callbacks in this structure can be set to null (DEF_NULL). Listing - Example of USBH_EVENT_FNCTS declaration in the *USB Host Event Callbacks Usage* page gives an example of how to create the structure of type USBH_EVENT_FNCTS. In the following sections, each callback function will be explained.

Listing - Example of USBH_EVENT_FNCTS declaration

```
USBH_EVENT_FNCTS  App_USBH_EventFncts = {
   .DevConnAccept   = App_USBH_DevConnAccept,
   .DevConfigAccept = App_USBH_DevConfigAccept,
   .DevConfig       = App_USBH_DevConfig,
   .FnctConnFail    = App_USBH_FnctConnFail,
   .DevConnFail     = App_USBH_DevConnFail,
   .DevResume       = DEF_NULL,            /* Device remote wake-up functionality not yet implemented. */
   .DevDisconn      = App_USBH_DevDisconn
};
```

DevConnAccept

This function will be called when a device is connected. When this function is called, the device is still in the addressed state (configuration information have not yet been retrieved). This is useful to deny specific device connections (for example, if too many devices are connected, or to filter device connections based on their vendor and product IDs).

If, while in this function, you call core functions that take a USBH_DEV_HANDLE as argument, the special value USBH_DEV_HANDLE_NOTIFICATION *must* be used.

This function returns a boolean. You must return DEF_YES if you accept the device connection, you must return DEF_NO otherwise. Note that if you set the .DevConnAccept field of your USBH_EVENT_FNCTS structure to DEF_NULL, all the device connections will be accepted.

Listing - Example of DevConnAccept callback function in the *USB Host Event Callbacks Usage* page shows an example of implementation of a DevConnAccept() function where a device that has a particular vendor ID will be denied.

Listing - Example of DevConnAccept callback function

```
CPU_BOOLEAN  App_USBH_DevConnAccept ()
{
    CPU_BOOLEAN  accept    = DEF_YES;
    CPU_INT16U   vendor_id;
    RTOS_ERR     err;

    vendor_id = USBH_DevVendorID_Get(USBH_DEV_HANDLE_NOTIFICATION,
                                    &err);
    if (err.Code != RTOS_ERR_NONE) {
        /* An error occurred. Error handling should be added here. */
    }

    if (vendor_id == <some denied vendor>) {
        accept = DEF_NO;
    }

    return (accept);
}
```

DevConfigAccept

This function will be called when a device is connected, just before setting the configuration. At this moment, all the configuration information has been retrieved. This is useful to deny enabling the configuration (for example, if the configuration will draw too much power).

If, while in this function, you call core functions that take a USBH_DEV_HANDLE as argument, the special value USBH_DEV_HANDLE_NOTIFICATION *must* be used.

This function returns a boolean. You must return DEF_YES if you accept the configuration, and you must return DEF_NO otherwise. Note that if you set the .DevConfigAccept field of your USBH_EVENT_FNCTS structure to DEF_NULL, all the configurations will be accepted.

Also note that if you deny the configuration, you can request another configuration to be set for this device later or before returning from this function by calling the function USBH_DevConfigSet().

Listing - Example of DevConfigAccept callback function in the *USB Host Event Callbacks Usage* page shows an example of implementation of a DevConfigAccept() function where a configuration that draws more than 250 mA will be denied.

Listing - Example of DevConfigAccept callback function

```
CPU_BOOLEAN  App_USBH_DevConfigAccept ()
{
    CPU_BOOLEAN  accept  = DEF_YES;
    CPU_INT16U   max_pwr;
    RTOS_ERR     err;

    max_pwr = USBH_ConfigMaxPwrGet(USBH_DEV_HANDLE_NOTIFICATION,
                                   &err);
    if (err.Code != RTOS_ERR_NONE) {
        /* An error occurred. Error handling should be added here. */
    }

    if ((max_pwr * 2u) > 250u) {
        accept = DEF_NO;
    }

    return (accept);
}
```

DevConfig

This function will be called when a device configuration has been set. At this moment, the device is ready for communication.

This function will also be called if the configuration set has failed. In this case, the argument err will give more information about the cause of the failure.

This function will provide a device handle to your application that can be saved for later use. From this point on (and within this function), the special value USBH_DEV_HANDLE_NOTIFICATION should *not* be used.

Listing - Example of DevConfig callback function in the *USB Host Event Callbacks Usage* page shows an example of implementation of a DevConfig() function where the application ensures that the configuration set was successful and saves the device handle if it was.

Listing - Example of DevConfig callback function

```
USBH_DEV_HANDLE  latest_dev_handle;

void  App_USBH_DevConfig (USBH_DEV_HANDLE  dev_handle,
                          CPU_INT08U       config_nbr,
                          RTOS_ERR         err)
{
    if (err.Code == RTOS_ERR_NONE) {
        latest_dev_handle = dev_handle;
    } else {
        /* Configuration set has failed. Error handling should be added here. */
    }
}
```

FnctConnFail

This function will be called when a function connection within a USB configuration has failed. This is useful to retrieve the cause of the failure. Also consider that if a multi-function device is connected, the configuration will be set as long as at least one function connected successfully.

Typical cause of failures are:

- Cannot allocate the required resources
- No class driver can handle the function
- Not enough bandwidth

The argument err should be checked to get more information on the cause of the failure.

Listing - Example of FnctConnFail callback function in the *USB Host Event Callbacks Usage* page shows an example of implementation of a FnctConnFail() function where the application retrieves the error code associated with the failure.

Listing - Example of FnctConnFail callback function

```
void  App_USBH_FnctConnFail (USBH_FNCT_HANDLE  fnct_handle,
                             RTOS_ERR          err)
{
    /* Function connection has failed. Error handling should be added here. */
}
```

DevConnFail

This function will be called when a device connection has failed. It can be use to retrieve the cause of the failure.

Typical cause of failures are:

- Cannot allocate the required resources
- No class driver can handle the device
- All the device's functions connection have failed

- Not enough bandwidth

The argument err should be checked to get more information on the cause of the failure.

Listing - Example of DevConnFail callback function in the *USB Host Event Callbacks Usage* page shows an example of implementation of a DevConnFail() function where the application retrieves the error code associated with the failure.

Listing - Example of DevConnFail callback function

```
void  App_USBH_DevConnFail (CPU_INT08U  hub_addr,
                            CPU_INT08U  port_nbr,
                            RTOS_ERR    err)
{
   /* Device connection has failed. Error handling should be added here. */
}
```

DevDisconn

This function will be called when a device is disconnected.

Listing - Example of DevDisconn callback function in the *USB Host Event Callbacks Usage* page shows an example of implementation of a DevDisconn() function.

Listing - Example of DevDisconn callback function

```
void  App_USBH_DevDisconn (USBH_DEV_HANDLE  dev_handle)
{
   /* Device was disconnected. All saved references to the device should be cleared. */
}
```

USB Host Device Resource Needs

USB Host Device Resource Needs

Configuring the USB Host module can be a challenge, as the configuration depends on the specifics of your application and the different devices you plan to connect. This guide is meant to help simplify the process of configuring the resources you need to allocate.

- Run-Time Configurations
  - General Configurations
  - Optimize Speed Configurations
  - Allocation at Initialization Configurations
- A Note on Each Device Type
  - Android Device
  - CDC ACM
  - HID
  - MSC
  - USB-To-Serial
  - External Hubs
- A Note on Composite Devices
- A Note on Compound Devices

Run-Time Configurations

Reference: USB Host Run-Time Application Specific Configurations .

General Configurations

Below is an estimate of the length, in octets, needed for each device type for the configuration descriptor. If you plan to support more than one device type, use the largest size.

Configuration	Device type	Value	Note
Maximum descriptor length	Android	35	Add 23 if debug interface is enabled.
	CDC ACM	90	If only one communication port.
	HID	100	
	MSC	40	
	USB-To-Serial adapter	10	If you have more than one serial port on your device, add 23 for each of them.
	External hub	30	

Optimize Speed Configurations

> These configurations are needed *only* if you set the configuration USBH_CFG_OPTIMIZE_SPD_EN to DEF_ENABLED.

If you plan to support more than one device type, use the largest size.

Field	Device type	Value	Note
`.FnctPerConfigQty`	Android	1	Add another function if debug interface is enabled.
	CDC ACM	1	
	HID	3	
	MSC	1	
	USB-To-serial adapter	1	Per serial port on device.
	External hub	1	
`.IF_PerFnctQty`	Android	1	
	CDC ACM	2	
	HID	1	
	MSC	1	
	USB-To-Serial adapter	1	
	External hub	1	

Allocation at Initialization Configurations

> These configurations are needed *only* if you set the configuration USBH_CFG_INIT_ALLOC_EN to DEF_ENABLED.

This is per device of each type. You must use the sum of all the devices.

Field	Device type	Value	Note
`.FnctQtyTot`	Android	1	Add another function if debug interface is enabled.
	CDC ACM	1	
	HID	3	
	MSC	1	
	USB-To-Serial adapter	1	Per serial port on device.
	External hub	1	
`.IF_QtyTot`	Android	1	Add another interface if debug interface is enabled.
	CDC ACM	2	
	HID	3	
	MSC	1	
	USB-To-Serial adapter	1	Per serial port on device.
	External hub	1	
`.EP_QtyTot`	Android	2	Add 2 other endpoints if debug interface is enabled.
	CDC ACM	3	
	HID	3	
	MSC	3	
	USB-To-Serial adapter	2	Per serial port on device, three for Prolific.
	External hub	1	
`.EP_QtyOpenTot*`	Android	2	
	CDC ACM	3	2 if `USBH_CDC_CFG_NOTIFICATIONS_RX_EN` set to `DEF_DISABLED`
	HID	3	
	MSC	2	
	USB-To-Serial adapter	2	3 if Prolific adapter driver is used and if `USBH_USB2SER_CFG_NOTIFICATIONS_RX_EN` is set to `DEF_ENABLED`
	External hub	1	

*The control endpoints must also be taken into consideration and represent 1 endpoint per device.

A Note on Each Device Type

Android Device

In accessory mode, an Android device is composed of a single interface with two bulk endpoints. These resources must be taken into consideration when configuring the USB host core module.

Also note that, when switching to the accessory mode, the Android device may switch into debug mode. The debug interface will show up only if the debugging mode has been enabled on the Android device. Since the USB host module does not offer a class driver for the debug interface, the host core module will not attempt to open the endpoints or allocate class a function. However, the core module will have to allocate the resources to represent this functionality.

So, if your plan is to support any kind of Android devices, you have to take into considerations the case where a device will present the debug functionality when configuring the core module.

CDC ACM

CDC ACM devices are special devices that use more than one interface per USB function. They are composed of one Communication Control Interface (CCI) and one or more (but generally only one) Data Control Interfaces. The CCI generally implements an interrupt endpoint and each DCI a pair of bulk endpoints.

HID

A typical HID device (such as a mouse or keyboard) has only one interface/function, which implements only one interrupt endpoint. However, some HID device manufacturers use the same firmware for all their HID device types and present three different functionalities to the host: Mouse, keyboard, and vendor-specific interface. Even though only one function is useful, they must all be allocated. The numbers given in the table take those exceptions into consideration. If you plan on supporting only a limited set of HID devices and you know they don't use such generic firmware, you can reduce the numbers accordingly.

MSC

MSC devices generally have very simple setups where only one function/interface and two bulk endpoints are declared. However, some MSC devices will declare an interrupt endpoint even though it is not used. This endpoint must be taken into consideration when configuring the value .EP_QtyTot.

USB-To-Serial

The USB-To-Serial class driver supports USB-to-serial adapters that do not follow any standard approved by the USB Implementers Forum. Hence, each device manufacturers use different protocols and configurations. Hence, each type of adapters may have different needs. While adapters from FTDI and Silicon Laboratories have similar setups where there is one function/interface with two bulk endpoints per serial port on the device, the adapters from Prolific Technology use an extra interrupt endpoint for notifications.

External Hubs

External hubs generally use only one function/interface per device with one interrupt endpoint. However, be aware that most external hubs that have more than 4 ports use multiple hubs. Hence, that must be taken into consideration when computing the needs in resources.

A Note on Composite Devices

A composite device (multi-function device) is composed of several functionalities. When computing the device needs, each function should be treated as a different device, except for the .MaxDescLen, for which you should use the sum of all the functions.

A Note on Compound Devices

Some common devices such as keyboards or media card readers embed a hub as well. They are called compound devices, as they are simply a hub with a permanently connected device (keyboard or MSC). That should be taken into consideration if you want to support any kind of devices.

USB Host Class Drivers

USB Host Class Drivers

The USB Host module offers several class drivers that allow you to provide support for a broad range of USB devices. In this section, each of the class drivers will be explained.

- USB Host Android Accessory Class Driver
- USB Host CDC Base Class Driver
- USB Host CDC ACM Class Driver
- USB Host HID Class Driver
- USB Host MSC Class Driver
- USB Host USB-to-Serial Class Driver

USB Host Android Accessory Class Driver

The Android Accessory class driver provided by Micrium OS USB Host is a class driver that will handle any Android devices (v3.1 and higher). It will switch them to the "Accessory Mode" and offers raw communication APIs to communicate with an embedded Android application.

The class driver implementation complies with AOAv1 as described here:https://source.android.com/devices/accessories/aoa.html.

USB Host Android Accessory Class Example Applications

Accessory Example

This example represents a template that you can use to start the development of your Android Accessory class project. When using this application, any Android devices connected to the host that supports the Android Open Accessory Protocol will be switched to the accessory mode.

The application offers blank functions that you can use to implement the communication with the Android device.

Location

The example implementation is located in /examples/usb/host/ex_usbh_aoap_accessory.c.

API

This example offers only one API named Ex_USBH_AOAP_Init(). This function is normally called from a USB host core example.

USB Host Android Accessory Class Programming Guide

- Prerequisites
- Initializing the USB Host Android Accessory Class Driver
- Using USB Host Android Accessory Class Application Functions
  - Conn
  - Disconn
- Communicating Using the USB Host Android Accessory Class Driver

Prerequisites

The following compile-time configurations from the USB Host core are required for this class driver to work properly.

Reference: USB Host Compile-Time Configurations .

- USBH_CFG_FIELD_EN_MASK: The following fields must be stored: USBH_CFG_FIELD_EN_DEV_VENDOR_ID and USBH_CFG_FIELD_EN_DEV_PRODUCT_ID.

Initializing the USB Host Android Accessory Class Driver

In order to support the Android function connections, you must first initialize the class driver. This is done by calling the function USBH_AOAP_Init().

Listing - Example of call to USBH_AOAP_Init() in the *USB Host Android Accessory Class Programming Guide* page shows an example of a call to USBH_AOAP_Init() using default arguments. For more information on the configuration arguments to pass to USBH_AOAP_Init(), see USB Host Android Accessory Class Configuration .

Listing - Example of call to USBH_AOAP_Init()

```
USBH_AOAP_APP_FNCTS  App_USBH_AOAP_Fncts = {
    .Conn     = App_USBH_AOAP_Conn,
    .Disconn = App_USBH_AOAP_Disconn
}

void  App_USBH_AOAP_Init()
{
    RTOS_ERR          err;
    USBH_AOAP_STR_CFG  aoap_str_cfg;

    aoap_str_cfg.AccStrManufacturer    = "Micrium inc";
    aoap_str_cfg.AccStrManufacturerLen =  11u;
    aoap_str_cfg.AccStrModel           = "AOAP demo";
    aoap_str_cfg.AccStrModelLen        =  9u;
    aoap_str_cfg.AccStrDescription     = "AOAP application for demonstration purposes";
    aoap_str_cfg.AccStrDescriptionLen  =  43u;
    aoap_str_cfg.AccStrVersion         = "1.0";
    aoap_str_cfg.AccStrVersionLen      =  3u;
    aoap_str_cfg.AccStrURI             = "www.micrium.com";
    aoap_str_cfg.AccStrURI_Len         =  15u;
    aoap_str_cfg.AccStrSerial          = "123456789";
    aoap_str_cfg.AccStrSerialLen       =  9u;

    USBH_AOAP_Init(&aoap_str_cfg,
                   &App_USBH_AOAP_Fncts,
                   &err);
    if (err.Code != RTOS_ERR_NONE) {
        /* An error occurred. Error handling should be added here. */
    }
}
```

Using USB Host Android Accessory Class Application Functions

The Android Accessory class driver provides two callback functions that will be called on connection and disconnection of an Android device.

These callbacks can have multiple purposes. Some of these include:

- Provide handles on the Android function to your application
- Execute some event driven simple task
- Trigger another task in your application
- Etc.

Below is a description of each of the available callback functions.

Conn

This function is called when an Android function is connected. It will provide all the necessary handles on the Android function to your application. Your application can also use it to return a pointer to an application object that is linked to this particular Android function. This pointer will be passed to any other callback function that is related to the Android function.

Listing - Example of implementation of Conn function in the *USB Host Android Accessory Class Programming Guide* page shows an example of implementation of the Conn callback function.

Listing - Example of implementation of Conn function

```
void  *App_USBH_AOAP_Conn(USBH_DEV_HANDLE        dev_handle,
                          USBH_FNCT_HANDLE       fnct_handle,
                          USBH_AOAP_FNCT_HANDLE  aoap_fnct_handle)
{
    /* An Android function was connected. aoap_fnct_handle should be saved in your application for later use. */

    /* You can allocate an object that will be linked to this Android function. */

    /* You can trigger one of your application task form here. */

    return (<pointer to application object>);   /* DEF_NULL can be returned as well. */
}
```

Disconn

This function is called when an Android function is disconnected.

Listing - Example of implementation of Disconn function in the *USB Host Android Accessory Class Programming Guide* page shows an example of implementation of the Conn callback function.

Listing - Example of implementation of Disconn function

```
void  App_USBH_AOAP_Disconn(USBH_AOAP_FNCT_HANDLE   aoap_fnct_handle,
                            void                   *p_arg)
{
    /* An Android function was disconnected. */

    /* p_arg is the pointer you returned in the Conn function associated to this function. */
    /* You can free the object if necessary.                                               */

    /* You can trigger one of your application task form here. */
}
```

Communicating Using the USB Host Android Accessory Class Driver

The Android Accessory Protocol does not define any data format for communication. So it is your responsibility to define one that will be understood by both your embedded and Android applications.

The USB Host Android Accessory class driver provides two raw communication functions: USBH_AOAP_AccDataRx() and USBH_AOAP_AccDataTx(). These functions will block until the data transfer is completed.

Listing - Simple loopback communication example in the *USB Host Android Accessory Class Programming Guide* page shows a simple example of a loopback communication.

Listing - Simple loopback communication example

```
void  App_USBH_AOAP_CommLoopback ()
{
    CPU_INT32U  xfer_len;
    CPU_INT08U  simple_buf[64u];
    RTOS_ERR    err;

                                                        /* Receive data from Android function.  */
    xfer_len = USBH_AOAP_AccDataRx(aoap_fnct_handle,
                                   simple_buf,
                                   64u,
                                   5000u,
                                  &err);
    if (err.Code != RTOS_ERR_NONE) {
        /* An error occurred. Error handling should be added here. */
    }

                                                        /* Send back data to Android function.  */
    (void)USBH_AOAP_AccDataTx(aoap_fnct_handle,
                              simple_buf,
                              xfer_len,
                              5000u,
                             &err);
    if (err.Code != RTOS_ERR_NONE) {
        /* An error occurred. Error handling should be added here. */
    }
}
```

USB Host Android Accessory Class Configuration

- AOAP initialization
  - p_str_cfg
  - p_app_fncts
- Optional Configurations
  - Buffer Alignment
  - Memory Segments
  - Optimize Speed Quantities
  - Allocation at initialization quantities
- Post-Init Configurations
  - Standard Requests Timeout

This section defines the configurations related to Micrium OS USB Host Android Accessory Class driver that are specified at run-time.

AOAP initialization

Initializing the USB Host Android Accessory class driver module of Micrium OS is done by calling the function USBH_AOAP_Init(). This function takes two configuration arguments that are described here.

p_str_cfg

p_str_cfg is a pointer to a configuration structure of type USBH_AOAP_STR_CFG. Its purpose is to inform the USB Host Android Accessory class driver module on Android Accessory descriptive Strings.

Note that all the descriptive strings MUST have a persistent storage.

Table - USBH_AOAP_STR_CFG configuration structure in the *USB Host Android Accessory Class Configuration* page describes each configuration field available in this configuration structure.

Table - USBH_AOAP_STR_CFG configuration structure

Field	Description
.AccStrManufacturer	String describing the Android Accessory manufacturer.
.AccStrManufacturerLen	Length, in octets, of the string describing the Android Accessory manufacturer.
.AccStrModel	String describing the Android Accessory model.
.AccStrModelLen	Length, in octets, of the string describing the Android Accessory model.
.AccStrDescription	String describing the Android Accessory.
.AccStrDescriptionLen	Length, in octets, of the string describing the Android Accessory.
.AccStrVersion	String of the Android Accessory version.
.AccStrVersionLen	Length, in octets, of the string of the Android Accessory version.
.AccStrURI	String of the Android Accessory URI.
.AccStrURI_Len	Length, in octets, of the string of the Android Accessory URI.
.AccStrSerial	String of the Android Accessory serial number.
.AccStrSerialLen	Length, in octets, of the string of the Android Accessory serial number.

p_app_fncts

p_app_fncts is a pointer to a structure of type USBH_AOAP_APP_FNCTS. Its purpose is to provide a set of application callbacks to be called when an Android Accessory function is connected and when it is disconnected. For more information on these callbacks, see USB Host Android Accessory Class Programming Guide .

The sections below describe each configuration field available in this configuration structure.

.Conn

An Android Accessory function was connected. This function provides the handles that were associated to it to your application.

Listing - .Conn function signature

```
void *App_USBH_AOAP_Conn (USBH_DEV_HANDLE dev_handle, USBH_FNCT_HANDLE fnct_handle,
USBH_AOAP_FNCT_HANDLE aoap_fnct_handle)
```

.Disconn

An Android Accessory function was disconnected. Once you return from this function, the handles passed at the Conn function for this Android Accessory function won't be valid anymore.

Listing - .Disconn function signature

```
void App_USBH_AOAP_Disconn(USBH_AOAP_FNCT_HANDLE aoap_fnct_handle, void *p_arg)
```

Optional Configurations

This section describes the configurations that are optional. If you do not set them in your application, the default configurations will apply.

The default values can be retrieved via the structure USBH_AOAP_InitCfgDflt.Note that these configurations must be set *before* you call the function USBH_AOAP_Init().

Buffer Alignment

This module allocates some buffers used for data transfers with the devices. You may have a specific need of address alignment for these buffers depending on your USB controller. If you use more than one USB controller, you must set the alignment to the largest value.

Type	Function to call	Default	Field from default configuration structure
CPU_SIZE_T	USBH_AOAP_ConfigureBufAlignOctets()	Size of cache line, or CPU alignment, if no cache.	.BufAlignOctets

Memory Segments

This module allocates some control data and buffers used for data transfers with the devices. It has the ability to use a different memory segment for the control data and for the data buffers.

Type	Function to call	Default	Field from default configuration structure
MEM_SEG*	USBH_AOAP_ConfigureMemSeg()	General-purpose heap .	.MemSegPtr .MemSegBufPtr

Optimize Speed Quantities

This configuration is mandatory IF you set USBH_CFG_OPTIMIZE_SPD_EN to DEF_ENABLED and therefore USBH_AOAP_ConfigureOptimizeSpdCfg() *must* be called from your application.

When the optimize speed mode is enabled, you have to provide further information to this module. In this case, this module will use indexes and arrays to retrieve its internal objects from your handles instead of browsing through a list. This configuration tells the size of each of these tables.This configuration is available only when USBH_CFG_OPTIMIZE_SPD_EN is set to DEF_ENABLED.

Type	Function to call	Default	Field from default configuration structure
USBH_AOAP_CFG_OPTIMIZE_SPD	USBH_AOAP_ConfigureOptimizeSpdCfg()	No default value.	.OptimizeSpd

Allocation at initialization quantities

This configuration is mandatory IF you set USBH_CFG_INIT_ALLOC_EN to DEF_ENABLED and therefore USBH_AOAP_ConfigureInitAllocCfg() *must* be called from your application.

When this module allocates all its objects at initialization, it must know how many objects of each type it must allocate. This configuration tells the number of objects to allocate.This configuration is available only when USBH_CFG_INIT_ALLOC_EN is set to DEF_ENABLED.

Type	Function to call	Default	Field from default configuration structure
USBH_AOAP_CFG_INIT_ALLOC	USBH_AOAP_ConfigureInitAllocCfg()	No default value.	.InitAlloc

Post-Init Configurations

This section describes the configurations that can be set at any time during execution AFTER you called the function USBH_AOAP_Init().

These configurations are optional. If you do not set them in your application, the default configurations will apply.

Standard Requests Timeout

Timeout, in milliseconds, for the standard requests executed by this module.

Type	Function to call	Default
CPU_INT32U	USBH_AOAP_StdReqTimeoutSet()	5000

USB Host Android Accessory Class Configuration For Speed Optimization

If you set USBH_CFG_OPTIMIZE_SPD_EN to DEF_ENABLED, you will have to provide additional information to the USB host Android Accessory class driver module. In this case, it will use indexes and arrays to retrieve its internal objects from your handles instead of browsing through a list. Hence, it is necessary to know the size of these tables.

Table - USBH_AOAP_CFG_OPTIMIZE_SPD configuration structure in the *USB Host Android Accessory Class Configuration For Speed Optimization* page describes each configuration field available in this configuration structure.

Table - USBH_AOAP_CFG_OPTIMIZE_SPD configuration structure

Field	Description
.FnctQty	Maximum quantity of connected Android Accessory functions.

Note that if you enabled both USBH_CFG_OPTIMIZE_SPD_EN and USBH_CFG_INIT_ALLOC_EN, the values of the field .FnctQty from both configuration structures *must* match.

USB Host Android Accessory Class Configuration For Allocation at Initialization

If you set USBH_CFG_INIT_ALLOC_EN to DEF_ENABLED, you will have to provide additional information to the USB host Android Accessory class driver module. In this case, it will allocate all its objects at initialization and will be unable to allocate more objects during execution. Hence, you must provide the number of each object type to allocate. This greatly depends on your needs and on the USB devices you plan to use.

Table - USBH_AOAP_CFG_INIT_ALLOC configuration structure in the *USB Host Android Accessory Class Configuration For Allocation at Initialization* page describes each configuration field available in this configuration structure.

Table - USBH_AOAP_CFG_INIT_ALLOC configuration structure

Field	Description
.FnctQty	Maximum number of connected Android Accessory functions.

Note that if you enabled both USBH_CFG_OPTIMIZE_SPD_EN and USBH_CFG_INIT_ALLOC_EN, the values of the field .FnctQty from both configuration structures *must* match.

USB Host CDC Base Class Driver

The CDC base class driver provided with Micrium OS USB Host is a class driver that handles any CDC devices for which a subclass driver is implemented.

The class driver implementation complies with the "Universal Serial Bus Class Definitions for Communication Devices",_revision_1.2,_November_3,_2010.

USB Host CDC Base Class Programming Guide

- Prerequisites
- Initializing the Class Driver
  - Class Driver Initialization
  - Class Driver Post-Initialization
- Communicating Using the Class Driver

Prerequisites

The following compile-time configurations in the USB Host core are required for this class driver to work properly.

Reference: USB Host Compile-Time Configurations .

- USBH_CFG_FIELD_EN_MASK: Following fields must be stored: USBH_CFG_FIELD_EN_IF_CLASS, USBH_CFG_FIELD_EN_IF_SUBCLASS and USBH_CFG_FIELD_EN_IF_PROTOCOL.
- USBH_CFG_PERIODIC_XFER_EN: Must be set to DEF_ENABLED if USBH_CDC_CFG_NOTIFICATIONS_RX_EN is set to DEF_ENABLED.

Initializing the Class Driver

Class Driver Initialization

In order to support the connections of CDC functions, you must first initialize the class driver. To do this, you call the function USBH_CDC_Init().

Listing - Example of call to USBH_CDC_Init() in the *USB Host CDC Base Class Programming Guide* page shows an example of a call to USBH_CDC_Init() using default arguments. For more information on the configuration arguments to pass to USBH_CDC_Init(), see USB Host CDC Base Class Configuration .

Listing - Example of call to USBH_CDC_Init()

```
void  App_USBH_CDC_Init()
{
   RTOS_ERR  err;

   USBH_CDC_Init(&err);
   if (err.Code != RTOS_ERR_NONE) {
      /* An error occurred. Error handling should be added here. */
   }

   :
   :
```

Class Driver Post-Initialization

Once the CDC class driver has been initialized, it is time to initialize all the CDC subclass driver(s). For more information on how to initialize your CDC subclass, refer to its section in USB Host Class Drivers.

Once, and only once you initialized all your CDC subclass driver(s), you must complete the initialization of the CDC base class driver. This is done by calling the function USBH_CDC_PostInit().

Listing - Example of call to USBH_CDC_PostInit() in the *USB Host CDC Base Class Programming Guide* page gives an example of a call to USBH_CDC_PostInit().

Listing - Example of call to USBH_CDC_PostInit()

```
:

   /* TODO: Initialize all your CDC subclass driver(s).    */

   :
   :

   USBH_CDC_PostInit(&err);
   if (err.Code != RTOS_ERR_NONE) {
      /* An error occurred. Error handling should be added here. */
   }
}
```

Communicating Using the Class Driver

The CDC base class driver is not meant to be used directly to communicate with a device via your application. You should always use a subclass driver (such as CDC ACM). For more information on how to communicate using your CDC subclass driver(s), refer to its section in USB Host Class Drivers .

USB Host CDC Base Class Configuration

- Compile-Time Configurations
- Run-Time Configurations
  - Class Driver Initialization

    - Buffer Alignment
    - Event URB Quantity
    - Memory Segments
    - Optimize Speed Quantities
    - Allocation at Initialization Quantities
- Post-Init Configurations
  - Standard Requests Timeout

This section describes the configurations related to Micrium OS USB Host CDC Class driver.

Compile-Time Configurations

The CDC class driver is configurable at compile time via one #define located in the usbh_cfg.h file.

We recommend that you begin the configuration process with the default configuration values, which in the next sections will be shown in **bold**.

Table - CDC Class Driver Compile-time Configurations in the *USB Host CDC Base Class Configuration* page describes the configuration.

Table - CDC Class Driver Compile-time Configurations

Constant	Description	Possible values
USBH_CDC_CFG_NOTIFICATIONS_RX_EN	Enables the notifications. CDC functions use interrupt endpoints to receive notifications. If your application does not need CDC notifications, you can disable them to save some resources. You can also disable the support for periodic transfers via the configuration USBH_CFG_PERIODIC_XFER_EN.	DEF_ENABLED or DEF_DISABLED

Run-Time Configurations

Class Driver Initialization

To initialize the USB Host CDC class driver module you call the function USBH_CDC_Init(). This function takes no configuration argument.

Optional Configurations

This section describes the configurations that are optional. If you do not set them in your application, the default configurations will apply. The default values can be retrieved via the structure USBH_CDC_InitCfgDflt.

> Note that these configurations must be set *before* you call the function USBH_CDC_Init().

Buffer Alignment

This module allocates buffers used for data transfers with the devices. You may have a specific need for address alignment for these buffers depending on your USB controller. If you use more than one USB controller, you must set the alignment to the largest value.

Type	Function to call	Default	Field from default configuration structure
CPU_SIZE_T	USBH_CDC_ConfigureBufAlignOctets()	Size of cache line, or CPU alignment, if no cache.	.BufAlignOctets

Event URB Quantity

Configures the number of event URBs to allocate and submit to the device. The more URBs submitted, the better will be the responsiveness to CDC events.

This configuration is ignored if USBH_CDC_CFG_NOTIFICATIONS_RX_EN is set to DEF_DISABLED.

Type	Function to call	Default	Field from default configuration structure
CPU_INT08U	USBH_CDC_ConfigureEventURB_Qty()	1	.EventURB_Qty

Memory Segments

This module allocates control data and buffers used for data transfers with the devices. It has the ability to use a different memory segment for the control data and for the data buffers.

Type	Function to call	Default	Field from default configuration structure
MEM_SEG*	USBH_CDC_ConfigureMemSeg()	General-purpose heap .	.MemSegPtr .MemSegBufPtr

Optimize Speed Quantities

This configuration is mandatory *if* you set USBH_CFG_OPTIMIZE_SPD_EN to DEF_ENABLED. Consequently, USBH_CDC_ConfigureOptimizeSpdCfg() *must* be called from your application.

When optimize speed mode is enabled, you must provide additional information to this module. The module will use indexes and arrays to retrieve its internal objects from your handles instead of browsing through a list. This configuration specifies the size of each of these tables.

This configuration is available only when USBH_CFG_OPTIMIZE_SPD_EN is set to DEF_ENABLED.

Type	Function to call	Default	Field from default configuration structure
USBH_CDC_CFG_OPTIMIZE_SPD	USBH_CDC_ConfigureOptimizeSpdCfg()	No default value.	.OptimizeSpd

Allocation at Initialization Quantities

This configuration is mandatory *if* you set USBH_CFG_INIT_ALLOC_EN to DEF_ENABLED . Consequently, USBH_CDC_ConfigureInitAllocCfg() *must* be called from your application.

When this module allocates its objects at initialization, it must know how many objects of each type to allocate. This configuration specifies the number of objects to allocate.

This configuration is available only when USBH_CFG_INIT_ALLOC_EN is set to DEF_ENABLED.

Type	Function to call	Default	Field from default configuration structure
USBH_CDC_CFG_INIT_ALLOC	USBH_CDC_ConfigureInitAllocCfg()	No default value.	.InitAlloc

Post-Init Configurations

This section describes the configurations that can be set at any time during execution *after* you called the function USBH_CDC_Init().

These configurations are optional. If you do not set them in your application, the default configurations will apply.

Standard Requests Timeout

Timeout, in milliseconds, for the standard requests executed by this module.

Type	Function to call	Default
CPU_INT32U	USBH_CDC_StdReqTimeoutSet()	5000

USB Host CDC Base Class Configuration for Speed Optimization

If you set the configuration USBH_CFG_OPTIMIZE_SPD_EN to DEF_ENABLED, you must provide additional information to the USB host CDC class driver module.It will use indexes and arrays to retrieve its internal objects from your handles instead of browsing through a list. So it is necessary to know the size of these tables.

Table - USBH_CDC_CFG_OPTIMIZE_SPD configuration structure in the *USB Host CDC Base Class Configuration for Speed Optimization* page describes each configuration field available in this configuration structure.

Table - USBH_CDC_CFG_OPTIMIZE_SPD configuration structure

Field	Description
.FnctQty	The maximum number of connected USB-to-serial functions.
.IF_PerFnctQty	The maximum number of interface per function.

Note that if you enabled both USBH_CFG_OPTIMIZE_SPD_EN and USBH_CFG_INIT_ALLOC_EN, the values of the field .FnctQty from both configuration structures *must* match.

USB Host CDC Base Class Configuration for Allocation at Initialization

If you set the configuration USBH_CFG_INIT_ALLOC_EN to DEF_ENABLED, you must provide additional information to the USB host CDC class driver module. It will allocate all its objects at the initialization and will be unable to allocate more objects during execution. So you must provide the number of each object type to allocate. This greatly depends on your needs and on the USB devices you plan to use.

Table - USBH_CDC_CFG_INIT_ALLOC configuration structure in the *USB Host CDC Base Class Configuration for Allocation at Initialization* page describes each configuration field available in this configuration structure.

Table - USBH_CDC_CFG_INIT_ALLOC configuration structure

Field	Description
.FnctQty	Maximum number of connected USB-to-serial functions.
.AsyncXferQty	Total number of TX asynchronous transfer. This represents the maximum quantity of asynchronous transfers that can be submitted at a given time for all the USB-to-serial functions.
.DCI_Qty	Total number of Data Control Interface (DCI).

Note that if you enabled both USBH_CFG_OPTIMIZE_SPD_EN and USBH_CFG_INIT_ALLOC_EN, the values of the field .FnctQty from both configuration structures *must* match.

USB Host CDC ACM Class Driver

The CDC ACM class driver provided by Micrium OS USB Host is a subclass driver implementation of the CDC base class. It handles CDC ACM devices such as USB-to-serial converters or modems.

The class driver implementation complies with the "Universal Serial Bus Communications Class Subclass Specification for PSTN Devices", revision 1.2, February 9, 2007.

CDC ACM Class Modem Example

This example will, upon connection of a CDC ACM device, issue a set of AT commands and display the results on the console. Note that in order to correctly execute this example, you will need a CDC ACM USB device that supports the AT commands.

The example accomplishes the following tasks:

- Initialize the CDC base class
- Initialize the ACM subclass
- Call the CDC base class Post Init function to inform the base class that all the subclasses have been initialized
- Allocate a buffer that is used to issue the AT commands and that holds the returned data
- Create a kernel task that will handle the AT commands issuing and that is triggered when a CDC ACM device is connected

The kernel task accomplishes the following tasks:

- Wait for the connection of a CDC ACM device
- Set a proper line coding
- Set the control line state
- Issue the following AT commands, and read and display the result:
  - *ATQ0V1E0*
  - *ATI0*
  - *ATI1*
  - *ATI2*
  - *ATI3*
  - *ATI7*

The example will also display the serial state changes on the console.

Location

The example implementation is located in /examples/usb/host/ex_usbh_cdc_acm_modem.c.

API

This example offers only one API named Ex_USBH_CDC_ACM_Init(). This function is normally called from a USB host core example.

USB Host CDC ACM Class Programming Guide

- Initializing the USB Host CDC ACM Class Driver
- Using USB Host CDC ACM Class Driver Application Functions
  - Conn
  - Disconn
  - NetConn
  - RespAvail
  - SerialStateChng
- Setting the Line Coding of a CDC ACM Function
- Communicating Using the CDC ACM Class Driver

Initializing the USB Host CDC ACM Class Driver

To support CDC ACM devices, you must first initialize the class driver. To begin, make sure you first have initialized the CDC base class. For more information, see USB Host CDC Base Class Programming Guide .

To initialize the CDC ACM class driver, you call the function USBH_ACM_Init().

Listing - Example of call to USBH_ACM_Init() in the *USB Host CDC ACM Class Programming Guide* page shows an example of a call to USBH_ACM_Init() using default arguments. For more information on the configuration arguments to pass to USBH_ACM_Init(), see USB Host CDC ACM Class Configuration .

Listing - Example of call to USBH_ACM_Init()

```
USBH_ACM_APP_FNCTS  App_USBH_ACM_Fncts = {
   .Conn            = App_USBH_ACM_Conn,
   .Disconn         = App_USBH_ACM_Disconn,
   .NetConn         = App_USBH_ACM_NetConn,
   .RespAvail       = App_USBH_ACM_RespAvail,
   .SerialStateChng = App_USBH_ACM_SerialStateChng
}

void  App_USBH_ACM_Init (void)
{
   RTOS_ERR  err;

   USBH_ACM_Init(&App_USBH_ACM_Fncts,
                 &err);
   if (err.Code != RTOS_ERR_NONE) {
      /* An error occured. Error handling should be added here. */
   }
}
```

Using USB Host CDC ACM Class Driver Application Functions

The CDC ACM class driver provides five callback functions that will be called when an event occurs on a CDC ACM function (such as connection and disconnection).

These callbacks can have multiple purposes. Some of these include:

- Provide handles on the CDC ACM function to your application
- Execute some event driven simple task
- Trigger another task in your application
- Etc.

The following is a description of each of the available callback functions.

Conn

This function is called when a CDC ACM function is connected. It will provide all the necessary handles on the CDC ACM function to your application. Your application can also use it to return a pointer to an application object that is linked to this particular CDC ACM function. This pointer will be passed to any other callback function that is related to the CDC ACM function.

Listing - Example of implementation of Conn function in the *USB Host CDC ACM Class Programming Guide* page shows an example of implementation of the *Conn* callback function.

Listing - Example of implementation of Conn function

```
void  *App_USBH_ACM_Conn(USBH_DEV_HANDLE       dev_handle,
                         USBH_FNCT_HANDLE      fnct_handle,
                         USBH_CDC_FNCT_HANDLE  cdc_fnct_handle,
                         USBH_ACM_FNCT_HANDLE  acm_fnct_handle)
{
   /* A CDC ACM function was connected. acm_fnct_handle should be saved in your application for later use. */

   /* You can allocate an object that will be linked to this CDC ACM function. */

   /* You can trigger one of your application task from here. */

   return (<pointer to application object>);   /* DEF_NULL can be returned as well. */
}
```

Disconn

This function is called when a CDC ACM function is disconnected.

Listing - Example of implementation of Disconn function in the *USB Host CDC ACM Class Programming Guide* page shows an example of implementation of the *Disconn* callback function.

Listing - Example of implementation of Disconn function

```
void  *App_USBH_ACM_Disconn(USBH_AOAP_FNCT_HANDLE   acm_fnct_handle,
                            void                   *p_arg)
{
    /* A CDC ACM function was disconnected. */

    /* p_arg is the pointer you returned in the Conn function associated to this function.    */
    /* You can free the object if necessary.                                                   */

    /* You can trigger one of your application task form here. */
}
```

NetConn

This function is called when a network connection/disconnection occurred on the CDC ACM function.

Listing - Example of implementation of NetConn function in the *USB Host CDC ACM Class Programming Guide* page shows an example of implementation of the *NetConn* callback function.

Listing - Example of implementation of NetConn function

```
void  App_USBH_ACM_NetConn(USBH_ACM_FNCT_HANDLE   acm_fnct_handle,
                           void                  *p_arg,
                           CPU_BOOLEAN            is_conn)
{
    /* A CDC ACM function has been connected (is_conn == DEF_YES) or                      */
    /* disconnected (is_conn == DEF_NO) from a network.                                   */

    /* p_arg is the pointer you returned in the Conn function associated to this function.  */

    /* You can trigger one of your application task form here.                            */
}
```

RespAvail

This function is called when a response is available from a CDC ACM function. The response can be retrieved by using the function USBH_ACM_EncapsulatedCmdRx().

Listing - Example of implementation of RespAvail function in the *USB Host CDC ACM Class Programming Guide* page shows an example of implementation of the *RespAvail* callback function.

Listing - Example of implementation of RespAvail function

```
void  App_USBH_ACM_RespAvail(USBH_ACM_FNCT_HANDLE   acm_fnct_handle,
                             void                  *p_arg)
{
    /* A CDC ACM function has received a response. */

    /* p_arg is the pointer you returned in the Conn function associated to this function. */

    /* You can trigger one of your application task form here. */
}
```

SerialStateChng

This function is called when the serial state changes on a CDC ACM function.

Listing - Example of implementation of SerialStateChng function in the *USB Host CDC ACM Class Programming Guide* page shows an example of implementation of the *SerialStateChng* callback function.

Listing - Example of implementation of SerialStateChng function

```
void  App_USBH_ACM_SerialStateChng(USBH_ACM_FNCT_HANDLE   acm_fnct_handle,
                                   void                  *p_arg,
                                   CPU_INT08U             serial_state)
{
    CPU_BOOLEAN  state_rx_carrier;
    CPU_BOOLEAN  state_tx_carrier;
    CPU_BOOLEAN  state_brk;
    CPU_BOOLEAN  state_ring_signal;
    CPU_BOOLEAN  state_framing;
    CPU_BOOLEAN  state_parity;
    CPU_BOOLEAN  state_overrun;

    /* A CDC ACM function serial state has changed. */

    /* p_arg is the pointer you returned in the Conn function associated to this function. */

    /* You can trigger one of your application task form here. */

    /* You can retrieve the new serial state like this: */

    state_rx_carrier  = DEF_BIT_IS_SET(serial_state, USBH_CDC_SERIAL_STATE_RXCARRIER);
    state_tx_carrier  = DEF_BIT_IS_SET(serial_state, USBH_CDC_SERIAL_STATE_TXCARRIER);
    state_brk         = DEF_BIT_IS_SET(serial_state, USBH_CDC_SERIAL_STATE_BRK);
    state_ring_signal = DEF_BIT_IS_SET(serial_state, USBH_CDC_SERIAL_STATE_RINGSIGNAL);
    state_rx_framing  = DEF_BIT_IS_SET(serial_state, USBH_CDC_SERIAL_STATE_FRAMING);
    state_rx_parity   = DEF_BIT_IS_SET(serial_state, USBH_CDC_SERIAL_STATE_PARITY);
    state_rx_overrun  = DEF_BIT_IS_SET(serial_state, USBH_CDC_SERIAL_STATE_OVERRUN);
}
```

Setting the Line Coding of a CDC ACM Function

The line coding of a CDC ACM function ca be set via the function USBH_ACM_LineCodingSet().

Listing - Example of CDC ACM function line coding set in the *USB Host CDC ACM Class Programming Guide* page shows an example of how to setting the line coding to standard values (9600 bps, 8 data bits, no parity, 1 stop bit).

Listing - Example of CDC ACM function line coding set

```
void App_USBH_ACM_LineCodingSet(USBH_ACM_FNCT_HANDLE  acm_fnct_handle)
{
    USBH_CDC_LINECODING  line_coding;
    RTOS_ERR             err;

    line_coding.Rate       = USBH_CDC_LINECODING_RATE_9600;
    line_coding.CharFmt    = USBH_CDC_LINECODING_STOP_BIT_1;
    line_coding.ParityType = USBH_CDC_LINECODING_PARITY_NONE;
    line_coding.DataBit    = USBH_CDC_LINECODING_DATA_BIT_8;

    USBH_ACM_LineCodingSet(acm_fnct_handle,
                          &line_coding,
                           5000u,
                          &err);
    if (err.Code != RTOS_ERR_NONE) {
        /* An error occured. Error handling should be added here. */
    }
}
```

Communicating Using the CDC ACM Class Driver

The CDC ACM class driver offers two functions to communicate with a CDC ACM function: USBH_ACM_RxAsync() and USBH_ACM_TxAsync().

Listing - Example of CDC ACM loopback communication in the *USB Host CDC ACM Class Programming Guide* page shows an example a simple loopback communication with a CDC ACM function.

Listing - Example of CDC ACM loopback communication

```
void  App_USBH_ACM_RxCmpl(USBH_ACM_FNCT_HANDLE   acm_fnct_handle,
                          CPU_INT08U            *p_buf,
                          CPU_INT32U             buf_len,
                          CPU_INT32U             xfer_len,
                          void                  *p_arg,
                          RTOS_ERR               err)
{
    RTOS_ERR  err_tx;

    if (err.Code != RTOS_ERR_NONE) {
        /* An error occured while receiving the data. Error handling should be added here. */

        return;
    }

    USBH_ACM_TxAsync(acm_fnct_handle,
                     p_buf,
                     buf_len,
                     App_USBH_ACM_TxCmpl,
                     DEF_NULL,
                    &err_tx);
    if (err_tx.Code != RTOS_ERR_NONE) {
        /* An error occured. Error handling should be added here. */
    }
}

void  App_USBH_ACM_TxCmpl(USBH_ACM_FNCT_HANDLE   acm_fnct_handle,
                          CPU_INT08U            *p_buf,
                          CPU_INT32U             buf_len,
                          CPU_INT32U             xfer_len,
                          void                  *p_arg,
                          RTOS_ERR               err)
{
    RTOS_ERR  err_rx;

    if (err.Code != RTOS_ERR_NONE) {
        /* An error occured while sending the data. Error handling should be added here. */

        return;
    }

    USBH_ACM_RxAsync(acm_fnct_handle,
                     p_buf,
                     buf_len,
                     App_USBH_ACM_RxCmpl,
                     DEF_NULL,
                    &err_rx);
    if (err_rx.Code != RTOS_ERR_NONE) {
        /* An error occured. Error handling should be added here. */
    }
}
```

USB Host CDC ACM Class Configuration

This section defines the configurations related to Micrium OS USB Host CDC ACM Class driver.

- Class Driver Initialization
  - p_acm_app_fncts
- Optional Configurations
  - Memory Segment
  - Optimize Speed Quantities
  - Allocation at Initialization Quantities
- Post-Init Configurations

- Standard Requests Timeout

Class Driver Initialization

To initialize the USB Host CDC ACM class driver module, you call the function USBH_ACM_Init() . This function takes one configuration argument that is described here.

p_acm_app_fncts

p_acm_app_fncts is a pointer to a structure of type USBH_ACM_APP_FNCTS. Its purpose is to provide a set of application callbacks to be called when an event occurs on a CDC ACM function (such as connection and disconnection).

For more information on these callbacks, see the USB Host CDC ACM Class Programming Guide .

The sections below describe each configuration field available in this configuration structure.

.Conn

A CDC ACM function was connected. This function provides the associated handles to your application.

Listing - .Conn function signature

```
void *App_USBH_ACM_Conn        (USBH_DEV_HANDLE       dev_handle,
                                USBH_FNCT_HANDLE      fnct_handle,
                                USBH_CDC_FNCT_HANDLE  cdc_fnct_handle,
                                USBH_ACM_FNCT_HANDLE  acm_fnct_handle)
```

.Disconn

A CDC ACM function was disconnected. Once you return from this function, the handles passed at the Conn function for this CDC ACM function won't be valid anymore.

Listing - .Disconn function signature

```
void App_USBH_ACM_Disconn      (USBH_ACM_FNCT_HANDLE  acm_fnct_handle,
                                void                 *p_arg)
```

.NetConn

Indicates a network connection or disconnection occurred on a CDC ACM function.

Listing - .NetConn function signature

```
void App_USBH_ACM_NetConn      (USBH_ACM_FNCT_HANDLE  acm_fnct_handle,
                                void                 *p_arg,
                                CPU_BOOLEAN           is_conn)
```

.RespAvail

Indicates a response is available on a CDC ACM function.

Listing - .RespAvail function signature

```
void App_USBH_ACM_RespAvail    (USBH_ACM_FNCT_HANDLE  acm_fnct_handle,
                                void                 *p_arg)
```

.SerialStateChng

Indicates the serial state has changed on a CDC ACM function. Also provides the new serial state.

Listing - .SerialStateChng function signature

```
void App_USBH_ACM_SerialStateChng(USBH_ACM_FNCT_HANDLE  acm_fnct_handle,
                                  void                 *p_arg,
                                  CPU_INT08U            serial_state)
```

Optional Configurations

This section describes the configurations that are optional. If you do not set them in your application, the default configurations will apply.

The default values can be retrieved via the structure USBH_ACM_InitCfgDflt.

Note that these configurations must be set *before* you call the function USBH_ACM_Init() .

Memory Segment

This module allocates control data from a memory segment.

Type	Function to call	Default	Field from default configuration structure
MEM_SEG*	USBH_ACM_ConfigureMemSeg()	General-purpose heap .	.MemSegPtr

Optimize Speed Quantities

This configuration is mandatory *if* you set USBH_CFG_OPTIMIZE_SPD_EN to DEF_ENABLED. Consequently, USBH_ACM_ConfigureOptimizeSpdCfg() *must* be called from your application.

When the optimize speed mode is enabled, you must provide additional information to this module. In this case, this module will use indexes and arrays to retrieve its internal objects from your handles instead of browsing through a list. This configuration specifies the size of each of these tables.

This configuration is available only when USBH_CFG_OPTIMIZE_SPD_EN is set to DEF_ENABLED.

Type	Function to call	Default	Field from default configuration structure
USBH_ACM_CFG_OPTIMIZE_SPD	USBH_ACM_ConfigureOptimizeSpdCfg()	No default value.	.OptimizeSpd

Allocation at Initialization Quantities

This configuration is mandatory *if* you set USBH_CFG_INIT_ALLOC_EN to DEF_ENABLED. Consequently, USBH_ACM_ConfigureInitAllocCfg() *must* be called from your application.

When this module allocates all its objects at initialization, it must know how many objects of each type it must allocate. This configuration specifies the number of objects to allocate.

This configuration is available only when USBH_CFG_INIT_ALLOC_EN is set to DEF_ENABLED.

Type	Function to call	Default	Field from default configuration structure
USBH_ACM_CFG_INIT_ALLOC	USBH_ACM_ConfigureInitAllocCfg()	No default value.	.InitAlloc

Post-Init Configurations

This section describes the configurations that can be set at any time during execution *after* you called the function USBH_ACM_Init() .

These configurations are optional. If you do not set them in your application, the default configurations will apply.

Standard Requests Timeout

Timeout, in milliseconds, for the standard requests executed by this module.

Type	Function to call	Default
CPU_INT32U	USBH_ACM_StdReqTimeoutSet()	5000

USB Host CDC ACM Class Configuration for Speed Optimization

If you set the configuration USBH_CFG_OPTIMIZE_SPD_EN to DEF_ENABLED, you must provide additional information to the USB host CDC ACM class driver module. It will use indexes and arrays to retrieve its internal objects from your handles instead of browsing through a list. Hence, it is necessary to know the size of these tables.

Table - USBH_ACM_CFG_OPTIMIZE_SPD configuration structure in the *USB Host CDC ACM Class Configuration for Speed Optimization* page describes each configuration field available in this configuration structure.

Table - USBH_ACM_CFG_OPTIMIZE_SPD configuration structure

Field	Description
.FnctQty	Maximum number of connected CDC ACM functions.

Note that if you enabled both USBH_CFG_OPTIMIZE_SPD_EN and USBH_CFG_INIT_ALLOC_EN, the values of the field .FnctQty from both configuration structures *must* match.

USB Host CDC ACM Class Configuration for Allocation at Initialization

If you set the configuration USBH_CFG_INIT_ALLOC_EN to DEF_ENABLED, you must provide additional information to the USB host ACM class driver module. It will allocate all its objects at the initialization and will be unable to allocate more objects during execution. Hence, you must provide the number of each object type to allocate. This greatly depends on your needs and on the USB devices you plan to use.

Table - USBH_ACM_CFG_INIT_ALLOC configuration structure in the *USB Host CDC ACM Class Configuration for Allocation at Initialization* page describes each configuration field available in this configuration structure.

Table - USBH_ACM_CFG_INIT_ALLOC configuration structure

Field	Description
.FnctQty	Maximum number of connected CDC ACM functions.
.AsyncXferQty	Total number of asynchronous transfers. This represents the maximum number of asynchronous transfers that can be submitted at a given time for all the CDC ACM functions.

Note that if you enabled both USBH_CFG_OPTIMIZE_SPD_EN and USBH_CFG_INIT_ALLOC_EN, the values of the field .FnctQty from both configuration structures *must* match.

USB Host HID Class Driver

The HID Class driver provided with Micrium OS USB Host is a class driver that will handle HID devices such as mice and keyboards.

The class driver implementation complies with the "Device Class Definition for Human Interface Devices (HID)", Version 1.11, 6/27/01.

USB Host HID Class Example Applications

This example presents a template that you can use to start the development of your HID class project. This application will accept the connection of mice and keyboards, parse their reports, and display either the position and button state (mouse) or key press (keyboard).

The following are the tasks accomplished by the example application when a HID device is connected:

- Allocate a memory block to represent the device
- Set the idle time to infinite
- Parse the report and determine:
  - The type of HID device (mouse, keyboard, or other)
  - The number of report IDs

Each time a key is pressed on a keyboard, the key will be displayed on the terminal. If the connected device is a mouse, each time the position changes or a button is pressed or release, the new position or button status will be displayed on the terminal.

Location

The example implementation is located in /examples/usb/host/ex_usbh_hid.c.

API

This example offers only one API named Ex_USBH_HID_Init(). This function is normally called from a USB host core example.

USB Host HID Class Programming Guide

- Prerequisites
- Initializing the USB Host HID Class driver
- Using USB Host HID Class Application Functions
  - Conn
  - DataRxd
  - Disconn

Prerequisites

The following compile-time configurations from the USB Host core are required for this class driver to work properly.

Reference: USB Host Compile-Time Configurations .

- USBH_CFG_PERIODIC_XFER_EN: Must be set to DEF_ENABLED.

Initializing the USB Host HID Class driver

In order to support the Human Interface Device (HID) function connections, you must first initialize the class driver. This is done by calling the function USBH_HID_Init().

Listing - Example of call to USBH_HID_Init() in the *USB Host HID Class Programming Guide* page gives an example of call to USBH_HID_Init() using default arguments. For more information on the configuration arguments to pass to USBH_HID_Init(), see USB Host HID Class Configuration .

Listing - Example of call to USBH_HID_Init()

```
USBH_HID_APP_FNCTS  App_USBH_HID_Fncts = {
    .Conn     = App_USBH_HID_Conn,
    .DataRxd = App_USBH_HID_DataRxd,
    .Disconn = App_USBH_HID_Disconn
}

void  App_USBH_HID_Init()
{
    RTOS_ERR  err;

    USBH_HID_Init(&App_USBH_HID_Fncts,
                  &err);
    if (err.Code != RTOS_ERR_NONE) {
        /* An error occured. Error handling should be added here. */
    }
}
```

Using USB Host HID Class Application Functions

The HID class driver provides three callback functions that will be called when an event occurs on a HID function (such as connection and disconnection).

These callbacks can have multiple purposes. Some of these include:

- Provide handles on the HID function to your application
- Execute some event driven simple task
- Trigger another task in your application
- Etc.

Below is a description of each of the available callback functions.

Conn

This function is called when a HID function is connected. It will provide all the necessary handles on the HID function to your application. Your application can also use it to return a pointer to an application object that is linked to this particular HID function. This pointer will be passed to any other callback function that is related to the HID function.

Listing - Example of implementation of Conn function in the *USB Host HID Class Programming Guide* page gives an example of implementation of the *Conn* callback function.

Listing - Example of implementation of Conn function

```
void  *App_USBH_HID_Conn(USBH_DEV_HANDLE        dev_handle,
                         USBH_FNCT_HANDLE       fnct_handle,
                         USBH_HID_FNCT_HANDLE   hid_fnct_handle,
                         USBH_HID_APP_COLL     *p_app_coll_head,
                         RTOS_ERR               err)
{
    /* A HID function was connected. hid_fnct_handle should be saved in your application for later use. */

    /* You can allocate an object that will be linked to this HID function. */

    /* You can trigger one of your application task from here. */

    return (<pointer to application object>);   /* DEF_NULL can be returned as well. */
}
```

DataRxd

This function is called when the class driver received a report from a HID function.

Listing - Example of implementation of DataRxd function in the *USB Host HID Class Programming Guide* page gives an example of implementation of the *DataRxd* callback function. The report is provided

Listing - Example of implementation of DataRxd function

```
void  App_USBH_HID_DataRxd(USBH_HID_FNCT_HANDLE   hid_fnct_handle,
                           void                  *p_arg,
                           CPU_INT08U             report_id,
                           CPU_INT08U            *p_buf,
                           CPU_INT32U             buf_len,
                           RTOS_ERR               err)
{
    /* A HID function has received a new report.                                  */

    /* p_arg is the pointer you returned in the Conn function associated to this function.    */
    /* You can free the object if necessary.                                      */

    /* You can trigger one of your application task form here.                    */
}
```

Disconn

This function is called when a HID function is disconnected.

Listing - Example of implementation of Disconn function in the *USB Host HID Class Programming Guide* page gives an example of implementation of the *Disconn* callback function.

Listing - Example of implementation of Disconn function

```
void  App_USBH_HID_Disconn(USBH_HID_FNCT_HANDLE   hid_fnct_handle,
                           void                  *p_arg)
{
    /* A HID function was disconnected.                                           */

    /* p_arg is the pointer you returned in the Conn function associated to this function.  */
    /* You can free the object if necessary.                                      */

    /* You can trigger one of your application task form here.                    */
}
```

USB Host HID Class Configuration

- Class Driver Initialization
  - p_hid_app_fncts
- Optional Configurations
  - Buffer Alignment
  - Receive Buffers
  - Report Descriptor Maximum Length
  - Maximum Number of Usages Per Items
  - Memory Segments
  - Optimize Speed Quantities
  - Allocation at Initialization Quantities
- Post-Init Configurations
  - Standard Requests Timeout

Class Driver Initialization

To initialize the USB Host Human Interface Device (HID) class driver module, you call the function USBH_HID_Init() . This function takes one configuration argument that is described below.

p_hid_app_fncts

p_hid_app_fncts is a pointer to a structure of type USBH_HID_APP_FNCTS. Its purpose is to provide a set of application callbacks to be called when an event occurs on a HID function (such as connection or disconnection). For more information on these callbacks, see USB Host HID Class Programming Guide .

The sections below describe each configuration field available in this configuration structure.

.Conn

A HID function was connected. This function provides the handles that were associated with it to your application.

Listing - .Conn function signature

```
void  *App_USBH_HID_Conn(USBH_DEV_HANDLE        dev_handle,
                   USBH_FNCT_HANDLE       fnct_handle,
                   USBH_HID_FNCT_HANDLE   hid_fnct_handle,
                   USBH_HID_APP_COLL     *p_app_coll_head,
                   RTOS_ERR              err)
```

.DataRxd

A HID function has received data (a new report).

Listing - .DataRxd function signature

```
void   App_USBH_HID_DataRxd(USBH_HID_FNCT_HANDLE   hid_fnct_handle,
                     void                  *p_arg,
                     CPU_INT08U             report_id,
                     CPU_INT08U            *p_buf,
                     CPU_INT32U             buf_len,
                     RTOS_ERR               err)
```

.Disconn

A HID function was disconnected. Once you return from this function, the handles passed at the Conn function for this HID function won't be valid anymore.

Listing - .Disconn function signature

```
void   App_USBH_HID_Disconn(USBH_HID_FNCT_HANDLE   hid_fnct_handle,
                     void                  *p_arg)
```

Optional Configurations

This section describes the configurations that are optional. If you do not set them in your application, the default configurations will apply.

The default values can be retrieved via the structure USBH_HID_InitCfgDflt.

> Note that these configurations must be set *before* you call the function USBH_HID_Init() .

Buffer Alignment

This module allocates buffers used for data transfers with the devices. You may have a specific need for address alignment for these buffers depending on your USB controller. If you use more than one USB controller, you must set the alignment to the largest value.

Type	Function to call	Default	Field from default configuration structure
CPU_SIZE_T	USBH_HID_ConfigureBufAlignOctets()	Size of cache line, or CPU alignment, if no cache.	.BufAlignOctets

Receive Buffers

Configures the quantity and the length of the receive buffers. The more receive buffers you allocate, the more responsive the HID device will be.

Type	Function to call	Default	Field from default configuration structure
CPU_INT08U	USBH_HID_ConfigureRxBuf()	2 receive buffers on 10 octets.	.RxBufQty .RxBufLen

Report Descriptor Maximum Length

Configures the maximum length of a report descriptor in octets.

Type	Function to call	Default	Field from default configuration structure
CPU_INT16U	USBH_HID_ConfigureReportDescMaxLen()	128	.ReportDescMaxLen

Maximum Number of Usages Per Items

Configures the maximum number of usages per items.

Type	Function to call	Default	Field from default configuration structure
CPU_INT08U	USBH_HID_ConfigureUsageMaxNbrPerItem()	5	.UsageMaxNbrPerItem

Memory Segments

This module allocates control data and buffers used for data transfers with the devices. It has the ability to use a different memory segment for the control data and for the data buffers.

Type	Function to call	Default	Field from default configuration structure
MEM_SEG*	USBH_HID_ConfigureMemSeg()	General-purpose heap .	.MemSegPtr .MemSegBufPtr

Optimize Speed Quantities

This configuration is mandatory *if* you set USBH_CFG_OPTIMIZE_SPD_EN to DEF_ENABLED and therefore USBH_HID_ConfigureOptimizeSpdCfg() *must* be called from your application.

When the Optimize Speed Mode is enabled, you must provide further information to this module. In this case, this module will use indexes and arrays to retrieve its internal objects from your handles instead of browsing through a list. This configuration specifies the size of each of these tables.

This configuration is available only when USBH_CFG_OPTIMIZE_SPD_EN is set to DEF_ENABLED.

Type	Function to call	Default	Field from default configuration structure
USBH_HID_CFG_OPTIMIZE_SPD	USBH_HID_ConfigureOptimizeSpdCfg()	No default value.	.OptimizeSpd

Allocation at Initialization Quantities

This configuration is mandatory *if* you set USBH_CFG_INIT_ALLOC_EN to DEF_ENABLED and therefore USBH_HID_ConfigureInitAllocCfg() *must* be called from your application.

When this module allocates all its objects at initialization, it must know how many objects of each type it must allocate. This configuration specifies the number of objects to allocate.

This configuration is available only when USBH_CFG_INIT_ALLOC_EN is set to DEF_ENABLED.

Type	Function to call	Default	Field from default configuration structure
USBH_HID_CFG_INIT_ALLOC	USBH_HID_ConfigureInitAllocCfg()	No default value.	.InitAlloc

Post-Init Configurations

This section describes the configurations that can be set at any time during execution *after* you called the function USBH_HID_Init() .

These configurations are optional. If you do not set them in your application, the default configurations will apply.

Standard Requests Timeout

Timeout, in milliseconds, for the standard requests executed by this module.

Type	Function to call	Default
CPU_INT32U	USBH_HID_StdReqTimeoutSet()	5000

USB Host HID Class Configuration for Speed Optimization

If you set the configuration USBH_CFG_OPTIMIZE_SPD_EN to DEF_ENABLED, you must provide additional information to the USB host HID class driver module. It will use indexes and arrays to retrieve its internal objects from your handles instead of browsing through a list. So, it is necessary to know the size of these tables.

Table - USBH_HID_CFG_OPTIMIZE_SPD configuration structure in the *USB Host HID Class Configuration for Speed Optimization* page describes each configuration field available in this configuration structure.

Table - USBH_HID_CFG_OPTIMIZE_SPD configuration structure

Field	Description
.FnctQty	Maximum number of connected HID functions. Keep in mind that some simple HID device (mouse/keyboard) may present themselves to the host as a multi HID function device.

Note that if you enabled both USBH_CFG_OPTIMIZE_SPD_EN and USBH_CFG_INIT_ALLOC_EN, the values of the field .FnctQty from both configuration structures *must* match.

USB Host HID Class Configuration for Allocation at Initialization

If you set the configuration USBH_CFG_INIT_ALLOC_EN to DEF_ENABLED, you must provide additional information to the USB host HID class driver module. It will allocate all its objects at initialization and will be unable to allocate more objects during execution. So, you must provide the number of each object type to allocate. This greatly depends on your needs and on the USB devices you plan to use.

Table - USBH_HID_CFG_INIT_ALLOC configuration structure in the *USB Host HID Class Configuration for Allocation at Initialization* page describes each configuration field available in this configuration structure.

Table - USBH_HID_CFG_INIT_ALLOC configuration structure

Field	Description
.FnctQty	Maximum number of connected HID functions. Keep in mind that some simple HID devices (such as keyboards) may present themselves to the host as a multi-HID-function device.
.ReportDescParseAppCollItemQty	Maximum number of application collections in report descriptor. A value of 2 here represents a good starting point.
.ReportDescParseGlobalItemQty	Maximum number of global items in report descriptor. A value of 2 here represents a good starting point.
.ReportDescParseCollItemQty	Maximum number of collection items in report descriptor. A value of 2 here represents a good starting point.
.ReportDescParseReportFmtItemQty	Maximum number of report formats in report descriptor. A value of 30 here represents a good starting point.

Note that if you enabled both USBH_CFG_OPTIMIZE_SPD_EN and USBH_CFG_INIT_ALLOC_EN, the values of the field .FnctQty from both configuration structures *must* match.

USB Host MSC Class Driver

The MSC Class driver provided by Micrium OS USB Host is a class driver that will handle mass storage devices such as flash memory drives, hard drives, and card readers).

The class driver implementation is in compliance with the following specifications:

- *Universal Serial Bus Mass Storage Class Specification Overview*, Revision 1.3 Sept. 5, 2008.
- *Universal Serial Bus Mass Storage Class Bulk-Only Transport*, Revision 1.0 Sept. 31, 1999.

MSC is a protocol that enables the transfer of information between a USB device and a host. The information is anything that can be stored electronically: executable programs, source code, documents, images, configuration data, or other text or numeric data.

A file system defines how the files are organized in the storage media. The USB mass storage class specification does not require any particular file system to be used on conforming devices. Instead, it provides a simple interface to read and write sectors of data using the Small Computer System Interface (SCSI) transparent command set.

The USB mass storage device class specification defines two transport protocols:

- Bulk-Only Transport (BOT)
- Control/Bulk/Interrupt (CBI) Transport.

The MSC class driver supports the BOT protocol only, which is by far the most widely used.

USB Host MSC Class Example Applications

This example performs only an initialization of the MSC host class. File accesses on the MSC device are handled by the Micrium OS File System module.

This application requires the SCSI driver from the Micrium OS File System Storage layer to work properly. The connection of MSC device will report as the connection of a removable media.

For concrete file access and removable media handling examples, refer to the Media Polling example in File System Example Applications .

Location

The example implementation is located in /examples/usb/host/ex_usbh_msc.c.

API

This example offers only one API named Ex_USBH_MSC_Init(). This function is normally called from a USB host core example.

USB Host MSC Class Programming Guide

- Prerequisites
- Initializing the USB Host MSC Class driver
- Communicating with an MSC device

Prerequisites

No prerequisites from core.

Initializing the USB Host MSC Class driver

In order to support the Mass Storage Class (MSC) function connections, you must first initialize the class driver. This is done by calling the function USBH_MSC_Init().

Listing - Example of call to USBH_MSC_Init() in the *USB Host MSC Class Programming Guide* page gives an example of call to USBH_MSC_Init() using default arguments. For more information on the configuration arguments to pass to USBH_MSC_Init(), see USB Host MSC Class Configuration .

Listing - Example of call to USBH_MSC_Init()

```
#include <rtos/fs/include/fs_scsi.h>              (1)

USBH_MSC_CMD_BLK_FNCTS  App_USBH_MSC_CmdBlkFncts = {   (2)
   .Conn           = FS_SCSI_LU_Conn,
   .Disconn         = FS_SCSI_LU_Disconn,
   .MaxRespBufLenGet = FS_SCSI_MaxRespBufLenGet
}

void  App_USBH_MSC_Init()
{
   RTOS_ERR  err;

   USBH_MSC_Init(&App_USBH_MSC_CmdBlkFncts,
             &err);
   if (err.Code != RTOS_ERR_NONE) {
      /* An error occured. Error handling should be added here. */
   }
}
```

(1) In this example, we use the Micrium OS File System SCSI device driver, which is the recommended method to communicate with the Mass Storage Device. Hence, inclusion is needed.

(2) The MSC event callbacks are handled by the File System SCSI device driver. The driver will handle the communication and add the MSC device as a regular File System Device. The application will be notified of a SCSI device connection or disconnection via the File System Media Poll task callbacks.

Communicating with an MSC device

The MSC class driver does not provide any public function to communicate with the device. You must use the Micrium OS File System API to communicate with the device. For more information, see File System Module User Manual .

USB Host MSC Class Configuration

- Run-Time Configurations
  - Class Driver Initialization
  - Optional Configurations
  - Buffer Alignment
  - Memory Segments
  - Optimize Speed Quantities

- Allocation at Initialization Quantities
- Post-Init Configurations
  - Standard Requests Timeout

Run-Time Configurations

Class Driver Initialization

To initialize the USB Host Mass Storage Class (MSC) driver module, you call the function USBH_MSC_Init(). This function takes one configuration argument which is described below.

p_cmd_blk_fncts

p_cmd_blk_fncts is a pointer to a structure of type USBH_MSC_CMD_BLK_FNCTS. Its purpose is to provide a set of application callbacks to be called when an event occurs on an MSC function (such as connection or disconnection).

We highly recommended that you map these callbacks to the Micrium OS FS SCSI driver functions. You can, however, implement your own functions that call the SCSI driver functions if you have some specific instructions to execute.

.Conn

An MSC function was connected.

Recommended handler function from SCSI device driver: FS_SCSI_LU_Conn()

Listing - .Conn function signature

```
void  *App_USBH_MSC_Conn(CPU_INT08U   lun,
                         CPU_INT16U   dev_id,
                         void        *p_transport_api,
                         void        *p_transport_arg)
```

.Disconn

An MSC function was disconnected.

Recommended handler function from SCSI device driver: FS_SCSI_LU_Disconn()

Listing - .Disconn function signature

```
void   App_USBH_MSC_Disconn(CPU_INT08U   lun,
                            CPU_INT16U   dev_id,
                            void        *p_transport_arg)
```

.MaxRespBufLenGet

Retrieve the maximum response buffer length, in octets. The response buffer is used by certain SCSI commands with a Data IN phase.

Recommended handler function from SCSI device driver: FS_SCSI_MaxRespBufLenGet()

Listing - .MaxRespBufLenGet function signature

```
void   App_USBH_MSC_MaxRespBufLenGet(void)
```

Optional Configurations

This section describes the configurations that are optional. If you do not set them in your application, the default configurations will apply.

The default values can be retrieved via the structure USBH_MSC_InitCfgDflt.Note that these configurations must be set *before* you call the function USBH_MSC_Init().

Buffer Alignment

This module allocates buffers used for data transfers with the devices. You may have a specific need of address alignment for these buffers depending on your USB controller. If you use more than one USB controller, you must set the alignment to the largest value.

Type	Function to call	Default	Field from default configuration structure
CPU_SIZE_T	USBH_MSC_ConfigureBufAlignOctets()	Size of cache line, or CPU alignment, if no cache.	.BufAlignOctets

Memory Segments

This module allocates control data and buffers used for data transfers with the devices. It has the ability to use a different memory segment for the control data and for the data buffers.

Type	Function to call	Default	Field from default configuration structure
MEM_SEG*	USBH_MSC_ConfigureMemSeg()	General-purpose heap .	.MemforSegPtr .MemSegBufPtr

Optimize Speed Quantities

This configuration is mandatory *if* you set USBH_CFG_OPTIMIZE_SPD_EN to DEF_ENABLED and therefore USBH_MSC_ConfigureOptimizeSpdCfg() *must* be called from your application.

When the optimize speed mode is enabled, you have to provide further information to this module. In this case, this module will use indexes and arrays to retrieve its internal objects from your handles instead of browsing through a list. This configuration specifies the size of each of these tables.

This configuration is available only when USBH_CFG_OPTIMIZE_SPD_EN is set to DEF_ENABLED.

Type	Function to call	Default	Field from default configuration structure
USBH_MSC_CFG_OPTIMIZE_SPD	USBH_MSC_ConfigureOptimizeSpdCfg()	No default value.	.OptimizeSpd

Allocation at Initialization Quantities

This configuration is mandatory *if* you set USBH_CFG_INIT_ALLOC_EN to DEF_ENABLED and therefore USBH_MSC_ConfigureInitAllocCfg() *must* be called from your application.

When this module allocates all its objects at initialization, it must know how many objects of each type it must allocate. This configuration specifies the number of objects to allocate.

This configuration is available only when USBH_CFG_INIT_ALLOC_EN is set to DEF_ENABLED.

Type	Function to call	Default	Field from default configuration structure
USBH_MSC_CFG_INIT_ALLOC	USBH_MSC_ConfigureInitAllocCfg()	No default value.	.InitAlloc

Post-Init Configurations

This section describes the configurations that can be set at any time during execution *after* you called the function USBH_MSC_Init().

These configurations are optional. If you do not set them in your application, the default configurations will apply.

Standard Requests Timeout

Timeout, in milliseconds, for the standard requests executed by this module.

Type	Function to call	Default
CPU_INT32U	USBH_MSC_StdReqTimeoutSet()	5000

USB Host MSC Class Configuration for Speed Optimization

If you set the configuration USBH_CFG_OPTIMIZE_SPD_EN to DEF_ENABLED, you must provide additional information to the USB host MSC class driver module. It will use indexes and arrays to retrieve its internal objects from your handles instead of browsing through a list. So, it is necessary to know the size of these tables.

Table - USBH_MSC_CFG_OPTIMIZE_SPD configuration structure in the *USB Host MSC Class Configuration for Speed Optimization* page describes each configuration field available in this configuration structure.

Table - USBH_MSC_CFG_OPTIMIZE_SPD configuration structure

Field	Description
.FnctQty	Maximum number of connected MSC functions.

Note that if you enabled both USBH_CFG_OPTIMIZE_SPD_EN and USBH_CFG_INIT_ALLOC_EN, the values of the field .FnctQty from both configuration structures *must* match.

USB Host MSC Class Configuration for Allocation at Initialization

If you set the configuration USBH_CFG_INIT_ALLOC_EN to DEF_ENABLED, you must provide additional information to the USB host MSC class driver module. In this case, it will allocate all its objects at initialization and will be unable to allocate more objects during execution. So, you must provide the number of each object type to allocate. This greatly depends on your needs and on the USB devices you plan to use.

Table - USBH_MSC_CFG_INIT_ALLOC configuration structure in the *USB Host MSC Class Configuration for Allocation at Initialization* page describes each configuration field available in this configuration structure.

Table - USBH_MSC_CFG_INIT_ALLOC configuration structure

Field	Description
.FnctQty	Maximum number of connected MSC functions.
.RespBufQty	Number of response buffers to allocate. Typical value is 2.

Note that if you enabled both USBH_CFG_OPTIMIZE_SPD_EN and USBH_CFG_INIT_ALLOC_EN, the values of the field .FnctQty from both configuration structures *must* match.

USB Host USB-to-Serial Class Driver

The USB-to-Serial Class driver provided with Micrium OS USB Host is a class driver that handles USB-to-serial adapters from different manufacturers (FTDI, Silicon Labs and Prolific).

USB Host USB-to-Serial Class Example Applications

This example detects the connection of a USB-to-serial adapter. After configuring the adapter, the example application sends back everything that was received from it, hence creating a loopback.

The following are the tasks accomplished by the example when a USB-To-Serial adapter device is connected:

- Set baud rate to 115200

Set data characteristics to 8 data bits, no parity, and 1 stop bit
- Disable hardware flow control
- Disable software flow control

The example then waits for data to be received from the device. Once data is received, a task is triggered to send it back to the device.

You can use you favorite terminal tool and connect to the USB-to-serial adapter from your PC using a serial port. All the data you type in should be sent back.

The example will also display on the terminal any changes in the serial status.

Location

The example implementation is located in /examples/usb/host/ex_usbh_usb2ser_loopback.c.

API

This example offers only one API named Ex_USBH_USB2SER_Init(). This function is normally called from a USB host core example.

USB Host USB-to-Serial Class Programming Guide

- Prerequisites
  - Initializing the Class Driver
- Using the Class Application Callback Functions
  - Conn
  - Disconn
  - DataRxd
  - SerialStatusChng
- Setting Line Parameters
  - Sending/Receiving Data
    - Receiving
    - Sending

Prerequisites

The following compile-time configurations in the USB Host core are required for this class driver to work properly.

Reference: USB Host Compile-Time Configurations .

- USBH_CFG_FIELD_EN_MASK: Following fields must be stored: USBH_CFG_FIELD_EN_DEV_VENDOR_ID and USBH_CFG_FIELD_EN_DEV_PRODUCT_ID. Storing USBH_CFG_FIELD_EN_DEV_REL_NBR is also recommended.
- USBH_CFG_PERIODIC_XFER_EN: Must be set to DEF_ENABLED if USBH_USB2SER_CFG_NOTIFICATIONS_RX_EN is set to DEF_ENABLED.

Initializing the Class Driver

In order to support the USB-to-serial adapter function connections, you must first initialize the class driver. To do this, you call the function USBH_USB2SER_Init().

Listing - Example of call to USBH_USB2SER_Init() in the *USB Host USB-to-Serial Class Programming Guide* page shows an example of a call to USBH_USB2SER_Init() using default arguments. For more information on the configuration arguments to pass to USBH_USB2SER_Init(), see USB Host USB-to-Serial Class Configuration .

Listing - Example of call to USBH_USB2SER_Init()

```
USBH_USB2SER_APP_FNCTS  App_USBH_USB2SER_Fncts = {
    .Conn            = App_USBH_USB2SER_Conn,
    .Disconn         = App_USBH_USB2SER_Disconn,
    .DataRxd         = App_USBH_USB2SER_DataRxd,
    .SerialStatusChng = App_USBH_USB2SER_SerStatusChng
};

void  App_USBH_USB2SER_Init()
{
    RTOS_ERR  err;

    USBH_USB2SER_Init(&App_USBH_USB2SER_Fncts,
                      &err);
    if (err.Code != RTOS_ERR_NONE) {
        /* An error occured. Error handling should be added here. */
    }
}
```

Using the Class Application Callback Functions

The USB-to-serial class driver provides four callback functions that will be called on connection and disconnection and when an event occurs on a USB-to-serial function.

These callbacks can have multiple purposes. Some of these include:

- Provide handles on the USB-to-serial function to your application
- Execute some event driven simple task
- Trigger another task in your application
- Etc.

Below is a description of each of the available callback functions.

Conn

This function is called when a USB-to-serial function is connected. It will provide all the necessary handles on the USB-to-serial function to your application. Your application can also use it to return a pointer to an application object that is linked to this particular USB-to-serial function. This pointer will be passed to any other callback function that is related to the USB-to-serial function.

Listing - Example of implementation of Conn function in the *USB Host USB-to-Serial Class Programming Guide* page shows an example of implementation of the Conn callback function.

Listing - Example of implementation of Conn function

```
void  *App_USBH_USB2SER_Conn(USBH_DEV_HANDLE           dev_handle,
                             USBH_FNCT_HANDLE          fnct_handle,
                             USBH_USB2SER_FNCT_HANDLE  usb2ser_fnct_handle)
{
    /* A USB-to-serial function was connected. usb2ser_fnct_handle should be saved in your application for later use. */

    /* You can allocate an object that will be linked to this usb-to-serial function. */

    /* You can trigger one of your application task from here. */

    return (<pointer to application object>);   /* DEF_NULL can be returned as well. */
}
```

Disconn

This function is called when a USB-to-serial function is disconnected.

Listing - Example of implementation of Disconn function in the *USB Host USB-to-Serial Class Programming Guide* page shows an example of implementation of the Disconn callback function.

Listing - Example of implementation of Disconn function

```
void  App_USBH_USB2SER_Disconn(USBH_AOAP_FNCT_HANDLE   usb2ser_fnct_handle,
                               void                    *p_arg)
{
    /* A USB-to-serial function was disconnected. */

    /* p_arg is the pointer you returned in the Conn function associated to this function. You can free the object if necessary. */

    /* You can trigger one of your application task from here. */
}
```

DataRxd

This function is called when a USB-to-serial function received data.

Listing - Example of implementation of DataRxd function in the *USB Host USB-to-Serial Class Programming Guide* page shows an example of implementation of the DataRxd callback function.

Listing - Example of implementation of DataRxd function

```
void  App_USBH_USB2SER_DataRxd(USBH_AOAP_FNCT_HANDLE   usb2ser_fnct_handle,
                               void                    *p_arg,
                               CPU_INT08U              *p_buf,
                               CPU_INT32U               buf_len)
{
    /* A USB-to-serial function received data. */

    /* p_arg is the pointer you returned in the Conn function associated to this function. */

    /* You can trigger one of your application task from here. */
}
```

SerialStatusChng

This function is called when the serial status of a USB-to-serial function has changed.

Listing - Example of implementation of SerialStatusChng function in the *USB Host USB-to-Serial Class Programming Guide* page gives an example of implementation of the SerialStatusChng callback function.

Listing - Example of implementation of SerialStatusChng function

```
void  App_USBH_USB2SER_SerStatusChng(USBH_AOAP_FNCT_HANDLE          usb2ser_fnct_handle,
                                     void                          *p_arg,
                                     const  USBH_USB2SER_SERIAL_STATUS   serial_status)
{
    /* The serial status of a USB-to-serial function has changed. */

    /* p_arg is the pointer you returned in the Conn function associated to this function. */

                                        /* Modem status.              */
    CPU_BOOLEAN  cts_state;
    CPU_BOOLEAN  dsr_state;
    CPU_BOOLEAN  ri_state;
    CPU_BOOLEAN  rlsd_state;
                                        /* Line status.               */
    CPU_BOOLEAN  oe_err_state;
    CPU_BOOLEAN  parity_err_state;
    CPU_BOOLEAN  framing_err_state;
    CPU_BOOLEAN  break_err_state;

                                        /* Retrieve serial status.    */
    cts_state         = DEF_BIT_IS_SET(serial_status.Modem, USBH_USB2SER_MODEM_STATUS_CTS);
    dsr_state         = DEF_BIT_IS_SET(serial_status.Modem, USBH_USB2SER_MODEM_STATUS_DSR);
    ri_state          = DEF_BIT_IS_SET(serial_status.Modem, USBH_USB2SER_MODEM_STATUS_RING);
    rlsd_state        = DEF_BIT_IS_SET(serial_status.Modem, USBH_USB2SER_MODEM_STATUS_CARRIER);

    oe_err_state      = DEF_BIT_IS_SET(serial_status.Line,  USBH_USB2SER_LINE_STATUS_RX_OVERFLOW_ERR);
    parity_err_state  = DEF_BIT_IS_SET(serial_status.Line,  USBH_USB2SER_LINE_STATUS_PARITY_ERR);
    framing_err_state = DEF_BIT_IS_SET(serial_status.Line,  USBH_USB2SER_LINE_STATUS_FRAMING_ERR);
    break_err_state   = DEF_BIT_IS_SET(serial_status.Line,  USBH_USB2SER_LINE_STATUS_BRK_INT);

    /* You can trigger one of your application task from here. */
}
```

Setting Line Parameters

You can configure the line parameters (baud rate, parity, data bits, etc) using the functions USBH_USB2SER_BaudRateSet() and USBH_USB2SER_DataSet().

Listing - Example of line configuration in the *USB Host USB-to-Serial Class Programming Guide* page shows an example of how to use the line parameters configuration functions to get 9600 bps, 8 data bits, no parity and 1 stop bit.

Listing - Example of line configuration

```
void App_USBH_USB2SER_LineCfg(USBH_USB2SER_FNCT_HANDLE usb2ser_fnct_handle)
{
    RTOS_ERR err;

    USBH_USB2SER_BaudRateSet(usb2ser_fnct_handle,
                             9600u,
                            &err);
    if (err.Code != RTOS_ERR_NONE) {
        /* An error occured. Error handling should be added here. */
    }

    USBH_USB2SER_DataSet(usb2ser_fnct_handle,
                         8u,
                         USBH_USB2SER_PARITY_NONE,
                         USBH_USB2SER_STOP_BITS_1,
                        &err);
    if (err.Code != RTOS_ERR_NONE) {
        /* An error occured. Error handling should be added here. */
    }
}
```

Sending/Receiving Data

Receiving

Receiving data is handled automatically by the class driver. Once a USB-to-serial adapter receives data, the DataRxd callback function will be called to notify your application.

Sending

The USB-to-serial class driver offers you a function to send data: USBH_USB2SER_TxAsync(). It is an asynchronous function which means that it won't block.

Listing - Example of data transmission in the *USB Host USB-to-Serial Class Programming Guide* page shows an example of how to send data to a USB-to-serial adapter.

Listing - Example of data transmission

```
void  App_USBH_USB2SER_TxCmpl(USBH_USB2SER_FNCT_HANDLE   usb2ser_fnct_handle,
                              void                      *p_arg,
                              CPU_INT08U                *p_buf,
                              CPU_INT32U                 buf_len,
                              CPU_INT32U                 xfer_len,
                              RTOS_ERR                   err)
{
    if (err.Code != RTOS_ERR_NONE) {
        /* An error occured. Error handling should be added here. */
    } else {
        /* Transfer completed successfully.  */
    }
}

CPU_INT08U  App_USBH_USB2SER_DataBuf[] = "Test buffer";

void  App_USBH_USB2SER_DataTx (USBH_USB2SER_FNCT_HANDLE  usb2ser_fnct_handle)
{
    RTOS_ERR  err;

    USBH_USB2SER_TxAsync(usb2ser_fnct_handle,
                         App_USBH_USB2SER_DataBuf,
                         sizeof(App_USBH_USB2SER_DataBuf),
                         App_USBH_USB2SER_TxCmpl,
                         DEF_NULL,
                        &err);
    if (err.Code != RTOS_ERR_NONE) {
        /* An error occured. Error handling should be added here. */
    }
}
```

USB Host USB-to-Serial Class Configuration

- Compile-Time Configurations
- Run-Time Configurations
  - Class Driver Initialization
  - Optional Configurations
  - Buffer Alignment
  - High-Speed Adapter Support Enable
  - Receive Buffer Quantity
  - Driver Entry Table
  - Memory Segments
  - Optimize Speed Quantities
  - Allocation at Initialization Quantities
  - Post-Init Configurations
  - Standard Requests Timeout

This section describes the configurations related to Micrium OS USB Host USB-to-serial Class driver.

Compile-Time Configurations

The USB-to-serial class driver is configurable at compile time via two #defines located in usbh_cfg.h file.

We recommend that you begin the configuration process with the default configuration values, which in the next sections will be shown in **bold**.

Table - USB-to-serial Class Driver Compile-time Configurations in the *USB Host USB-to-Serial Class Configuration* page describes the configurations.

Table - USB-to-serial Class Driver Compile-time Configurations

Constant	Description	Possible values
USBH_USB2SER_CFG_NOTIFICATIONS_RX_EN	Enables notifications. Some adapter drivers (such as the Prolific adapter driver) use interrupt endpoints to receive status notifications. If your application does not need status notifications, you can disable them to save resources. Likewise, you can also disable support for periodic transfers to save resources via the configuration USBH_CFG_PERIODIC_XFER_EN.	DEF_ENABLED or DEF_DISABLED
USBH_USB2SER_CFG_ADAPTER_CAPABILITIES_CHK_EN	Enables validation of the adapter's capabilities. Some adapter drivers (such as the Silicon Labs' adapter driver) can report the supported features of the adapter (baud rates, data bits, flow control, etc.). This is used to make sure that the parameters requested by your application are supported by the adapter. This can be disabled if you intend to always use adapters that you have already confirmed to support your parameters.	DEF_ENABLED or DEF_DISABLED

Run-Time Configurations

Class Driver Initialization

To initialize the USB Host USB-to-serial class driver module, you call the function USBH_USB2SER_Init(). This function takes one configuration argument that is described here.

p_app_fncts

p_app_fncts is a pointer to a structure of type USBH_USB2SER_APP_FNCTS. Its purpose is to provide a set of application callbacks to be called when a USB-to-serial function is connected, when data is received, and when it is disconnected. For more information on these callbacks, see USB Host USB-to-Serial Class Programming Guide .

The sections below describe each configuration field available in this configuration structure.

.Conn

A USB-to-serial function was connected. This function provides the associated handles to your application.

Listing - .Conn function signature

```
void  *App_USBH_USB2SER_Conn(USBH_DEV_HANDLE          dev_handle,
                             USBH_FNCT_HANDLE         fnct_handle,
                             USBH_USB2SER_FNCT_HANDLE  usb2ser_fnct_handle)
```

.Disconn

A USB-to-serial function was disconnected. Once you return from this function, the handles passed at the Conn function for this USB-to-serial function won't be valid anymore.

Listing - .Disconn function signature

```
void   App_USBH_USB2SER_Disconn(USBH_USB2SER_FNCT_HANDLE   usb2ser_fnct_handle,
                                void                       *p_arg)
```

.DataRxd

A USB-to-serial function has received data.

Listing - .DataRxd function signature

```
void   App_USBH_USB2SER_DataRxd(USBH_USB2SER_FNCT_HANDLE   usb2ser_fnct_handle,
                                void                      *p_arg,
                                CPU_INT08U                *p_buf,
                                CPU_INT32U                 buf_len)
```

.SerialStatusChng

A USB-to-serial function serial status has changed.

Listing - .SerialStatusChng function signature

```
void   App_USBH_USB2SER_SerStatusChng(USBH_USB2SER_FNCT_HANDLE          usb2ser_fnct_handle,
                                      void                             *p_arg,
                                      const  USBH_USB2SER_SERIAL_STATUS   serial_status)
```

Optional Configurations

This section describes the configurations that are optional. If you do not set them in your application, the default configurations will apply.

The default values can be retrieved via the structure USBH_USB2SER_InitCfgDflt.Note that these configurations must be set *before* you call the function USBH_USB2SER_Init().

Buffer Alignment

This module allocates buffers used for data transfers with the devices. You may have a specific need for address alignment for these buffers depending on your USB controller. If you use more than one USB controller, you must set the alignment to the largest value.

Type	Function to call	Default	Field from default configuration structure
CPU_SIZE_T	USBH_USB2SER_ConfigureBufAlignOctets()	Size of cache line, or CPU alignment, if no cache.	.BufAlignOctets

High-Speed Adapter Support Enable

Configures support for High-Speed adapters. Enabling support for high-speed adapters will require you to allocate larger receive buffers for all the adapters (even the Full-Speed ones).

Note that most common adapters are Full-Speed.

Type	Function to call	Default	Field from default configuration structure
CPU_BOOLEAN	USBH_USB2SER_ConfigureHS_En()	High-Speed adapters supported.	.HS_En

Receive Buffer Quantity

Configures the number of receive buffers allocated/submitted.

Type	Function to call	Default	Field from default configuration structure
CPU_INT08U	USBH_USB2SER_ConfigureRxBufQty()	1 buffer	.RxBufQty

Driver Entry Table

The USB-to-serial class driver uses drivers to support the different adapter protocols that are used on the market. Your application can use only the drivers for the adapter types you plan to use. A null-terminated table of type

USBH_USB2SER_ADAPTER_DRV_ENTRY can be configured to provide a list of supported adapter types. The currently available adapter drivers are listed here: USB Host USB-to-Serial Class Adapter Drivers .

Type	Function to call	Default	Field from default configuration structure
USBH_USB2SER_ADAPTER_DRV_ENTRY*	USBH_USB2SER_ConfigureDrvEntryTbl()	All adapter drivers available in your project.	.DevDrvEntryTbl

Memory Segments

This module allocates some control data and buffers used for data transfers with the devices. It has the ability to use a different memory segment for the control data and for the data buffers.

Type	Function to call	Default	Field from default configuration structure
MEM_SEG*	USBH_USB2SER_ConfigureMemSeg()	General-purpose heap .	.MemSegPtr .MemSegBufPtr

Optimize Speed Quantities

This configuration is mandatory *if* you set USBH_CFG_OPTIMIZE_SPD_EN to DEF_ENABLED.

Also, USBH_USB2SER_ConfigureOptimizeSpdCfg() *must* be called from your application when USBH_CFG_OPTIMIZE_SPD_EN is set to DEF_ENABLED.

When optimize speed mode is enabled, you must provide additional information to this module. This module will use indexes and arrays to retrieve its internal objects from your handles instead of browsing through a list. This configuration tells the size of each of these tables.This configuration is available only when USBH_CFG_OPTIMIZE_SPD_EN is set to DEF_ENABLED.

Type	Function to call	Default	Field from default configuration structure
USBH_USB2SER_CFG_OPTIMIZE_SPD	USBH_USB2SER_ConfigureOptimizeSpdCfg()	No default value.	.OptimizeSpd

Allocation at Initialization Quantities

This configuration is mandatory *if* you set USBH_CFG_INIT_ALLOC_EN to DEF_ENABLED.

Also, USBH_USB2SER_ConfigureInitAllocCfg() *must* be called from your application when USBH_CFG_OPTIMIZE_SPD_EN is set to DEF_ENABLED.

When this module allocates its objects at initialization, it must know how many objects of each type to allocate. This configuration specifies the number of objects to allocate.This configuration is available only when USBH_CFG_INIT_ALLOC_EN is set to DEF_ENABLED.

Type	Function to call	Default	Field from default configuration structure
USBH_USB2SER_CFG_INIT_ALLOC	USBH_USB2SER_ConfigureInitAllocCfg()	No default value.	.InitAlloc

Post-Init Configurations

This section describes the configurations that can be set at any time during execution *after* you called the function USBH_USB2SER_Init().

These configurations are optional. If you do not set them in your application, the default configurations will apply.

Standard Requests Timeout

Timeout, in milliseconds, for the standard requests executed by this module.

Type	Function to call	Default
CPU_INT32U	USBH_USB2SER_StdReqTimeoutSet()	5000

USB Host USB-to-Serial Class Configuration - Adapter Driver Entry

Table - USBH_USB2SER_ADAPTER_DRV_ENTRY configuration structure in the *USB Host USB-to-Serial Class Configuration - Adapter Driver Entry* page describes each configuration field available in the driver entry structure.

Table - USBH_USB2SER_ADAPTER_DRV_ENTRY configuration structure

Field	Description
.DrvPtr	Pointer to the adapter driver. See the "Driver structure" table entry on this page: USB Host USB-to-Serial Class Adapter Drivers .
.CustomIdTbl	Some adapter types allow you to customize the adapter by burning in a specific vendor and product id. If you want to support specific adapter(s) that uses those custom IDs, you must pass a table of type USBH_USB2SER_APP_ID here. A null pointer (DEF_NULL) can be passed otherwise.
.CustomIdTblLen	Length of the custom ID table. Should be 0 if none.

USB Host USB-to-Serial Class Configuration for Speed Optimization

If you set the configuration USBH_CFG_OPTIMIZE_SPD_EN to DEF_ENABLED, you must provide additional information to the USB-to-serial class driver module. It will use indexes and arrays to retrieve its internal objects from your handles instead of browsing through a list. So, it is necessary to know the size of these tables.

Table - USBH_USB2SER_CFG_OPTIMIZE_SPD configuration structure in the *USB Host USB-to-Serial Class Configuration for Speed Optimization* page describes each configuration field available in this configuration structure.

Table - USBH_USB2SER_CFG_OPTIMIZE_SPD configuration structure

Field	Description
.FnctQty	Maximum number of connected USB-to-serial functions.

Note that if you enabled both USBH_CFG_OPTIMIZE_SPD_EN and USBH_CFG_INIT_ALLOC_EN, the values of the field .FnctQty from both configuration structures *must* match.

USB Host USB-to-Serial Class Configuration for Allocation at Initialization

If you set the configuration USBH_CFG_INIT_ALLOC_EN to DEF_ENABLED, you must provide additional information to the USB-to-serial class driver module. It will allocate all its objects at the initialization and will be unable to allocate more objects during execution. So, you must provide the number of each object type to allocate. This greatly depends on your needs and on the USB devices you plan to use.

Table - USBH_USB2SER_CFG_INIT_ALLOC configuration structure in the *USB Host USB-to-Serial Class Configuration for Allocation at Initialization* page describes each configuration field available in this configuration structure.

Table - USBH_USB2SER_CFG_INIT_ALLOC configuration structure

Field	Description
.FnctQty	Maximum number of connected USB-to-serial functions.
.TxAsyncXferQty	Total number of Tx asynchronous transfers. This represents the maximum number of asynchronous transfers that can be submitted at a given time for all the USB-to-serial functions.

> Note that if you enabled both USBH_CFG_OPTIMIZE_SPD_EN and USBH_CFG_INIT_ALLOC_EN, the values of the field .FnctQty from both configuration structures *must* match.

> Note that if you enabled both USBH_CFG_OPTIMIZE_SPD_EN and USBH_CFG_INIT_ALLOC_EN, the values of the field .FnctQty from both configuration structures *must* match.

USB Host USB-to-Serial Class Adapter Drivers

The USB-to-serial class supports adapters from different vendors. Each vendor uses different protocols with their adapters, and so the USB-to-serial class uses different "adapter drivers" to support the different adapter available on the market. Following is a list of the different adapter drivers currently available and supported.

- FTDI
- SiLabs
- Prolific

FTDI

Name	FTDI
Adapter manufacturer	Future Technology Devices International Ltd.
Supported adapters model	• FT232B • FT232R • FT232H • FT2232F • FT2232H • FT4232H • FT8U232AM
Driver structure	`USBH_USB2SER_FTDI_Drv` defined in `usbh_usb2ser_ftdi.h`.
Note(s)	None.

SiLabs

Name	SILABS
Adapter manufacturer	Silicon Laboratories Inc.
Supported adapters model	• CP2102 • CP2103 • CP2104 • CP2105 • CP2108
Driver structure	`USBH_USB2SER_SILABS_Drv` defined in `usbh_usb2ser_silabs.h` .
Note(s)	None.

Prolific

Name	Prolific
Adapter manufacturer	Prolific Technology Inc.
Supported adapters model	• PL2303 series
Driver structure	`USBH_USB2SER_PROLIFIC_Drv` defined in `usbh_usb2ser_prolific.h`.
Note(s)	None.

USB Host Hardware Porting Guide

USB Host Hardware Porting Guide

In order to work properly, Micrium OS USB Host requires a hardware driver that is implemented for any specific USB host controller IP. Moreover, each USB host controller must have a Board Support Package (BSP). The BSP has two purposes:

- initializing and configuring any resources needed by the USB controller, but which are provided by an external module,
- and providing hardware information to the USB host driver and core.

Micrium provides examples of BSPs for some popular platforms. If one is available for your platform, we recommend that you use it as a starting point. However, if no example BSP is available for your platform, the information in this section will help you understand how to correctly port the USB device module.

Note that each USB host controller (that you are planning to use) will require a BSP, and all the steps described in this section must be performed for each of them.

- USB Host Controller Driver Types
- USB Host BSP Functions Guide
- USB Host HCD Hardware Information
- USB Host PBHCD Hardware Information
- USB Host Controller Registration to the Platform Manager

USB Host Controller Driver Types

Micrium OS USB Host provides two different types of drivers:

- Host Controller Drivers (HCD)
- Pipe-Based Host Controller Drivers (PBHCD)
  Table - Description of the Different Types of USB Host Controller Drivers in the *USB Host Controller Driver Types* page describes each type of USB host controller driver.

Table - Description of the Different Types of USB Host Controller Drivers

Type	Description
HCD	The HCD type targets complete controllers that are list-based and handle periodic transfers. Only the OHCI and EHCI drivers are of type HCD.
PBHCD	The PBHCD type is for simple controllers that have a limited set of Pipes (used as endpoints). They are not list-based. These controllers assume that transfer scheduling and other tasks are handled by the software. These drivers require an optional module called PBHCI (Pipe-Based Host Controller Interface).

USB Host BSP Functions Guide

The Board Support Package contains a set of functions that support your hardware platform on Micrium OS. These functions are called by the USB host driver to perform hardware configuration or initialization of I/O pins, interrupts, etc. It is your responsibility to either create these functions from scratch or to modify example code to fit your needs and the specifics of your hardware.

Each of these functions is described below.

- Initialize
- Clock Configuration
- I/O Configuration
- Interrupt Configuration

- Power Configuration
- Start
- Stop
- Extended BSP API
- ISR Handling

Initialize

The Initialize function is called by the driver only once at initialization. Depending on whether you have an HCD or a PBHCD, the function will have a slightly different signature. Listing - Init function for a HCD in the *USB Host BSP Functions Guide* page and Listing - Init function for a PBHCD in the *USB Host BSP Functions Guide* page show the signatures of the functions.

Listing - Init function for a HCD

```
CPU_BOOLEAN  BSP_USBH_EFM32GG_Init (USBH_HCD_ISR_HANDLE_FNCT   isr_fnct,
                                    USBH_HC_DRV               *p_hc_drv);
```

Listing - Init function for a PBHCD

```
CPU_BOOLEAN  BSP_USBH_EFM32GG_Init (USBH_PBHCD_ISR_HANDLE_FNCT   isr_fnct,
                                   USBH_PBHCI_HC_DRV          *p_pbhci_hc_drv);
```

Its purpose is to initialize and allocate any resource needed by the USB controller BSP.

The initialize function receives two arguments: isr_fnct and p_hc_drv (or p_pbhci_hc_drv, if your driver is a PBHCD), which are used when handling interrupts from the USB host controller. The argument isr_fnct is a pointer to a driver function that is called when a USB interrupt occurs, and this driver function takes p_hc_drv (or p_pbhci_hc_drv, if your driver is a PBHCD) as an argument. So it is important to save these as global variables in your BSP.

You must return DEF_OK from this function if it executed successfully, or DEF_FAIL, otherwise.

Clock Configuration

The Clock Configuration function is called by the driver when the USB host controller is started. Listing - Clock configure function signature in the *USB Host BSP Functions Guide* page shows the signature of the function.

Listing - Clock configure function signature

```
CPU_BOOLEAN  BSP_USBH_EFM32GG_ClkCfg (void);
```

Its purpose is to configure the clock of the USB host controller.

You must return DEF_OK from this function if it executed successfully, or DEF_FAIL, otherwise.

I/O Configuration

The I/O Configuration function is called by the driver when the USB host controller is started. Listing - IO configure function signature in the *USB Host BSP Functions Guide* page shows the signature of the function.

Listing - IO configure function signature

```
CPU_BOOLEAN  BSP_USBH_EFM32GG_IO_Cfg (void);
```

Its purpose is to configure the I/O pins for the USB host controller.

You must return DEF_OK from this function if it executed successfully, or DEF_FAIL, otherwise.

Interrupt Configuration

The Interrupt Configuration function is called by the driver when the USB host controller is started. Listing - Interrupt configure function signature in the *USB Host BSP Functions Guide* page shows the signature of the function.

Listing - Interrupt configure function signature

```
CPU_BOOLEAN  BSP_USBH_EFM32GG_IntCfg (void);
```

Its purpose is to configure the interrupt(s) of the USB host controller.

You must return DEF_OK from this function if it executed successfully, or DEF_FAIL, otherwise.

Power Configuration

The Power Configuration function is called by the driver when the USB host controller is started. Listing - Power configure function signature in the *USB Host BSP Functions Guide* page shows the signature of the function.

Listing - Power configure function signature

```
CPU_BOOLEAN  BSP_USBH_EFM32GG_PwrCfg (void);
```

Its purpose is to configure the power of the USB host controller.

You must return DEF_OK from this function if it executed successfully, or DEF_FAIL, otherwise.

Start

The Start function is called by the driver each time the USB host controller is started. It is always called after all the "configure" functions. Listing - Start function signature in the *USB Host BSP Functions Guide* page shows the signature of the function.

Listing - Start function signature

```
CPU_BOOLEAN  BSP_USBH_EFM32GG_Start (void);
```

Its main purpose is to perform any operation required when the USB controller is started.

You must return DEF_OK from this function if it executed successfully, or DEF_FAIL, otherwise.

Stop

The Stop function is called by the driver when the USB host controller is stopped. Listing - Stop function signature in the *USB Host BSP Functions Guide* page shows the signature of the function.

Listing - Stop function signature

```
CPU_BOOLEAN  BSP_USBH_EFM32GG_Stop (void);
```

Its purpose is to perform any operation required when the USB controller is stopped.

You must return DEF_OK from this function if it executed successfully, or DEF_FAIL, otherwise.

Extended BSP API

The BSP for the PBHCD may require extended features from the BSP. These features, if required, are specific to each driver.

ISR Handling

Each USB host driver implements an ISR handler function, which is called each time a USB host interrupt is triggered. For most platforms, it will be necessary to implement an intermediate ISR handler in the BSP, which in turn will call the driver's

ISR handler. This is necessary because some interrupt controllers may require the interrupt status to be cleared each time it is triggered.

An intermediate ISR handler is also necessary if your interrupt controller does not pass an argument to the interrupt vector, as the driver's ISR handler takes p_hc_drv (or p_pbhci_hc_drv), which was received in the Init() function of the BSP, as an argument.

Listing - Example of BSP ISR implementation in the *USB Host BSP Functions Guide* page shows an example of an ISR handler implemented in the BSP (that follows the CMSIS naming convention).

Listing - Example of BSP ISR implementation

```
void USB_IRQHandler (void)
{
   /* TODO: Clear interrupt status, if needed. */

   OSIntEnter();
   BSP_USBH_EFM32GG_ISR_Fnct(BSP_USBH_EFM32GG_DrvPtr);              (1)
   OSIntExit();
}
```

(1) Here, the driver's ISR is called. Note that the global variable BSP_USBH_EFM32GG_ISR_Fnct is a copy of the isr_fnct argument and the global variable BSP_USBH_EFM32GG_DrvPtr is a copy of the p_hc_drv (or p_pbhci_hc_drv, if your driver is a PBHCD) argument received from the BSP's Init() function.

USB Host HCD Hardware Information

- Hardware Driver Information
  - Hardware Information Structure
  - Dedicated Memory Information Structure
- BSP API Structure
- Host Controller Hardware Information

Hardware Driver Information

Hardware Information Structure

The USB host driver requires specific information about the USB controller on your MCU. You provide this information using a structure of type USBH_HC_HW_INFO.

The information describing the USB host controller can be found in the manual of your MCU.

Table - USBH_HC_HW_INFO structure in the *USB Host HCD Hardware Information* page describes each configuration field available in this structure.

Table - USBH_HC_HW_INFO structure

Field	Description
.BaseAddr	Base address of the USB host controller registers set. This corresponds to the address of the first register.
.RH_Spd	Maximum speed of operation your USB host controller can support. This depends on your host controller capabilities and on how the controller is set up in your BSP. For example, a High-Speed capable USB host controller can be set to operate at Full-Speed in order to supply a slower clock. Possible values are: - USBH_DEV_SPD_FULL - USBH_DEV_SPD_HIGH
.HC_Type	Type of host controller. Possible values are:
Value	Description
USBH_HC_TYPE_LIST	List-based controller. The controller uses a driver of type HCD. Only OHCI- and EHCI-based controllers fall into this category.

Field	Description
USBH_HC_TYPE_PIPE	Pipe-based controller. The controller uses a driver of type PBHCD.
.HW_DescDedicatedMemInfoPtr	Pointer to a structure of type USBH_HC_DEDICATED_MEM_INFO. Describes the dedicated memory to use for the hardware descriptor. Most of the time, this configuration is optional. Set this to DEF_NULL if not used.
.DataBufDedicatedMemInfoPtr	Pointer to a structure of type USBH_HC_DEDICATED_MEM_INFO. Describes the dedicated memory to use for the data buffers. Most of the time, this configuration is optional. Set this to DEF_NULL if not used.

Dedicated Memory Information Structure

When using a dedicated memory region for either the hardware descriptors or the data buffers, or both, a structure of type USBH_HC_DEDICATED_MEM_INFO must be used.

Table - USBH_HC_DEDICATED_MEM_INFO structure in the *USB Host HCD Hardware Information* page describes each configuration field available in this structure.

Table - USBH_HC_DEDICATED_MEM_INFO structure

Field	Description
.MemSegPtr	Pointer to the memory segment in dedicated memory. See Memory Segments and LIB Heap for more information on the memory segments.
.BufAlignOctets	Minimum alignment, in octets.

BSP API Structure

In order to provide a pointer to the BSP functions that you created (that are described here: USB Host BSP Functions Guide) for the USB host controller driver, you have to create a structure of type USBH_HC_BSP_API.

Table - USBH_HC_BSP_API structure in the *USB Host HCD Hardware Information* page describes each field available in this structure.

Table - USBH_HC_BSP_API structure

Field	Description
.Init	Pointer to the BSP initialization function.
.ClkCfg	Pointer to the BSP clock configure function.
.IO_Cfg	Pointer to the BSP IO configure function.
.IntCfg	Pointer to the BSP interrupt configure function.
.PwrCfg	Pointer to the BSP power configure function.
.Start	Pointer to the BSP start function.
.Stop	Pointer to the BSP stop function.

Listing - Example of BSP API structure in the *USB Host HCD Hardware Information* page gives an example of a BSP API structure.

Listing - Example of BSP API structure

```
static USBH_PBHCI_BSP_API BSP_USBH_EFM32GG_BSP_API_Ptr = {
  BSP_USBH_EFM32GG_Init,
  BSP_USBH_EFM32GG_ClkCfg,
  BSP_USBH_EFM32GG_IO_Cfg,
  BSP_USBH_EFM32GG_IntCfg,
  BSP_USBH_EFM32GG_PwrCfg,
  BSP_USBH_EFM32GG_Start,
  BSP_USBH_EFM32GG_Stop,
```

```
  DEF_NULL
};
```

Host Controller Hardware Information

The last step is to create the main host hardware information structure.

Table - USBH_HC_HCD_HW_INFO structure in the *USB Host HCD Hardware Information* page describes each configuration field available in this structure.

Table - USBH_HC_HCD_HW_INFO structure

Field	Description
.HW_Info	Hardware information structure described at step Hardware Driver Information above.
.DrvAPI_Ptr	Pointer to the driver API structure that you are using with your driver. Some drivers may provide more than one API structure.
.RH_API_Ptr	Pointer to the driver root hub API structure that you are using with your driver. Some drivers may provide more than one root hub API structure.
.BSP_API_Ptr	Pointer to the BSP API structure you created at step BSP API Structure .

USB Host PBHCD Hardware Information

- Hardware Driver Information
  - Hardware Information Structure
  - Dedicated Memory Information Structure
- Endpoint Information Table
- BSP API Structure
- Host Controller Hardware Information

Hardware Driver Information

Hardware Information Structure

The USB host driver requires specific information about the USB controller on your MCU. You provide this information using a structure of type USBH_HC_HW_INFO.

The information describing the USB host controller can be found in the manual of your MCU.

Table - USBH_HC_HW_INFO structure in the *USB Host PBHCD Hardware Information* page describes each configuration field available in this structure.

Table - USBH_HC_HW_INFO structure

Field	Description	
.BaseAddr	Base address of the USB host controller registers set. This corresponds to the address of the first register.	
.RH_Spd	Maximum speed of operation your USB host controller can support. This depends on your host controller capabilities and on how the controller is set up in your BSP. For example, a High-Speed capable USB host controller can be set to operate at Full-Speed in order to supply a slower clock. Possible values are: - USBH_DEV_SPD_FULL - USBH_DEV_SPD_HIGH	
.HC_Type	Type of host controller. Possible values are:	
Value	Description	
USBH_HC_TYPE_LIST	List-based controller. The controller uses a driver of type HCD. Only OHCI- and EHCI-based controllers fall into this category.	

Field	Description
USBH_HC_TYPE_PIPE	Pipe-based controller. The controller uses a driver of type PBHCD.
.HW_DescDedicatedMemInfoPtr	Pointer to a structure of type USBH_HC_DEDICATED_MEM_INFO. Describes the dedicated memory to use for the hardware descriptor. Most of the time, this configuration is optional. Set this to DEF_NULL if not used.
.DataBufDedicatedMemInfoPtr	Pointer to a structure of type USBH_HC_DEDICATED_MEM_INFO. Describes the dedicated memory to use for the data buffers. Most of the time, this configuration is optional. Set this to DEF_NULL if not used.

Dedicated Memory Information Structure

When using a dedicated memory region for either the hardware descriptors or the data buffers, or both, a structure of type USBH_HC_DEDICATED_MEM_INFO must be used.

Table - USBH_HC_DEDICATED_MEM_INFO structure in the *USB Host PBHCD Hardware Information* page describes each configuration field available in this structure.

Table - USBH_HC_DEDICATED_MEM_INFO structure

Field	Description
.MemSegPtr	Pointer to the memory segment in dedicated memory. See Memory Segments and LIB Heap for more information on the memory segments.
.BufAlignOctets	Minimum alignment, in octets.

Endpoint Information Table

Your USB host controller offers a set of pipes (used as endpoints) that are available to the USB host software. The Micrium OS USB Host module requires you to have information about these available pipes. The way to provide this information is by using a table of elements of types USBH_PBHCI_PIPE_INFO. The first element of the table *must* always be the default control pipe.

Note that the last element of the table *must* be a null entry.

The information regarding the pipes of your USB host controller can be found in the manual of your MCU.

Table - USBH_PBHCI_PIPE_INFO structure in the *USB Host PBHCD Hardware Information* page describes each configuration field available in this structure.

Table - USBH_PBHCI_PIPE_INFO structure

Field	Description
.Attrib	This is a bitmap that represents the different endpoint types and directions that this pipe can support. Possible values of the bitmap are:
Value	Description
USBH_PIPE_INFO_TYPE_CTRL	Pipe supports transfers of type Control.
USBH_PIPE_INFO_TYPE_BULK	Pipe supports transfers of type Bulk.
USBH_PIPE_INFO_TYPE_INTR	Pipe supports transfers of type Interrupt.
USBH_PIPE_INFO_TYPE_ISOC	Pipe supports transfers of type Isochronous.
USBH_PIPE_INFO_DIR_OUT	Pipe supports OUT (host-to-device) transfers.
Special values:	
Value	Description
USBH_PIPE_INFO_TYPE_ALL	Pipe supports all transfer types.

Field	Description
USBH_PIPE_INFO_DIR_BOTH	Pipe supports both IN and OUT transfers.
.Nbr	Pipe number.
.MaxPktSize	Maximum packet size (in octets) this pipe supports.
.MaxBuFlen	Maximum transfer buffer length (in octets) this pipe supports.

Listing - Example of pipe description table in the *USB Host PBHCD Hardware Information* page gives an example of a pipe information table. In this case, the controller has 12 bi-directional pipes. All but one are capable of all transfer types, the other one is the default control pipe. Also, note the null entry at the end of the table.

Listing - Example of pipe description table

```
USBH_PBHCI_PIPE_INFO BSP_USBH_EFM32GG_PipeInfoTbl[] =
{
 { USBH_PIPE_INFO_TYPE_CTRL                                                    | USBH_PIPE_INFO_DIR_BOTH, 0u, 64u, 0u },
 { USBH_PIPE_INFO_TYPE_CTRL | USBH_PIPE_INFO_TYPE_BULK | USBH_PIPE_INFO_TYPE_INTR | USBH_PIPE_INFO_DIR_BOTH, 1u, 512u, 0u },
 { USBH_PIPE_INFO_TYPE_CTRL | USBH_PIPE_INFO_TYPE_BULK | USBH_PIPE_INFO_TYPE_INTR | USBH_PIPE_INFO_DIR_BOTH, 2u, 512u, 0u },
 { USBH_PIPE_INFO_TYPE_CTRL | USBH_PIPE_INFO_TYPE_BULK | USBH_PIPE_INFO_TYPE_INTR | USBH_PIPE_INFO_DIR_BOTH, 3u, 512u, 0u },
 { USBH_PIPE_INFO_TYPE_CTRL | USBH_PIPE_INFO_TYPE_BULK | USBH_PIPE_INFO_TYPE_INTR | USBH_PIPE_INFO_DIR_BOTH, 4u, 512u, 0u },
 { USBH_PIPE_INFO_TYPE_CTRL | USBH_PIPE_INFO_TYPE_BULK | USBH_PIPE_INFO_TYPE_INTR | USBH_PIPE_INFO_DIR_BOTH, 5u, 512u, 0u },
 { DEF_BIT_NONE, 0u, 0u, 0u }
};
```

BSP API Structure

In order to provide a pointer to the BSP functions you created (that are described here: USB Host BSP Functions Guide) to the USB host controller driver, you have to create a structure of type USBH_PBHCI_BSP_API.

Table - USBH_PBHCI_BSP_API structure in the *USB Host PBHCD Hardware Information* page describes each field available in this structure.

Table - USBH_PBHCI_BSP_API structure

Field	Description
.Init	Pointer to the BSP initialization function.
.ClkCfg	Pointer to the BSP clock configure function.
.IO_Cfg	Pointer to the BSP IO configure function.
.IntCfg	Pointer to the BSP interrupt configure function.
.PwrCfg	Pointer to the BSP power configure function.
.Start	Pointer to the BSP start function.
.Stop	Pointer to the BSP stop function.
.ExtBSP_API	Pointer to optional driver specific extended BSP API. Can be set to DEF_NULL.

Listing - Example of BSP API structure in the *USB Host PBHCD Hardware Information* page gives an example of a BSP API structure.

Listing - Example of BSP API structure

```
static USBH_PBHCI_BSP_API BSP_USBH_EFM32GG_BSP_API_Ptr = {
 BSP_USBH_EFM32GG_STK3700A_Init,
 BSP_USBH_EFM32GG_STK3700A_ClkCfg,
 BSP_USBH_EFM32GG_STK3700A_IO_Cfg,
 BSP_USBH_EFM32GG_STK3700A_IntCfg,
 BSP_USBH_EFM32GG_STK3700A_PwrCfg,
 BSP_USBH_EFM32GG_STK3700A_Start,
 BSP_USBH_EFM32GG_STK3700A_Stop,
```

```
    DEF_NULL
};
```

Host Controller Hardware Information

The last step is to create the main host hardware information structure.

Table - USBH_PBHCI_HC_HW_INFO structure in the *USB Host PBHCD Hardware Information* page describes each configuration field available in this structure.

Table - USBH_PBHCI_HC_HW_INFO structure

Field	Description
.HW_Info	Hardware information structure described at step Hardware driver information .
.DrvAPI_Ptr	Pointer to the driver API structure you are using with your driver. Some drivers may provide more than one API structure.
.BSP_API_Ptr	Pointer to the BSP API structure you created at step BSP API Structure .
.PipeInfoTblPtr	Pointer to the table you created at step Endpoint information table .

USB Host Controller Registration to the Platform Manager

Once the hardware information structure for your USB host controller is ready, it must be registered to the Platform Manager. This should normally be done from the BSP_OS_Init() function that is located in the file bsp_os.c.

There exist three different macros located in the file usb_ctrlr.h you can call to register a USB Host controller. Table - USB Host Controller Register Macros in the *USB Host Controller Registration to the Platform Manager* page describes the different macros.

Table - USB Host Controller Register Macros

Macro	Description
USB_CTRLR_HW_INFO_REG()	Registers hardware information for a USB controller that has both Device and Host functionalities. Refer to USB Device Hardware Porting Guide for more information on the device side.
USB_CTRLR_HW_INFO_HOST_ONLY_REG()	Registers hardware information for a USB host controller.
USB_CTRLR_HW_INFO_HOST_COMPANION_REG()	Registers hardware information for a USB host companion controller.

Listing - Example of USB Controller Registration in the *USB Host Controller Registration to the Platform Manager* page gives an example of how to register a USB Host controller.

Listing - Example of USB Controller Registration

```
#if defined(RTOS_MODULE_USB_HOST_AVAIL) && defined (RTOS_MODULE_USB_HOST_PBHCI_AVAIL)    (1)
BSP_HW_INFO_EXT(const USBH_PBHCI_HC_HW_INFO, BSP_USBH_EFM32_PBHCI_HW_Info);
#endif

#if defined(RTOS_MODULE_USB_DEV_AVAIL)
BSP_HW_INFO_EXT(const USBD_DEV_CTRLR_HW_INFO, BSP_USBD_EFM32_HwInfo);
#endif

void  BSP_OS_Init (void)
{

    /* ... */
                                /* ------------- REGISTER USB CONTROLLERS ------------- */
    USB_CTRLR_HW_INFO_REG("usb0",                         (2)
                &BSP_USBD_EFM32_HwInfo,
```

```
&BSP_USBH_EFM32_PBHCI_HW_Info);}
```

(1) Since the hardware information global variables are declared in another file, you have to declare them as external in your bsp_os.c file. Always use the BSP_HW_INFO_EXT() macro.

(2) Registering a USB controller that implements both host and device functionalities. The controller name is "usb0". The tag will be used later on when calling the USBD_DevAdd() and/or USBH_HC_Add() function.

USB Host Troubleshooting

USB Host Troubleshooting

The following are some common issues that you may encounter while developing an application using Micrium OS USB Host.

- When I connect a device to my host controller, nothing seems to happen.
- When connecting a given device to my host, I get an error RTOS_ERR_rsrc_ALLOC. However, when I connect another similar device, I don't get any error. Why's that?

When I connect a device to my host controller, nothing seems to happen.

There can be many causes for this issue. We highly recommended that you implement the USB Host event functions. The event functions offer a function that will be called when a device connection fails and will provide an error code that will help to identify the issue. The event functions can be implemented by calling the function USBH_ConfigureEventFncts() before the function USBH_Init(). For more information on the event functions, refer to USB Host Event Callbacks Usage .

Also, double check that your USB bus is properly supplied with 5 volts.

When connecting a given device to my host, I get an error RTOS_ERR_rsrc_ALLOC. However, when I connect another similar device, I don't get any error. Why's that?

Event though two different devices may appear to have exactly the same features, they may present themselves quite differently to the host. This is especially true for the HID devices.

If you set the configuration USBH_CFG_OPTIMIZE_SPD_EN and/or USBH_CFG_INIT_ALLOC_EN to DEF_ENABLED, refer to USB Host Device Resource Needs for more information on how many resources you should allocate to account for the worst cases.

If you didn't enable either of these configurations, make sure that the memory segments used by the USB Host module are large enough.

CAN

CAN

A controller area network (CAN) is a high-integrity serial bus system for networking intelligent devices. CAN has emerged as the standard in-vehicle network thanks to its low cost, performance, and upgradeability. Several higher-level protocols have been standardized on CAN, such as CANopen and DeviceNet. Other markets have widely adopted these additional protocols, which are now standards for industrial communications.

Micrium OS provides two stacks to deploy a CAN device network: Micrium OS CAN Bus and Micrium OS CANopen. Micrium OS CANopen allows high-level data transfers on the CAN network using the CAN Bus stack.

CANopen

CANopen

CANopen is a high-level communications protocol and device-profile specification based on the CAN (Controller Area Network) protocol. It is aimed at embedded networking applications, such as in-vehicle networks. CANopen is a widespread protocol used in various industries, especially in automation and motion applications.

Micrium OS CANopen is a module designed specifically for embedded systems. Built from the ground up with Micrium's quality, scalability, and reliability, it allows you to create networked CANopen devices with the required features.

This manual describes how to initialize, start, and use Micrium OS CANopen. It explains the various configuration values and their uses, as well as how to set up and interact with a CANopen device (also known as a node). It also provides information such as overview, configuration possibilities, implementation details and examples of typical usage.

More information on Micrium OS CANopen can be found in the following sections:

- CANopen Overview
- Integrating CANopen Into Your Project
- CANopen Example Applications
- CANopen Configuration
- CANopen Programming Guide

CANopen Overview

- Specifications
- Features
- Limitations

Specifications

The Micrium CANopen stack implementation complies with the following specification:

- CiA 301: CANopen application layer and communication profile

Features

- Object Dictionary
  - Unlimited number of objects
  - Support for String objects
  - Support for Domain objects
- Network Management (NMT) service
  - NMT heartbeat producer
  - NMT heartbeat consumer
- Service Data Object (SDO) service
  - SDO server (up to 127 servers)
  - SDO expedited transfer
  - SDO normal transfer
- Emergency (EMCY) service
  - EMCY producer
  - EMCY history
- Process Data Object (PDO) service
  - PDO producer (up to 512 transmit PDOs)
  - PDO consumer (up to 512 receive PDOs)
  - PDO communication parameter (static and dynamic)
  - PDO mapping parameter (static and dynamic)

  - PDO synchronous cyclic/acyclic and asynchronous
- Synchronization (SYNC) service
  - Used in conjunction with PDO synchronous cyclic/acyclic
- One unique Service task within the CANopen stack, which centralizes the communication objects processing. The application does not access directly the CAN bus and can perform safe accesses to the object dictionary.
- Application callbacks available to fine-tune some operations during communication objects processing by the CANopen Service task. Application callbacks are available for the following objects: NMT state transitions, PDO transfers, NMT heartbeat consumer, parameters store/restore.
- Node specifications defined thanks to the CANopen Configurator tool.

Limitations

- No support for NMT error control known as node guarding (specification CiA 301). NMT node guarding is not recommended by CiA for new designs. Since the CANopen stack already supports NMT heartbeat, node guarding is not required.
- No support for SDO client (specification CiA 301).
- No support for SDO block transfer (specification CiA 301).
- No support for EMCY consumer (specification CiA 301).
- No support for PDO remote transfer (specification CiA 301).
- No support for LSS used to configure the node-ID and the bit rate via the CAN network (specification CiA 305).
- No support for Time Stamp object (specification CiA 301).

Integrating CANopen Into Your Project

Micrium OS CANopen is composed of several components, each of which is a set of files that implement specific functions. To use CANopen, you must add these files to your project and populate your RTOS Description File .

Starting the CANopen Module Quickly

Micrium offers a quick example application that demonstrates how to initialize, add a controller, and start the Micrium OS CANopen module. We highly recommend that you start with one of these examples.

The section CANopen Example Applications describes each example application.

Configuring CANopen

Micrium OS CANopen module can be configured to optimize memory usage and features. The page CANopen Compile-Time Configuration explains how the CANopen can be configured at compile-time. The page CANopen Run-Time Configuration explains how the CANopen can be configured at run-time.

CANopen Example Applications

This section describes the example applications for the Micrium OS CANopen stack. See Example Applications section for further information about how to enable an example application.

- CANopen Module Initialization Example
  - Description
  - Configuration
  - Location
  - API
- CANopen Node Start Example
  - Description
  - Configuration
  - Location
  - API
- CANopen Object Dictionary Read/Write Example
  - Description
  - Configuration
  - Location
  - API

CANopen Module Initialization Example

Description

This example shows how to initialize the CANopen stack.

The CANopen module initialization example accomplishes the following tasks:

- Call CANopen_Init() to initialize the CANopen resources and service task.
- Call CANopen_NodeAdd() to add a node instance to the stack and configure it.

Configuration

No specific configuration applies to this example.

Location

- /examples/canopen/ex_canopen.c
- /examples/canopen/ex_canopen.h

API

API	Description
Ex_CANopen_Init()	This function performs the different example steps mentioned in the section Description . The function must be called by your application task prior to calling any other CANopen examples.

CANopen Node Start Example

Description

This example shows how to start the CANopen node(s). It accomplishes the following tasks:

- Call CANopen_NodeStart() to start the CAN controller associated with the node instance.

Configuration

No specific configuration applies to this example.

Location

- /examples/canopen/ex_canopen.c
- /examples/canopen/ex_canopen.h

API

API	Description
Ex_CANopen_NodeStart()	This function performs the example steps mentioned in the section Description . The function must be called by your application task prior to calling any other CANopen examples and after calling the initialization example.

CANopen Object Dictionary Read/Write Example

Description

This example shows how to perform read and write operations with the CANopen Object Dictionary. It accomplishes the following tasks:

- It reads the value of an object at index 0x1017 (CANopen Producer Heartbeat Time) and shows it on the console.
-

It writes a new value for this object entry.
- It reads again the value of object 0x1017 and shows it on the console to confirm that the write operation is successful.
- It sends a Node Reset message and reads again the value of object 0x1017 to show that it has come back to its default value.
  - It also demonstrates how the Parameter Groups and the callbacks structure work to reload parameters from a non-volatile memory. Refer to page Communication Objects Callbacks Usage for more details about the available callbacks.

Configuration

No specific configuration applies to this example.

Location

- /examples/canopen/ex_canopen.c
- /examples/canopen/ex_canopen.h

API

API	Description
Ex_CANopen_DictRdWrReset()	This function performs the example steps mentioned in the section Description .

CANopen Configuration

To configure Micrium OS CANopen, there are two groups of configuration parameters:

- CANopen Compile-Time Configuration
- CANopen Run-Time Configuration

This section explains how to set up these configuration groups.

CANopen Compile-Time Configuration

- Object Dictionary Configuration
- SDO Server Configuration
- Emergency Producer Configuration
- Timer Configuration
- Receive PDO Configuration
- Transmit PDO Configuration
- SYNC Consumer Configuration
- Debug Configuration

You can configure Micrium OS CANopen at compile time via a set of #defines located in canopen_cfg.h file. CANopen uses #defines when possible because they allow code and data sizes to be scaled at compile time based on enabled features. This allows the read-only memory (ROM) and random-access memory (RAM) footprints of Micrium OS CANopen to be adjusted based on application requirements.

We recommend that you begin the configuration process with the default configuration values, which are shown in **bold** in the next sections.

These sections are organized according to the order in Micrium OS CANopen's template configuration file, canopen_cfg.h.

Object Dictionary Configuration

Table - Object Dictionary Configuration

Constant	Description	Possible values
CANOPEN_OBJ_PARAM_EN	This constant enables or disables the loading/saving of parameters from/into non-volatile memory. If enabled, the parameter callback functions are used when accessing the standard object directory entries at index 0x1010 (Store) and 0x1011 (Restore default parameters). The CANopen stack will perform some plausibility checks on the accessed entries. In case of a configuration error, the corresponding error code will be set within the node error.	DEF_ENABLED, DEF_DISABLED
CANOPEN_OBJ_STRING_EN	This constant enables or disables the string object management inside the object dictionary.	DEF_ENABLED, DEF_DISABLED
CANOPEN_OBJ_DOMAIN_EN	This constant enables or disables the domain object management inside the object dictionary.	DEF_ENABLED, DEF_DISABLED

SDO Server Configuration

Table - SDO Server Configuration

Constant	Description	Possible values
CANOPEN_SDO_MAX_SERVER_QTY	This constant defines the number of SDO servers for a given node.	Integer between 1 and 128 (default is **2**)
CANOPEN_SDO_DYN_ID_EN	This constant enables or disables the ability to change an SDO server COB-ID at runtime. It is possible to dynamically assign an SDO server COB-ID if certain SDO server parameters (located in the indexes range 0x1200-0x127F) have been configured to allow it.	DEF_ENABLED, DEF_DISABLED
CANOPEN_SDO_SEG_EN	This constant enables or disables the SDO segmented transfer. The segmented transfer, also referred as a non-expedited transfer, allows you to transfer data of any size during the segment phase (upload/download SDO segment). If it is disabled, user data is limited to four bytes during the init phase of an SDO transfer (initiate SDO upload/download).	DEF_ENABLED, DEF_DISABLED

Emergency Producer Configuration

Table - Emergency Producer Configuration

Constant	Description	Possible values
CANOPEN_EMCY_MAX_ERR_QTY	This constant sets the number of emergency codes for a given node. If the emergency module is enabled, the object entries 0x1001 (Error register) and 0x1014 (COB-ID EMCY) must be configured correctly. The CANopen stack will perform some plausibility checks on the emergency entries. In case of a configuration error, the corresponding error code will be set within the node error.When this configuration is set to 0, all the Emergency service support is disabled within the stack.	Integer between 1 and 255 (default is **6**)

Constant	Description	Possible values
CANOPEN_EMCY_REG_CLASS_EN	This constant enables or disables the emergency error classes such as current, voltage, and temperature within the error register object entry (index 0x1001) considered as optional by the CiA 301 specification.	DEF_ENABLED, DEF_DISABLED
CANOPEN_EMCY_EMCY_MAN_EN	This constant enables or disables the manufacturer-specific information field in the emergency messages. The manufacturer-specific field corresponds to 5 bytes in each 8-byte emergency message.	DEF_ENABLED, DEF_DISABLED
CANOPEN_EMCY_HIST_EN	This constant enables or disables the emergency error history. If this configuration is set to DEF_ENABLED, the standard object directory entry 0x1003 (pre-defined error field) must be configured according to the CANopen standard. The CANopen stack will perform some plausibility checks on the emergency history entries. In case of a configuration error, the corresponding error code will be set within the node error.	DEF_ENABLED, DEF_DISABLED
CANOPEN_EMCY_HIST_MAN_EN	This constant enables or disables the manufacturer-specific information field within the EMCY history entries, located in the standard object "pre-defined error field" at index 0x1003. If it is set to DEF_ENABLED , you can specify your own specific information by passing the correct argument to the function CANopen_EmcySet() (cf. Listing - CANopen EmcySet() Usage in the *Using Communication Objects* page for an example).	DEF_ENABLED, DEF_DISABLED

Timer Configuration

Table - Timer Configuration

Constant	Description	Possible values
CANOPEN_TMR_MAX_QTY	This constant sets the timer action queue length. The timer is used by some communication objects such as NMT heartbeat and by PDOs.	Integer between 1 and 255 (default is **5**)

Receive PDO Configuration

Table - Receive PDO Configuration

Constant	Description	Possible values
CANOPEN_RPDO_MAX_QTY	This constant sets the number of active Receive PDOs (RPDO) for a given node. A Receive PDO is described in the node's object dictionary by the RPDO communication parameter (starting at index 0x1400) and mapping parameter (starting at index 0x 1600). You must correctly configure these parameters for each RPDO you use. The CANopen stack performs some plausibility checks on the communication and mapping profile of the RPDOs. In case of a configuration error, the corresponding error code will be set within the node error.When this configuration is set to 0, all the RPDO service support is disabled within the stack.	Integer between 1 and 512 (default is **4**)

Constant	Description	Possible values
CANOPEN_RPDO_MAX_MAP_QTY	This constant sets the number of receive PDO mapping objects.	Integer between 1 and 64 (default is 2)
CANOPEN_RPDO_DYN_COM_EN	This constant enables or disables support for dynamic communication profiles within the standard RPDO object entries at index 0x1400 (RPDO communication parameter).	DEF_ENABLED, DEF_DISABLED
CANOPEN_RPDO_DYN_MAP_EN	This constant enables or disables support for dynamic mapping profiles within the standard RPDO object entries at index 0x1600 (RPDO mapping parameter). Setting this configuration to DEF_ENABLED requires you to also enable support for dynamic communication profiles, constant CANOPEN_RPDO_DYN_COM_EN.	DEF_ENABLED, DEF_DISABLED

Transmit PDO Configuration

Table - Transmit PDO Configuration

Constant	Description	Possible values
CANOPEN_TPDO_MAX_QTY	This constant sets the number of active Transmit PDOs (TPDO) for a given node. A Transmit PDO is described in the node's object dictionary by the TPDO communication parameter (starting at index 0x1800) and mapping parameter (starting at index 0x1A00). You must correctly configure these parameters for each TPDO you use. The CANopen stack performs some plausibility checks on the communication and mapping profile of the TPDOs. In case of a configuration error, the corresponding error code will be set within the node error.When this configuration is set to 0, all the TPDO service support is disabled within the stack.	Integer between 1 and 512 (default is 4)
CANOPEN_TPDO_MAX_MAP_QTY	This constant sets the number of transmit PDO mapping objects.	Integer between 1 and 64 (default is 2)
CANOPEN_TPDO_DYN_COM_EN	This constant enables or disables the support of dynamic communication profile within the standard TPDO object entries at index 0x1800 (TPDO communication parameter).	DEF_ENABLED, DEF_DISABLED
CANOPEN_TPDO_DYN_MAP_EN	This constant enables or disables the support of dynamic mapping profile within the standard TPDO object entries at index 0x1A00 (TPDO mapping parameter).	DEF_ENABLED, DEF_DISABLED

SYNC Consumer Configuration

Table - SYNC Consumer Configuration

Constant	Description	Possible values
CANOPEN_SYNC_EN	This constant enables or disables the use of synchronous PDOs.	DEF_ENABLED, DEF_DISABLED

Debug Configuration

Table - Debug Configuration

Constant	Description	Possible values
CANOPEN_DBG_CTR_ERR_EN	This constant enables or disables the error counters inside the stack. When enabled, if errors occur during the CAN frames decoding and processing by the CANopen Service task, these errors will be recorded into internal counters. The error counters are per node instance.	DEF_ENABLED, DEF_DISABLED

CANopen Run-Time Configuration

- Core Configuration
- Optional Pre-Init Configuration
  - Memory Segment
  - Maximum Number of Service Task Events
  - Service Task Stack
  - Hardware Timer Period
- Optional Post-Init Configuration
  - Service Task Priority
  - Node Lock Timeout

This section describes the application-specific configurations of the Micrium OS CANopen module, that are specified at run-time. For more information on how to initialize any Micrium OS module, see Stacks Initialization Methods .

Core Configuration

To initialize the Micrium OS CANopen module, you must call the function CANopen_Init(). This function does not have any configuration argument.

Optional Pre-Init Configuration

To perform pre-init configuration, you must call the dedicated configuration functions before calling CANopen_Init(). If no explicit pre-init configuration is performed, default values will be used. These default values are stored in CANopen_InitCfgDflt defined in canopen_core.c.

Memory Segment

Configures the memory segment where the core internal data structures will be allocated.

Type	Function to call	Default	Field from default configuration structure
MEM_SEG*	CANopen_ConfigureMemSeg()	General-purpose heap	.MemSegPtr

Maximum Number of Service Task Events

Configures the maximum number of service task events.

Type	Function to call	Default	Field from default configuration structure
CPU_SIZE_T	CANopen_ConfigureEventQty()	25	.EventQtyTot

Service Task Stack

Configures the size and the start address of the Service task stack.

Type	Function to call	Default	Field from default configuration structure
CPU_INT32U	CANopen_ConfigureSvcTaskStk()	A stack of 512 elements.	.SvcTaskStkSizeElements
void *	CANopen_ConfigureSvcTaskStk()	A stack allocated on Common 's memory segment,	.SvcTaskStkPtr

Hardware Timer Period

Configures the hardware timer period in microseconds. A hardware timer is used by some communication object services such as the NMT heartbeat error control and PDOs.

Type	Function to call	Default	Field from default configuration structure
CPU_INT32U	CANopen_ConfigureTmrPeriod()	1000 microseconds	.HwTmrPeriod

Optional Post-Init Configuration

This section describes configurations that can be set at any time during execution after CANopen_Init() has been called. These configurations are optional. If you do not set them in your application, the default configurations will apply.

Service Task Priority

Sets the priority of the Service task.

Type	Function to call	Default
RTOS_TASK_PRIO	CANopen_SvcTaskPrioSet()	See Appendix A - Internal Tasks

Node Lock Timeout

Sets the Node Lock timeout expressed in milliseconds. The node lock is used to protect access to the object dictionary.

Type	Function to call	Default
CPU_INT32U	CANopen_NodeLockTimeoutSet()	Infinite timeout

CANopen Programming Guide

This section explains how to use the CANopen module.

- Initial Setup of CANopen Module
- Defining Node Specifications
- Communication Objects Callbacks Usage
- Accessing Object Dictionary
- Using Communication Objects

Initial Setup of CANopen Module

This section describes the basic steps required to initialize the CANopen module, and to add, start and stop a CANopen node.

- Initializing the CANopen Module
- Adding Your CANopen Node(s)
- Starting Your CANopen Node(s)

Initializing the CANopen Module

The first step is to initialize the CANopen module core. This is done by calling the function CANopen_Init() . When called, this function will allocate any internal resources needed by the stack and will create the service stack responsible for the CANopen communication objects.

Listing - Example of Call to CANopen Init() in the *Initial Setup of CANopen Module* page shows an example of a call to the function CANopen_Init().

Listing - Example of Call to CANopen_Init()

```
RTOS_ERR err;

CANopen_Init(&err);
```

```
/* An error occurred. Error handling should be added here. */}
```

Adding Your CANopen Node(s)

The CANopen stack supports multiple nodes. A node refers to a CANopen device. The function CANopen_NodeAdd() allows you to add a node to the CANopen stack and to configure it. The function CANopen_NodeAdd() can be called for each node you need to support. A node is associated with one unique CAN bus controller.

Listing - Example of Call to CANopen NodeAdd() in the *Initial Setup of CANopen Module* page shows an example that adds a node. More information about the node specifications configuration can be found in Defining Node Specifications . You can look at Communication Objects Callbacks Usage for more details about the different object callbacks available to your application. If you need to override the default CANopen stack configuration, you can call the CANopen_ConfigureXxxx() functions (cf. CANopen Run-Time Configuration for more details) before calling CANopen_NodeAdd().

Listing - Example of Call to CANopen_NodeAdd()

```
const  CANOPEN_NODE_SPEC Ex_CANopen_NodeSpec = {             /* CONFIGURATION FOR NODE 1.          */
   .NodeId      = 0x1,                                /* Pre-defined Node-ID.              */
   .Baudrate    = 125000,                             /* Default baudrate.                 */
   .DictPtr     = (CANOPEN_OBJ *)&Ex_ObjDict1,               /* Start of object directory.          */
   .DictLen     = sizeof(Ex_ObjDict1)/sizeof(CANOPEN_OBJ),   /* Number of objects in directory.    */
   .EmcyCodePtr = (CANOPEN_EMCY_TBL *)&Ex_EmcyTbl,           /* Start of emergency code table.     */
   .TmrMemPtr   = (CANOPEN_TMR_MEM *)&Ex_TmrMem1,            /* Start of timer manager memory.     */
   .TmrQty      = sizeof(Ex_TmrMem1)/sizeof(CANOPEN_TMR_MEM), /* Maximum number of timers/actions.  */
   .SdoBufPtr   = (CPU_INT08U *)&Ex_SdoBuf1                  /* Start of SDO transfer buffer.      */
};

const  CANOPEN_EVENT_FNCTS  Ex_EventsCB = {                  /* EVENTS CALLBACKS FOR NODE 1.       */
   .RpdoOnRx      = DEF_NULL,
   .TpdoOnTx      = DEF_NULL,
   .StateOnChange = DEF_NULL,
   .HbcOnEvent    = DEF_NULL,
   .HbcOnChange   = DEF_NULL,
   .ParamOnLoad   = Ex_Param_OnLoad,
   .ParamOnSave   = DEF_NULL,
   .ParamOnDflt   = DEF_NULL
};

CANOPEN_NODE_HANDLE  Ex_CANopen_Node1Handle;

void  Ex_CANopen_Init (void)
{
   RTOS_ERR  err;

  /* The CANopen stack has been previously initialized with CANopen_Init(). */

  /* Add node 1 to the CANopen stack with the default stack configuration. */
  Ex_CANopen_Node1Handle = CANopen_NodeAdd("can0",           /* Node 1 associated w/ controller "can0". */
                                     &Ex_CANopen_NodeSpec, /* Reference to node specifications.     */
                                     &Ex_EventsCB,         /* Reference to node event callbacks.     */
                                     &err);
  if (err.Code != RTOS_ERR_NONE) {
      /* An error occurred. Error handling should be added here. */
  }
}
```

Starting Your CANopen Node(s)

Once you have successfully added your node(s), you must start it/them. This is done by calling the function CANopen_NodeStart() . This function must be called for each node you have added. The function will start the operations of the CAN bus controller associated with this node. If the node is already in the "initializing" state according to the CANopen NMT (Network Management), the bootup message will be sent to the NMT master node. Later, you may want to

stop your node(s) by calling CANopen_NodeStop() . Once the node is started, it advertises its presence on the bus and is put in the pre-operational state. At that point, a master device can issue an NMT message to put it in the Operational state.

Listing - Example of Call to CANopen NodeStart() in the *Initial Setup of CANopen Module* page shows an example of a call to CANopen_NodeStart().

Listing - Example of Call to CANopen_NodeStart()

```
RTOS_ERR            err;
CANOPEN_NODE_HANDLE  node1_handle;

/* The CANopen stack has been previously initialized with CANopen_Init(). */
/* The node has been added to the stack with CANopen_NodeAdd(), which     */
/* returned the handle to use for further operations on the node.         */

                        /* Start node 1 operations.            */
CANopen_NodeStart(&node1_handle,
              &err);
if (err.Code != RTOS_ERR_NONE) {
    /* An error occurred. Error handling should be added here. */
}
```

Defining Node Specifications

This section describes how to define a specifications structure for a given node.

The node specifications structure is used to configure a node with parameters such as the node ID, the speed of communication between the nodes, and others. To save time, you can use CANopen Configurator to generate the configuration structure for each node on the network.

- Node Specification
  - Structure Details
    - .NodeId
    - .Baudrate
    - .DictPtr
    - .EmcyCodePtr
    - .TmrMemPtr / .TmrQty
    - .SdoBufPtr
- CANopen Configurator Tool
  - Node ID
  - Baud Rate
  - Object Dictionary
  - Emergency Codes
  - Timers
  - SDO Buffer

Node Specification

When you add a node (by using CANopen NodeAdd()), you must provide the node's specifications. The node specification contains information that ties the CAN bus and your application together to the CANopen stack. This structure can be generated automatically using CANopen Configurator. This tool simplifies the generation of the object dictionary associated with the node, and it also produces an EDS file for your node. Some fields of the node specifications structure can be entered directly in the tool and some are derived from other configuration values. Table - Node Specification Description in the *Defining Node Specifications* page shows each element of the node specification structure.

Listing - Node Specification Structure

```
typedef struct canopen_node_spec {
    CPU_INT08U        NodeId;
    CPU_INT32U        Baudrate;
    CANOPEN_OBJ       *DictPtr;
    CANOPEN_EMCY_TBL  *EmcyCodePtr;
    CANOPEN_TMR_MEM   *TmrMemPtr;
    CPU_INT16U        TmrQty;
    CPU_INT08U        *SdoBufPtr;
} CANOPEN_NODE_SPEC;
```

Table - Node Specification Description

Type	Name	Description	Configuration when using CANopen Configurator tool
CPU_INT08U	NodeId	Default Node ID	User-defined directly within the configuration tool
CPU_INT32U	Baudrate	Default baud rate	User-defined directly within the configuration tool
CANOPEN_OBJ *	DictPtr	Node object dictionary pointer	Name of the array is predefined and derived from NodeId Size of the array is based on how many entries you have defined in the Object Dictionary tab of the configuration tool.
CANOPEN_EMCY_TBL *	EmcyCodePtr	Application emergency information pointer	Name of the array is predefined and derived from NodeId Size of the array is based on a user-defined value of CANOPEN_EMCY_MAX_ERR_QTY set within the configuration tool.
CANOPEN_TMR_MEM *	TmrMemPtr	Timer memory block	Name of the array is predefined and derived from NodeId Size of the array is based on a user-defined value of CANOPEN TMR MAX QTY set within the configuration tool.
CPU_INT16U	TmrQty	Number of timer memory blocks	Number of TmrMemPtr array elements
CPU_INT08U *	SdoBufPtr	SDO Transfer Buffer Memory Start	Name of the array is predefined and derived from NodeId Size of array is based on a user-defined value of CANOPEN SDO MAX SERVER QTY set within the configuration tool.

Listing - Node Specification Example

```
const CANOPEN_OBJ Ex_ObjDict1[] = {
    { CANOPEN_KEY(0x1000,0x0, CANOPEN_OBJ_UNSIGNED32|CANOPEN_OBJ___R_), 0, (CPU_INT32U)&Ex_Var_1000_0 },
    { CANOPEN_KEY(0x1001,0x0, CANOPEN_OBJ_UNSIGNED8 |CANOPEN_OBJ___R_), 0, (CPU_INT32U)&Ex_Var_1001_0 },
    CANOPEN_OBJ_DICT_ENDMARK
};

CANOPEN_TMR_MEM   Ex_TmrMem1[CANOPEN_TMR_MAX_QTY];          /* Allocation of Timer memory blocks.        */

CANOPEN_EMCY_TBL  Ex_EmcyTbl[CANOPEN_EMCY_MAX_ERR_QTY] = {
    { 0, 0x1000 },
    { 1, 0x1001 },
};

CPU_INT08U        Ex_SdoBuf1[CANOPEN_SDO_MAX_SERVER_QTY];    /* Allocation of SDO transfer memory.        */

const CANOPEN_NODE_SPEC Ex_CANopen_NodeSpec = {
    .NodeId      =                     0x1,
    .Baudrate    =                     125000,
    .DictPtr     = (CANOPEN_OBJ      *)&Ex_ObjDict1,
    .EmcyCodePtr = (CANOPEN_EMCY_TBL *)&Ex_EmcyTbl,
    .TmrMemPtr   = (CANOPEN_TMR_MEM  *)&Ex_TmrMem1,
    .TmrQty      =                     sizeof(Ex_TmrMem1)/sizeof(CANOPEN_TMR_MEM),
```

```
.SdoBufPtr     = (CPU_INT08U       *)&Ex_SdoBuf1
};
```

Structure Details

.NodeId

The Node ID uniquely identifies one CAN device on a network. Because a device could have more than one CAN bus, the Node ID is also used when naming variables to prevent name clashes. For instance, if your device uses nodes 1 and 3, their object dictionary, as generated by the configuration tool, will be named Ex_ObjDict1 and Ex_ObjDict3.

.Baudrate

The baud rate is used to configure the CAN bus to operate at the specified speed, expressed in kbit/s. Its value must be among the ones set by the CANopen standard: 10k, 20k, 50k, 125k, 250k, 500k, 800k, 1000k.

.DictPtr

DictPtr is a pointer to an array of CANOPEN OBJ , representing the node's Object Dictionary. We recommend that you use the configuration tool to generate the object dictionary, as its content may be too complex to handle manually.

.EmcyCodePtr

EmcyCodePtr is a pointer to an array containing a map of user emergency codes. The array is of type CANOPEN_EMCY_TBL . Each element in this array has two parts: an emergency code and a number that maps the error code to a bit in the Error Register (index 0x1001).

For instance, in the Ex_EmcyTbl[] variable above, error 0x1000 maps to bit 0 of the error register.

During the node setup, an emergency structure is assigned to the node. The aforementioned *user* emergency table is placed in this structure, as well as counters to keep track of each type of error code received.

.TmrMemPtr / .TmrQty

This is a pointer to an array of one or more timer memory structures. A timer memory structure is made up of two elements:

- The Timer structure, which manages the timer queue
- The Action structure, which manages the action queue for a timer, including settings such as callback function and parameters, and cycle time in the case of periodic actions

.SdoBufPtr

The SDO Buffer is used to temporarily hold data in an SDO transfer. Because an SDO transfer can be broken into multiple messages, a buffer is required to reassemble the transfer into a consistent state.

CANopen Configurator Tool

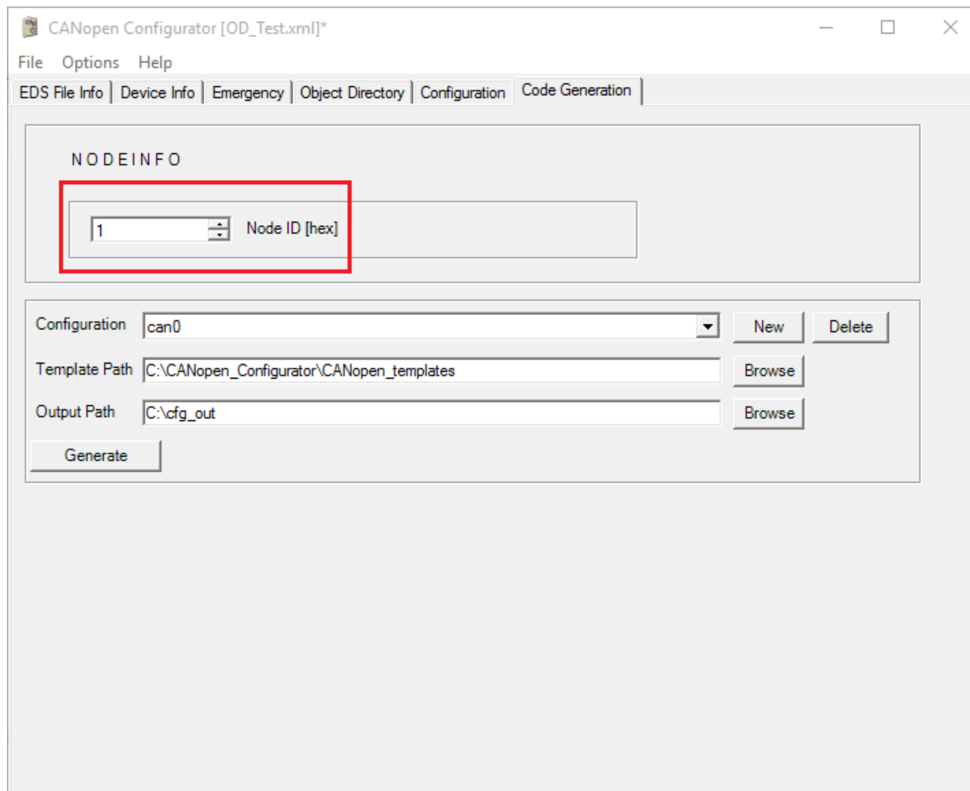
This application is provided to help you create and maintain an object dictionary and generate relevant C code, including the structures and configuration files needed by the CANopen module.

This section explains the relationship between the above structural elements and the configuration tool.

Node ID

The default node ID (.NodeId) can be changed by editing the "Node ID" field in the *Code Generation* tab. Figure - Default Node-ID in the *Defining Node Specifications* page shows where to change it.

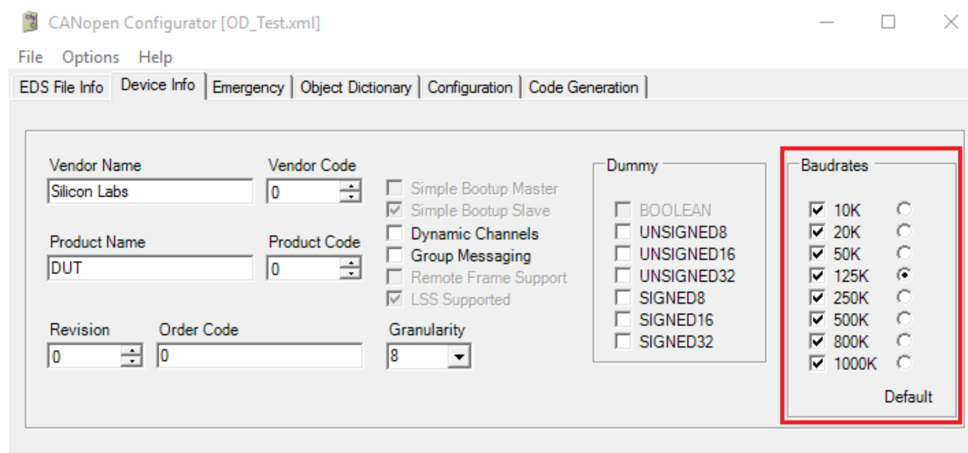
Figure - Default Node-ID



Baud Rate

The default CAN Bus baud rate (.Baudrate) can be changed by selecting a new baud rate in the *Device Info* tab. Figure - Default Baudrate in the *Defining Node Specifications* page shows where to change it.

Figure - Default Baudrate

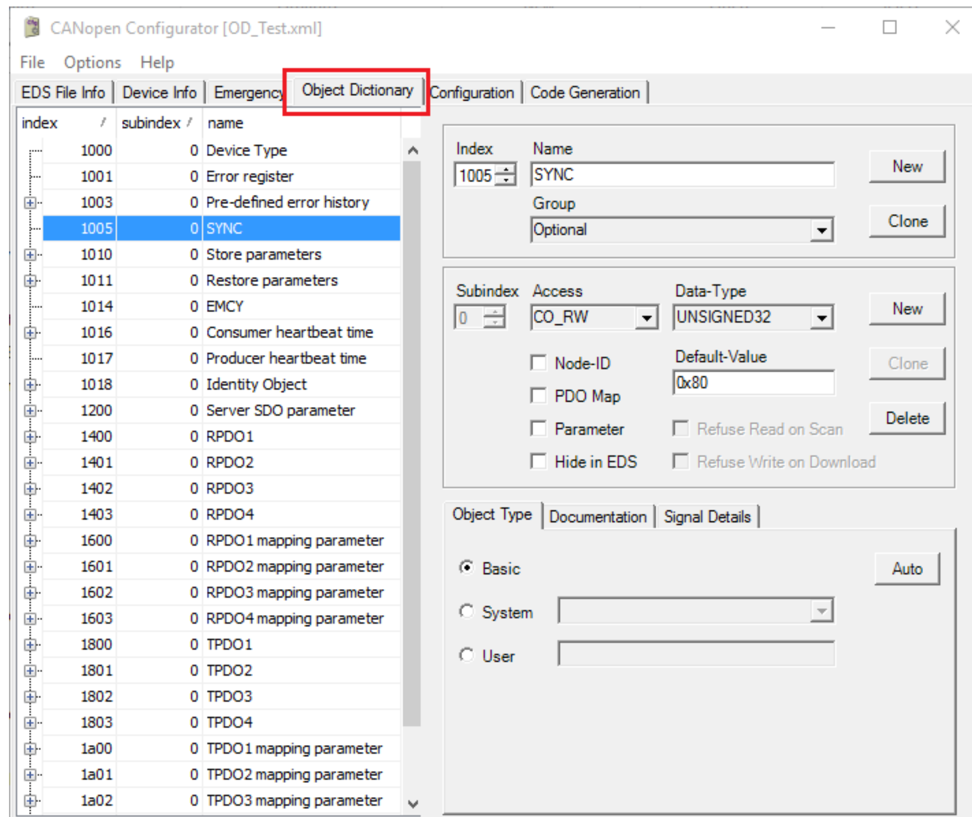


Object Dictionary

.DictPtr is related to the Object Dictionary. The object dictionary is set up through its own tab named *Object Dictionary*. Figure - Object Dictionary in the *Defining Node Specifications* page shows where to change it.

The number of entries in the dictionary (plus 1 for an end marker) establishes the size of the array pointed to by DictPtr.

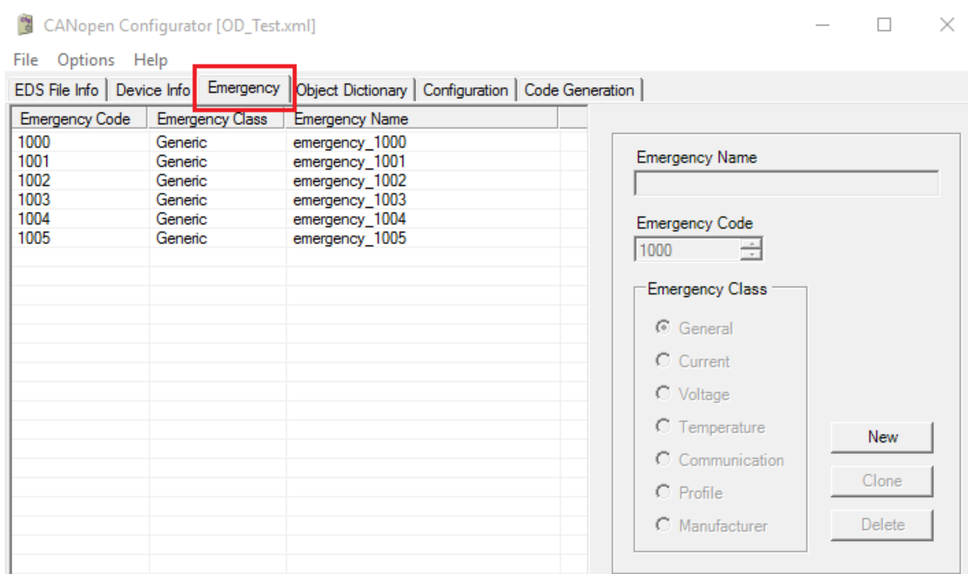
Figure - Object Dictionary



Emergency Codes

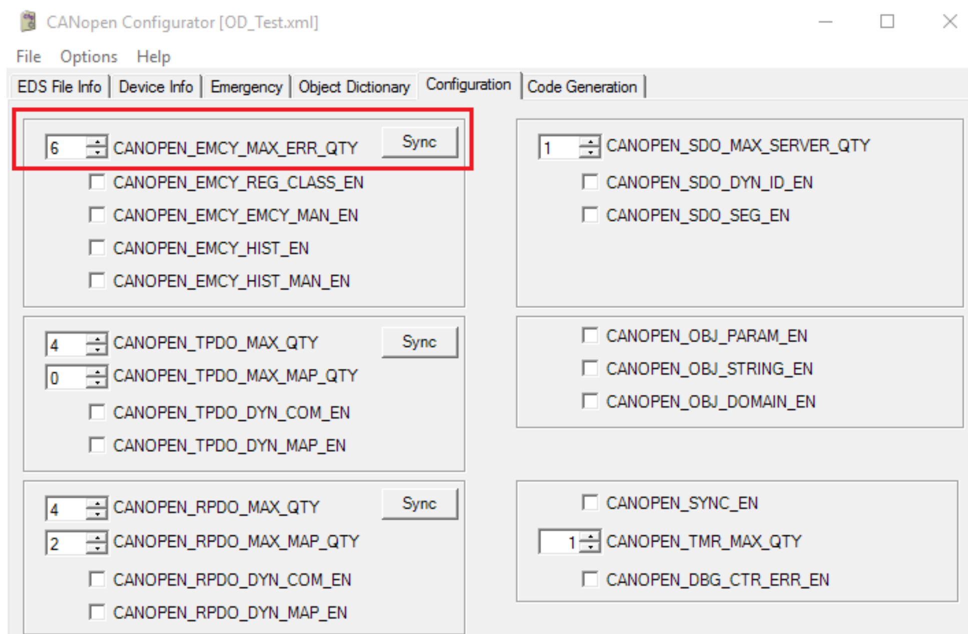
.EmcyCodePtr is related to the Emergency objects. The Emergency objects are configured through the *Emergency* tab. Figure - Emergency Objects in the *Defining Node Specifications* page shows where to change it.

Figure - Emergency Objects



The size of the .EmcyCodePtr array (Ex_EmcyTbl[] in the example above) is determined by the value of CANOPEN_EMCY_MAX_ERR_QTY, which can be set in the *Configuration* tab of the tool. The value can be entered manually, or the Sync button may be used to retrieve the number of entries defined in the Emergency tab. Figure - Emergency Configuration in the *Defining Node Specifications* page shows where to change it.

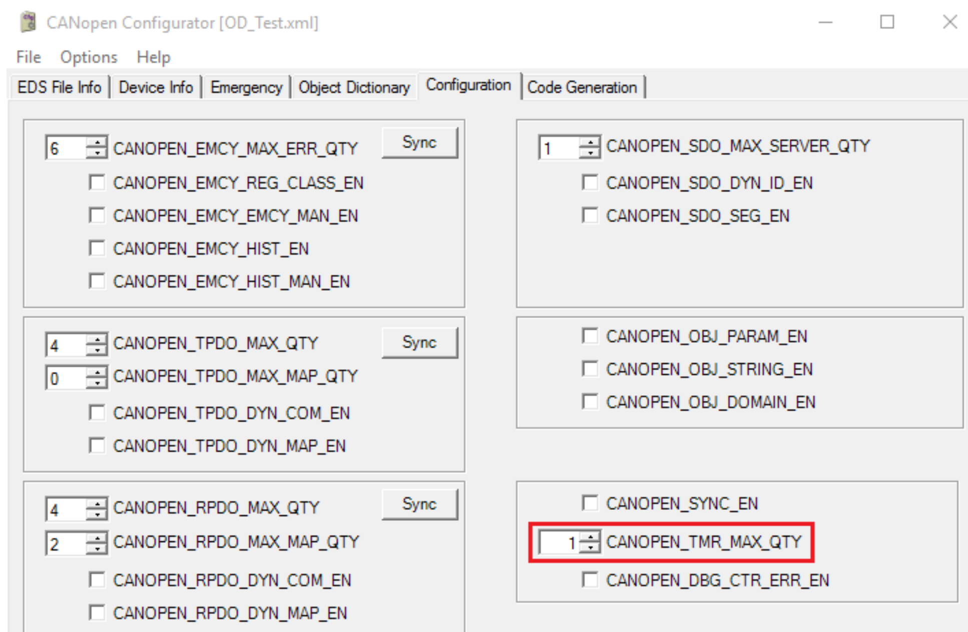
Figure - Emergency Configuration



Timers

.TmrMemPtr and .TmrQty are related to CANopen Timers management. The only relevant user-defined value is the number of timers, which can be set in the *Configuration* tab. Figure - Timers in the *Defining Node Specifications* page shows where to change it. The number of timers (CANOPEN_TMR_MAX_QTY) is used to determine the size of the Timers memory block array (Ex_TmrMem1[] in the example above).

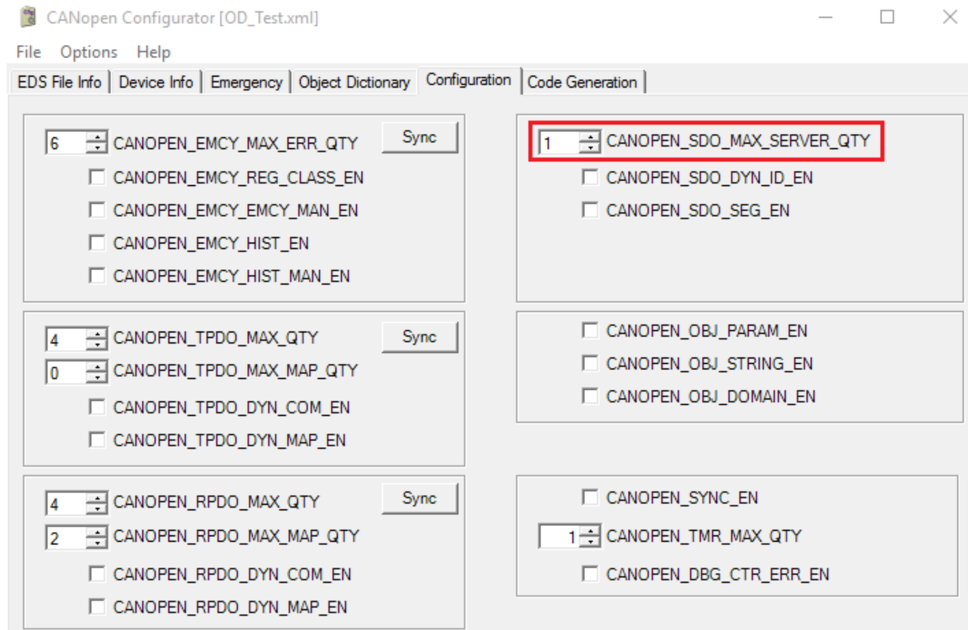
Figure - Timers



SDO Buffer

.SdoBufPtr is related to the handling of SDO segments. The only relevant user-defined value is the number of SDO servers (CANOPEN_SDO_MAX_SERVER_QTY), which can be set in the *Configuration* tab. Figure - SDO Buffer in the *Defining Node Specifications* page shows where to change it. This value is used to determine the size of the SDO Transfer Memory array (Ex_SdoBuf1[] in the example above).

Figure - SDO Buffer



Communication Objects Callbacks Usage

This section explains the callback functions that are available for use by some communication objects. These callbacks allow your application to customize certain actions while the CANOpen stack is processing communication objects.

The callbacks are available for the following CANopen services processing:

- Parameters
- Process Data Object (PDO)
- Network Management (NMT)
- Heartbeat consumer (NMT error control)
- Initializing Callbacks
- Parameters
  - ParamOnLoad()
  - ParamOnSave()
  - ParamOnDflt()
- PDO
  - TpdoOnTx()
  - RpdoOnRx()
- NMT
  - StateOnChange()
- Heartbeat Consumer
  - HbcOnEvent()
  - HbcOnChange()

Initializing Callbacks

Communication object callbacks can be set when calling CANopen_NodeAdd(). The second parameter of this function takes a pointer to a CANOPEN_EVENT_FNCTS structure. Listing - Communication Object Callbacks in the *Communication Objects Callbacks Usage* page provides an example of how to declare the callbacks.

Listing - Communication Object Callbacks

```
RTOS_ERR           err;
CANOPEN_NODE_HANDLE  node_handle;
CANOPEN_NODE_SPEC    spec_node;

CANOPEN_EVENT_FNCTS  event_fnt = {
   .RpdoOnRx      = &RpdoOnRx,
   .TpdoOnTx      = &TpdoOnTx,
   .StateOnChange = &StateOnChange,
   .HbcOnEvent    = &HBCOnEvent,
   .HbcOnChange   = &HbcOnChange,
   .ParamOnLoad   = &ParamOnLoad,
   .ParamOnSave   = &ParamOnSave,
   .ParamOnDflt   = &ParamOnDflt
};

node_handle = CANopen_NodeAdd(&spec_node,
                              &event_fnt,
                              &err);
APP_RTOS_ASSERT_DBG((err.Code == RTOS_ERR_NONE), ;);
```

Parameters

Your application has access to three callbacks to manage the storing and restoring of object dictionary parameters. Listing - Example of CANOPEN PARAM Usage in the *Communication Objects Callbacks Usage* page provides a basic definition of an object dictionary with the store and restore features enabled and shows an example of how to declare the parameters group.

Listing - Example of CANOPEN_PARAM Usage

```
typedef struct APP_PARAM_GROUP {
    CPU_INT32U    AppVar_1005_0;     /* ------------------ [1005:0] SYNC ------------------ */
    CPU_INT32U    AppVar_1010_0;     /* [1010:0] Store parameters                           */
    CPU_INT32U    AppVar_1010_1;     /* [1010:1] Save all parameter                         */
    CPU_INT08U    AppVar_1011_0;     /* [1011:0] Restore parameters                         */
    CPU_INT32U    AppVar_1011_1;     /* [1011:1] Restore all parameter                      */
} APP_PG;

...

APP_PG  App_ParamGrp  = {               (1)
        .AppVar_1005_0 = 0x80,
        .AppVar_1010_0 = 1,
        .AppVar_1010_1 = 1,
        .AppVar_1011_0 = 1,
        .AppVar_1011_1 = 1,
};

APP_PG  App_ParamDef = {                (2)
        .AppVar_1005_0 = 0x80,
        .AppVar_1010_0 = 0,
        .AppVar_1010_1 = 0,
        .AppVar_1011_0 = 0,
        .AppVar_1011_1 = 0,
};

...

const CANOPEN_PARAM App_ParamGrpCfg = {   (3)
    .MemBlkSize     = sizeof(APP_PG),
    .StartMemBlkPtr = (CPU_INT08U *)&App_ParamGrp,
    .DfltMemBlkPtr  = (CPU_INT08U *)&App_ParamDef,
    .ResetType      = CANOPEN_RESET_COMM,  (4)
    .IdPtr          = (void*)"AppParam",
    .Val            = CANOPEN_PARAM___E
};

...

const CANOPEN_OBJ AppObjDir[] = {
    /*------------------------------------------------------------------------------------------*/
    /* [1005:xx] - SYNC */
    { CANOPEN_KEY(0x1005,0x0, CANOPEN_OBJ_UNSIGNED32|CANOPEN_OBJ___RW), 0,
      (CPU_INT32U)&App_ParamGrp.AppVar_1005_0 },
    /*------------------------------------------------------------------------------------------*/
    /* [1010:xx] - Store parameters */
    { CANOPEN_KEY(0x1010,0x0, CANOPEN_OBJ_UNSIGNED8|CANOPEN_OBJ___R_), 0,
      (CPU_INT32U)&App_ParamGrp.AppVar_1010_0 },
    { CANOPEN_KEY(0x1010,0x1, CANOPEN_OBJ_UNSIGNED32|CANOPEN_OBJ___RW), CANOPEN_OBJ_TYPE_PARAM,
      (CPU_INT32U)&App_ParamGrpCfg },      (5)
    /*------------------------------------------------------------------------------------------*/
    /* [1011:xx] - Restore parameters */
    { CANOPEN_KEY(0x1011,0x0, CANOPEN_OBJ_UNSIGNED8|CANOPEN_OBJ___R_), 0,
      (CPU_INT32U)&App_ParamGrp.AppVar_1011_0 },
    { CANOPEN_KEY(0x1011,0x1, CANOPEN_OBJ_UNSIGNED32|CANOPEN_OBJ___RW), CANOPEN_OBJ_TYPE_PARAM,
      (CPU_INT32U)&App_ParamGrpCfg },
    CANOPEN_OBJ_DICT_ENDMARK
};
```

(1) This structure holds all the values used by the object dictionary. All these values can be restored to their original state using the communication object at index 0x1011 (restore parameters).

(2) This structure holds the default values of a specific part of the object dictionary. These values are written back to the object dictionary when using the communication object at index 0x1011 (restore parameters).

(3) This structure is used to store pointers to data structures such as the default and current values of the object dictionary.

(4) Specifies the reset type for a parameter group. This can be CANOPEN_RESET_NODE or CANOPEN_RESET_COMM. If CANOPEN_RESET_NODE is used, the parameters of the manufacturer-specific profile area and the standardized device profile area are both set to their power-on values. If CANOPEN_RESET_COMM is used, the parameters of the communication profile area are set to their power-on values.

(5) A pointer to the configuration structure must be passed as data to the store/restore communication object into the object dictionary.

ParamOnLoad()

This callback is of type ParamOnLoad(). This callback is optional and must be defined in your application.

The callback restores the current object dictionary with default values from a memory segment. Listing - ParamOnLoad() Example in the *Communication Objects Callbacks Usage* page show a basic implementation of the ParamOnLoad() callback.

Listing - ParamOnLoad() Example

```
CPU_BOOLEAN Ex_ParamOnLoad (CANOPEN_NODE_HANDLE   handle,
                            CANOPEN_PARAM        *p_pg)
{
   PP_UNUSED_PARAM(handle);

   Mem_Copy(p_pg->StartMemBlkPtr,            /* Pointer to parameter memory block in object dictionary.   */
            p_pg->DfltMemBlkPtr,             /* Pointer to memory segment from where values are restored. */
            p_pg->MemBlkSize);

   return (DEF_TRUE);
}
```

ParamOnSave()

This callback is of type ParamOnSave(). This callback is optional and must be defined in your application.

The callback is called when a master writes a valid signature at the index 0x1010 (store parameters*)* of the node's object dictionary. The signature that enables the parameters to be stored is "save" – that is, in hexadecimal, 0x65766173, which represents 'e', 'v', 'a', 's'.

The current state of the object dictionary is saved into memory for future use.

Listing - ParamOnSave() Example

```
CPU_BOOLEAN  Ex_ParamOnSave (CANOPEN_NODE_HANDLE   handle,
                             CANOPEN_PARAM        *p_pg)
{
   /* The object dictionary must be stored somewhere in memory for future use.
      The pointer p_pg->StartMemBlkPtr contains the address of the first entry
      of the object dictionary. The number of bytes in the object dictionary
      is given by p_pg->MemBlkSize. */

   return (DEF_TRUE);
}
```

ParamOnDflt()

This callback is of type ParamOnDflt(). This callback is optional and must be defined in your application.

The callback is called when a master writes a valid signature at the index 0x1011 (restore default parameters*)* of the node's object dictionary. The signature that enables the default parameters to be restored is "load" – that is, in hexadecimal,

0x64616F6C, which represents 'd', 'a', 'o', 'l'. Listing - ParamOnDflt() Example in the *Communication Objects Callbacks Usage* page shows a basic implementation of the ParamOnDflt() callback.

Listing - ParamOnDflt() Example

```
CPU_BOOLEAN Ex_ParamOnDflt (CANOPEN_NODE_HANDLE   handle,
                            CANOPEN_PARAM        *p_pg)
{
    PP_UNUSED_PARAM(handle);

    Mem_Copy(p_pg->StartMemBlkPtr,            /* Pointer to parameter memory block in object dictionary.   */
             p_pg->DfltMemBlkPtr,             /* Pointer to memory segment from where values are restored. */
             p_pg->MemBlkSize);

    return (DEF_TRUE);
}
```

PDO

PDO callbacks are useful for checking data payloads after a PDO message is received or before it is transmitted. This payload data is stored in the second argument of each PDO callback.

This structure is used to store various information about a message received from or transmitted to the bus. CANOPEN_IF_FRM shows a brief description of each argument of the following structure.

Listing - CANopen Frame Structure

```
typedef struct canopen_if_frm {
    CPU_INT32U  MsgId;
    CPU_INT08U  Data[8];
    CPU_INT08U  DLC;
    CPU_INT08U  MsgNum;
} CANOPEN_IF_FRM;
```

TpdoOnTx()

TpdoOnTx() is called before every TPDO transmission. It can be used to change the PDO data before sending it.

This callback is of type TpdoOnTx(). TpdoOnTx() is optional and must be defined in your application. Listing - TpdoOnTx() Example in the *Communication Objects Callbacks Usage* page shows a basic implementation of the TpdoOnTx() callback.

Listing - TpdoOnTx() Example

```
void Ex_TpdoOnTx (CANOPEN_NODE_HANDLE   handle,
                  CANOPEN_IF_FRM       *p_frm)
{
    PP_UNUSED_PARAM(handle);
                                      /* Change payload data before sending it over the bus.  */
    p_frm->Data[0u] = 1u;
    p_frm->Data[1u] = 2u;
    p_frm->Data[2u] = 3u;
    p_frm->Data[3u] = 4u;

    p_frm->DLC = 4u;                  /* Change the message length.                           */
}
```

RpdoOnRx()

RpdoOnRx() is called every time an asynchronous RPDO message is received.

This callback is of type RpdoOnRx(). RpdoOnRx() is optional and must be defined in your application. Listing - RpdoOnRx() Example in the *Communication Objects Callbacks Usage* page shows a basic implementation of the RpdoOnRx() callback.

Listing - RpdoOnRx() Example

```
void Ex_RpdoOnRx (CANOPEN_NODE_HANDLE   handle,
                  CANOPEN_IF_FRM       *p_frm)
{
   PP_UNUSED_PARAM(handle);

   for (CPU_INT32U loop = 0; loop < p_frm->DLC; loop ++) {
       printf("%d ", p_frm->Data[loop]);   /* Print every characters from the data payload.        */
   }
}
```

NMT

The node's NMT feature (Network Management) allows a master to control the internal node states by sending commands over the CAN bus. One unique callback allows the user application to be notified about a state transition.

Listing - CANopen State Enumeration in the *Communication Objects Callbacks Usage* page is used to define the different states taken by a node. CANOPEN_NODE_STATE shows a brief description of each enumeration state.

Listing - CANopen State Enumeration

```
typedef enum canopen_node_state {
   CANOPEN_INVALID = 0,
   CANOPEN_INIT,
   CANOPEN_PREOP,
   CANOPEN_OPERATIONAL,
   CANOPEN_STOP,
   CANOPEN_STATE_QTY
} CANOPEN_NODE_STATE;
```

StateOnChange()

This callback is used when the NMT state is changed. This callback is of type StateOnChange(). This callback is optional and must be defined in your application.

Heartbeat Consumer

Two callbacks are available to notify your application about the heartbeat producer activity.

HbcOnEvent()

This callback is used when a heartbeat consumer timer elapses and is triggered before receiving the corresponding heartbeat message from the heartbeat producer being monitored.

This callback is of type HbcOnEvent() . This callback is optional and must be defined in your application.

HbcOnChange()

This callback is used when a heartbeat consumer monitor detects a state change in a monitored node.

This callback is of type HbcOnChange(). This callback is optional and must be defined in your application.

HbcOnChange Callback

This callback function is called during the CANopen activities for a heartbeat consumer detected NMT mode change. The function is intended to allow the application to perform specific functions on NMT state change in received heartbeats.

Prototype

```
void (*HbcOnChange) (CANOPEN_NODE_HANDLE handle, CPU_INT08U node_id, CANOPEN_NODE_STATE state);
```

Arguments

handle

Handle to CANopen node object.

node_id

Node ID of the consumed heartbeat message.

state

Node state.

Return Values

None.

Notes / Warnings

None.

HbcOnEvent Callback

This callback function is called during the CANopen activities when a heartbeat consumer detects missed deadline of the monitored node. The function is intended to allow the application to perform specific actions on missing heartbeat events.

Prototype

```
void (*HbcOnEvent) (CANOPEN_NODE_HANDLE handle, CPU_INT08U node_id);
```

Arguments

handle

Handle to CANopen node object.

node_id

Node ID of the consumed heartbeat message.

Return Values

None.

Notes / Warnings

None.

ParamOnDflt Callback

This callback is used when the standard object "Restore default parameters" at index 0x1011 is written. The function is intended to restore the default parameters values from the non-volatile memory to the object dictionary.

Prototype

```
CPU_BOOLEAN (*ParamOnDflt) (CANOPEN_NODE_HANDLE handle, CANOPEN_PARAM *p_pg);
```

Arguments

handle

Handle to CANopen node object.

p_pg

Pointer to a parameter group information.

Return Values

DEF_YES, if load was successful.

DEF_NO, if load encountered an error.

Notes / Warnings

None.

ParamOnLoad Callback

The callback is called during reset and power-up events, including the NMT reset node and reset communication request. The function is intended to access the non-volatile memory and setting the parameter variables to the stored values.

Prototype

```
CPU_BOOLEAN (*ParamOnLoad) (CANOPEN_NODE_HANDLE handle, CANOPEN_PARAM *p_pg);
```

Arguments

handle

Handle to CANopen node object.

p_pg

Pointer to a parameter group information.

Return Values

DEF_YES, if load was successful.

DEF_NO, if load encountered an error.

Notes / Warnings

None.

ParamOnSave Callback

This callback is used when the standard object "Store" at index 0x1010 is written. The function is intended to save the parameters values to the non-volatile memory.

Prototype

```
CPU_BOOLEAN (*ParamOnSave) (CANOPEN_NODE_HANDLE handle, CANOPEN_PARAM *p_pg);
```

Arguments

handle

Handle to CANopen node object.

p_pg

Pointer to a parameter group information.

Return Values

DEF_YES, if load was successful.

DEF_NO, if load encountered an error.

Notes / Warnings

None.

RpdoOnRx Callback

This callback function is called after the reception of a RPDO CAN frame. The function is intended to allow a filter mechanism or modifications on the RPDO before distribution of the CAN frame into the CANopen stack.

Prototype

CPU_INT16S (*RpdoOnRx) (CANOPEN_NODE_HANDLE handle, CANOPEN_IF_FRM *p_frm);

Arguments

handle

Handle to CANopen node object.

p_frm

Pointer to the received frame buffer.

Return Values

1, if message was consumed by the callback.

0, if message was not consumed by the callback.

Notes / Warnings

None.

StateOnChange Callback

This callback function is called during the CANopen activities for a specific NMT state change. The function is intended to allow the application to perform specific actions on NMT state change events from the CANopen network (e.g., Reset, Stop, etc.).

Prototype

void (*StateOnChange) (CANOPEN_NODE_HANDLE handle, CANOPEN_NODE_STATE state);

Arguments

handle

Handle to CANopen node object.

state

Node state.

Return Values

None.

Notes / Warnings

None.

TpdoOnTx Callback

This callback function is called just before transmission of the built transmit PDO CAN frame. The function is intended to allow a 'last minute' customer specific modification on the TPDO data payload.

Prototype

```
void (*TpdoOnTx) (CANOPEN_NODE_HANDLE handle, CANOPEN_IF_FRM *p_frm);
```

Arguments

handle

Handle to CANopen node object.

p_frm

Pointer to the transmitted frame buffer.

Return Values

None.

Notes / Warnings

None.

Accessing Object Dictionary

This section shows how the object dictionary works, how it is structured, and how to access it from your application.

- Object Dictionary
  - CANopen Object
    - Object Key
    - Object Type
    - Object Data Payload
- Object Dictionary Functions
  - CANopen DictByteRd()
  - CANopen DictWordRd()
  - CANopen DictLongRd()
  - CANopen DictByteWr()
  - CANopen DictWordWr()
  - CANopen DictLongWr()
  - CANopen DictBufRd()
  - CANopen DictBufWr()

Object Dictionary

CANopen Object

The object dictionary is used to configure and communicate with a given CANopen node. An object dictionary is linked with an EDS (electronic data sheet) file, which is read by the CANopen master to discover the object dictionary entries for a given node.

The object dictionary consists of an array of CANOPEN_OBJ . Each entry of this array has:

- A key to encode index, sub-index, and information about the data
- An optional type that contains a callback structure related to the data

- A data payload that contains the direct object value, or an address to a memory region that holds the object's value, which can be simple or complex

Listing - Object Dictionary Example in the *Accessing Object Dictionary* page shows a basic example of an object dictionary.

> Defining an object dictionary, as presented in Listing - Object Dictionary Example in the *Accessing Object Dictionary* page, can be challenging. A tool called "CANopen Configurator" can be used to generate the node configuration structure and its associated object dictionary. Refer to Defining Node Specifications for more details about the tool "CANopen Configurator".

Listing - Object Dictionary Example

```
const CANOPEN_OBJ AppObjDir[] = {
  /*-----------------------------------------------------------------------------------*/
  /* [1000:xx] - Device Type */
  { CANOPEN_KEY(0x1000,0x0, CANOPEN_OBJ_UNSIGNED32|CANOPEN_OBJ___R_), 0,
    (CPU_INT32U)&App_ParamGrp.AppVar_1000_0 },
  /*-----------------------------------------------------------------------------------*/
  /* [1001:xx] - Error register */
  { CANOPEN_KEY(0x1001,0x0, CANOPEN_OBJ_UNSIGNED8|CANOPEN_OBJ___R_), 0,
    (CPU_INT32U)&App_ParamGrp.AppVar_1001_0 },
  /*-----------------------------------------------------------------------------------*/
  /* [1005:xx] - SYNC */
  { CANOPEN_KEY(0x1005,0x0, CANOPEN_OBJ_UNSIGNED32|CANOPEN_OBJ___RW), 0,
    (CPU_INT32U)&App_ParamGrp.AppVar_1005_0 },
  /*-----------------------------------------------------------------------------------*/
  /* [1010:xx] - Store parameters */
  { CANOPEN_KEY(0x1010,0x0, CANOPEN_OBJ_UNSIGNED8|CANOPEN_OBJ___R_), 0,
    (CPU_INT32U)&App_ParamGrp.AppVar_1010_0 },
  { CANOPEN_KEY(0x1010,0x1, CANOPEN_OBJ_UNSIGNED32|CANOPEN_OBJ___RW), CANOPEN_OBJ_TYPE_PARAM,
    (CPU_INT32U)&AppParamGrp },
  /*-----------------------------------------------------------------------------------*/
  /* [1011:xx] - Restore parameters */
  { CANOPEN_KEY(0x1011,0x0, CANOPEN_OBJ_UNSIGNED8|CANOPEN_OBJ___R_), 0,
    (CPU_INT32U)&App_ParamGrp.AppVar_1011_0 },
  { CANOPEN_KEY(0x1011,0x1, CANOPEN_OBJ_UNSIGNED32|CANOPEN_OBJ___RW), CANOPEN_OBJ_TYPE_PARAM,
    (CPU_INT32U)&AppParamGrp },
  /*-----------------------------------------------------------------------------------*/
  /* [1014:xx] - EMCY */
  { CANOPEN_KEY(0x1014,0x0, CANOPEN_OBJ_UNSIGNED32|CANOPEN_OBJ_N_RW), 0,
    (CPU_INT32U)&App_ParamGrp.AppVar_1014_0 },
  /*-----------------------------------------------------------------------------------*/
  /* [1016:xx] - Consumer heartbeat time */
  { CANOPEN_KEY(0x1016,0x0, CANOPEN_OBJ_UNSIGNED8|CANOPEN_OBJ___R_), 0,
    (CPU_INT32U)&App_ParamGrp.AppVar_1016_0 },
  { CANOPEN_KEY(0x1016,0x1, CANOPEN_OBJ_UNSIGNED32|CANOPEN_OBJ___RW), CANOPEN_OBJ_TYPE_HB_CONS,
    (CPU_INT32U)&App_ParamGrp.AppVar_1016_1 },
  /*-----------------------------------------------------------------------------------*/
  /* [1017:xx] - Producer heartbeat time */
  { CANOPEN_KEY(0x1017,0x0, CANOPEN_OBJ_UNSIGNED16|CANOPEN_OBJ___RW), CANOPEN_OBJ_TYPE_HB_PROD,
    (CPU_INT32U)&App_ParamGrp.AppVar_1017_0 },
  /*-----------------------------------------------------------------------------------*/
  /* [1018:xx] - Identity Object */
  { CANOPEN_KEY(0x1018,0x0, CANOPEN_OBJ_UNSIGNED8|CANOPEN_OBJ___R_), 0,
    (CPU_INT32U)&App_ParamGrp.AppVar_1018_0 },
  { CANOPEN_KEY(0x1018,0x1, CANOPEN_OBJ_UNSIGNED32|CANOPEN_OBJ___R_), 0,
    (CPU_INT32U)&App_ParamGrp.AppVar_1018_1 },
  { CANOPEN_KEY(0x1018,0x2, CANOPEN_OBJ_UNSIGNED32|CANOPEN_OBJ___R_), 0,
    (CPU_INT32U)&App_ParamGrp.AppVar_1018_2 },
  { CANOPEN_KEY(0x1018,0x3, CANOPEN_OBJ_UNSIGNED32|CANOPEN_OBJ___R_), 0,
    (CPU_INT32U)&App_ParamGrp.AppVar_1018_3 },
  { CANOPEN_KEY(0x1018,0x4, CANOPEN_OBJ_UNSIGNED32|CANOPEN_OBJ___R_), 0,
    (CPU_INT32U)&App_ParamGrp.AppVar_1018_4 },
  /*-----------------------------------------------------------------------------------*/
  /* [1200:xx] - Server SDO parameter */
```

```
(CPU_INT32U)&App_ParamGrp.AppVar_1200_0 },{CANOPEN_KEY(0x1200,0x1, CANOPEN_OBJ_UNSIGNED32|CANOPEN_OBJ__N_RW),
CANOPEN_OBJ_TYPE_SDO_ID,(CPU_INT32U)&App_ParamGrp.AppVar_1200_1 },{CANOPEN_KEY(0x1200,0x2,
CANOPEN_OBJ_UNSIGNED32|CANOPEN_OBJ__N_RW), CANOPEN_OBJ_TYPE_SDO_ID,(CPU_INT32U)&App_ParamGrp.AppVar_1200_2 },/*----------------
---------------------------------------------------------------------------*//* [1201:xx] - Server SDO parameter */{CANOPEN_KEY(0x1201,0x0,
CANOPEN_OBJ_UNSIGNED8|CANOPEN_OBJ___RW),0,(CPU_INT32U)&App_ParamGrp.AppVar_1200_0 },{CANOPEN_KEY(0x1201,0x1,
CANOPEN_OBJ_UNSIGNED32|CANOPEN_OBJ__N_R_), CANOPEN_OBJ_TYPE_SDO_ID,(CPU_INT32U)&App_ParamGrp.AppVar_1200_1 },
{CANOPEN_KEY(0x1201,0x2, CANOPEN_OBJ_UNSIGNED32|CANOPEN_OBJ__N_R_), CANOPEN_OBJ_TYPE_SDO_ID,
(CPU_INT32U)&App_ParamGrp.AppVar_1200_2 },
    CANOPEN_OBJ_DICT_ENDMARK
};
```

Object Key

A CANopen object key is composed of a 16-bit index, an 8-bit sub-index, and an 8-bit encoded flag. The key is generally encoded using the macro CANOPEN_KEY(idx, sub, flags) in canopen_obj.h. Table - CANopen Object Key (Flag) in the *Accessing Object Dictionary* page shows the object key datatype and access type that can be used when declaring an object dictionary. Datatype and access type will be OR'ed to form the complete flag field of the object key.

Table - CANopen Object Key (Flag)

Flag	Description
Datatype	
CANOPEN_OBJ_UNSIGNED8	CANopen Datatype: UNSIGNED8
CANOPEN_OBJ_UNSIGNED16	CANopen Datatype: UNSIGNED16
CANOPEN_OBJ_UNSIGNED32	CANopen Datatype: UNSIGNED32
CANOPEN_OBJ_UNSIGNED64	CANopen Datatype: UNSIGNED64
CANOPEN_OBJ_SIGNED8	CANopen Datatype: SIGNED8
CANOPEN_OBJ_SIGNED16	CANopen Datatype: SIGNED16
CANOPEN_OBJ_SIGNED32	CANopen Datatype: SIGNED32
CANOPEN_OBJ_FLOAT	CANopen Datatype: FLOAT
CANOPEN_OBJ_DOMAIN	CANopen Datatype: DOMAIN
CANOPEN_OBJ_STRING	CANopen Datatype: STRING
Access type	
CANOPEN*OBJ___R*	Read only access
CANOPEN_OBJ_W	Write only access
CANOPEN_OBJ___RW	Read/Write access
CANOPEN_OBJ_**P**	PDO Mappable
CANOPEN_OBJ_*_PR*	Read only access + PDO Mappable
CANOPEN_OBJ__P_W	Write only access + PDO Mappable
CANOPEN_OBJ__PRW	Read/Write access + PDO Mappable
CANOPEN_OBJ**N_**	Consider Node-ID in COB-ID
CANOPEN*OBJ__N_R*	Read only access + Node-ID
CANOPEN_OBJ**N**W	Write only access + Node-ID
CANOPEN_OBJ__N_RW	Read/Write access + Node-ID
CANOPEN*OBJ__NPR*	Read only access + PDO Mappable + Node-ID
CANOPEN_OBJ__NP_W	Write only access + PDO Mappable+ Node-ID
CANOPEN_OBJ__NPRW	Read/Write access + PDO Mappable+ Node-ID
CANOPEN_OBJ_D___	Direct access (Data field contains the object direct value)

Flag	Description
CANOPEN*OBJ_D_R*	Direct access + Read only access
CANOPEN_OBJ_D__W	Direct access + Write only access
CANOPEN_OBJ_D__RW	Direct access + Read/Write access
CANOPEN*OBJ_DN_R*	Direct access + Node-ID + Read only access
CANOPEN_OBJ_DN__W	Direct access + Node-ID + Write only access
CANOPEN_OBJ_DN_RW	Direct access + Node-ID + Read/Write access

Object Type

A CANopen type can be used to make a specialized action when the object dictionary is accessed at a specific index/sub-index. Table - CANopen Object Type in the *Accessing Object Dictionary* page shows the available types for different communication objects.

Table - CANopen Object Type

Type	Description
CANOPEN_OBJ_TYPE_EMCY	CANopen object type EMCY history.
CANOPEN_OBJ_TYPE_STR	CANopen object type string.
CANOPEN_OBJ_TYPE_DOMAIN	CANopen object type domain.
CANOPEN_OBJ_TYPE_PDO_NBRS	CANopen object type PDO map number.
CANOPEN_OBJ_TYPE_PDO_MAP	CANopen object type PDO mapping.
CANOPEN_OBJ_TYPE_PDO_ID	CANopen object type dynamic PDO identifier.
CANOPEN_OBJ_TYPE_PDO_TYPE	CANopen object type dynamic PDO transmission type.
CANOPEN_OBJ_TYPE_SDO_ID	CANopen object type dynamic SDO identifier.
CANOPEN_OBJ_TYPE_PARAM	CANopen object type parameter.
CANOPEN_OBJ_TYPE_TPDO_ASYNC	CANopen object type asynchronous TPDO object.
CANOPEN_OBJ_TYPE_TPDO_SYNC	CANopen object type synchronous TPDO object.
CANOPEN_OBJ_TYPE_TPDO_EVENT	CANopen object type TPDO event timer.
CANOPEN_OBJ_TYPE_HB_CONS	CANopen object type heartbeat consumer.
CANOPEN_OBJ_TYPE_HB_PROD	CANopen object type heartbeat producer.

Object Data Payload

The data payload can contain the object value, for direct access, or the address of a memory region, which holds the object value or a more complex object data structure. Listing - CANopen Object Data in the *Accessing Object Dictionary* page shows an example of how the data is structured inside the object dictionary.

Listing - CANopen Object Data

```
CPU_INT32U            Ex_Var_2008_0 = 0xAABBCCDD;
APP_CUSTOM_STRUCT_TYPE  Ex_Var_1016_0;
...

const CANOPEN_OBJ AppObjDir[] = {
  { CANOPEN_KEY(0x2008,0x0, CANOPEN_OBJ_DOMAIN|CANOPEN_OBJ___RW), 0, (CPU_INT32U)&Ex_Var_2008_0 },     (1)
  { CANOPEN_KEY(0x2009,0x0, CANOPEN_OBJ_STRING|CANOPEN_OBJ_D_R_), 0, 1234u },                    (2)
  { CANOPEN_KEY(0x1016,0x0, CANOPEN_OBJ_UNSIGNED32|CANOPEN_OBJ___R_), CANOPEN_OBJ_TYPE_HB_CONS,        (3)
    (CPU_INT32U)&Ex_Var_1016_0 },
    CANOPEN_OBJ_DICT_ENDMARK
};
```

(1) The data payload of this object is accessed by a pointer pointing to the variable AppVar. The field .Data of structure CANOPEN_OBJ is used as a pointer in that case.

(2) The data payload of this object is accessed directly via the field .Data of structure CANOPEN_OBJ .

(3) The object data payload is a more complex value that requires interpretation. Thus the object type CANOPEN_OBJ_TYPE_HB_CONS is specified to provide specialized callbacks to read/write the object value. The field .Data of structure CANOPEN_OBJ is not used in that case. The field .TypePtr will be used instead.

Object Dictionary Functions

This section shows how to read data from, and write data to, the object dictionary.

CANopen_DictByteRd()

The function CANopen_DictByteRd() reads an 8-bit value from the given object dictionary. The object entry is addressed with the given key, and the value to read will be written to the given destination pointer. Listing - CANopen DictByteRd() Usage in the *Accessing Object Dictionary* page gives an example of how to use it.

Listing - CANopen_DictByteRd() Usage

```
CANOPEN_NODE_HANDLE   handle;
CPU_INT08U           *p_val;
RTOS_ERR              local_err;

CANopen_DictByteRd(handle,
                   CANOPEN_DEV(0x1234, 0),
                   p_val,
                   &local_err);
if (RTOS_ERR_CODE_GET(local_err) != RTOS_ERR_NONE) {        /* see, if an error is detected */
    ...
}
```

CANopen_DictWordRd()

The function CANopen_DictWordRd() reads a 16-bit value from the given object dictionary. The object entry is addressed with the given key and the value to read will be written to the given destination pointer. Listing - CANopen DictWordRd() Usage in the *Accessing Object Dictionary* page gives an example of how to use it.

Listing - CANopen_DictWordRd() Usage

```
CANOPEN_NODE_HANDLE   handle;
CPU_INT16U           *p_val;
RTOS_ERR              local_err;

CANopen_DictWordRd(handle,
                   CANOPEN_DEV(0x1234, 0),
                   p_val,
                   &local_err);
if (RTOS_ERR_CODE_GET(local_err) != RTOS_ERR_NONE) {        /* see, if an error is detected */
    ...
}
```

CANopen_DictLongRd()

The function CANopen_DictLongRd() reads a 32-bit value from the given object dictionary. The object entry is addressed with the given key and the value to read will be written to the given destination pointer. Listing - CANopen DictLongRd() Usage in the *Accessing Object Dictionary* page gives an example of how to use it.

Listing - CANopen_DictLongRd() Usage

```
CANOPEN_NODE_HANDLE   handle;
CPU_INT32U           *p_val;
RTOS_ERR              local_err;

CANopen_DictLongRd(handle,
                   CANOPEN_DEV(0x1234, 0),
                   p_val,
                   &local_err);
if (RTOS_ERR_CODE_GET(local_err) != RTOS_ERR_NONE) {        /* see, if an error is detected */
    ...
}
```

CANopen_DictByteWr()

The function CANopen_DictByteWr() writes an 8-bit value to the given object directory. The object entry is addressed with the given key. Listing - CANopen DictByteWr() Usage in the *Accessing Object Dictionary* page gives an example of how to use it.

Listing - CANopen_DictByteWr() Usage

```
CANOPEN_NODE_HANDLE  handle;
CPU_INT08U           val;
RTOS_ERR             local_err;

val = 0xFF;
CANopen_DictByteWr(handle,
                   CANOPEN_DEV(0x1234, 0),
                   val,
                   &local_err);
if (RTOS_ERR_CODE_GET(local_err) != RTOS_ERR_NONE) {        /* see, if an error is detected */
    ...
}
```

CANopen_DictWordWr()

The function CANopen_DictWordWr() writes a 16-bit value to the given object dictionary. The object entry is addressed with the given key. Listing - CANopen DictWordWr() Usage in the *Accessing Object Dictionary* page gives an example of how to use it.

Listing - CANopen_DictWordWr() Usage

```
CANOPEN_NODE_HANDLE  handle;
CPU_INT16U           val;
RTOS_ERR             local_err;

val = 0xFFFF;
CANopen_DictWordWr(handle,
                   CANOPEN_DEV(0x1234, 0),
                   val,
                   &local_err);
if (RTOS_ERR_CODE_GET(local_err) != RTOS_ERR_NONE) {        /* see, if an error is detected */
    ...
}
```

CANopen_DictLongWr()

The function CANopen_DictLongWr() writes a 32-bit value to the given object dictionary. The object entry is addressed with the given key. Listing - CANopen DictLongWr() Usage in the *Accessing Object Dictionary* page gives an example of how to use it.

Listing - CANopen_DictLongWr() Usage

```
CANOPEN_NODE_HANDLE  handle;
CPU_INT32U           val
RTOS_ERR             local_err;

val = 0xFFFFFFFF;
CANopen_DictLongWr(handle,
                   CANOPEN_DEV(0x1234, 0),
                   val,
                   &local_err);
if (RTOS_ERR_CODE_GET(local_err) != RTOS_ERR_NONE) {       /* see, if an error is detected */
    ...
}
```

CANopen_DictBufRd()

The function CANopen_DictBufRd() reads a buffer byte stream from the given object dictionary. The object entry is addressed with the given key and the bytes to read will be written to the given destination buffer of the given length. Listing - CANopen DictBufRd() Usage in the *Accessing Object Dictionary* page gives an example of how to use it.

Listing - CANopen_DictBufRd() Usage

```
CANOPEN_NODE_HANDLE  handle;
CPU_INT08U           val_array[8u];
RTOS_ERR             local_err;

CANopen_DictLongWr(handle,
                   CANOPEN_DEV(0x1234, 0),
                   &val_array,
                   8u,
                   &local_err);
if (RTOS_ERR_CODE_GET(local_err) != RTOS_ERR_NONE) {       /* see, if an error is detected */
    ...
}
```

CANopen_DictBufWr()

The function CANopen_DictBufWr() writes a buffer byte stream to the given object dictionary. The object entry is addressed with the given key and the bytes to write will be read from to the given source buffer of the given length. Listing - CANopen DictBufWr() Usage in the *Accessing Object Dictionary* page gives an example of how to use it.

Listing - CANopen_DictBufWr() Usage

```
CANOPEN_NODE_HANDLE  handle;
CPU_INT08U           val_array[8u];
RTOS_ERR             local_err;

val_array[0u] = 0x01;
val_array[1u] = 0x02;
val_array[2u] = 0x03;
val_array[3u] = 0x04;
val_array[4u] = 0x05;
val_array[5u] = 0x06;
val_array[6u] = 0x07;
val_array[7u] = 0x08;

CANopen_DictBufWr(handle,
                  CANOPEN_DEV(0x1234, 0),
                  &val_array,
                  8u,
                  &local_err);
if (RTOS_ERR_CODE_GET(local_err) != RTOS_ERR_NONE) {        /* see, if an error is detected */
    ...
}
```

Using Communication Objects

The Micrium OS CANopen stack allows you to set up a CANopen node device that acts as a slave on the CAN network. Once configured, a CANopen node device is almost completely autonomous. A master node on the network can remotely configure and communicate with the slave node, and your application may have little interaction with the configured node during the master-slave node communication. But depending on the type of ongoing communication, your application can use a few API interfaces. The following sections describe these API interfaces for the Emergency object, Network Management, and the Heartbeat consumer.

- Emergency Object (EMCY)
  - Emergency Error Set
  - Emergency Error Clear
  - Emergency Error Get
  - Emergency Error Reset
  - Emergency Error Count
  - Emergency Error History Reset
- Network Management (NMT)
  - Node Reset
  - Node State Set
  - Node State Get
- Heartbeat Consumer
  - Events Get
  - Last state Get

Emergency Object (EMCY)

The CANopen stack provides service API functions for managing emergency events within the application. EMCY transmission events, the error register, and EMCY history management within the object directory are all handled within the CANopen stack. For managing emergency errors, the service function group CANopen_EmcyXxxx() is provided. The EMCY API is optional and can be disabled by setting the configuration CANOPEN_EMCY_MAX_ERR_QTY to 0.

An EMCY message uses the 8 bytes available in the payload of the CAN frame. The EMCY message carries information that can help determine which error has occurred on the node application side. The message has the following format.

Table - Emergency Message Format

Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8
Emergency error code	Error register	Manufacturer-specific error code					

Emergency Error Set

The function CANopen_EmcySet() sets the EMCY error status and updates the object dictionary. The function does not send an EMCY message directly over the CAN Bus; the internal CANopen Service task performs the EMCY transfer.

On the first occurrence of a given error code, an EMCY message is sent over the CAN Bus. But if the *same* error is set again by the application, the Service task does not send the error message. The task will send a new EMCY message only if the error code is different from the previous one.

The given manufacturer-specific fields are optional, e.g., the pointer may be DEF_NULL to set all manufacturer specific values to 0. If you plan to define manufacturer-specific error codes, you must enable the configuration constant CANOPEN_EMCY_EMCY_MAN_EN. Listing - CANopen EmcySet() Usage in the *Using Communication Objects* page shows how to configure the structure CANOPEN_EMCY_USR to specify your own error codes.

Listing - CANopen_EmcySet() Usage

```
CPU_INT08U      err_code_ix;
CANOPEN_EMCY_USR  usr;
RTOS_ERR        err;

err_code_ix  = 2u;                  (1)
usr.Hist     = 0x2233;              (2)
                              (3)
usr.Emcy[0u] = 0xAA;
usr.Emcy[1u] = 0xBB;
usr.Emcy[2u] = 0xCC;
usr.Emcy[3u] = 0xDD;
usr.Emcy[4u] = 0xEE;

CANopen_EmcySet(&node1.Emcy,
                 err_code_ix,
                 &usr,
                 &err);
```

(1) Specify the index of the user emergency error code to set. The index is linked to the table defined as part of the node specifications. See the pointer EmcyCodePtr in Table - Node Specification Description in the *Defining Node Specifications* page for more details about the user emergency error code table. This error code is a standard emergency error code, and will be sent, if it is necessary, as part of the EMCY message in the field 'Emergency error code' (cf. Table - Emergency Message Format in the *Using Communication Objects* page).The index should not be greater than the configuration CANOPEN_EMCY_MAX_ERR_QTY. Otherwise, the function CANopen_EmcySet() will return an error.

(2) Set an additional 16-bit error code, which is manufacturer-specific. This additional error code will be stored in the error history located in the object dictionary at index 0x1003 (which is a pre-defined error field). The consumer node may interrogate the producer node object dictionary at index 0x1003 to read the error history.

(3) Set a 5-byte field representing the manufacturer-specific error code, as shown in Table - Emergency Message Format in the *Using Communication Objects* page. This error code will be part of the emergency message along with the standard error code explained in note (1). Here, the specific user information data for this event is: 0xAABBCCDDEE.

Emergency Error Clear

The function CANopen_EmcyClr() clears the EMCY error status and updates the object dictionary as needed. If the EMCY error status is already set, a clear EMCY message is sent over the CAN Bus.

Emergency Error Get

The function CANopen_EmcyGet() gets the current EMCY error status.

Emergency Error Reset

The function CANopen_EmcyReset() clears all EMCY errors, mapped in error register at object index 0x1001.

Emergency Error Count

The function CANopen_EmcyCnt() returns the number of currently detected EMCY errors.

Emergency Error History Reset

The function CANopen_EmcyHistReset() clears the EMCY history in the object directory, pre-defined error field at object index 0x1003.

Network Management (NMT)

NMT allows the CANopen master on the network to initialize, start, monitor, reset or stop NMT slaves. The NMT master remotely controls the internal NMT slave states via specific messages. The Micrium OS CANopen stack offers a few functions to your application to interact with the NMT communication object.

Node Reset

Resetting a node is typically performed by the CANopen master via NMT protocol messages. In general, this is done by the CANopen stack without any action required by your application. However, your application may need to reset the NMT slave. In that case, the CANopen stack provides a service function, CANopen_NmtReset(), to reset the node.

There are two different kinds of resets possible:

- Reset Communication and Application (also called "Reset Node"): the parameters of the manufacturer-specific profile area (object dictionary indexes 0x2000 to 0x5FFF) and of the standardized device profile area (object dictionary indexes 0x6000 to 0x9FFF) are set to their power-on values.
- Reset Communication: the parameters of the communication profile area (object dictionary indexes 0x1000 to 0x1FFF) are set to their power-on values.

Node State Set

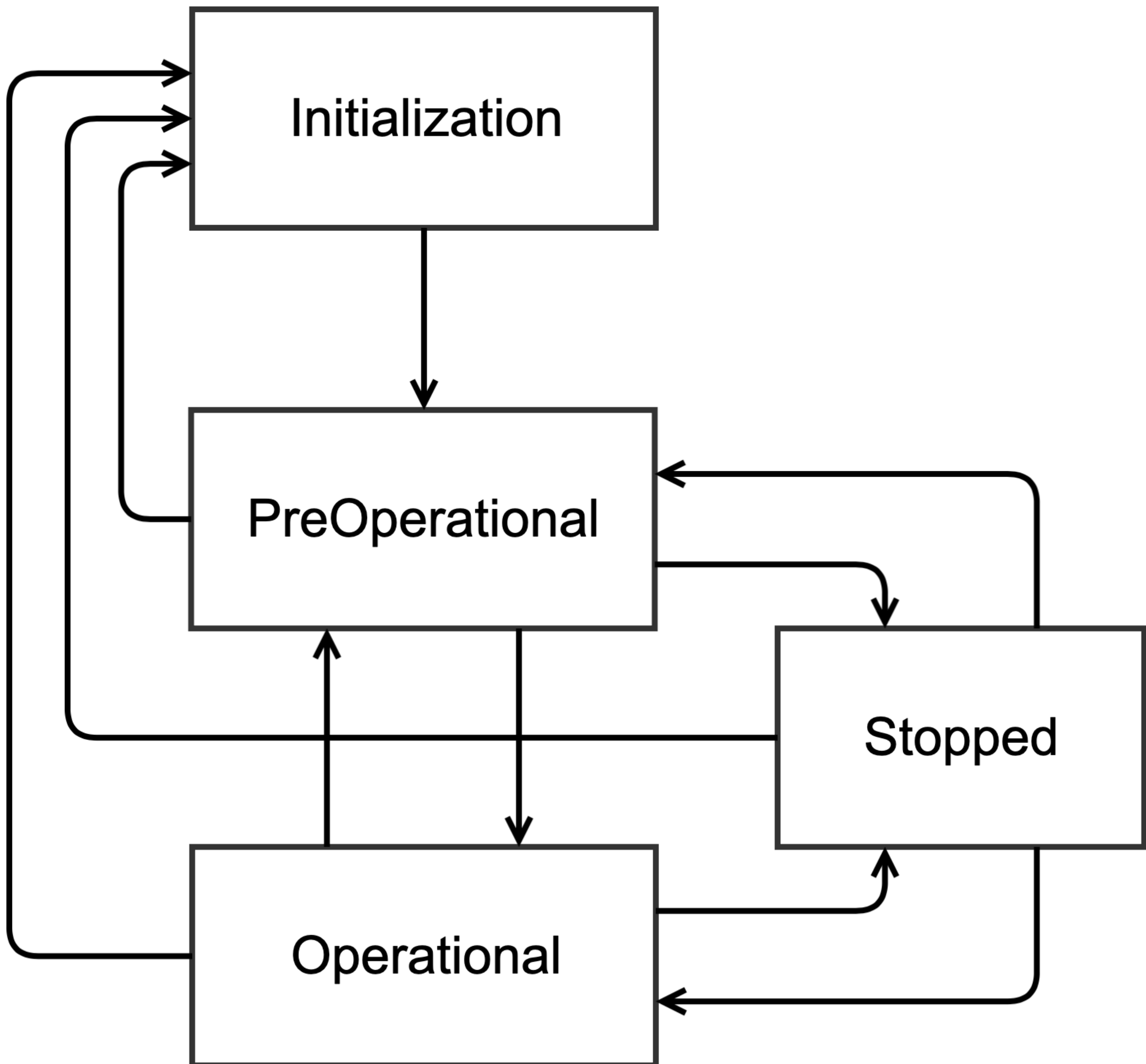
The function CANopen_NmtStateSet() sets the requested CANopen NMT state machine mode:

NMT Mode	Description
CANOPEN_INIT	Node transitions to NMT state Initialization.
CANOPEN_PREOP	Node transitions to NMT state Pre-operational.
CANOPEN_OPERATIONAL	Node transitions to NMT state Operational.
CANOPEN_STOP	Node transitions to NMT state Stopped.

Table - NMT Slave States

Figure - NMT State Diagram of a CANopen Device in the *Using Communication Objects* page shows a typical NMT slave state diagram, and where the function CANopen_NmtStateSet() allows you to transition the state.

Figure - NMT State Diagram of a CANopen Device



Node State Get

The function CANopen_NmtStateGet() returns the current CANopen NMT state.

NMT Mode	Description
CANOPEN_INIT	Node is in NMT state Initialization.
CANOPEN_PREOP	Node is in NMT state Pre-operational.
CANOPEN_OPERATIONAL	Node is in NMT state Operational.
CANOPEN_STOP	Node is in NMT state Stopped.

Table - NMT Slave States

Heartbeat Consumer

If a node enables the heartbeat consumer, this node will be able to monitor heartbeat producer messages sent by other nodes on the network. Multiple producer nodes can be linked together to form a list of monitored nodes. Listing - CANopen

Heartbeat Structure in the *Using Communication Objects* page shows the heartbeat structure used to monitor a producer node.

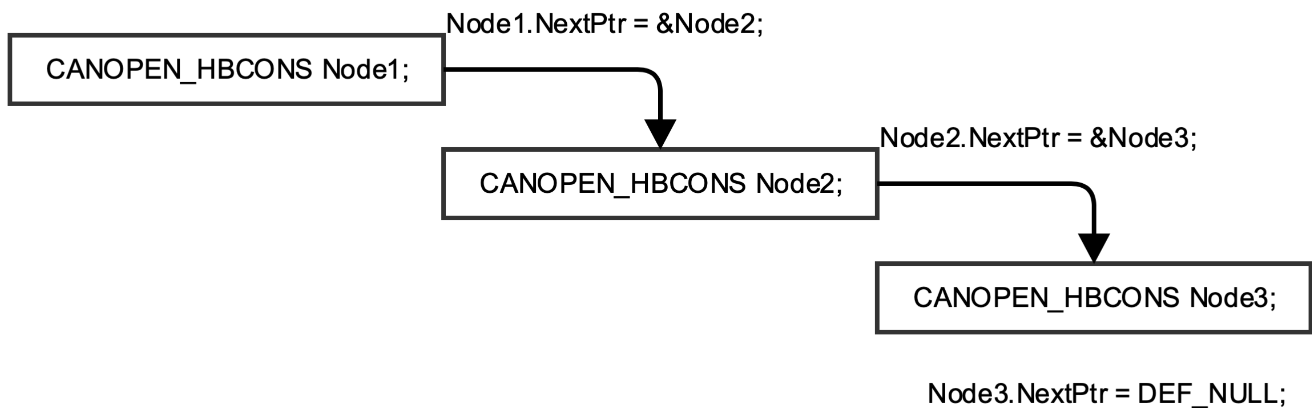
Figure - Monitored Nodes List in the *Using Communication Objects* page shows how to link these structures together to monitor more than one node at the same time.

A few functions, described below, are available to get pieces of information about a given producer node.

Listing - CANopen Heartbeat Structure

```
struct canopen_hbcons {
    CANOPEN_HBCONS      *NextPtr;          /**< Link to next consumer in active chain  */
    CANOPEN_NODE_STATE  State;             /**< Received Node-State                    */
    CPU_INT16S          TmrId;             /**< Timer Identifier                       */
    CPU_INT16U          TimeMs;            /**< Time   (Bit00-15 when read object)     */
    CPU_INT08U          NodeId;            /**< NodeId (Bit16-23 when read object)     */
    CPU_INT08U          MissedEventCnt;    /**< Event Counter                          */
};
```

Figure - Monitored Nodes List



Events Get

The function CANopen_NmtHbConsEventsGet() returns the number of missed heartbeat for a given node. That is the number of times the producer heartbeat message has not been received by the consumer node within the heartbeat consumer time.

Last State Get

The function CANopen_NmtHbConsLastStateGet() returns the last NMT state (Stopped, Pre-operational, Operational, Initialization) for a given node.

CAN Bus Hardware Porting Guide

CAN Bus Hardware Porting Guide

In order to work properly, Micrium OS CAN Bus requires a specific hardware driver for any CAN Bus controller IP. Moreover, each CAN Bus controller must have a board support package (BSP). The BSP has two purposes:

- Initializing and configuring resources needed by the CAN Bus controller that are provided by an external module
- Providing hardware information to the CAN Bus driver

Micrium provides examples of BSPs for some popular platforms. If one is available for your platform, we recommend that you use it as a starting point. However, if no example BSP is available for your platform, the information in this section will help you understand how to correctly port the CAN Bus module.

Note that each CAN Bus controller (that you are planning to use) will require a BSP, and all the steps described in this section must be performed for each of them.

- CAN Bus BSP Functions Guide
- CAN Hardware Information
- CAN Bus Controller Registration to the Platform Manager

CAN Bus BSP Functions Guide

A Board Support Package contains a set of functions that support your specific hardware platform on Micrium OS. These functions are called by the CAN Bus driver to perform hardware configuration, or initialization of IO pins, interrupts, and so on. It is your responsibility to either create these functions from scratch or to modify example code to fit your needs and the specifics of your hardware.

Each of these functions is described below.

- Open
- Close
- Timer Configuration
- Interrupt Control
- ISR Handling
- Timer ISR Handling

Open

The Open function is used to set the CAN Bus clock, and set the IO pins and route them to the desired pins. Listing - Open Function Signature in the *CAN Bus BSP Functions Guide* page shows the signature of the function.

Listing - Open Function Signature

```
void BSP_CAN_Open (void);
```

This function is called each time the CAN controller operations are started by the CAN Bus module.

Close

The Close function is used to disable the CAN Bus clock, unset IO route, and disable CAN Bus interrupt. Listing - Close Function Signature in the *CAN Bus BSP Functions Guide* page shows the signature of the function.

Listing - Close Function Signature

```
void BSP_CAN_Close (void);
```

This function is called each time the CAN controller operations are stopped by the CAN Bus module.

Timer Configuration

The Timer Config function is used to configure and enable a timer that can be used by a CAN high-level protocol. Listing - Timer Configuration Function Signature in the *CAN Bus BSP Functions Guide* page shows the signature of the function.

Listing - Timer Configuration Function Signature

```
void  BSP_CAN_TmrCfg  (CPU_INT32U  tmr_period)
```

The period of this timer is set to the value of tmr_period in microseconds. The timer configuration typically encompasses:

- Initializing the input clock required by the timer peripheral
- Configuring the interrupt service routine associated with the timer
- Enabling the interrupt line for the timer
- Configuring the timer period to periodically generate an interrupt

Refer to the section Timer ISR Handling for more details about the timer ISR.

Interrupt Control

The Interrupt Control function is used to register the interrupt service routine associated with the CAN controller and enable the CAN Bus interrupt. Listing - Interrupt Control Function Signature in the *CAN Bus BSP Functions Guide* page shows the signature of the function.

Listing - Interrupt Control Function Signature

```
void  BSP_CAN_IntCtrl (CAN_BUS_HANDLE  bus_handle);
```

In addition, the function should store a bus handle associated to the CAN controller. The bus handle is provided by the parameter bus_handle. It will be needed inside the controller's interrupt service routine to indicate to the CAN Bus module which CAN controller the interrupt is associated to (see ISR Handling for more details).

ISR Handling

Each CAN Bus controller driver has an ISR handler function that must be called each time a CAN Bus interrupt is triggered. However, for most platforms, it will be necessary to implement an intermediate ISR handler in the BSP. This BSP ISR will then call the driver's ISR handler. This is necessary because some interrupt controllers may require the interrupt status to be cleared each time it is triggered. It is also necessary if your interrupt controller does not support passing an argument to the interrupt vector, as the driver's ISR handler takes the bus_handle received in the IntCtrl () function of the BSP as an argument.

Listing - Example of BSP ISR Implementation in the *CAN Bus BSP Functions Guide* page shows an example of an ISR handler implemented in the BSP if using a Silicon Labs chip (that follows the CMSIS naming standard). Note that the global variable BSP_CAN_BusHandle will have been set in the function IntCtrl () .

Listing - Example of BSP ISR Implementation

```
void CAN0_IRQHandler (void)
{
  OSIntEnter();
  CanBus_ISRHandler(BSP_CAN0_BusHandle);
  OSIntExit();
}
```

Timer ISR Handling

The timer ISR has been configured in the function TmrCfg() . This timer ISR represents the BSP Timer ISR which should call the CAN driver's timer ISR handler. This is necessary, as some interrupt controllers may require the interrupt status to be

cleared each time it is triggered.

Listing - Example of BSP Timer ISR Implementation in the *CAN Bus BSP Functions Guide* page shows an example of a Timer ISR handler implemented in the BSP if using a Silicon Labs chip (that follows the CMSIS naming standard).

Listing - Example of BSP Timer ISR Implementation

```
void TIMER0_IRQHandler (void)
{
   CPU_INT32U timer_get;

   OSIntEnter();
   timer_get = TIMER_IntGet(TIMER0);
   CanBus_TmrISRHandler();
   TIMER_IntClear(TIMER0, timer_get);
   OSIntExit();
}
```

CAN Hardware Information

- Hardware Driver Information
- BSP API Structure
- Device Hardware Information

Hardware Driver Information

The CAN Bus driver requires information about the CAN Bus controller on your MCU, which you can provide using a structure of type CAN_CTRLR_HW_INFO. This information can be found in the manual for your MCU.

Table - CAN_CTRLR_HW_INFO Structure in the *CAN Hardware Information* page describes each configuration field available in this structure.

Table - CAN_CTRLR_HW_INFO Structure

Field	Description
.BaseAddr	Base address of the CAN Bus controller registers set.
.InfoExtPtr	Pointer to an extended hardware information structure. Some drivers may require extra information, so the format of this structure is specific to your driver. Most of the time this can be set to DEF_NULL.
.IF_Rx	Reception interface number. This interface can take the value of 0 or 1.
.IF_Tx	Transmission interface number. This interface can take the value of 0 or 1.

BSP API Structure

In order to provide a pointer to the BSP functions for the CAN Bus controller driver, you must create a structure of type CAN_CTRLR_BSP_API.

Table - CAN_CTRLR_BSP_API Structure in the *CAN Hardware Information* page describes each field available in this structure.

Table - CAN_CTRLR_BSP_API Structure

Field	Description
.Open	Pointer to the BSP open function.
.Close	Pointer to the BSP close function.
.IntCtrl	Pointer to the BSP interrupt control function.
.TmrCfg	Pointer to the BSP timer configure function.

Listing - Example of BSP API Structure in the *CAN Hardware Information* page shows an example of a BSP API structure.

Listing - Example of BSP API Structure

```
const  CAN_CTRLR_DRV_INFO  BSP_CAN0_BSP_DrvInfo = {
    .BSP_API_Ptr        = &CAN0_BSP_API,
    .HW_Info.BaseAddr   = (CPU_ADDR)CAN0,
    .HW_Info.InfoExtPtr = DEF_NULL,
    .HW_Info.IF_Rx      = 1u,
    .HW_Info.IF_Tx      = 0u
};
```

Device Hardware Information

The last step is to create the main device hardware information structure.

Table - CAN_CTRLR_DRV_INFO Structure in the *CAN Hardware Information* page describes each configuration field available in this structure.

Table - CAN_CTRLR_DRV_INFO Structure

Field	Description
.HW_Info	Structure explained above in Hardware Driver Information .
.BSP_API_Ptr	Pointer to the BSP API structure as described above in BSP API Structure .

CAN Bus Controller Registration to the Platform Manager

Once the hardware information structure for your CAN Bus controller is ready, it must be registered to the Platform Manager . This should typically be done from the BSP_OS_Init() function that is located in the file bsp_os.c.

There is a macro located in the file can_bus.h that you must call to register a CAN Bus controller. The macro is named CAN_CTRLR_REG().

Listing - Example of CAN Bus Controller Registration in the *CAN Bus Controller Registration to the Platform Manager* page shows an example of how to register a CAN Bus controller.

Listing - Example of CAN Bus Controller Registration

```
#include  <rtos_description.h>
#if defined(RTOS_MODULE_CAN_BUS_AVAIL)
#include  <rtos/can/include/can_bus.h>
#endif

#if defined(RTOS_MODULE_CAN_BUS_AVAIL)                    (1)
BSP_HW_INFO_EXT(const  CAN_CTRLR_DRV_INFO, BSP_CAN0_BSP_DrvInfo);
BSP_HW_INFO_EXT(const  CAN_CTRLR_DRV_INFO, BSP_CAN1_BSP_DrvInfo);
#endif

void  BSP_OS_Init (void)
{

    /* ... */

                                    /* ------- REGISTER CAN BUS CONTROLLERS ------ */
#if defined(RTOS_MODULE_CAN_BUS_AVAIL)
    CAN_CTRLR_REG("can0", &BSP_CAN0_BSP_DrvInfo);
    CAN_CTRLR_REG("can1", &BSP_CAN1_BSP_DrvInfo);
#endif
}
```

(1) Since the hardware information global variables are declared in another file, you must declare them as external in your bsp_os.c file. Always use the BSP_HW_INFO_EXT() macro.

SILICON LABS

Hardware Porting Guide

Hardware Porting Guide

Micrium OS includes board support packages for some key Silicon Labs evaluation boards. However, your specific evaluation board may not be supported. And of course, any custom board will not be supported. If your board is similar to one that is supported, we highly recommended that you start from the similar board's BSP and modify it as needed. Otherwise, you can use the template BSP files as a starting point. This section explains the different steps needed to port Micrium OS to your board from scratch.

Standard BSP Functions

Standard BSP Functions

Each BSP is expected to implement a function called BSP_OS_Init(). This function is called at a specific moment by the application during initialization.

The definitions of this functions is located in the file bsp/siliconlabs/generic/include/bsp_os.h.

Table - Standard BSP Functions in the *Hardware Porting Guide* page describes the function that must be implemented.

Table - Standard BSP Functions

Function	Function signature	Description	Called...
Peripheral Initialization	void BSP_OS_Init(void)	Initializes other peripherals and registers the controllers to the Platform Manager . Refer to Controller Registration to the Platform Manager for more information.	After the call to the function Common_Init().

CPU Module

Micrium OS CPU is the first module you should port since all other Micrium OS modules rely on it. To port CPU, you will need some simple board-specific adaptations. Refer to CPU Port Board Support Package for more information.

Other Micrium OS Modules

Other Micrium OS Modules

Once the Kernel is working properly, you can start porting the other Micrium OS modules to your hardware. If you have the Micrium OS File System, it is probably a good idea to start by porting this module, as other modules may have a dependency on it.

The other Micrium OS modules will require a BSP for each controller you plan on using. The porting procedure is similar for each kind of controller and consists of the steps described in the following sections.

Table - Hardware Controller Types in the *Hardware Porting Guide* page lists the different types of controllers that must be ported to Micrium OS, along with the modules that handle them, and a link to the documentation section that better describes the porting process.

Table - Hardware Controller Types

Hardware Controller type	Micrium OS Module	Dedicated Section(s) to refer
SPI	IO-Serial	SPI Hardware Porting Guide
Flash (NAND, NOR), SD Card	File System	File System Hardware Porting Guide
Ethernet, WiFi	Network	Network Core Hardware Porting Guide
USB	USB Host, USB Device	USB Device Hardware Porting Guide , USB Host Hardware Porting Guide
CAN Bus	CAN, CANopen	CAN Bus Hardware Porting Guide

Selecting Drivers

The first step is to select the appropriate driver for your controller from among the ones provided with Micrium OS. Refer to the appropriate section for more information:

- Page:File_System_Memory_Driver_Selection_Guide
- Page:Network_Driver_Selection_Guide

Implementing BSP Functions

You will need to implement a set of functions in the BSP for each controller. The typical purpose of these functions is configure everything that relies on other controllers, such as GPIO, interrupts, power, and clock supply.

The set of functions is specific for each module and controller type. Refer to the appropriate section for more information:

- Page:CAN_Bus_BSP_Functions_Guide
- Page:File_System_Memory_BSP_Functions_Guide
- Page:Network_BSP_Functions_Guide
- Page:SD_BSP_Functions_Guide
- Page:SPI_BSP_Functions_Guide
- Page:USB_Device_BSP_Functions_Guide
- Page:USB_Host_BSP_Functions_Guide

Creating Hardware Information Structure(s)

Each controller must be provided with a hardware information structure. This structure typically contains information such as the controller's register set base address, controller speed, a pointer to the BSP functions, a pointer to the controller driver, and so on.

The format of the hardware information structure is specific for each controller type. Refer to the appropriate section for more information:

- Page:CAN_Hardware_Information
- Page:File_System_Memory_Hardware_Information
- Page:Network_Hardware_Information
- Page:SD_Hardware_Information
- Page:SPI_Hardware_Information
- Page:USB_Device_Hardware_Information
- Page:USB_Host_HCD_Hardware_Information
- Page:USB_Host_PBHCD_Hardware_Information

Registering the Controller to the Platform Manager

Once all the above steps have been completed, you must register the controller to the Platform Manager . The Platform Manager is a database that contains information on all the hardware controllers that can be used by the various Micrium OS modules.

The registration should be done from the BSP_OS_Init() function using the available macros:

- Page:CAN_CTRLR_REG()
- Page:FS_NAND_HW_INFO_REG()
- Page:FS_NOR_QUAD_SPI_HW_INFO_REG()
- Page:FS_SD_CARD_HW_INFO_REG()
- Page:FS_SD_SPI_HW_INFO_REG()
- Page:IO_SERIAL_CTRLR_REG()
- Page:NET_CTRLR_ETHER_REG()
- Page:NET_CTRLR_WIFI_SPI_REG()
- Page:USB_CTRLR_HW_INFO_DEV_ONLY_REG()
- Page:USB_CTRLR_HW_INFO_HOST_COMPANION_REG()
- Page:USB_CTRLR_HW_INFO_HOST_ONLY_REG()
- Page:USB_CTRLR_HW_INFO_REG()

Refer to the appropriate section for more information:

- Page:CAN_Bus_Controller_Registration_to_the_Platform_Manager
- Page:File_System_Memory_Controller_Registration_to_the_Platform_Manager
- Page:Network_Controller_Registration_to_the_Platform_Manager
- Page:SD_Controller_Registration_to_the_Platform_Manager
- Page:SPI_Controller_Registration_to_the_Platform_Manager
- Page:USB_Device_Controller_Registration_to_the_Platform_Manager
- Page:USB_Host_Controller_Registration_to_the_Platform_Manager

Common API

Common API

- Common Core API
- Authentication API
- LIB API
- Logging API
- RTOS_ERR API
- Toolchain Abstraction API
- Utilities API
- Shell API
- RTOS_TASK_CFG

Common Core API

Common Core API

- Common_ConfigureLogging()
- Common_ConfigureMemSegLogging()
- Common_ConfigureMemSeg()
- Common_Init()

Common_ConfigureLogging()

Description

Configure the properties of the logging module.

Files

`common.h/common.c`

Prototype

```
void  Common_ConfigureLogging (COMMON_CFG_LOGGING  *p_cfg)
```

Arguments

`p_cfg`

Pointer to the structure containing the new logging parameters.

Returned Value

None.

Notes / Warnings

1. This function is optional. If it is called, it must be called before `Common_Init()`. If it is not called, default values will be used to initialize the module.

Common_ConfigureMemSegLogging()

Description

Configure the memory segment that will be used to allocate internal data needed by the logging module instead of the default memory segment.

Files

`common.h/common.c`

Prototype

```
void  Common_ConfigureMemSegLogging (MEM_SEG  *p_mem_seg)
```

Arguments

`p_mem_seg`

Pointer to the memory segment from which the internal data will be allocated. If `DEF_NULL` , the internal data will be allocated from the global Heap.

Returned Value

None.

Notes / Warnings

1. This function is optional. If it is called, it must be called before `Common_Init()` . If it is not called, default values will be used to initialize the module.

Common_ConfigureMemSeg()

Description

Configure the memory segment that will be used to allocate internal data needed by Common modules instead of the default memory segment.

Files

`common.h/common.c`

Prototype

```
void Common_ConfigureMemSeg (MEM_SEG *p_mem_seg)
```

Arguments

`p_mem_seg`

Pointer to the memory segment from which the internal data will be allocated. If `DEF_NULL` , the internal data will be allocated from the global Heap.

Returned Value

None.

Notes / Warnings

1. This function is optional. If it is called, it must be called before `Common_Init()` . If it is not called, default values will be used to initialize the module.

Common_Init()

Description

Initializes all the common modules (Lib Mem, Lib Math, Logging, platform manager) in the correct order with the specified configurations.

Files

`common.h/common.c`

Prototype

```
void Common_Init (RTOS_ERR *p_err)
```

Arguments

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`

Returned Value

None.

Notes / Warnings

1. The functions `Common_Configure...()` can be used to configure more specific properties of the Common module, when `RTOS_CFG_EXTERNALIZE_OPTIONAL_CFG_EN` is set to `DEF_DISABLED`. If set to `DEF_ENABLED`, the structure `Common_InitCfg` needs to be declared and filled by the application to configure these specific properties for the module.

Authentication API

Authentication API

- Auth_ConfigureRootUser()
- Auth_ConfigureResource()
- Auth_ConfigureMemSeg()
- Auth_Init()
- Auth_CreateUser()
- Auth_GetUser()
- Auth_GetUserRight()
- Auth_ValidateCredentials()
- Auth_GrantRight()
- Auth_RevokeRight()

Auth_ConfigureRootUser()

Description

Configure the properties of the root user, in the Auth sub-module.

Files

`auth.h/auth.c`

Prototype

```
void  Auth_ConfigureRootUser (AUTH_CFG_ROOT_USER  *p_cfg)
```

Arguments

`p_cfg`

Pointer to the structure containing the new root user parameters.

Returned Value

None.

Notes / Warnings

1. This function is optional. If it is called, it must be called before `Auth_Init()`. If it is not called, default values will be used to initialize the module.

Auth_ConfigureResource()

Description

Configure the properties of the resources used by the Auth sub-module.

Files

`auth.h/auth.c`

Prototype

```
void Auth_ConfigureResource (AUTH_CFG_RESOURCE *p_cfg)
```

Arguments

`p_cfg`

Pointer to the structure containing the new resource usage parameters.

Returned Value

None.

Notes / Warnings

1. This function is optional. If it is called, it must be called before `Auth_Init()`. If it is not called, default values will be used to initialize the module.

Auth_ConfigureMemSeg()

Description

Configure the memory segment that will be used to allocate internal data needed by Auth instead of the default memory segment.

Files

`auth.h/auth.c`

Prototype

```
void Auth_ConfigureMemSeg (MEM_SEG *p_mem_seg)
```

Arguments

`p_mem_seg`

Pointer to the memory segment from which the internal data will be allocated. If `DEF_NULL`, the internal data will be allocated from the global Heap.

Returned Value

None.

Notes / Warnings

1. This function is optional. If it is called, it must be called before `Auth_Init()`. If it is not called, default values will be used to initialize the module.

Auth_Init()

Description

Initializes authentication module.

Files

`auth.h/auth.c`

Prototype

```
void Auth_Init (RTOS_ERR *p_err)
```

Arguments

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`

Returned Value

None.

Notes / Warnings

1. The functions `Auth_Configure...()` can be used to configure more specific properties of the Auth sub-module, when `RTOS_CFG_EXTERNALIZE_OPTIONAL_CFG_EN` is set to `DEF_DISABLED`. If set to `DEF_ENABLED`, the structure `Auth_InitCfg` needs to be declared and filled by the application to configure these specific properties for the module.

Auth_CreateUser()

Description

Creates a user profile and fills the properties of the user profile structure provided.

Files

`auth.h/auth.c`

Prototype

```
AUTH_USER_HANDLE  Auth_CreateUser (const  CPU_CHAR    *p_name,
                                   const  CPU_CHAR    *p_pwd,
                                   RTOS_ERR           *p_err)
```

Arguments

`p_name`

Pointer to the user name string.

`p_pwd`

Pointer to the password string.

`p_err`

Pointer to the variable that receives the following return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_ALREADY_EXISTS`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_NO_MORE_RSRC`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

Handle of the created user profile.

Notes / Warnings

None.

Auth_GetUser()

Description

Gets the specific user profile structure according to the name provided.

Files

`auth.h/auth.c`

Prototype

```
AUTH_USER_HANDLE  Auth_GetUser (const  CPU_CHAR    *p_name,
                               RTOS_ERR          *p_err)
```

Arguments

`p_name`

Pointer to the user name string to retrieve.

`p_err`

Pointer to the variable that receives the following return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

Handle of the user associated with the name provided.

Notes / Warnings

None.

Auth_GetUserRight()

Description

Gets the rights of the user associated with the specified user handle.

Files

`auth.h/auth.c`

Prototype

```
AUTH_RIGHT  Auth_GetUserRight (AUTH_USER_HANDLE   user_handle,
                               RTOS_ERR          *p_err)
```

Arguments

`user_handle`

Handle name of the user profile from which to retrieve the rights.

`p_err`

Pointer to the variable that receives the following return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

User's rights, if no errors.

`AUTH_RIGHT_NONE` , if any errors are returned.

Notes / Warnings

None.

Auth_ValidateCredentials()

Description

Validates the user and password tuple with known users.

Files

`auth.h/auth.c`

Prototype

```
AUTH_USER_HANDLE  Auth_ValidateCredentials (const  CPU_CHAR   *p_name,
                                            const  CPU_CHAR   *p_pwd,
                                            RTOS_ERR          *p_err)
```

Arguments

`p_name`

Pointer to the user name string.

`p_pwd`

Pointer to the password string.

`p_err`

Pointer to the variable that receives the following return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_INVALID_CREDENTIALS
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_IS_OWNER
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_ABORT
- RTOS_ERR_TIMEOUT

Returned Value

Handle of the user associated with the provided name and password, if no errors are returned.

Notes / Warnings

None.

Auth_GrantRight()

Description

Grants a specific right to a user as another user (limits the rights granted).

Files

auth.h/auth.c

Prototype

```
void  Auth_GrantRight (AUTH_RIGHT          right,
                       AUTH_USER_HANDLE    user_handle,
                       AUTH_USER_HANDLE    as_user_handle,
                       RTOS_ERR           *p_err)
```

Arguments

right

The new right to grant.

user_handle

User handle of the user that will receive the new right.

as_user_handle

User handle of the user with the permission level to perform this task.

p_err

Pointer to the variable that receives the following return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_NOT_FOUND
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_PERMISSION
- RTOS_ERR_IS_OWNER

- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

None.

Notes / Warnings

None.

Auth_RevokeRight()

Description

Revokes a specific right of a specified user.

Files

`auth.h/auth.c`

Prototype

```
void  Auth_RevokeRight (AUTH_RIGHT         right,
                        AUTH_USER_HANDLE    user_handle,
                        AUTH_USER_HANDLE    as_user_handle,
                        RTOS_ERR           *p_err)
```

Arguments

`right`

Right to revoke.

`user_handle`

User handle of the user that will have right revoked.

`as_user_handle`

User handle of the user with the permission level to perform this task.

`p_err`

Pointer to variable that receives the following return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_PERMISSION`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

None.

Notes / Warnings

None.

Lib API

LIB API

- API LIB Ascii
- API LIB Def
- API LIB Math
- API LIB Memory
- API LIB String
- API LIB Utilities

API LIB Ascii

- ASCII_IS_ALPHA() / ASCII_IsAlpha()
- ASCII_IS_ALPHA_NUM() / ASCII_IsAlphaNum()
- ASCII_IS_LOWER() / ASCII_IsLower()
- ASCII_IS_UPPER() / ASCII_IsUpper()
- ASCII_IS_DIG() / ASCII_IsDig()
- ASCII_IS_DIG_OCT() / ASCII_IsDigOct()
- ASCII_IS_DIG_HEX() / ASCII_IsDigHex()
- ASCII_IS_BLANK() / ASCII_IsBlank()
- ASCII_IS_SPACE() / ASCII_IsSpace()
- ASCII_IS_PRINT() / ASCII_IsPrint()
- ASCII_IS_GRAPH() / ASCII_IsGraph()
- ASCII_IS_PUNCT() / ASCII_IsPunct()
- ASCII_IS_CTRL() / ASCII_IsCtrl()
- ASCII_TO_LOWER() / ASCII_ToLower()
- ASCII_TO_UPPER() / ASCII_ToUpper()
- ASCII_Cmp()

ASCII_IS_ALPHA() / ASCII_IsAlpha()

Description

Determines whether a character is an alphabetic character.

Files

`lib_ascii.h/lib_ascii.c`

Prototype

```
CPU_BOOLEAN  ASCII_IsAlpha (CPU_CHAR   c)
```

Arguments

`c`

Character to examine.

Returned Value

- `DEF_YES` , the character is alphabetic.
- `DEF_NO` , the character is NOT alphabetic.

Notes / Warnings

1. ISO/IEC 9899:TC2, Section 7.4.1.2.(2) states that " `isalpha()` returns true only for the characters for which `isupper()` or `islower()` is true".

ASCII_IS_ALPHA_NUM() / ASCII_IsAlphaNum()

Description

Determines whether a character is an alphanumeric character.

Files

`lib_ascii.h/lib_ascii.c`

Prototype

```
CPU_BOOLEAN  ASCII_IsAlphaNum (CPU_CHAR   c)
```

Arguments

`c`

Character to examine.

Returned Value

- `DEF_YES` , the character is alphanumeric.
- `DEF_NO` , the character is NOT alphanumeric.

Notes / Warnings

1. ISO/IEC 9899:TC2, Section 7.4.1.1.(2) states that " `isalnum()` ... tests for any character for which `isalpha()` or `isdigit()` is true".

ASCII_IS_LOWER() / ASCII_IsLower()

Description

Determines whether a character is a lowercase alphabetic character.

Files

`lib_ascii.h/lib_ascii.c`

Prototype

```
CPU_BOOLEAN  ASCII_IsLower (CPU_CHAR   c)
```

Arguments

`c`

Character to examine.

Returned Value

- `DEF_YES` , the character is lowercase alphabetic.
- `DEF_NO` , the character is lowercase alphabetic.

Notes / Warnings

1. ISO/IEC 9899:TC2, Section 7.4.1.7.(2) states that " `islower()` returns true only for the lowercase letters".

ASCII_IS_UPPER() / ASCII_IsUpper()

Description

Determines whether a character is an uppercase alphabetic character.

Files

`lib_ascii.h/lib_ascii.c`

Prototype

```
CPU_BOOLEAN  ASCII_IsUpper (CPU_CHAR   c)
```

Arguments

`c`

Character to examine.

Returned Value

- `DEF_YES` , the character is uppercase alphabetic.
- `DEF_NO` , the character is uppercase alphabetic.

Notes / Warnings

1. ISO/IEC 9899:TC2, Section 7.4.1.11.(2) states that " `isupper()` returns true only for the uppercase letters".

ASCII_IS_DIG() / ASCII_IsDig()

Description

Determines whether a character is a decimal-digit character.

Files

`lib_ascii.h/lib_ascii.c`

Prototype

```
CPU_BOOLEAN  ASCII_IsDig (CPU_CHAR   c)
```

Arguments

`c`

Character to examine.

Returned Value

- `DEF_YES`, the character is a decimal-digit character.
- `DEF_NO`, the character is NOT a decimal-digit character.

Notes / Warnings

1. ISO/IEC 9899:TC2, Section 7.4.1.5.(2) states that " `isdigit()` ... tests for any decimal-digit character".

ASCII_IS_DIG_OCT() / ASCII_IsDigOct()

Description

Determines whether a character is an octal-digit character.

Files

`lib_ascii.h/lib_ascii.c`

Prototype

```
CPU_BOOLEAN  ASCII_IsDigOct (CPU_CHAR   c)
```

Arguments

`c`

Character to examine.

Returned Value

- `DEF_YES`, the character is an octal-digit.
- `DEF_NO`, the character is NOT an octal-digit.

Notes / Warnings

None.

ASCII_IS_DIG_HEX() / ASCII_IsDigHex()

Description

Determine whether a character is a hexadecimal-digit character.

Files

`lib_ascii.h/lib_ascii.c`

Prototype

```
CPU_BOOLEAN  ASCII_IsDigHex (CPU_CHAR  c)
```

Arguments

`c`

Character to examine.

Returned Value

- `DEF_YES` , the character is a hexadecimal-digit character.
- `DEF_NO` , the character is NOT a hexadecimal-digit character.

Notes / Warnings

1. ISO/IEC 9899:TC2, Section 7.4.1.12.(2) states that " `isxdigit()` ... tests for any hexadecimal-digit character".

ASCII_IS_BLANK() / ASCII_IsBlank()

Description

Determines whether a character is a standard blank character.

Files

`lib_ascii.h/lib_ascii.c`

Prototype

```
CPU_BOOLEAN  ASCII_IsBlank (CPU_CHAR  c)
```

Arguments

`c`

Character to examine.

Returned Value

- `DEF_YES` , the character is a standard blank character.
- `DEF_NO` , the character is NOT a standard blank character.

Notes / Warnings

1. (a) ISO/IEC 9899:TC2, Section 7.4.1.3.(2) states that " `isblank()` returns true only for the standard blank characters".
   (b) ISO/IEC 9899:TC2, Section 7.4.1.3.
2. defines "the standard blank characters" as the "space (' '), and horizontal tab ('\t')".

ASCII_IS_SPACE() / ASCII_IsSpace()

Description

Determines whether a character is a white-space character.

Files

`lib_ascii.h/lib_ascii.c`

Prototype

```
CPU_BOOLEAN  ASCII_IsSpace (CPU_CHAR   c)
```

Arguments

`c`

Character to examine.

Returned Value

- `DEF_YES` , the character is a white-space character.
- `DEF_NO` , the character is NOT a white-space character.

Notes / Warnings

1. (a) ISO/IEC 9899:TC2, Section 7.4.1.10.(2) states that " `isspace()` returns true only for the standard white-space characters".
   (b) ISO/IEC 9899:TC2, Section 7.4.1.10.(2) defines "the standard white-space characters" as the "space (' ` ` '), form feed (' `\f` '), new-line (' `\n` '), carriage return (' `\r` '), horizontal tab (' `\t` '), and vertical tab (' `\v` ')".

ASCII_IS_PRINT() / ASCII_IsPrint()

Description

Determines whether a character is a printing character.

Files

`lib_ascii.h/lib_ascii.c`

Prototype

```
CPU_BOOLEAN  ASCII_IsPrint (CPU_CHAR   c)
```

Arguments

`c`

Character to examine.

Returned Value

- `DEF_YES` , the character is a printing character.
- `DEF_NO` , the character is NOT a printing character.

Notes / Warnings

1. (a) ISO/IEC 9899:TC2, Section 7.4.1.8.(2) states that " `isprint()` ... tests for any printing character including space (' ')".
   (b) ISO/IEC 9899:TC2, Section 7.4.(3), Note 169, states that in "the 7-bit US ASCII character set, the printing characters are those whose values lie from 0x20 (space) through 0x7E (tilde)".

ASCII_IS_GRAPH() / ASCII_IsGraph()

Description

Determines whether a character is any printing character except a space character.

Files

`lib_ascii.h/lib_ascii.c`

Prototype

```
CPU_BOOLEAN  ASCII_IsGraph (CPU_CHAR  c)
```

Arguments

`c`

Character to examine.

Returned Value

- `DEF_YES` , the character is a graphic.
- `DEF_NO` , the character is NOT a graphic.

Notes / Warnings

1. (a) ISO/IEC 9899:TC2, Section 7.4.1.6.(2) states that " `isgraph()` ... tests for any printing character except space (' ')".
   (b) ISO/IEC 9899:TC2, Section 7.4.(3), Note 169, states that in "the 7-bit US ASCII character set, the printing characters are those whose values lie from 0x20 (space) through 0x7E (tilde)".

ASCII_IS_PUNCT() / ASCII_IsPunct()

Description

Determines whether a character is a punctuation character.

Files

`lib_ascii.h/lib_ascii.c`

Prototype

```
CPU_BOOLEAN  ASCII_IsPunct (CPU_CHAR   c)
```

Arguments

`c`

Character to examine.

Returned Value

- `DEF_YES` , the character is punctuation.
- `DEF_NO` , the character is NOT punctuation.

Notes / Warnings

1. ISO/IEC 9899:TC2, Section 7.4.1.9.(2) states that " `ispunct()` returns true for every printing character for which neither isspace() nor isalnum() is true".

ASCII_IS_CTRL() / ASCII_IsCtrl()

Description

Determines whether a character is a control character.

Files

`lib_ascii.h/lib_ascii.c`

Prototype

```
CPU_BOOLEAN  ASCII_IsCtrl (CPU_CHAR   c)
```

Arguments

`c`

Character to examine.

Returned Value

- `DEF_YES` , the character is a control character.
- `DEF_NO` , the character is NOT a control character.

Notes / Warnings

1. (a) ISO/IEC 9899:TC2, Section 7.4.1.4.(2) states that "iscntrl() ... tests for any control character".
   (b) ISO/IEC 9899:TC2, Section 7.4.(3), Note 169, states that in "the 7-bit US ASCII character set, ... the control characters are those whose values lie from 0 (NUL) through 0x1F (US), and the character 0x7F (DEL)".

ASCII_TO_LOWER() / ASCII_ToLower()

Description

Converts an uppercase alphabetic character to its corresponding lowercase alphabetic character.

Files

`lib_ascii.h/lib_ascii.c`

Prototype

```
CPU_CHAR  ASCII_ToLower (CPU_CHAR   c)
```

Arguments

`c`

Character to convert.

Returned Value

Lowercase equivalent of 'c', the character 'c' is an uppercase character (see Note #1b1).

Notes / Warnings

1. (a) ISO/IEC 9899:TC2, Section 7.4.2.1.(2) states that " `tolower()` ... converts an uppercase letter to a corresponding lowercase letter".
   (b) ISO/IEC 9899:TC2, Section 7.4.2.1.(3) states that :
   1. (A) "if the argument is a character for which `isupper()` is true and there are one or more corresponding characters ... for which `islower()` is true,".
      (B) " `tolower()` ... returns one of the corresponding characters".
   2. "otherwise, the argument is returned unchanged."

ASCII_TO_UPPER() / ASCII_ToUpper()

Description

Converts a lowercase alphabetic character to its corresponding uppercase alphabetic character.

Files

`lib_ascii.h/lib_ascii.c`

Prototype

```
CPU_CHAR  ASCII_ToUpper (CPU_CHAR  c)
```

Arguments

`c`

Character to convert.

Returned Value

Uppercase equivalent of 'c', the character 'c' is a lowercase character (see Note #1b1).

Notes / Warnings

1. (a) ISO/IEC 9899:TC2, Section 7.4.2.2. (2) states that " `toupper()` ... converts a lowercase letter to a corresponding uppercase letter".
   (b) ISO/IEC 9899:TC2, Section 7.4.2.2.(3) states that :
   1. (A)"if the argument is a character for which `islower()` is true and there are one or more corresponding characters ... for which `isupper()` is true," ...
      (B) " `toupper()` ... returns one of the corresponding characters" ...
   2. "otherwise, the argument is returned unchanged."

ASCII_Cmp()

Description

Determines if two characters are identical (case-insensitive).

Files

`lib.ascii.h/lib.ascii.c`

Prototype

```
ASCII_Cmp (CPU_CHAR    c1,
           CPU_CHAR    c2)
```

Arguments

`c1`

First character.

`c2`

Second character.

Returned Value

- `DEF_YES` , the characters are identical.
- `DEF_NO` , the characters are different.

Notes / Warnings

None.

API LIB Def

- DEF_CHK_VAL_MIN()
- DEF_CHK_VAL_MAX()
- DEF_CHK_VAL()
- DEF_MIN()
- DEF_MAX()
- DEF_ABS()
- DEF_IS_ADDR_ALIGNED()

DEF_CHK_VAL_MIN()

Description

Validate a value as greater than or equal to a specified minimum value.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
DEF_CHK_VAL_MIN (val, val_min)
```

Arguments

`val`

Value to validate.

`val_min`

Minimum value to test.

Returned Value

`DEF_OK` , Value is greater than or equal to minimum value.

Notes / Warnings

1. `DEF_CHK_VAL_MIN()` avoids directly comparing any two values if only one of the values is negative since the negative value might be incorrectly promoted to an arbitrary unsigned value if the other value to compare is unsigned.
2. Validation of values is limited to the range supported by the compiler &/or target environment. All other values that underflow/overflow the supported range will modulo/wrap into the supported range as arbitrary signed or unsigned values. (a) Note that the most negative value, -2^(N-1), is NOT included in the supported range since many compilers do NOT always correctly handle this value.
3. ' `val` ' and ' `val_min` ' are compared to 1 instead of 0 to avoid warning generated for unsigned numbers.

DEF_CHK_VAL_MAX()

Description

Validate a value as less than or equal to a specified maximum value.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
DEF_CHK_VAL_MAX (val, val_max)
```

Arguments

`val`

Value to validate.

`val_max`

Maximum value to test.

Returned Value

`DEF_OK` , Value is less than or equal to maximum value.

Notes / Warnings

1. `DEF_CHK_VAL_MAX()` avoids directly comparing any two values if only one of the values is negative since the negative value might be incorrectly promoted to an arbitrary unsigned value if the other value to compare is unsigned.
2. Validation of values is limited to the range supported by the compiler &/or target environment. All other values that underflow/overflow the supported range will modulo/wrap into the supported range as arbitrary signed or unsigned values. (a) Note that the most negative value, -2^(N-1), is NOT included in the supported range since many compilers do NOT always correctly handle this value.
3. ' `val` ' and ' `val_max` ' are compared to 1 instead of 0 to avoid warning generated for unsigned numbers.

DEF_CHK_VAL()

Description

Validate a value as greater than or equal to a specified minimum value & less than or equal to a specified maximum value.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
DEF_CHK_VAL (val, val_min, val_max)
```

Arguments

`val`

Value to validate.

`val_min`

Minimum value to test.

`val_max`

Maximum value to test.

Returned Value

`DEF_OK` , Value is greater than or equal to minimum value AND less than or equal to maximum value.

Notes / Warnings

1. `DEF_CHK_VAL()` avoids directly comparing any two values if only one of the values is negative since the negative value might be incorrectly promoted to an arbitrary unsigned value if the other value to compare is unsigned.
2. Validation of values is limited to the range supported by the compiler &/or target environment. All other values that underflow/overflow the supported range will modulo/wrap into the supported range as arbitrary signed or unsigned values. (a) Note that the most negative value, -2^(N-1), is NOT included in the supported range since many compilers do NOT always correctly handle this value.
3. `DEF_CHK_VAL()` does NOT validate that the maximum value ('val_max') is greater than or equal to the minimum value ('val_min').

DEF_MIN()

Description

Determine the minimum of two values.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
DEF_MIN (a, b)
```

Arguments

`a`

First value.

`b`

Second value.

Returned Value

Minimum of the two values.

Notes / Warnings

None.

DEF_MAX()

Description

Determine the maximum of two values.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
DEF_MAX (a, b)
```

Arguments

`a`

First value.

`b`

Second value.

Returned Value

Maximum of the two values.

Notes / Warnings

None.

DEF_ABS()

Description

Determine the absolute value of a value.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
DEF_ABS (a)
```

Arguments

`a`

Value to calculate absolute value.

Returned Value

Absolute value of the value.

Notes / Warnings

None.

DEF_IS_ADDR_ALIGNED()

Description

Determine if specified address is aligned.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
DEF_ADDR_IS_ALIGNED (addr, align)
```

Arguments

`addr`

Memory address to evaluate.

`align`

Alignment needed (in bytes).

Returned Value

`DEF_YES` Address is aligned.

Notes / Warnings

1. Align value MUST be a power of 2.

API LIB Math

- Math_RandSetSeed()
- Math_Rand()
- Math_RandSeed()
- Math_Init()
- MATH_IS_PWR2()
- MATH_ROUND_INC_UP_PWR2()
- MATH_ROUND_INC_UP()

Math_RandSetSeed()

Description

Sets the current pseudo-random number generator seed.

Files

`lib_math.h/lib_math.c`

Prototype

```
void Math_RandSetSeed (RAND_NBR  seed)
```

Arguments

`seed`

Initial (or current) value to set for the pseudo-random number sequence.

Returned Value

None.

Notes / Warnings

1. IEEE Std 1003.1, 2004 Edition, Section '`rand()` : DESCRIPTION' states that "`srand()` ... uses the argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to rand()".
2. 'Math_RandSeedCur' MUST always be accessed exclusively in critical sections. See also '`Math_Rand()` Note #1b'.

Math_Rand()

Description

Calculates the next pseudo-random number.

Files

`lib_math.h/lib_math.c`

Prototype

```
RAND_NBR Math_Rand (void)
```

Arguments

None.

Returned Value

Next pseudo-random number in the sequence after 'Math_RandSeedCur'.

Notes / Warnings

1. See '`Math_RandSeed()` Note #1'.
   (a) The pseudo-random number generator is implemented as a Linear Congruential Generator (LCG).
   (b) The pseudo-random number generated is in the range `[0, RAND_LCG_PARAM_M]` .
2. See '`Math_RandSeed()` Note #2'.
   (a) IEEE Std 1003.1, 2004 Edition, Section 'rand() : DESCRIPTION' states that "`rand()` ... need not be reentrant ... [and] is not required to be thread-safe".

(b) However, to implement `Math_Rand()` as re-entrant; '`Math_RandSeedCur`' MUST always be accessed & updated exclusively in critical sections.

Math_RandSeed()

Description

Calculates the next pseudo-random number.

Files

`lib_math.h/lib_math.c`

Prototype

```
RAND_NBR Math_RandSeed (RAND_NBR   seed)
```

Arguments

`seed`

Initial (or current) value for the pseudo-random number sequence.

Returned Value

Next pseudo-random number in the sequence after 'seed'.

Notes / Warnings

1. See 'lib_math.h RANDOM NUMBER DEFINES Note #1b'.
   (a) BSD/ANSI-C implements rand() as a Linear Congruential Generator (LCG):
   `random_number = [(a * random_number ) + b] modulo m`
   1. (a) random_number on the left: Next random number to generate (b) random_number on the right: Previous random number generated
   2. a = `RAND_LCG_PARAM_A` LCG multiplier
   3. b = `RAND_LCG_PARAM_B` LCG incrementor
   4. m = `RAND_LCG_PARAM_M` + 1 LCG modulus

   (b) The pseudo-random number generated is in the range [0,][0, `RAND_LCG_PARAM_M`].
2. (a) IEEE Std 1003.1, 2004 Edition, Section '`rand()` : DESCRIPTION' states that "`rand()` ... need not be reentrant ... [and] is not required to be thread-safe".
   (b) However, `Math_RandSeed()` is re-entrant since it calculates the next random number using ONLY local variables.

Math_Init()

Description

Initializes the Mathematic Module.

Files

`lib_math.h/lib_math.c`

Prototype

```
void Math_Init (void)
```

Arguments

None.

Returned Value

None.

Notes / Warnings

1. IEEE Std 1003.1, 2004 Edition, Section '`rand()` : DESCRIPTION' states that "if `rand()` is called before any calls to `srand()` are made, the same sequence shall be generated as when `srand()` is first called with a seed value of 1".
2. The following function is deprecated. Unless already used, it should no longer be used, since they will be removed in a future version.

MATH_IS_PWR2()

Description

Determine if a value is a power of 2.

Files

`lib_math.h/lib_math.c`

Prototype

```
MATH_IS_PWR2 (nbr)
```

Arguments

`nbr`

Value.

Returned Value

`DEF_YES` , 'nbr' is a power of 2.

Notes / Warnings

None.

MATH_ROUND_INC_UP_PWR2()

Description

Round value up to the next (power of 2) increment.

Files

`lib_math.h/lib_math.c`

Prototype

```
MATH_ROUND_INC_UP_PWR2(nbr, inc)
```

Arguments

`nbr`

Value to round.

`inc`

Increment to use. MUST be a power of 2.

Returned Value

Rounded up value.

Notes / Warnings

None.

MATH_ROUND_INC_UP()

Description

Round value up to the next increment.

Files

`lib_math.h/lib_math.c`

Prototype

```
MATH_ROUND_INC_UP(nbr, inc)
```

Arguments

`nbr`

Value to round.

`inc`

Increment to use.

Returned Value

Rounded up value.

Notes / Warnings

None.

API LIB Memory

- Mem_Clr()
- Mem_Set()
- Mem_Copy()
- Mem_Move()
- Mem_Cmp()
- Mem_SegCreate()
- MEM_SEG_INIT()

- Mem_SegReg()
- Mem_SegClr()
- Mem_SegAlloc()
- Mem_SegAllocExt()
- Mem_SegAllocHW()
- Mem_SegRemSizeGet()
- Mem_OutputUsage()
- Mem_DynPoolCreate()
- Mem_DynPoolCreatePersistent()
- Mem_DynPoolCreateHW()
- Mem_DynPoolBlkNbrAvailGet()
- Mem_DynPoolBlkGet()
- Mem_DynPoolBlkFree()
- MEM_VAL_BIG_TO_LITTLE_xx() / MEM_VAL_LITTLE_TO_BIG_xx()
- MEM_VAL_BIG_TO_HOST_xx() / MEM_VAL_HOST_TO_BIG_xx()
- MEM_VAL_LITTLE_TO_HOST_xx() / MEM_VAL_HOST_TO_LITTLE_xx()
- MEM_VAL_GET_INTxxU_xxx()
- MEM_VAL_SET_INTxxU_xxx()
- MEM_VAL_COPY_GET_INTxxU_xxx()
- MEM_VAL_COPY_GET_INTU_xxx()
- MEM_VAL_COPY_SET_INTxxU_xxx()
- MEM_VAL_COPY_SET_INTU_xxx()
- MEM_VAL_COPY_xx()

Mem_Clr()

Description

Clears the data buffer (see Note #2).

Files

`lib_mem.h/lib_mem.c`

Prototype

```
#if (LIB_MEM_CFG_STD_C_LIB_EN == DEF_DISABLED)
void  Mem_Clr (void       \*p_mem,
          CPU_SIZE_T   size)
```

Arguments

`p_mem`

Pointer to the memory buffer to clear.

`size`

Number of data buffer octets to clear (see Note #1).

Returned Value

None.

Notes / Warnings

1. Null clears are allowed (i.e. zero-length clears). See also '`Mem_Set()` Note #1'.

Mem_Set()

Description

Fills the data buffer with specified data octet.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
#if (LIB_MEM_CFG_STD_C_LIB_EN == DEF_DISABLED)

void Mem_Set (void        \*p_mem,
          CPU_INT08U   data_val,
          CPU_SIZE_T   size)
```

Arguments

`p_mem`

Pointer to the memory buffer to fill with the specified data octet.

`data_val`

Data filled octet value.

`size`

Number of data buffer octets to fill (see Note #1).

Returned Value

None.

Notes / Warnings

1. Null sets are allowed (i.e. zero-length sets).
2. For the best CPU performance, this function fills the data buffer using '`CPU_ALIGN`'-sized data words. Since many word-aligned processors REQUIRE that multi-octet words be accessed on word-aligned addresses, '`CPU_ALIGN`'-sized words MUST be accessed on '`CPU_ALIGN`' addresses.*
3. Modulo arithmetic determines if a memory buffer starts on a '`CPU_ALIGN`' address boundary.

Modulo arithmetic in ANSI-C REQUIREs operations are performed on integer values, so address values MUST be cast to an appropriately-sized integer value before any '`mem_align_mod`' arithmetic operation.

Mem_Copy()

Description

Copies data octets from one memory buffer to another memory buffer.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
#if (LIB_MEM_CFG_STD_C_LIB_EN == DEF_DISABLED)
void Mem_Copy (      void       \*p_dest,
               const void       \*p_src,
                    CPU_SIZE_T   size)
```

Arguments

`p_dest`

Pointer to the destination memory buffer.

`p_src`

Pointer to the source memory buffer.

`size`

Number of octets to copy (see Note #1).

Returned Value

None.

Notes / Warnings

1. Null copies are allowed (i.e. zero-length copies).
2. Memory buffers NOT checked for overlapping.
   (a) IEEE Std 1003.1, 2004 Edition, Section '`memcpy()` : DESCRIPTION' states that "if copying takes place between objects that overlap, the behavior is undefined".
   (b) Data octets from a source memory buffer at a higher address value SHOULD successfully copy to a destination memory buffer at a lower address value. This occurs even if any octets of the memory buffers overlap, as long as there are no individual, atomic CPU word copy overlaps.
3. For the best CPU performance, this function copies data buffers using 'CPU_ALIGN'-sized data words. Since many word-aligned processors REQUIRE that multi-octet words be accessed on word-aligned addresses, '`CPU_ALIGN`'-sized words MUST be accessed on '`CPU_ALIGN`' addresses.
4. Modulo arithmetic determines if a memory buffer starts on a '`CPU_ALIGN`' address boundary.

Modulo arithmetic in ANSI-C REQUIRE operations are performed on integer values, so address values MUST be cast to an appropriately-sized integer value before any '`mem_align_mod`' arithmetic operation.

Mem_Move()

Description

Moves data octets from one memory buffer to another memory buffer, or within the same memory buffer. Overlapping is correctly handled for all move operations.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
#if (LIB_MEM_CFG_STD_C_LIB_EN == DEF_DISABLED)
void Mem_Move (      void       *p_dest,
               const void       *p_src,
                    CPU_SIZE_T   size)
```

Arguments

`p_dest`

Pointer to destination memory buffer.

`p_src`

Pointer to source memory buffer.

`size`

Number of octets to move (see Note #1).

Returned Value

None.

Notes / Warnings

1. Null move operations are allowed (i.e. zero-length).
2. Memory buffers checked for overlapping.
3. For the best CPU performance, this function copies data buffer using '`CPU_ALIGN`'-sized data words. Since many word-aligned processors REQUIRE that multi-octet words be accessed on word-aligned addresses, '`CPU_ALIGN`'-sized words MUST be accessed on '`CPU_ALIGN`'d addresses.
4. Modulo arithmetic determines if a memory buffer starts on a '`CPU_ALIGN`' address boundary.

Modulo arithmetic in ANSI-C REQUIREs operations performed on integer values, so address values MUST be cast to an appropriately-sized integer value before any '`mem_align_mod`' arithmetic operation.

Mem_Cmp()

Description

Verifies that ALL data octets in the two memory buffers are identical in sequence.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
#if (LIB_MEM_CFG_STD_C_LIB_EN == DEF_DISABLED)
CPU_BOOLEAN Mem_Cmp (const void      *p1_mem,
                     const void      *p2_mem,
                       CPU_SIZE_T    size)
```

Arguments

`p1_mem`

Pointer to first memory buffer.

`p2_mem`

Pointer to second memory buffer.

`size`

Number of data buffer octets to compare (see Note #1).

Returned Value

- `DEF_YES` , if the 'size' of data octets is identical in both memory buffers.
- `DEF_NO` , if the 'size' of data octets is different in both memory buffers.

Notes / Warnings

1. Null compares are allowed (i.e. zero-length compares); ' `DEF_YES` ' is returned to indicate identical null compare.
2. Many memory buffer comparisons vary ONLY in the least significant octets (e.g., network address buffers). Consequently, memory buffer comparison is more efficient if the comparison starts from the end of the memory buffers. This aborts sooner on dissimilar memory buffers that vary only in the least significant octets.
3. For the best CPU performance, this function compares data buffers using ' `CPU_ALIGN` '-sized data words. Since many word-aligned processors REQUIRE that multi-octet words be accessed on word-aligned addresses, ' `CPU_ALIGN` '-sized words MUST be accessed on ' `CPU_ALIGN` 'd addresses.
4. Modulo arithmetic determines if a memory buffer starts on a ' `CPU_ALIGN` ' address boundary.

Modulo arithmetic in ANSI-C REQUIREs operations performed on integer values, so address values MUST be cast to an appropriately-sized integer value before any ' `mem_align_mod` ' arithmetic operation.

Mem_SegCreate()

Description

Creates a new memory segment to be used for runtime memory allocation.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
void Mem_SegCreate (const CPU_CHAR    *p_name,
                    MEM_SEG      *p_seg,
                    CPU_ADDR      seg_base_addr,
                    CPU_SIZE_T    size,
                    CPU_SIZE_T    padding_align,
                    RTOS_ERR     *p_err)
```

Arguments

`p_name`

Pointer to segment name.

`p_seg`

Pointer to segment data. Must be allocated by caller.

`seg_base_addr`

Address of segment's first byte.

`size`

Total size of segment (in bytes).

`padding_align`

Padding alignment (in bytes) that will be added to any allocated buffer from this memory segment. MUST be a power of 2. `LIB_MEM_PADDING_ALIGN_NONE` means no padding.

`p_err`

Pointer to a variable that will receive the following return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_ARG`

Returned Value

None.

Notes / Warnings

1. New segments are checked for overlap with existing segments. A critical section must be maintained during the whole list search and adds a procedure to prevent a re-entrant call from creating another segment that would overlaps with the new one.

MEM_SEG_INIT()

Description

Creates a new memory segment to be used for runtime memory allocation. Created segment will not be added to list checked for overlap or to debug, it must be added afterwards by calling the `Mem_SegReg()` function.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
MEM_SEG_INIT (p_name, seg_base_addr, size, padding_align)
```

Arguments

`p_name`

Pointer to segment name.

`seg_base_addr`

Address of segment's first byte.

`size`

Total size of segment (in bytes).

`padding_align`

Padding alignment (in bytes) that will be added to any allocated buffer from this memory segment. MUST be a power of 2. `LIB_MEM_PADDING_ALIGN_NONE` means no padding.

Returned Value

None.

Notes / Warnings

1. Contrary to the `Mem_SegCreate()` function, this segment is not checked for overlap at run-time. Therefore, extra care should be taken when using this macro.

2. Usage is as follows:

```
MEM_SEG My_MemSeg = MEM_SEG_INIT("My Segment", (CPU_ADDR)&My_SegmentData[0u],
                         MY_SEGMENT_SIZE, LIB_MEM_CFG_HEAP_PADDING_ALIGN);
```

Mem_SegReg()

Description

Registers a memory segment that was created at compile-time to enable both usage output and overlap checks when creating a new memory segment.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
void Mem_SegReg (MEM_SEG   *p_seg,
            RTOS_ERR  *p_err)
```

Arguments

`p_seg`

Pointer to segment data already created.

`p_err`

Pointer to a variable that will receive the following return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ALREADY_EXISTS`

Returned Value

None.

Notes / Warnings

None.

Mem_SegClr()

Description

Clears a memory segment.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
#if (LIB_MEM_CFG_DBG_INFO_EN == DEF_DISABLED)
void Mem_SegClr (MEM_SEG   \*p_seg,
            RTOS_ERR  \*p_err)
```

Arguments

`p_seg`

Pointer to segment data and must be allocated by caller.

`p_err`

Pointer to a variable that will receive the following return error code from this function :

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

1. Use this function with extreme caution. It must only be called on memory segments that are no longer used.
2. This function is disabled when debug mode is enabled to avoid heap memory leaks.

Mem_SegAlloc()

Description

Allocates memory from a specified segment. The returned memory block will be aligned on a CPU word boundary.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
void *Mem_SegAlloc (const CPU_CHAR   *p_name,
                    MEM_SEG          *p_seg,
                    CPU_SIZE_T        size,
                    RTOS_ERR         *p_err)
```

Arguments

`p_name`

Pointer to the allocated object name, which may track allocations. May be `DEF_NULL` .

`p_seg`

Pointer to the segment from which to allocate memory. If NULL, it will allocate from the general-purpose heap.

`size`

Size of memory block to allocate (bytes).

`p_err`

Pointer to a variable that will receive the following return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_SEG_OVF`

Returned Value

- Pointer to allocated memory block, if successful.
- `DEF_NULL` , if the memory is not allocated.

Notes / Warnings

1. The memory block returned by this function will be aligned on a word boundary. To specify an alignment value, use either `Mem_SegAllocExt()` or `Mem_SegAllocHW()` .
2. No logging can be done in this function or called functions, since they may be used before the logging mechanism is initialized.

Mem_SegAllocExt()

Description

Allocates memory from specified memory segment.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
void *Mem_SegAllocExt (const CPU_CHAR    *p_name,
                       MEM_SEG      *p_seg,
                       CPU_SIZE_T   size,
                       CPU_SIZE_T   align,
                       CPU_SIZE_T  *p_bytes_reqd,
                       RTOS_ERR    *p_err)
```

Arguments

`p_name`

Pointer to allocated object name, which may track allocations. May be `DEF_NULL` .

`p_seg`

Pointer to segment from which to allocate memory. If NULL, it will allocate from the general-purpose heap.

`size`

Size of memory block to allocate (in bytes).

`align`

Required alignment of memory block (in bytes). MUST be a power of 2.

`p_bytes_reqd`

Pointer to a variable that will receive the number of free bytes missing for the allocation to succeed. Set to `DEF_NULL` to skip calculation.

`p_err`

Pointer to a variable that will receive the following return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_SEG_OVF`

Returned Value

Pointer to allocated memory block, if successful.

`DEF_NULL`, if memory is not allocated.

Notes / Warnings

None.

Mem_SegAllocHW()

Description

Allocates memory from specified segment. The returned buffer will be padded in function of memory segment's properties.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
void *Mem_SegAllocHW (const CPU_CHAR    *p_name,
                      MEM_SEG      *p_seg,
                      CPU_SIZE_T    size,
                      CPU_SIZE_T    align,
                      CPU_SIZE_T   *p_bytes_reqd,
                      RTOS_ERR     *p_err)
```

Arguments

`p_name`

Pointer to allocated object name, which may track allocations. May be DEF_NULL.

`p_seg`

Pointer to segment from which to allocate memory. If NULL, it will allocate from the general-purpose heap.

`size`

Size of memory block to allocate (in bytes).

`align`

Required alignment of memory block (in bytes). MUST be a power of 2.

`p_bytes_reqd`

Pointer to a variable that will receive the number of free bytes missing for the allocation to succeed. Set to `DEF_NULL` to skip calculation.

`p_err`

Pointer to a variable that will receive the following return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_SEG_OVF`

Returned Value

Pointer to allocated memory block, if successful.

DEF_NULL , if memory is not allocated.

Notes / Warnings

None.

Mem_SegRemSizeGet()

Description

Calculates the remaining free space in the memory segment.

Files

lib_mem.h/lib_mem.c

Prototype

```
CPU_SIZE_T Mem_SegRemSizeGet (MEM_SEG       *p_seg,
                              CPU_SIZE_T     align,
                              MEM_SEG_INFO  *p_seg_info,
                              RTOS_ERR      *p_err)
```

Arguments

p_seg

Pointer to segment data.

align

Alignment in bytes to assume for calculation of free space.

p_seg_info

Pointer to structure that will receive further segment info data (used size, total size, base address, and next allocation address).

p_err

Pointer to a variable that will receive the following return error code from this function :

- RTOS_ERR_NONE

Returned Value

- The amount of free space in the memory segment (bytes), if successful.
- 0, otherwise or if the memory segment empty.

Notes / Warnings

None.

Mem_OutputUsage()

Description

Outputs the memory usage report through 'out_fnct'.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
#if (LIB_MEM_CFG_DBG_INFO_EN == DEF_ENABLED)
void  Mem_OutputUsage(void      (\*out_fnct) (CPU_CHAR  \*p_str),
               RTOS_ERR  \*p_err)
```

Arguments

`out_fnct`

Pointer to output function.

`p_err`

Pointer to a variable that will receive the following return error code from this function :

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

None.

Mem_DynPoolCreate()

Description

Creates a dynamic memory pool.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
void Mem_DynPoolCreate (const CPU_CHAR     *p_name,
                        MEM_DYN_POOL  *p_pool,
                        MEM_SEG       *p_seg,
                        CPU_SIZE_T     blk_size,
                        CPU_SIZE_T     blk_align,
                        CPU_SIZE_T     blk_qty_init,
                        CPU_SIZE_T     blk_qty_max,
                        RTOS_ERR      *p_err)
```

Arguments

`p_name`

Pointer to a pool name.

`p_pool`

Pointer to a pool data.

`p_seg`

Pointer to segment from which to allocate memory. If NULL, it will allocate from the general-purpose heap.

`blk_size`

Size of memory block to allocate from pool (in bytes). See Note #1.

`blk_align`

Required alignment of memory block (in bytes). MUST be a power of 2.

`blk_qty_init`

Initial number of elements to be allocated in pool.

`blk_qty_max`

Maximum number of elements that can be allocated from this pool. Set to `LIB_MEM_BLK_QTY_UNLIMITED` if there is no limit.

`p_err`

Pointer to a variable that will receive the following return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`

Returned Value

None.

Notes / Warnings

1. 'blk_size' must be big enough to fit a pointer since the pointer to the next free block is stored in the block itself (only when free/unused).

Mem_DynPoolCreatePersistent()

Description

Creates a persistent dynamic memory pool.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
void Mem_DynPoolCreatePersistent (const CPU_CHAR           *p_name,
                                  MEM_DYN_POOL             *p_pool,
                                  MEM_SEG                  *p_seg,
                                  CPU_SIZE_T               blk_size,
                                  CPU_SIZE_T               blk_align,
                                  CPU_SIZE_T               blk_qty_init,
                                  CPU_SIZE_T               blk_qty_max,
                                  MEM_DYN_POOL_ALLOC_FNCT  alloc_callback,
                                  void                     *p_callback_arg,
                                  RTOS_ERR                 *p_err)
```

Arguments

`p_name`

Pointer to a pool name.

`p_pool`

Pointer to a pool data.

`p_seg`

Pointer to segment from which to allocate memory.

`blk_size`

Size of memory block to allocate from pool (in bytes). See Note #1.

`blk_align`

Required alignment of memory block (in bytes). MUST be a power of 2.

`blk_qty_init`

Initial number of elements to be allocated in pool.

`blk_qty_max`

Maximum number of elements that can be allocated from this pool. Set to `LIB_MEM_BLK_QTY_UNLIMITED` if there is no limit.

`alloc_callback`

Function that will be called the first time each block is allocated.

`p_callback_arg`

Pointer to argument that will be passed to callback.

`p_err`

Pointer to a variable that will receive the following return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`

Returned Value

None.

Notes / Warnings

1. 'blk_size' must be big enough to fit a pointer since the pointer to the next free block is stored in the block itself (only when free/unused).

Mem_DynPoolCreateHW()

Description

Creates a dynamic memory pool. Memory blocks will be padded according to memory segment's properties.

Files

lib_mem.h/lib_mem.c

Prototype

```
void Mem_DynPoolCreateHW (const CPU_CHAR     *p_name,
                          MEM_DYN_POOL  *p_pool,
                          MEM_SEG       *p_seg,
                          CPU_SIZE_T     blk_size,
                          CPU_SIZE_T     blk_align,
                          CPU_SIZE_T     blk_qty_init,
                          CPU_SIZE_T     blk_qty_max,
                          RTOS_ERR      *p_err)
```

Arguments

p_name

Pointer to a pool name.

p_pool

Pointer to a pool data.

p_seg

Pointer to the segment from which to allocate memory. If NULL, it will allocate from the general-purpose heap.

blk_size

Size of memory block to allocate from pool (in bytes). See Note #1.

blk_align

Required alignment of memory block (in bytes). MUST be a power of 2.

blk_qty_init

Initial number of elements to be allocated in pool.

blk_qty_max

Maximum number of elements that can be allocated from this pool. Set to LIB_MEM_BLK_QTY_UNLIMITED if no limit.

p_err

Pointer to a variable that will receive the following return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_BLK_ALLOC_CALLBACK
- RTOS_ERR_SEG_OVF

Returned Value

None.

Notes / Warnings

1. 'blk_size' must be big enough to fit a pointer since the pointer to the next free block is stored in the block itself (only when free/unused).

Mem_DynPoolBlkNbrAvailGet()

Description

Gets a number of available blocks in dynamic memory pool. If 'p_pool_info' is `DEF_NULL`, this call will fail with a dynamic memory pool for which no limit was set at creation.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
CPU_SIZE_T Mem_DynPoolBlkNbrAvailGet (MEM_DYN_POOL       *p_pool,
                                      MEM_DYN_POOL_INFO  *p_pool_info,
                                      RTOS_ERR           *p_err)
```

Arguments

`p_pool`

Pointer to a pool data.

`p_pool_info`

Pointer to `MEM_DYN_POOL_INFO` that will be filled by this function. If `DEF_NULL` and if pool has a block limit, only the number of blocks remaining is returned.

`p_err`

Pointer to a variable that will receive the following return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_POOL_UNLIMITED`

Returned Value

Number of blocks remaining in dynamic memory pool, if successful.

0, if pool is empty or if an error occurred.

Notes / Warnings

None.

Mem_DynPoolBlkGet()

Description

Gets a memory block from specified pool, growing it if needed.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
void *Mem_DynPoolBlkGet (MEM_DYN_POOL  *p_pool,
                         RTOS_ERR      *p_err)
```

Arguments

`p_pool`

Pointer to a pool data.

`p_err`

Pointer to a variable that will receive the following return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`

Returned Value

Pointer to memory block, if successful.

`DEF_NULL` , if it is not successful.

Notes / Warnings

None.

Mem_DynPoolBlkFree()

Description

Frees a memory block, making it available for future use.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
void Mem_DynPoolBlkFree (MEM_DYN_POOL  *p_pool,
                    void          *p_blk,
                    RTOS_ERR      *p_err)
```

Arguments

`p_pool`

Pointer to a pool data.

`p_blk`

Pointer to first byte of memory block.

`p_err`

Pointer to a variable that will receive the following return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_POOL_FULL`

Returned Value

None.

Notes / Warnings

None.

MEM_VAL_BIG_TO_LITTLE_xx() / MEM_VAL_LITTLE_TO_BIG_xx()

Description

Convert data values to & from big- or little-endian CPU word order.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
MEM_VAL_BIG_TO_LITTLE_16 (val)

MEM_VAL_BIG_TO_LITTLE_32 (val)

MEM_VAL_LITTLE_TO_BIG_16 (val)

MEM_VAL_LITTLE_TO_BIG_32 (val)
```

Arguments

`val`

Data value to convert (see Notes #1 & #2).

Returned Value

Converted data value (see Notes #1 & #2).

Notes / Warnings

1. Convert data values to the desired data-word order :
2. 'val' data value to convert & any variable to receive the returned conversion MUST start on appropriate CPU word-aligned addresses.
3. `MEM_VAL_COPY_xxx()` macro's are more efficient than generic endian word order macro's & are also independent of CPU data-word-alignment & SHOULD be used whenever possible.
4. Generic endian word order macro's are NOT atomic operations & MUST NOT be used on any non-static (i.e., volatile) variables, registers, hardware, etc.; without the caller of the macro's providing some form of additional protection (e.g., mutual exclusion).
5. The ' `CPU_CFG_ENDIAN_TYPE` ' pre-processor 'else'-conditional code SHOULD never be compiled/ linked since each 'cpu.h' SHOULD ensure that the CPU data-word-memory order configuration constant (`CPU_CFG_ENDIAN_TYPE`) is configured with an appropriate data-word-memory order value (see 'cpu.h CPU WORD CONFIGURATION Note #2'). The 'else'-conditional code is included as an extra precaution in case 'cpu.h' is incorrectly configured.

MEM_VAL_BIG_TO_HOST_xx() / MEM_VAL_HOST_TO_BIG_xx()

Description

Convert data values to & from big- or host-endian CPU word order.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
MEM_VAL_BIG_TO_HOST_16 (val)

MEM_VAL_BIG_TO_HOST_32 (val)

MEM_VAL_HOST_TO_BIG_16 (val)

MEM_VAL_HOST_TO_BIG_32 (val)
```

Arguments

`val`

Data value to convert (see Notes #1 & #2).

Returned Value

Converted data value (see Notes #1 & #2).

Notes / Warnings

1. Convert data values to the desired data-word order :
2. 'val' data value to convert & any variable to receive the returned conversion MUST start on appropriate CPU word-aligned addresses.
3. `MEM_VAL_COPY_xxx()` macro's are more efficient than generic endian word order macro's & are also independent of CPU data-word-alignment & SHOULD be used whenever possible.
4. Generic endian word order macro's are NOT atomic operations & MUST NOT be used on any non-static (i.e., volatile) variables, registers, hardware, etc.; without the caller of the macro's providing some form of additional protection (e.g., mutual exclusion).
5. The ' `CPU_CFG_ENDIAN_TYPE` ' pre-processor 'else'-conditional code SHOULD never be compiled/ linked since each 'cpu.h' SHOULD ensure that the CPU data-word-memory order configuration constant (`CPU_CFG_ENDIAN_TYPE`) is configured with an appropriate data-word-memory order value (see 'cpu.h CPU WORD CONFIGURATION Note #2'). The 'else'-conditional code is included as an extra precaution in case 'cpu.h' is incorrectly configured.

MEM_VAL_LITTLE_TO_HOST_xx() / MEM_VAL_HOST_TO_LITTLE_xx()

Description

Convert data values to & from little- or host-endian CPU word order.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
MEM_VAL_LITTLE_TO_HOST_16 (val)

MEM_VAL_LITTLE_TO_HOST_32 (val)

MEM_VAL_HOST_TO_LITTLE_16 (val)

MEM_VAL_HOST_TO_LITTLE_32 (val)
```

Arguments

`val`

Data value to convert (see Notes #1 & #2).

Returned Value

Converted data value (see Notes #1 & #2).

Notes / Warnings

1. Convert data values to the desired data-word order :
2. 'val' data value to convert & any variable to receive the returned conversion MUST start on appropriate CPU word-aligned addresses.
3. `MEM_VAL_COPY_xxx()` macro's are more efficient than generic endian word order macro's & are also independent of CPU data-word-alignment & SHOULD be used whenever possible.
4. Generic endian word order macro's are NOT atomic operations & MUST NOT be used on any non-static (i.e., volatile) variables, registers, hardware, etc.; without the caller of the macro's providing some form of additional protection (e.g., mutual exclusion).
5. The ' `CPU_CFG_ENDIAN_TYPE` ' pre-processor 'else'-conditional code SHOULD never be compiled/ linked since each 'cpu.h' SHOULD ensure that the CPU data-word-memory order configuration constant (`CPU_CFG_ENDIAN_TYPE`) is configured with an appropriate data-word-memory order value (see 'cpu.h CPU WORD CONFIGURATION Note #2'). The 'else'-conditional code is included as an extra precaution in case 'cpu.h' is incorrectly configured.

MEM_VAL_GET_INTxxU_xxx()

Description

Decode data values from any CPU memory address.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
MEM_VAL_GET_INT08U (addr)

MEM_VAL_GET_INT16U (addr)

MEM_VAL_GET_INT32U (addr)

MEM_VAL_GET_INT64U (addr)

MEM_VAL_GET_INT08U_BIG (addr)

MEM_VAL_GET_INT16U_BIG (addr)

MEM_VAL_GET_INT32U_BIG (addr)

MEM_VAL_GET_INT64U_BIG (addr)

MEM_VAL_GET_INT08U_LITTLE (addr)

MEM_VAL_GET_INT16U_LITTLE (addr)

MEM_VAL_GET_INT32U_LITTLE (addr)

MEM_VAL_GET_INT64U_LITTLE (addr)
```

Arguments

`addr`

Lowest CPU memory address of data value to decode (see Notes #2 & #3a).

Returned Value

Decoded data value from CPU memory address (see Notes #1 & #3b).

Notes / Warnings

1. Decode data values based on the values' data-word order in CPU memory :
2. CPU memory addresses/pointers NOT checked for NULL.
3. (a) `MEM_VAL_GET_xxx()` macro's decode data values without regard to CPU word-aligned addresses. Thus for processors that require data word alignment, data words can be decoded from any CPU address, word-aligned or not, without generating data-word-alignment exceptions/faults.
   (b) However, any variable to receive the returned data value MUST start on an appropriate CPU word-aligned address.
4. `MEM_VAL_COPY_GET_xxx()` macro's are more efficient than MEM_VAL_GET_xxx() macro's & are also independent of CPU data-word-alignment & SHOULD be used whenever possible.
5. `MEM_VAL_GET_xxx()` macro's are NOT atomic operations & MUST NOT be used on any non-static (i.e., volatile) variables, registers, hardware, etc.; without the caller of the macro's providing some form of additional protection (e.g., mutual exclusion).
6. The ' `CPU_CFG_ENDIAN_TYPE` ' pre-processor 'else'-conditional code SHOULD never be compiled/ linked since each 'cpu.h' SHOULD ensure that the CPU data-word-memory order configuration constant (`CPU_CFG_ENDIAN_TYPE`) is configured with an appropriate data-word-memory order value (see 'cpu.h CPU WORD CONFIGURATION Note #2'). The 'else'-conditional code is included as an extra precaution in case 'cpu.h' is incorrectly configured.

MEM_VAL_SET_INTxxU_xxx()

Description

Encode data values to any CPU memory address.

Files

lib_mem.h/lib_mem.c

Prototype

```
MEM_VAL_SET_INT08U (addr, val)

MEM_VAL_SET_INT16U (addr, val)

MEM_VAL_SET_INT32U (addr, val)

MEM_VAL_SET_INT64U (addr, val)

MEM_VAL_SET_INT08U_BIG (addr, val)

MEM_VAL_SET_INT16U_BIG (addr, val)

MEM_VAL_SET_INT32U_BIG (addr, val)

MEM_VAL_SET_INT64U_BIG (addr, val)

MEM_VAL_SET_INT08U_LITTLE (addr, val)

MEM_VAL_SET_INT16U_LITTLE (addr, val)

MEM_VAL_SET_INT32U_LITTLE (addr, val)

MEM_VAL_SET_INT64U_LITTLE (addr, val)
```

Arguments

`addr`

Lowest CPU memory address to encode data value (see Notes #2 & #3a).

`val`

Data value to encode (see Notes #1 & #3b).

Returned Value

None.

Notes / Warnings

1. Encode data values into CPU memory based on the values' data-word order :
2. CPU memory addresses/pointers NOT checked for NULL.
3. (a) `MEM_VAL_SET_xxx()` macro's encode data values without regard to CPU word-aligned addresses. Thus for processors that require data word alignment, data words can be encoded to any CPU address, word-aligned or not, without generating data-word-alignment exceptions/faults.
   (b) However, '`val`' data value to encode MUST start on an appropriate CPU word-aligned address.
4. `MEM_VAL_COPY_SET_xxx()` macro's are more efficient than `MEM_VAL_SET_xxx()` macro's & are also independent of CPU data-word-alignment & SHOULD be used whenever possible.
5. `MEM_VAL_SET_xxx()` macro's are NOT atomic operations & MUST NOT be used on any non-static (i.e., volatile) variables, registers, hardware, etc.; without the caller of the macro's providing some form of additional protection (e.g., mutual exclusion).
6. The '`CPU_CFG_ENDIAN_TYPE`' pre-processor 'else'-conditional code SHOULD never be compiled/ linked since each '`cpu.h`' SHOULD ensure that the CPU data-word-memory order configuration constant (CPU_CFG_ENDIAN_TYPE) is configured with an appropriate data-word-memory order value (see '`cpu.h` CPU WORD CONFIGURATION Note #2'). The 'else'-conditional code is included as an extra precaution in case '`cpu.h`' is incorrectly configured.

MEM_VAL_COPY_GET_INTxxU_xxx()

Description

Copy & decode data values from any CPU memory address to any CPU memory address.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
MEM_VAL_COPY_GET_INTxxU (addr_dest, addr_src)

MEM_VAL_COPY_GET_INTxxU_BIG (addr_dest, addr_src)

MEM_VAL_COPY_GET_INTxxU_LITTLE (addr_dest, addr_src)
```

Arguments

`addr_dest`

Lowest CPU memory address to copy/decode source address's data value (see Notes #2 & #3).

`addr_src`

Lowest CPU memory address of data value to copy/decode (see Notes #2 & #3).

Returned Value

None.

Notes / Warnings

1. Copy/decode data values based on the values' data-word order :
2. (a) CPU memory addresses/pointers NOT checked for NULL.
   (b) CPU memory addresses/buffers NOT checked for overlapping.
   1. IEEE Std 1003.1, 2004 Edition, Section '`memcpy()` : DESCRIPTION' states that "copying ... between objects that overlap ... is undefined".
3. `MEM_VAL_COPY_GET_xxx()` macro's copy/decode data values without regard to CPU word-aligned addresses. Thus for processors that require data word alignment, data words can be copied/ decoded to/from any CPU address, word-aligned or not, without generating data-word-alignment exceptions/faults.
4. `MEM_VAL_COPY_GET_xxx()` macro's are more efficient than `MEM_VAL_GET_xxx()` macro's & are also independent of CPU data-word-alignment & SHOULD be used whenever possible.
5. Since octet-order copy/conversion are inverse operations, `MEM_VAL_COPY_GET_xxx()` & `MEM_VAL_COPY_SET_xxx()` macros are inverse, but identical, operations & are provided in both forms for semantics & consistency.
6. `MEM_VAL_COPY_GET_xxx()` macro's are NOT atomic operations & MUST NOT be used on any non- static (i.e., volatile) variables, registers, hardware, etc.; without the caller of the macro's providing some form of additional protection (e.g., mutual exclusion).
7. The '`CPU_CFG_ENDIAN_TYPE`' pre-processor 'else'-conditional code SHOULD never be compiled/ linked since each 'cpu.h' SHOULD ensure that the CPU data-word-memory order configuration constant (`CPU_CFG_ENDIAN_TYPE`) is configured with an appropriate data-word-memory order value (see 'cpu.h CPU WORD CONFIGURATION Note #2'). The 'else'-conditional code is included as an extra precaution in case 'cpu.h' is incorrectly configured.

MEM_VAL_COPY_GET_INTU_xxx()

Description

Copy & decode data values from any CPU memory address to any CPU memory address for any sized data values.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
MEM_VAL_COPY_GET_INTU (addr_dest, addr_src, val_size)

MEM_VAL_COPY_GET_INTU_BIG (addr_dest, addr_src, val_size)

MEM_VAL_COPY_GET_INTU_LITTLE (addr_dest, addr_src, val_size)
```

Arguments

`addr_dest`

Lowest CPU memory address to copy/decode source address's data value (see Notes #2 & #3).

`addr_src`

Lowest CPU memory address of data value to copy/decode (see Notes #2 & #3).

`val_size`

Number of data value octets to copy/decode.

Returned Value

None.

Notes / Warnings

1. Copy/decode data values based on the values' data-word order :
2. (a) CPU memory addresses/pointers NOT checked for NULL.
   (b) CPU memory addresses/buffers NOT checked for overlapping.
   1. IEEE Std 1003.1, 2004 Edition, Section ' `memcpy()` : DESCRIPTION' states that "copying ... between objects that overlap ... is undefined".
3. `MEM_VAL_COPY_GET_INTU_xxx()` macro's copy/decode data values without regard to CPU word- aligned addresses. Thus for processors that require data word alignment, data words can be copied/decoded to/from any CPU address, word-aligned or not, without generating data-word-alignment exceptions/faults.
4. `MEM_VAL_COPY_GET_xxx()` macro's are more efficient than `MEM_VAL_COPY_GET_INTU_xxx()` macro's & SHOULD be used whenever possible.
5. Since octet-order copy/conversion are inverse operations, `MEM_VAL_COPY_GET_INTU_xxx()` & `MEM_VAL_COPY_SET_INTU_xxx()` macros are inverse, but identical, operations & are provided in both forms for semantics & consistency.
6. `MEM_VAL_COPY_GET_INTU_xxx()` macro's are NOT atomic operations & MUST NOT be used on any non-static (i.e., volatile) variables, registers, hardware, etc.; without the caller of the macro's providing some form of additional protection (e.g., mutual exclusion).
7. MISRA-C 2004 Rule 5.2 states that "identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier".
8. The ' `CPU_CFG_ENDIAN_TYPE` ' pre-processor 'else'-conditional code SHOULD never be compiled/ linked since each ' `cpu.h` ' SHOULD ensure that the CPU data-word-memory order configuration constant (`CPU_CFG_ENDIAN_TYPE`) is configured with an appropriate data-word-memory order value (see ' `cpu.h` CPU WORD CONFIGURATION Note #2'). The 'else'-conditional code is included as an extra precaution in case ' `cpu.h` ' is incorrectly configured.

MEM_VAL_COPY_SET_INTxxU_xxx()

Description

Copy & encode data values from any CPU memory address to any CPU memory address.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
MEM_VAL_COPY_SET_INTxxU (addr_dest, addr_src)

MEM_VAL_COPY_SET_INTxxU_BIG (addr_dest, addr_src)

MEM_VAL_COPY_SET_INTxxU_LITTLE (addr_dest, addr_src)
```

Arguments

`addr_dest`

Lowest CPU memory address to copy/encode source address's data value (see Notes #2 & #3).

`addr_src`

Lowest CPU memory address of data value to copy/encode (see Notes #2 & #3).

Returned Value

None.

Notes / Warnings

1. Copy/encode data values based on the values' data-word order :
2. (a) CPU memory addresses/pointers NOT checked for NULL.
   (b) CPU memory addresses/buffers NOT checked for overlapping.
   1. IEEE Std 1003.1, 2004 Edition, Section '`memcpy()` : DESCRIPTION' states that "copying ... between objects that overlap ... is undefined".
3. `MEM_VAL_COPY_SET_xxx()` macro's copy/encode data values without regard to CPU word-aligned addresses. Thus for processors that require data word alignment, data words can be copied/ encoded to/from any CPU address, word-aligned or not, without generating data-word-alignment exceptions/faults.
4. `MEM_VAL_COPY_SET_xxx()` macro's are more efficient than `MEM_VAL_SET_xxx()` macro's & are also independent of CPU data-word-alignment & SHOULD be used whenever possible.
5. Since octet-order copy/conversion are inverse operations, `MEM_VAL_COPY_GET_xxx()` & `MEM_VAL_COPY_SET_xxx()` macros are inverse, but identical, operations & are provided in both forms for semantics & consistency.
6. `MEM_VAL_COPY_SET_xxx()` macro's are NOT atomic operations & MUST NOT be used on any non-static (i.e., volatile) variables, registers, hardware, etc.; without the caller of the macro's providing some form of additional protection (e.g., mutual exclusion).

MEM_VAL_COPY_SET_INTU_xxx()

Description

Copy & encode data values from any CPU memory address to any CPU memory address for any sized data values.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
MEM_VAL_COPY_SET_INTU (addr_dest, addr_src, val_size)

MEM_VAL_COPY_SET_INTU_BIG (addr_dest, addr_src, val_size)

MEM_VAL_COPY_SET_INTU_LITTLE (addr_dest, addr_src, val_size)
```

Arguments

`addr_dest`

Lowest CPU memory address to copy/encode source address's data value (see Notes #2 & #3).

`addr_src`

Lowest CPU memory address of data value to copy/encode (see Notes #2 & #3).

`val_size`

Number of data value octets to copy/encode.

Returned Value

None.

Notes / Warnings

1. Copy/encode data values based on the values' data-word order :
2. (a) CPU memory addresses/pointers NOT checked for NULL.
   (b) CPU memory addresses/buffers NOT checked for overlapping.
   1. IEEE Std 1003.1, 2004 Edition, Section '`memcpy()` : DESCRIPTION' states that "copying ... between objects that overlap ... is undefined".
3. `MEM_VAL_COPY_SET_INTU_xxx()` macro's copy/encode data values without regard to CPU word- aligned addresses. Thus for processors that require data word alignment, data words can be copied/encoded to/from any CPU address, word-aligned or not, without generating data-word-alignment exceptions/faults.
4. `MEM_VAL_COPY_SET_xxx()` macro's are more efficient than `MEM_VAL_COPY_SET_INTU_xxx()` macro's & SHOULD be used whenever possible.
5. Since octet-order copy/conversion are inverse operations, `MEM_VAL_COPY_GET_INTU_xxx()` & `MEM_VAL_COPY_SET_INTU_xxx()` macros are inverse, but identical, operations & are provided in both forms for semantics & consistency.
6. `MEM_VAL_COPY_SET_INTU_xxx()` macro's are NOT atomic operations & MUST NOT be used on any non-static (i.e., volatile) variables, registers, hardware, etc.; without the caller of the macro's providing some form of additional protection (e.g., mutual exclusion).

MEM_VAL_COPY_xx()

Description

Copy data values from any CPU memory address to any CPU memory address.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
MEM_VAL_COPY_08 (addr_dest, addr_src)

MEM_VAL_COPY_16 (addr_dest, addr_src)

MEM_VAL_COPY_32 (addr_dest, addr_src)

MEM_VAL_COPY (addr_dest, addr_src, val_size)
```

Arguments

`addr_dest`

Lowest CPU memory address to copy source address's data value (see Notes #2 & #3).

`addr_src`

Lowest CPU memory address of data value to copy (see Notes #2 & #3).

`val_size`

Number of data value octets to copy. Present only for the `MEM_VAL_COPY()` macro.

Returned Value

None.

Notes / Warnings

1. `MEM_VAL_COPY_xxx()` macro's copy data values based on CPU's native data-word order.
2. (a) CPU memory addresses/pointers NOT checked for NULL.
   (b) CPU memory addresses/buffers NOT checked for overlapping.
   1. IEEE Std 1003.1, 2004 Edition, Section ' `memcpy()` : DESCRIPTION' states that "copying ... between objects that overlap ... is undefined".
3. `MEM_VAL_COPY_xxx()` macro's copy data values without regard to CPU word-aligned addresses. Thus for processors that require data word alignment, data words can be copied to/from any CPU address, word-aligned or not, without generating data-word-alignment exceptions/faults.
4. `MEM_VAL_COPY_xxx()` macro's are more efficient than `MEM_VAL_COPY()` macro & SHOULD be used whenever possible.
5. `MEM_VAL_COPY_xxx()` macro's are NOT atomic operations & MUST NOT be used on any non-static (i.e., volatile) variables, registers, hardware, etc.; without the caller of the macro's providing some form of additional protection (e.g., mutual exclusion).
6. MISRA-C 2004 Rule 5.2 states that "identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier".

API LIB String

- Str_Len()
- Str_Len_N()
- Str_Copy()
- Str_Copy_N()
- Str_Cat()
- Str_Cat_N()
- Str_Cmp()
- Str_Cmp_N()
- Str_CmpIgnoreCase()
- Str_CmpIgnoreCase_N()
- Str_Char()
- Str_Char_N()
- Str_Char_Last()

- Str_Char_Last_N()
- Str_Char_Replace()
- Str_Char_Replace_N()
- Str_Str()
- Str_Str_N()
- Str_Printf()
- Str_FmtNbr_Int32U()
- Str_FmtNbr_Int32S()
- Str_FmtNbr_32()
- Str_ParseNbr_Int32U()
- Str_ParseNbr_Int32S()

Str_Len()

Description

Calculates the length of a string.

Files

`lib_str.h/lib_str.c`

Prototype

```
CPU_SIZE_T Str_Len (const CPU_CHAR *p_str)
```

Arguments

`p_str`

Pointer to the string (see Note #1).

Returned Value

Length of a string; number of characters in a string before the terminating NULL character (see Note #2b1).

Notes / Warnings

1. String buffer NOT modified.
2. (a) IEEE Std 1003.1, 2004 Edition, Section '`strlen()` : DESCRIPTION' states that :
    1. "The `strlen()` function shall compute the number of bytes in the string to which 's' ('p_str') points," ...
    2. "not including the terminating null byte."

   (b) IEEE Std 1003.1, 2004 Edition, Section '`strlen()` : RETURN VALUE' states that :
    1. "The `strlen()` function shall return the length of 's' ('p_str');" ...
    2. "no return value shall be reserved to indicate an error."
3. String length calculation terminates when :

   (a) String pointer points to NULL.
    1. String buffer overlaps with NULL address.
    2. String length calculated for string up to but NOT beyond or including the NULL address.

   (b) Terminating NULL character found.
    1. String length calculated for string up to but NOT including the NULL character (see Note #2a2).

Str_Len_N()

Description

Calculates the length of a string, up to a maximum number of characters.

Files

`lib_str.h/lib_str.c`

Prototype

```
CPU_SIZE_T Str_Len_N (const CPU_CHAR   *p_str,
                      CPU_SIZE_T   len_max)
```

Arguments

`p_str`

Pointer to the string (see Note #1).

`len_max`

Maximum number of characters to search (see Note #3c).

Returned Value

Length of string; number of characters in string before terminating NULL character, if terminating NULL character found (see Note #2b1).

Requested maximum number of characters to search, if terminating NULL character NOT found (see Note #3c).

Notes / Warnings

1. String buffer NOT modified.
2. (a) IEEE Std 1003.1, 2004 Edition, Section ' `strlen()` : DESCRIPTION' states that :
    1. "The `strlen()` function shall compute the number of bytes in the string to which 's' ('p_str') points," ...
    2. "not including the terminating null byte."

   (b) IEEE Std 1003.1, 2004 Edition, Section ' `strlen()` : RETURN VALUE' states that :
    1. "The `strlen()` function shall return the length of 's' (' `p_str` ');" ...
    2. "no return value shall be reserved to indicate an error."
3. String length calculation terminates when :

   (a) String pointer points to NULL.
    1. String buffer overlaps with NULL address.
    2. String length calculated for string up to but NOT beyond or including the NULL address.

   (b) Terminating NULL character found.
    1. String length calculated for string up to but NOT including the NULL character (see Note #2a2).

   (c) ' `len_max` ' number of characters searched.
    1. ' `len_max` ' number of characters does NOT include the terminating NULL character.

Str_Copy()

Description

Copies the source string to destination string buffer.

Files

`lib_str.h/lib_str.c`

Prototype

```
CPU_CHAR *Str_Copy (      CPU_CHAR  *p_str_dest,
                    const CPU_CHAR  *p_str_src)
```

Arguments

`p_str_dest`

Pointer to the destination string buffer to receive source string copy (see Note #1a).

`p_str_src`

Pointer to the source string to copy into destination string buffer (see Note #1b).

Returned Value

Pointer to the destination string, if NO error(s) [see Note #2b1].

Pointer to NULL, if any errors occur (see Note #2b2A).

Notes / Warnings

1. (a) Destination buffer size is NOT validated; buffer overruns MUST be prevented by caller.
    1. Destination buffer size MUST be large enough to accommodate the entire source string size including the terminating NULL character.

   (b) Source buffer NOT modified.
2. (a) IEEE Std 1003.1, 2004 Edition, Section '`strcpy()` : DESCRIPTION' states that :
    1. "The `strcpy()` function shall copy the string pointed to by 's2' ('`p_str_src`') ... into the array pointed to by 's1' ('`p_str_dest`')" ...
    2. "(including the terminating null byte)."

   (b) IEEE Std 1003.1, 2004 Edition, Section '`strcpy()` : RETURN VALUE' states that :
    1. "The `strcpy()` function shall return 's1' ('`p_str_dest`');" ...
    2. "no return value is reserved to indicate an error."
       (A) #### This requirement is intentionally NOT implemented in order to return NULL for any error(s).

   (c) IEEE Std 1003.1, 2004 Edition, Section '`strcpy()` : DESCRIPTION' states that "if copying takes place between objects that overlap, the behavior is undefined".
3. String copy terminates when :
   (a) Destination/Source string pointer(s) are passed as NULL pointers.
    1. No string copy performed; NULL pointer is returned.

   (b) Destination/Source string pointer(s) point to NULL.
    1. String buffer(s) overlap with NULL address; NULL pointer is returned.

   (c) Source string's terminating NULL character found.
    1. Entire source string copied into destination string buffer (see Note #2a).

Str_Copy_N()

Description

Copies the source string to the destination string buffer, up to a maximum number of characters.

Files

`lib_str.h/lib_str.c`

Prototype

```
CPU_CHAR *Str_Copy_N (      CPU_CHAR   *p_str_dest,
                      const CPU_CHAR   *p_str_src,
                            CPU_SIZE_T  len_max)
```

Arguments

`p_str_dest`

Pointer to the destination string buffer to receive the source string copy (see Note #1a).

`p_str_src`

Pointer to the source string to copy into the destination string buffer (see Note #1b).

`len_max`

Maximum number of characters to copy (see Notes #2a2 and #3d).

Returned Value

Pointer to destination string, if NO error(s) [see Note #2b1].

Pointer to NULL, if any errors occur (see Note #2b2A).

Notes / Warnings

1. (a) Destination buffer size is NOT validated; buffer overruns MUST be prevented by caller.
    1. Destination buffer size MUST be large enough to accommodate the entire source string size including the terminating NULL character.

   (b) Source string buffer NOT modified.
2. (a)
    1. IEEE Std 1003.1, 2004 Edition, Section ' `strncpy()` : DESCRIPTION' states that :
       (A) "The `strncpy()` function shall copy ... the array pointed to by 's2' (' `p_str_src` ') to the array pointed to by 's1' (' `p_str_dest` ')";
       (B) but "not more than 'n' (' `len_max` ') bytes"
       (C) & "(bytes that follow a null byte are not copied)".
    2. (A) IEEE Std 1003.1, 2004 Edition, Section ' `strncpy()` : DESCRIPTION' adds that "if the array pointed to by 's2' (' `p_str_src` ') is a string that is shorter than 'n' (' `len_max` ') bytes, null bytes shall be appended to the copy in the array pointed to by 's1' (' `p_str_dest` '), until 'n' (' `len_max` ') bytes in all are written."
        1. #### Since Str_Copy() limits the maximum number of characters to copy via Str_Copy_N() by the CPU's maximum number of addressable characters, this requirement is intentionally NOT implemented to avoid appending a potentially large number of unnecessary terminating NULL characters.

       (B) IEEE Std 1003.1, 2004 Edition, Section ' `strncpy()` : APPLICATION USAGE' also states that "if there is no null byte in the first 'n' (' `len_max` ') bytes of the array pointed to by ' `s2` ' (' `p_str_src` '), the result is not null-terminated".

   (b) IEEE Std 1003.1, 2004 Edition, Section ' `strncpy()` : RETURN VALUE' states that :
    1. "The `strncpy()` function shall return ' `s1` ' (' `p_str_dest` ');"
    2. "no return value is reserved to indicate an error."
       (A) #### This requirement is intentionally ignored in order to return NULL for any error(s).

   (c) IEEE Std 1003.1, 2004 Edition, Section ' `strncpy()` : DESCRIPTION' states that "if copying takes place between objects that overlap, the behavior is undefined".
3. String copy terminates when :
   (a) Destination/Source string pointer(s) are passed as NULL pointers.
    1. No string copy performed; NULL pointer is returned.

   (b) Destination/Source string pointer(s) point to NULL.
    1. String buffer(s) overlap with NULL address; NULL pointer is returned.

   (c) Source string's terminating NULL character found.
    1. Entire source string copied into destination string buffer (see Note #2a1A).

   (d) ' `len_max` ' number of characters copied.
    1. ' `len_max` ' number of characters MAY include the terminating NULL character (see Note #2a1C).

(A) No string copy performed; destination string returned (see Note #2b1).

Str_Cat()

Description

Appends the concatenation string to the destination string.

Files

`lib_str.h/lib_str.c`

Prototype

```
CPU_CHAR *Str_Cat (      CPU_CHAR  *p_str_dest,                const CPU_CHAR  *p_str_cat)
```

Arguments

`p_str_dest`

Pointer to the destination string to append concatenation string (see Note #1a).

`p_str_cat`

Pointer to the concatenation string to append to destination string (see Note #1b).

Returned Value

Pointer to the destination string, if NO error(s) [see Note #2b1].

Pointer to NULL, if any errors occur (see Note #2b2A).

Notes / Warnings

1. (a) Destination buffer size is NOT validated; buffer overruns MUST be prevented by caller.
    1. Destination buffer size MUST be large enough to accommodate the entire concatenated string size including the terminating NULL character.

   (b) Concatenation string buffer NOT modified.
2. (a) IEEE Std 1003.1, 2004 Edition, Section ' `strcat()` : DESCRIPTION' states that :
    1. "The `strcat()` function shall append a copy of the string pointed to by 's2' (' `p_str_cat` ') ... to the end of the string pointed to by 's1' (' `p_str_dest` ')."
    2. (A) "The initial byte of 's2' (' `p_str_cat` ') overwrites the null byte at the end of 's1' (' `p_str_dest` ')."
       (B) A "terminating null byte" is appended at the end of the concatenated destination strings.

   (b) IEEE Std 1003.1, 2004 Edition, Section ' `strcat()` : RETURN VALUE' states that :
    1. "The `strcat()` function shall return 's1' (' `p_str_dest` ');" ...
    2. "no return value shall be reserved to indicate an error."
       (A) #### This requirement is intentionally NOT implemented in order to return NULL for any error(s).

   (c) IEEE Std 1003.1, 2004 Edition, Section ' `strcat()` : DESCRIPTION' states that "if copying takes place between objects that overlap, the behavior is undefined."
3. String concatenation terminates when :
   (a) Destination/Concatenation string pointer(s) are passed as NULL pointers.
    1. No string concatenation performed; NULL pointer is returned.

   (b) Destination/Concatenation string pointer(s) point to NULL.
    1. String buffer(s) overlap with NULL address; NULL pointer is returned.

   (c) Concatenation string's terminating NULL character found.
    1. Entire concatenation string appended to destination string (see Note #2a1).

Str_Cat_N()

Description

Appends concatenation string to destination string, up to a maximum number of characters.

Files

`lib_str.h/lib_str.c`

Prototype

```
CPU_CHAR *Str_Cat_N (      CPU_CHAR   *p_str_dest,
                     const CPU_CHAR   *p_str_cat,
                           CPU_SIZE_T  len_max)
```

Arguments

`p_str_dest`

Pointer to the destination string to append concatenation string (see Note #1a).

`p_str_cat`

Pointer to the concatenation string to append to destination string (see Note #1b).

`len_max`

Maximum number of characters to concatenate (see Notes #2a1B and #3d).

Returned Value

Pointer to the destination string, if NO error(s) [see Note #2b1].

Pointer to NULL, if any errors occur (see Note #2b2A).

Notes / Warnings

1. (a) Destination buffer size is NOT validated; buffer overruns MUST be prevented by caller.
    1. Destination buffer size MUST be large enough to accommodate the entire concatenated string size including the terminating NULL character.

   (b) Concatenation string buffer NOT modified.
2. (a) IEEE Std 1003.1, 2004 Edition, Section ' `strncat()` : DESCRIPTION' states that :
    1. (A) "The `strncat()` function shall append ... the array pointed to by 's2' (' `p_str_cat` ') to the end of the string pointed to by 's1' (' `p_str_dest` ')" ...

       (B) but "not more than 'n' ('len_max') bytes".
    2. (A) "The initial byte of 's2' (' `p_str_cat` ') overwrites the null byte at the end of 's1' (' `p_str_dest` ')."

       (B) "(a null byte and bytes that follow it are not appended)."

       (C) "A terminating null byte is always appended to the result."

   (b) IEEE Std 1003.1, 2004 Edition, Section ' `strncat()` : RETURN VALUE' states that :
    1. "The `strncat()` function shall return 's1' ('p_str_dest');" ...
    2. "no return value shall be reserved to indicate an error."

       (A) #### This requirement is intentionally NOT implemented in order to return NULL for any error(s).

   (c) IEEE Std 1003.1, 2004 Edition, Section ' `strncat()` : DESCRIPTION' states that "if copying takes place between objects that overlap, the behavior is undefined."
3. String concatenation terminates when :

   (a) Destination/Concatenation string pointer(s) are passed as NULL pointers.
    1. No string concatenation performed; NULL pointer is returned.

   (b) Destination/Concatenation string pointer(s) point to NULL.
    1. String buffer(s) overlap with NULL address; NULL pointer is returned.

   (c) Concatenation string's terminating NULL character found.

1. Entire concatenation string appended to destination string (see Note #2a1A).
   (d) ' len_max ' number of characters concatenated.
   1. ' len_max ' number of characters does NOT include the terminating NULL character (see Note #2a2).
   2. Null concatenations allowed (i.e. zero-length concatenations).
      (A) No string concatenation performed; destination string returned (see Note #2b1).

Str_Cmp()

Description

Determines if two strings are identical.

Files

lib_str.h/lib_str.c

Prototype

```
CPU_INT16S Str_Cmp (const CPU_CHAR  *p1_str,
                    const CPU_CHAR  *p2_str)
```

Arguments

p1_str

Pointer to the first string (see Note #1).

p2_str

Pointer to the second string (see Note #1).

Returned Value

0, if strings are identical (see Notes #3a1A, #3a2A, and #3b).

Negative value, if ' p1_str ' is less than ' p2_str ' (see Notes #3a1B1, #3a2B1, and #3c).

Positive value, if ' p1_str ' is greater than ' p2_str ' (see Notes #3a1B2, #3a2B2, and #3c).

See also Note #2b.

Notes / Warnings

1. String buffers are NOT modified.
2. (a) IEEE Std 1003.1, 2004 Edition, Section ' strcmp() : DESCRIPTION' states that "the strcmp() function shall compare the string pointed to by 's1' (' p1_str ') to the string pointed to by 's2' (' p2_str)".
   (b)
   1. IEEE Std 1003.1, 2004 Edition, Section ' strcmp() : RETURN VALUE' states that "upon successful completion, strcmp() shall return an integer greater than, equal to, or less than 0".
   2. IEEE Std 1003.1, 2004 Edition, Section ' strcmp() : DESCRIPTION' adds that "the sign of a non-zero return value shall be determined by the sign of the difference between the values of the first pair of bytes ... that differ in the strings being compared".
3. String comparison terminates when :
   (a)
   1. (A) BOTH string pointer(s) are passed as NULL pointers.
         1. NULL strings identical; 0 returned.
      (B)
         1. ' p1_str ' passed a NULL pointer.
            (a) Return negative value of character pointed to by ' p2_str '.
         2. ' p2_str ' passed a NULL pointer.

(a) Return positive value of character pointed to by ' p1_str '.
2. (A) BOTH strings point to NULL.
   1. Strings overlap with NULL address.
   2. Strings identical up to but NOT beyond or including the NULL address; 0 returned.

   (B)
   1. ' p1_str_cmp_next ' points to NULL.
      (a) ' p1_str ' overlaps with NULL address.
      (b) Strings compared up to but NOT beyond or including the NULL address.
      (c) Return negative value of character pointed to by ' p2_str_cmp_next '.
   2. ' p2_str_cmp_next ' points to NULL.
      (a) ' p2_str ' overlaps with NULL address.
      (b) Strings compared up to but NOT beyond or including the NULL address.
      (c) Return positive value of character pointed to by ' p1_str_cmp_next '.

(b) Terminating NULL character found in both strings.
1. Strings identical; 0 returned.
2. Only one NULL character test required in conditional since the previous condition tested character equality.

(c) Non-matching characters found.
1. Return signed-integer difference of the character pointed to by ' p2_str ' from the character pointed to by ' p1_str '.

4. Since 16-bit signed arithmetic is performed to calculate a non-identical comparison return value, ' CPU_CHAR ' native data type size MUST be 8-bit.

Str_Cmp_N()

Description

Determines if two strings are identical for up to a maximum number of characters.

Files

lib_str.h/lib_str.c

Prototype

```
CPU_INT16S Str_Cmp_N (const CPU_CHAR    *p1_str,
                      const CPU_CHAR    *p2_str,
                            CPU_SIZE_T    len_max)
```

Arguments

p1_str

Pointer to the first string (see Note #1).

p2_str

Pointer to the second string (see Note #1).

len_max

Maximum number of characters to compare (see Note #3d).

Returned Value

0, if strings are identical (see Notes #3a1A, #3a2A, #3b, and #3d).

Negative value, if ' p1_str ' is less than ' p2_str ' (see Notes #3a1B1, #3a2B1, and #3c).

Positive value, if ' p1_str ' is greater than ' p2_str ' (see Notes #3a1B2, #3a2B2, and #3c).

See also Note #2b.

Notes / Warnings

1. String buffers are NOT modified.
2. (a) IEEE Std 1003.1, 2004 Edition, Section ' `strncmp()` : DESCRIPTION' states that :
    1. "The `strncmp()` function shall compare ... the array pointed to by 's1' (' `p1_str` ') to the array pointed to by 's2' (' `p2_str`)" ...
    2. but "not more than 'n' (' `len_max` ') bytes" of either array.

   (b)
    1. IEEE Std 1003.1, 2004 Edition, Section ' `strncmp()` : RETURN VALUE' states that "upon successful completion, `strncmp()` shall return an integer greater than, equal to, or less than 0".
    2. IEEE Std 1003.1, 2004 Edition, Section ' `strncmp()` : DESCRIPTION' adds that "the sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of bytes ... that differ in the strings being compared".
    3. String comparison terminates when :

       (a)
        1. (A) BOTH string pointer(s) are passed as NULL pointers.
            1. NULL strings identical; 0 returned.

           (B)
            1. ' `p1_str` ' passed a NULL pointer.
               (a) Return negative value of character pointed to by ' `p2_str` '.
            2. ' `p2_str` ' passed a NULL pointer.
               (a) Return positive value of character pointed to by ' `p1_str` '.
        2. (A) BOTH strings point to NULL.
            1. Strings overlap with NULL address.
            2. Strings identical up to but NOT beyond or including the NULL address; 0 returned.

           (B)
            1. ' `p1_str_cmp_next` ' points to NULL.
               (a) ' `p1_str` ' overlaps with NULL address.
               (b) Strings compared up to but NOT beyond or including the NULL address.
               (c) Return negative value of character pointed to by ' `p2_str_cmp_next` '.
            2. ' `p2_str_cmp_next` ' points to NULL.
               (a) ' `p2_str` ' overlaps with NULL address.
               (b) Strings compared up to but NOT beyond or including the NULL address.
               (c) Return positive value of character pointed to by ' `p1_str_cmp_next` '.

       (b) Terminating NULL character found in both strings.
        1. Strings identical; 0 returned.
        2. Only one NULL character test required in conditional since the previous condition tested character equality.

       (c) Non-matching characters found.
        1. Return signed-integer difference of the character pointed to by ' `p2_str` ' from the character pointed to by ' `p1_str` '.

       (d)
        1. ' `len_max` ' passed a zero length.
           (A) Zero-length strings identical; 0 returned.
        2. First ' `len_max` ' number of characters identical.
           (A) Strings identical; 0 returned.

       See also Note #2a2.
3. Since 16-bit signed arithmetic is performed to calculate a non-identical comparison return value, ' `CPU_CHAR` ' native data type size MUST be 8-bit.

Str_CmpIgnoreCase()

Description

Determines if two strings are identical, ignoring case.

Files

`lib_str.h/lib_str.c`

Prototype

```
CPU_INT16S Str_CmpIgnoreCase (const CPU_CHAR   *p1_str,
                              const CPU_CHAR   *p2_str)
```

Arguments

`p1_str`

Pointer to the first string (see Note #1).

`p2_str`

Pointer to the second string (see Note #1).

Returned Value

0, if strings are identical (see Notes #3a1A, #3a2A, and #3b).

Negative value, if '`p1_str`' is less than '`p2_str`' (see Notes #3a1B1, #3a2B1, and #3c).

Positive value, if '`p1_str`' is greater than '`p2_str`' (see Notes #3a1B2, #3a2B2, and #3c).

See also Note #2b.

Notes / Warnings

1. String buffers are NOT modified.
2. (a) IEEE Std 1003.1, 2004 Edition, Section '`strcasecmp()` : DESCRIPTION' states that :
    1. (A) "The `strcasecmp()` function shall compare ... the string pointed to by '`s1`' ('`p1_str`') to the string pointed to by 's2' ('p2_str')" ...
       (B) "ignoring differences in case".
    2. "`strcasecmp()` ... shall behave as if the strings had been converted to lowercase and then a byte comparison performed."

   (b)
    1. IEEE Std 1003.1, 2004 Edition, Section '`strcasecmp()` : RETURN VALUE' states that "upon successful completion, `strcasecmp()` shall return an integer greater than, equal to, or less than 0".
    2. IEEE Std 1003.1, 2004 Edition, Section '`strcmp()` : DESCRIPTION' adds that "the sign of a non-zero return value shall be determined by the sign of the difference between the values of the first pair of bytes ... that differ in the strings being compared".
3. String comparison terminates when :
   (a)
    1. (A) BOTH string pointer(s) are passed as NULL pointers.
        1. NULL strings identical; 0 returned.

       (B)
        1. '`p1_str`' passed a NULL pointer.
           (a) Return negative value of character pointed to by '`p2_str`', converted to lower case (see Note #2a2).
        2. '`p2_str`' passed a NULL pointer.
           (a) Return positive value of character pointed to by '`p1_str`', converted to lower case (see Note #2a2).
    2. (A) BOTH strings point to NULL.
        1. Strings overlap with NULL address.
        2. Strings identical up to but NOT beyond or including the NULL address; 0 returned.

       (B)
        1. '`p1_str_cmp_next`' points to NULL.
           (a) '`p1_str`' overlaps with NULL address.
           (b) Strings compared up to but NOT beyond or including the NULL address.
           (c) Return negative value of character pointed to by '`p2_str_cmp_next`', converted to lower case (see Note #2a2).

'p2_str_cmp_next' points to NULL.
(a) 'p2_str' overlaps with NULL address.
(b) Strings compared up to but NOT beyond or including the NULL address.
(c) Return positive value of character pointed to by 'p1_str_cmp_next', converted to lower case (see Note #2a2).
(b) Terminating NULL character found in both strings.
1. Strings identical; 0 returned.
2. Only one NULL character test required in conditional since the previous condition tested character equality.
(c) Non-matching characters found.
1. Return signed-integer difference of the character pointed to by 'p2_str', converted to lower case, from the character pointed to by 'p1_str', converted to lower case.

4. Since 16-bit signed arithmetic is performed to calculate a non-identical comparison return value, 'CPU_CHAR' native data type size MUST be 8-bit.

Str_CmpIgnoreCase_N()

Description

Determines if two strings are identical for up to a maximum number of characters, ignoring case.

Files

lib_str.h/lib_str.c

Prototype

```
CPU_INT16S Str_CmpIgnoreCase_N (const CPU_CHAR    *p1_str,
                                const CPU_CHAR    *p2_str,
                                      CPU_SIZE_T   len_max)
```

Arguments

p1_str

Pointer to the first string (see Note #1).

p2_str

Pointer to the second string (see Note #1).

len_max

Maximum number of characters to compare (see Note #3d).

Returned Value

0, if strings are identical (see Notes #3a1A, #3a2A, #3b, and #3d).

Negative value, if 'p1_str' is less than 'p2_str' (see Notes #3a1B1, #3a2B1, and #3c).

Positive value, if 'p1_str' is greater than 'p2_str' (see Notes #3a1B2, #3a2B2, and #3c).

See also Note #2b.

Notes / Warnings

1. String buffers are NOT modified.
2. (a) IEEE Std 1003.1, 2004 Edition, Section 'strncasecmp() : DESCRIPTION' states that :
   1. (A) "The strncasecmp() function shall compare ... the string pointed to by 's1' ('p1_str') to the string pointed to by 's2' ('p2_str')" ...
   (B) "ignoring differences in case" ...
   (C) but "not more than 'n' ('len_max') bytes" of either string.

    2. " `strncasecmp()` shall behave as if the strings had been converted to lowercase and then a byte comparison performed."
  (b)
    1. IEEE Std 1003.1, 2004 Edition, Section ' `strncasecmp()` : RETURN VALUE' states that "upon successful completion, `strncasecmp()` shall return an integer greater than, equal to, or less than 0".
    2. IEEE Std 1003.1, 2004 Edition, Section ' `strcmp()` : DESCRIPTION' adds that "the sign of a non-zero return value shall be determined by the sign of the difference between the values of the first pair of bytes ... that differ in the strings being compared".
3. String comparison terminates when :
  (a)
    1. (A) BOTH string pointer(s) are passed as NULL pointers.
        1. NULL strings identical; 0 returned.
      (B)
        1. ' `p1_str` ' passed a NULL pointer.
          (a) Return negative value of character pointed to by ' `p2_str` ', converted to lower case (see Note #2a2).
        2. ' `p2_str` ' passed a NULL pointer.
          (a) Return positive value of character pointed to by ' `p1_str` ', converted to lower case (see Note #2a2).
    2. (A) BOTH strings point to NULL.
        1. Strings overlap with NULL address.
        2. Strings identical up to but NOT beyond or including the NULL address; 0 returned.
      (B)
        1. ' `p1_str_cmp_next` ' points to NULL.
          (a) ' `p1_str` ' overlaps with NULL address.
          (b) Strings compared up to but NOT beyond or including the NULL address.
          (c) Return negative value of character pointed to by ' `p2_str_cmp_next` ', converted to lower case (see Note #2a2).
        2. ' `p2_str_cmp_next` ' points to NULL.
          (a) ' `p2_str` ' overlaps with NULL address.
          (b) Strings compared up to but NOT beyond or including the NULL address.
          (c) Return positive value of character pointed to by ' `p1_str_cmp_next` ', converted to lower case (see Note #2a2).
  (b) Terminating NULL character found in both strings.
    1. Strings identical; 0 returned.
    2. Only one NULL character test required in conditional since the previous condition tested character equality.
  (c) Non-matching characters found.
    1. Return signed-integer difference of the character pointed to by ' `p2_str` ', converted to lower case, from the character pointed to by 'p1_str', converted to lower case.
  (d) See also Note #2a1C.
    1. ' `len_max` ' passed a zero length.
      (A) Zero-length strings identical; 0 returned.
    2. First ' `len_max` ' number of characters identical.
      (A) Strings identical; 0 returned.
4. Since 16-bit signed arithmetic is performed to calculate a non-identical comparison return value, ' `CPU_CHAR` ' native data type size MUST be 8-bit.

Str_Char()

Description

Searches the string for first occurrence of specific character.

Files

`lib_str.h/lib_str.c`

Prototype

```
CPU_CHAR *Str_Char (const CPU_CHAR   *p_str,
                    CPU_CHAR    srch_char)
```

Arguments

`p_str`

Pointer to the string (see Note #1).

`srch_char`

Search character.

Returned Value

Pointer to the first occurrence of search character in string, if any occur (see Note #2b1).

Pointer to NULL, if no search character is found (see Note #2b2).

Notes / Warnings

1. String buffer NOT modified.
2. (a) IEEE Std 1003.1, 2004 Edition, Section ' `strchr()` : DESCRIPTION' states that :
    1. "The `strchr()` function shall locate the first occurrence of 'c' (' `srch_char` ') ... in the string pointed to by 's' (' `p_str` ')."
    2. "The terminating null byte is considered to be part of the string."

   (b) IEEE Std 1003.1, 2004 Edition, Section ' `strchr()` : RETURN VALUE' states that "upon completion, `strchr()` shall return" :
    1. "a pointer to the byte," ...
    2. "or a null pointer if the byte was not found."
       (A) #### Although NO `strchr()` specification states to return NULL for any other reason (s), NULL is also returned for any error(s).
3. String search terminates when :
   (a) String pointer passed a NULL pointer.
    1. No string search performed; NULL pointer is returned.

   (b) String pointer points to NULL.
    1. String overlaps with NULL address; NULL pointer is returned.

   (c) String's terminating NULL character found.
    1. Search character NOT found in search string; NULL pointer is returned (see Note #2b2).
    2. Applicable even if search character is the terminating NULL character (see Note #2a2).

   (d) Search character found.
    1. Return pointer to first occurrence of search character in search string (see Note #2a1).

Str_Char_N()

Description

Searches the string for first occurrence of specific character, up to a maximum number of characters.

Files

`lib_str.h/lib_str.c`

Prototype

```
CPU_CHAR *Str_Char_N (const CPU_CHAR    *p_str,
                      CPU_SIZE_T   len_max,
                      CPU_CHAR     srch_char)
```

Arguments

`p_str`

Pointer to the string (see Note #1).

`len_max`

Maximum number of characters to search (see Notes #2c and #3e).

`srch_char`

Search character.

Returned Value

Pointer to the first occurrence of search character in string, if any occur (see Note #2b1).

Pointer to the NULL, if no search character is found (see Note #2b2).

Notes / Warnings

1. String buffer NOT modified.
2. (a) IEEE Std 1003.1, 2004 Edition, Section ' `strchr()` : DESCRIPTION' states that :
    1. "The `strchr()` function shall locate the first occurrence of 'c' (' `srch_char` ') ... in the string pointed to by 's' (' `p_str` ')."
    2. "The terminating null byte is considered to be part of the string."

   (b) IEEE Std 1003.1, 2004 Edition, Section ' `strchr()` : RETURN VALUE' states that "upon completion, `strchr()` shall return" :
    1. "a pointer to the byte," ...
    2. "or a null pointer if the byte was not found."
       (A) #### Although NO `strchr()` specification states to return NULL for any other reason(s), NULL is also returned for any error(s).

   (c) Ideally, the 'len_max' argument would be the last argument in this function's argument list for consistency with all other custom string library functions. However, the 'len_max' argument is sequentially ordered as the second argument to comply with most standard library's `strnchr()` argument list.
3. String search terminates when :
   (a) String pointer passed a NULL pointer.
    1. No string search performed; NULL pointer is returned.

   (b) String pointer points to NULL.
    1. String overlaps with NULL address; NULL pointer is returned.

   (c) String's terminating NULL character found.
    1. Search character NOT found in search string; NULL pointer is returned (see Note #2b2).
    2. Applicable even if search character is the terminating NULL character (see Note #2a2).

   (d) Search character found.
    1. Return pointer to first occurrence of search character in search string (see Note #2a1).

   (e) ' `len_max` ' number of characters searched.
    1. Search character NOT found in search string within first ' `len_max` ' number of characters; NULL pointer is returned.
    2. ' `len_max` ' number of characters MAY include terminating NULL character (see Note #2a2).

Str_Char_Last()

Description

Searches the string for last occurrence of specific character.

Files

`lib_str.h/lib_str.c`

Prototype

```
CPU_CHAR *Str_Char_Last (const CPU_CHAR   *p_str,
                               CPU_CHAR    srch_char)
```

Arguments

`p_str`

Pointer to the string (see Note #1).

`srch_char`

Search character.

Returned Value

Pointer to the last occurrence of search character in string, if any occur (see Note #2b1).

Pointer to NULL, if no search character is found (see Note #2b2).

Notes / Warnings

1. String buffer NOT modified.
2. (a) IEEE Std 1003.1, 2004 Edition, Section ' `strrchr()` : DESCRIPTION' states that :
    1. "The `strrchr()` function shall locate the last occurrence of 'c' (' `srch_char` ') ... in the string pointed to by 's' (' `p_str` ')."
    2. "The terminating null byte is considered to be part of the string."

   (b) IEEE Std 1003.1, 2004 Edition, Section ' `strrchr()` : RETURN VALUE' states that "upon successful completion, `strrchr()` shall return" :
    1. "a pointer to the byte"
    2. "or a null pointer if 'c' (' `srch_char` ') does not occur in the string."
       (A) #### Although NO `strrchr()` specification states to return NULL for any other reason(s), NULL is also returned for any error(s).
3. String search terminates when :
   (a) String pointer passed a NULL pointer.
    1. No string search performed; NULL pointer is returned.

   (b) String pointer points to NULL.
    1. String overlaps with NULL address; NULL pointer is returned.

   (c) String searched from end to beginning.
    1. Search character NOT found in search string; NULL pointer is returned.
    2. Applicable even if search character is the terminating NULL character (see Note #2a2).

   (d) Search character found.
    1. Return pointer to last occurrence of search character in search string (see Note #2a1).

Str_Char_Last_N()

Description

Searches the string for last occurrence of specific character, up to a maximum number of characters.

Files

`lib_str.h/lib_str.c`

Prototype

```
CPU_CHAR *Str_Char_Last_N (const CPU_CHAR    *p_str,
                           CPU_SIZE_T   len_max,
                           CPU_CHAR     srch_char)
```

Arguments

`p_str`

Pointer to the string (see Note #1).

`len_max`

Maximum number of characters to search (see Notes #2c and #3e).

`srch_char`

Search character.

Returned Value

Pointer to the last occurrence of search character in string, if any occur (see Note #2b1).

Pointer to NULL, if no search character is found (see Note #2b2).

Notes / Warnings

1. String buffer NOT modified.
2. (a) IEEE Std 1003.1, 2004 Edition, Section '`strrchr()` : DESCRIPTION' states that :
    1. "The `strrchr()` function shall locate the last occurrence of 'c' ('`srch_char`') ... in the string pointed to by 's' ('`p_str`')."
    2. "The terminating null byte is considered to be part of the string."

   (b) IEEE Std 1003.1, 2004 Edition, Section '`strrchr()` : RETURN VALUE' states that "upon successful completion, `strrchr()` shall return" :
    1. "a pointer to the byte"
    2. "or a null pointer if 'c' ('`srch_char`') does not occur in the string."
       (A) #### Although NO `strrchr()` specification states to return NULL for any other reason(s), NULL is also returned for any error(s).

   (c) Ideally, the '`len_max`' argument would be the last argument in this function's argument list for consistency with all other custom string library functions. However, the '`len_max`' argument is sequentially ordered as the second argument to comply with most standard library's `strnrchr()` argument list.
3. String search terminates when :
   (a) String pointer passed a NULL pointer.
    1. No string search performed; NULL pointer is returned.

   (b) String pointer points to NULL.
    1. String overlaps with NULL address; NULL pointer is returned.

   (c) String searched from end to beginning.
    1. Search character NOT found in search string; NULL pointer is returned (see Note #2b2).
    2. Applicable even if search character is the terminating NULL character (see Note #2a2).

   (d) Search character found.
    1. Return pointer to last occurrence of search character in search string (see Note #2a1).

   (e) '`len_max`' number of characters searched.
    1. Search character NOT found in search string within last '`len_max`' number of characters; NULL pointer is returned.
    2. '`len_max`' number of characters MAY include terminating NULL character (see Note #2a2).

Str_Char_Replace()

Description

Searches the string for specific character and replace it by another specific character.

Files

`lib_str.h/lib_str.c`

Prototype

```
CPU_CHAR *Str_Char_Replace (CPU_CHAR   *p_str,
                            CPU_CHAR    char_srch,
                            CPU_CHAR    char_replace)
```

Arguments

`p_str`

Pointer to the string (see Note #1).

`char_srch`

Search character.

`char_replace`

Replace character.

Returned Value

Pointer to the string, if NO error(s).

Pointer to NULL, if any errors occur.

Notes / Warnings

1. String buffer modified.
2. String search terminates when :
   - (a) String pointer passed a NULL pointer.
     1. No string search performed; NULL pointer is returned.
   - (b) String pointer points to NULL.
     1. String overlaps with NULL address; NULL pointer is returned.
   - (c) String's terminating NULL character found.
     1. Search character NOT found in search string; NULL pointer is returned
     2. Applicable even if search character is the terminating NULL character
   - (d) Search character found.
     1. Replace character found by the specified character.

Str_Char_Replace_N()

Description

Searches the string for specific character and replace it by another specific character, up to a maximum number of characters.

Files

`lib_str.h/lib_str.c`

Prototype

```
CPU_CHAR *Str_Char_Replace_N (CPU_CHAR    *p_str,
                              CPU_CHAR     char_srch,
                              CPU_CHAR     char_replace,
                              CPU_SIZE_T   len_max)
```

Arguments

`p_str`

Pointer to the string (see Note #1).

`char_srch`

Search character.

`char_replace`

Replace character.

`len_max`

Maximum number of characters to search (see Notes #2c and #3e).

Returned Value

Pointer to the string, if NO error(s).

Pointer to NULL, if any errors occur.

Notes / Warnings

1. String buffer modified.
2. String search terminates when :
   - (a) String pointer passed a NULL pointer.
     1. No string search performed; NULL pointer is returned.
   - (b) String pointer points to NULL.
     1. String overlaps with NULL address; NULL pointer is returned.
   - (c) String's terminating NULL character found.
     1. Search character NOT found in search string; NULL pointer is returned
     2. Applicable even if search character is the terminating NULL character
   - (d) Search character found.
     1. Replace character found by the specified character.
   - (e) '`len_max`' number of characters searched.
     1. Search character NOT found in search string within first '`len_max`' number of characters; NULL pointer is returned.
     2. '`len_max`' number of characters MAY include terminating NULL character (see Note #2a2).

Str_Str()

Description

Searches the string for the first occurrence of a specific search string.

Files

`lib_str.h/lib_str.c`

Prototype

```
CPU_CHAR *Str_Str (const CPU_CHAR   *p_str,
                   const CPU_CHAR   *p_str_srch)
```

Arguments

`p_str`

Pointer to string (see Note #1).

`p_str_srch`

Pointer to the search string (see Note #1).

Returned Value

- Pointer to the first occurrence of a search string in the string, if any occur (see Note #2b1A).
- Pointer to the string, if search string has a zero length (see Note #2b2).
- Pointer to NULL, if no search character is found (see Note #2b1B).

Notes / Warnings

1. String buffers are NOT modified.
2. (a) IEEE Std 1003.1, 2004 Edition, Section ' `strstr()` : DESCRIPTION' states that :
    1. "The `strstr()` function shall locate the first occurrence in the string pointed to by 's1' ('p_str') of the sequence of bytes ... in the string pointed to by 's2' ('p_str_srch')"
    2. "(excluding the terminating null byte)."

   (b) IEEE Std 1003.1, 2004 Edition, Section ' `strstr()` : RETURN VALUE' states that :
    1. "Upon successful completion, `strstr()` shall return" :
       (A) "a pointer to the located string"
       (B) "or a null pointer if the string is not found."
        1. #### Although NO `strstr()` specification states to return NULL for any other reason(s), NULL is also returned for any error(s).
    2. "If 's2' (' `p_str_srch` ') points to a string with zero length, the function shall return 's1' (' `p_str` ')."
3. String search terminates when :
   (a) String pointer(s) are passed as NULL pointers.
    1. No string search performed; NULL pointer is returned.

   (b) String pointer(s) point to NULL.
    1. String buffer(s) overlap with NULL address; NULL pointer is returned.

   (c) for the first occurrence length equal to zero.
    1. No string search performed; string pointer returned (see Note #2b2).

   (d) Search string length greater than string length.
    1. No string search performed; NULL pointer returned (see Note #2b1B).

   (e) Entire string has been searched.
    1. Search string not found; NULL pointer is returned (see Note #2b1B).

   (f) Search string found.
    1. Return pointer to first occurrence of search string in string (see Note #2b1A).

Str_Str_N()

Description

Searches the string for the first occurrence of a specific search string, up to a maximum number of characters.

Files

`lib_str.h/lib_str.c`

Prototype

```
CPU_CHAR *Str_Str_N (const CPU_CHAR    *p_str,
                     const CPU_CHAR    *p_str_srch,
                          CPU_SIZE_T    len_max)
```

Arguments

`p_str`

Pointer to the string (see Note #1).

`p_str_srch`

Pointer to the search string (see Note #1).

`len_max`

Maximum number of characters to search (see Note #3g).

Returned Value

Pointer to the first occurrence of search string in string, if any occur (see Note #2b1A).

Pointer to the string, if NULL search string (see Note #2b2).

Pointer to NULL, otherwise (see Note #2b1B).

Notes / Warnings

1. String buffers are NOT modified.
2. (a) IEEE Std 1003.1, 2004 Edition, Section '`strstr()` : DESCRIPTION' states that :
    1. "The `strstr()` function shall locate the first occurrence in the string pointed to by 's1' ('p_str') of the sequence of bytes ... in the string pointed to by 's2' ('p_str_srch')"
    2. "(excluding the terminating null byte)."

    (b) IEEE Std 1003.1, 2004 Edition, Section '`strstr()` : RETURN VALUE' states that :
    1. "Upon successful completion, `strstr()` shall return" :
        (A) "a pointer to the located string"
        (B) "or a null pointer if the string is not found."
        1. #### Although NO `strstr()` specification states to return NULL for any other reason(s), NULL is also returned for any error(s).
    2. "If 's2' ('`p_str_srch`') points to a string with zero length, the function shall return 's1' ('`p_str`')."
3. String search terminates when :
    (a) String pointer(s) are passed as NULL pointers.
    1. No string search performed; NULL pointer is returned.

    (b) String pointer(s) point to NULL.
    1. String buffer(s) overlap with NULL address; NULL pointer is returned.

    (c) Search string length equal to zero.
    1. No string search performed; string pointer returned (see Note #2b2).

    (d) Search string length greater than string length.
    1. No string search performed; NULL pointer returned (see Note #2b1B).

    (e) Entire string has been searched.
    1. Search string not found; NULL pointer is returned (see Note #2b1B).
    2. Maximum size of the search is defined as the subtraction of the search string length from the string length.

    (f) Search string found.
    1. Return pointer to first occurrence of search string in string (see Note #2b1A).
    2. Search string found via `Str_Cmp_N()` .

    (g) '`len_max`' number of characters searched.
    1. '`len_max`' number of characters does NOT include terminating NULL character (see Note #2a2).

Str_Printf()

Description

Lightweight `printf()` implementation.

Files

lib_str.h/lib_str.c

Prototype

```
CPU_SIZE_T Str_Printf (STR_PRINTF_OUT_CB   out_cb,
                       void               *p_out_cb_arg,
                       CPU_CHAR           *p_fmt,
                                           ...)
```

Arguments

out_cb

Output callback (putchar-like with an extra callback-specific argument).

p_out_cb_arg

Character output callback argument.

p_fmt

Pointer to the format string.

...

Variable number of arguments that will be formatted according to the format specifiers in the format string.

Returned Value

None.

Notes / Warnings

None.

Str_FmtNbr_Int32U()

Description

Formats a 32-bit unsigned integer into a multi-digit character string.

Files

lib_str.h/lib_str.c

Prototype

```
CPU_CHAR *Str_FmtNbr_Int32U (CPU_INT32U    nbr,
                             CPU_INT08U    nbr_dig,
                             CPU_INT08U    nbr_base,
                             CPU_CHAR      lead_char,
                             CPU_BOOLEAN   lower_case,
                             CPU_BOOLEAN   nul,
                             CPU_CHAR     *p_str)
```

Arguments

nbr

Number to format.

`nbr_dig`

Number of digits to format (see Note #1).

The following may be used to specify the number of digits to format :

- `DEF_INT_32U_NBR_DIG_MIN` Minimum number of 32-bit unsigned digits
- `DEF_INT_32U_NBR_DIG_MAX` Maximum number of 32-bit unsigned digits

`nbr_base`

Base of number to format (see Note #2).

The following may be used to specify the number base :

- `DEF_NBR_BASE_BIN` Base 2
- `DEF_NBR_BASE_OCT` Base 8
- `DEF_NBR_BASE_DEC` Base 10
- `DEF_NBR_BASE_HEX` Base 16

`lead_char`

Prepend leading character (see Note #3) :

- '\0' Do NOT prepend leading character to a string.
- Printable character Prepend leading character to a string.
- Unprintable character Format invalid string (see Note #6).

`lower_case`

Format alphabetic characters (if any) in lower case :

- `DEF_NO` Format alphabetic characters in upper case.
- `DEF_YES` Format alphabetic characters in lower case.

`nul`

Append terminating NULL-character (see Note #4) :

- `DEF_NO` Do NOT append terminating NULL-character to a string.
- `DEF_YES` Append terminating NULL-character to a string.

`p_str`

Pointer to the character array to return formatted number string (see Note #5).

Returned Value

- Pointer to the formatted string, if NO error(s).
- Pointer to NULL, if any errors occur.

Notes / Warnings

1. (a) If the number of digits to format ('nbr_dig') is zero; then NO formatting is performed except possible NULL-termination of the string (see Note #4).
   Example :

```
nbr = 23456
nbr_dig = 0
nbr_base = 10

p_str = ""
```

See Note #6a
(b) If the number of digits to format ('nbr_dig') is less than the number of significant integer digits of the number to format ('nbr'); then an invalid string is formatted instead of truncating any significant integer digits.
Example :

```
   nbr = 23456
   nbr_dig = 3
   nbr_base = 10

   p_str = "???"
```

See Note #6b

2. The number's base MUST be between 2 and 36, inclusive.
3. Leading character option prepends leading characters prior to the first non-zero digit.
   (a)
   1. Leading character MUST be a printable ASCII character.
   2. (A) Leading character MUST NOT be a number base digit,
      (B) with the exception of '0'.

   (b) The number of leading characters is such that the total number of significant integer digits plus the number of leading characters is equal to the requested number of integer digits to format ('nbr_dig').
   Example :

```
 nbr = 23456
 nbr_dig = 7
 nbr_base = 10
 lead_char = ' '

 p_str = " 23456"
```

   (c)
   1. If the value of the number to format is zero
   2. & the number of digits to format is non-zero,
   3. but NO leading character available;
   4. then one digit of '0' value is formatted.
      This is NOT a leading character; but a single integer digit of '0' value.
4. (a) NULL-character terminate option DISABLED prevents overwriting previous character array formatting.
   (b) WARNING: Unless '`p_str`' character array is pre-/post-terminated, NULL-character terminate option DISABLED will cause character string run-on.
5. (a) Format buffer size is NOT validated; buffer overruns MUST be prevented by caller.
   (b) To prevent character buffer overrun :
6. For any unsuccessful string format or error(s), an invalid string of question marks ('?') will be formatted, where the number of question marks is determined by the number of digits to format ('`nbr_dig`') :

```
Invalid string's { (a) 0 (NULL string) , if 'nbr_dig' = 0
  number of = {
question marks { (b) 'nbr_dig' , if 'nbr_dig' > 0
```

Str_FmtNbr_Int32S()

Description

Formats a 32-bit signed integer into a multi-digit character string.

Files

`lib_str.h/lib_str.c`

Prototype

```
CPU_CHAR *Str_FmtNbr_Int32S (CPU_INT32S   nbr,
                             CPU_INT08U   nbr_dig,
                             CPU_INT08U   nbr_base,
                             CPU_CHAR     lead_char,
                             CPU_BOOLEAN  lower_case,
                             CPU_BOOLEAN  nul,
                             CPU_CHAR    *p_str)
```

Arguments

`nbr`

Number to format.

`nbr_dig`

Number of digits to format (see Note #1).

The following may be used to specify the number of digits to format :

- `DEF_INT_32S_NBR_DIG_MIN` + 1 Minimum number of 32-bit signed digits
- `DEF_INT_32S_NBR_DIG_MAX` + 1 Maximum number of 32-bit signed digits

(plus 1 digit for possible negative sign)

`nbr_base`

Base of number to format (see Note #2).

The following may be used to specify the number base :

- `DEF_NBR_BASE_BIN` Base 2
- `DEF_NBR_BASE_OCT` Base 8
- `DEF_NBR_BASE_DEC` Base 10
- `DEF_NBR_BASE_HEX` Base 16

`lead_char`

Prepend leading character (see Note #3) :

'\0' Do NOT prepend leading character to a string. Printable character Prepend leading character to a string. Unprintable character Format invalid string (see Note #6).

`lower_case`

Format alphabetic characters (if any) in lower case :

- `DEF_NO` Format alphabetic characters in upper case.
- `DEF_YES` Format alphabetic characters in lower case.

`nul`

Append terminating NULL-character (see Note #4) :

- `DEF_NO` Do NOT append terminating NULL-character to a string.
- `DEF_YES` Append terminating NULL-character to a string.

`p_str`

Pointer to the character array to return formatted number string (see Note #5).

Returned Value

- Pointer to the formatted string, if NO error(s).
- Pointer to NULL, if any errors occur.

Notes / Warnings

1. (a) If the number of digits to format ('nbr_dig') is zero; then NO formatting is performed except possible NULL-termination of the string (see Note #4).
   Example :

   ```
   nbr = -23456
   nbr_dig = 0
   nbr_base = 10
   p_str = ""
   ```

   See Note #6a
   (b) If the number of digits to format ('nbr_dig') is less than the number of significant integer digits of the number to format ('nbr'); then an invalid string is formatted instead of truncating any significant integer digits.
   Example :

   ```
   nbr = 23456
   nbr_dig = 3
   nbr_base = 10
   p_str = "???"
   ```

   See Note #6b
   (c) If the number to format ('nbr') is negative but the number of digits to format ('nbr_dig') is equal to the number of significant integer digits of the number to format ('nbr'); then an invalid string is formatted instead of truncating the negative sign.
   Example :

   ```
   nbr = -23456
   nbr_dig = 5
   nbr_base = 10
   p_str = "?????"
   ```

   See Note #6b
2. The number's base MUST be between 2 and 36, inclusive.
3. Leading character option prepends leading characters prior to the first non-zero digit.
   (a)
   1. Leading character MUST be a printable ASCII character.
   2. (A) Leading character MUST NOT be a number base digit,
      (B) with the exception of '0'.

   (b)
   1. The number of leading characters is such that the total number of significant integer digits plus the number of leading characters plus possible negative sign character is equal to the requested number of integer digits to format (' `nbr_dig` ').
      Examples :

      ```
      nbr = 23456
      nbr_dig = 7
      nbr_base = 10
      lead_char = ' '
      p_str = " 23456"
      nbr = -23456
      nbr_dig = 7
      nbr_base = 10
      lead_char = ' '
      p_str = " -23456"
      ```

   2. (A) If the number to format (' `nbr` ') is negative AND the leading character (' `lead_char` ') is a '0' digit; then the negative sign character prefixes all leading characters prior to the formatted number.
      Examples :

```
nbr = -23456
nbr_dig = 8
nbr_base = 10
lead_char = '0'
p_str = "-0023456"
nbr = -43981
nbr_dig = 8
nbr_base = 16
lead_char = '0'
lower_case = DEF_NO
p_str = "-000ABCD"
```

(B) If the number to format (' `nbr` ') is negative AND the leading character (' `lead_char` ') is NOT a '0' digit; then the negative sign character immediately prefixes the most significant digit of the formatted number.
Examples :

```
nbr = -23456
nbr_dig = 8
nbr_base = 10
lead_char = '#'
p_str = "##-23456"
nbr = -43981
nbr_dig = 8
nbr_base = 16
lead_char = '#'
lower_case = DEF_YES
p_str = "###-abcd"
```

(c)

1. If the value of the number to format is zero
2. & the number of digits to format is non-zero,
3. but NO leading character available;
4. then one digit of '0' value is formatted.

4. (a) NULL-character terminate option DISABLED prevents overwriting previous character array formatting.
   (b) WARNING: Unless 'p_str' character array is pre-/post-terminated, NULL-character terminate option DISABLED will cause character string run-on.
5. (a) Format buffer size is NOT validated; buffer overruns MUST be prevented by caller.
   (b) To prevent character buffer overrun :
   Character array size MUST be >= ('nbr_dig' + 1 negative sign + 1 'NUL' terminator) characters
6. For any unsuccessful string format or error(s), an invalid string of question marks ('?') will be formatted, where the number of question marks is determined by the number of digits to format ('nbr_dig') :

```
Invalid string's { (a) 0 (NULL string) ,   if 'nbr_dig' = 0
  number of   = {
question marks  { (b) 'nbr_dig'       ,   if 'nbr_dig' > 0
```

Str_FmtNbr_32()

Description

Formats a number into a multi-digit character string.

Files

`lib_str.h/lib_str.c`

Prototype

```
#if (LIB_STR_CFG_FP_EN == DEF_ENABLED)
CPU_CHAR  *Str_FmtNbr_32 (CPU_FP32     nbr,
                          CPU_INT08U   nbr_dig,
                          CPU_INT08U   nbr_dp,
                          CPU_CHAR     lead_char,
                          CPU_BOOLEAN  nul,
                          CPU_CHAR    *p_str)
```

Arguments

`nbr`

Number to format (see Note #1).

`nbr_dig`

Number of decimal digits to format (see Note #2).

`nbr_dp`

Number of decimal point digits to format.

`lead_char`

Prepend leading character (see Note #3) :

- '\0' : Do NOT prepend leading character to a string.
- Printable character : Prepend leading character to a string.
- Unprintable character : Format invalid string (see Note #6d).

`nul`

Append terminating NULL-character (see Note #4) :

- `DEF_NO` Do NOT append terminating NULL-character to a string.
- `DEF_YES` Append terminating NULL-character to a string.

`p_str`

Pointer to the character array to return formatted number string (see Note #5).

Returned Value

Pointer to the formatted string, if NO error(s) [see Note #6c].

Pointer to NULL, if any errors occur.

Notes / Warnings

1. (a) The maximum accuracy for 32-bit floating-point numbers :

```
Maximum Accuracy              log [Internal-Base ^ (Number-Internal-Base-Digits)]
32-bit Floating-point Number = -----------------------------------------------------
                              log [External-Base]
                    log [2 ^ 24]
                  = -------------
                     log [10]

                  < 7.225 Base-10 Digits
where
      Internal-Base              Internal number base of floating-
                               point numbers (i.e. 2)
      External-Base               External number base of floating-
                               point numbers (i.e. 10)
      Number-Internal-Base-Digits   Number of internal number base
                               significant digits (i.e. 24)
```

(b) Some CPUs' &/or compilers' floating-point implementations MAY further reduce the maximum accuracy.

2. (a) If the total number of digits to format (' nbr_dig + nbr_dp ') is zero; then NO formatting is performed except possible NULL-termination of the string (see Note #4).
   Example :

   ```
   nbr = -23456.789
   nbr_dig = 0
   nbr_dp = 0
   p_str = ""
   ```

   See Note #7a
   (b)
   1. If the number of digits to format (' nbr_dig ') is less than the number of significant integer digits of the number to format (' nbr '); then an invalid string is formatted instead of truncating any significant integer digits.
      Example :

      ```
      nbr = 23456.789
      nbr_dig = 3
      nbr_dp = 2
      p_str = "??????"
      ```

      See Note #7d
   2. If the number to format (' nbr ') is negative but the number of digits to format ('nbr_dig') is equal to the number of significant integer digits of the number to format ('nbr'); then an invalid string is formatted instead of truncating the negative sign.
      Example :

      ```
      nbr = -23456.789
      nbr_dig = 5
      nbr_dp = 2
      p_str = "????????"
      ```

      See Note #7d
   3. If the number to format (' nbr ') is negative but the number of significant integer digits is zero, and the number of digits to format (' nbr_dig ') is one but the number of decimal point digits to format (' nbr_dp ') is zero; then an invalid string is formatted instead of truncating the negative sign.
      Example :

      ```
      nbr = -0.7895
      nbr_dig = 1
      nbr_dp = 0
      p_str = "?"
      ```

      See Note #7d
   4. (A) If the number to format (' nbr ') is negative but the number of significant integer digits is zero, and the number of digits to format (' nbr_dig ') is zero but the number of decimal point digits to format (' nbr_dp ') is non- zero; then the

negative sign immediately prefixes the decimal point -- with NO decimal digits formatted, NOT even a single decimal digit of '0'.
Example :

```
nbr = -0.7895
nbr_dig = 0
nbr_dp = 2
p_str = "-.78"
```

(B) If the number to format (' `nbr` ') is positive but the number of significant integer digits is zero, and the number of digits to format (' `nbr_dig` ') is zero but the number of decimal point digits to format (' `nbr_dp` ') is non- zero; then a single decimal digit of '0' prefixes the decimal point.
This '0' digit is used whenever a negative sign is not formatted (see Note #2b4A) so that the formatted string's decimal point is not floating, but fixed in the string as the second character.
Example :

```
nbr = 0.7895
nbr_dig = 0
nbr_dp = 2

p_str = "0.78"
```

(c)

1. If the total number of digits to format (' `nbr_dig` + `nbr_dp` ') is greater than :
   (A) the maximum accuracy of the CPU's &/or compiler's 32-bit floating-point numbers, digits following all significantly-accurate digits of the number to format (' `nbr` ') will be inaccurate; ...
   (B) the configured maximum accuracy (' `LIB_STR_CFG_FP_MAX_NBR_DIG_SIG` '), all digits or decimal places following all significantly-accurate digits of the number to format (' `nbr` ') will be replaced & formatted with zeros ('0').
2. Therefore, one or more least-significant digit(s) of the number to format (' `nbr` ') MAY be rounded & not necessarily truncated due to the inaccuracy of the CPU's &/or compiler's floating-point implementation.

3. Leading character option prepends leading characters prior to the first non-zero digit.

(a)

1. Leading character MUST be a printable ASCII character.
2. (A) Leading character MUST NOT be a base-10 digit,
   (B) with the exception of '0'.

(b)

1. The number of leading characters is such that the total number of significant integer digits plus the number of leading characters plus possible negative sign character is equal to the requested number of integer digits to format (' `nbr_dig` ').
   Examples :

```
nbr = 23456.789
nbr_dig = 7
nbr_dp = 2
lead_char = ' '

p_str = " 23456.78"

nbr = -23456.789
nbr_dig = 7
nbr_dp = 2
lead_char = ' '

p_str = " -23456.78"
```

2. (A) If the number to format (' `nbr` ') is negative AND the leading character (' `lead_char` ') is a '0' digit; then the negative sign character prefixes all leading characters prior to the formatted number.
   Example :

```
nbr = -23456.789
nbr_dig = 8
nbr_dp = 2
lead_char = '0'


p_str = "-0023456.78"
```

(B) If the number to format (' `nbr` ') is negative AND the leading character (' `lead_char` ') is NOT a '0' digit; then the negative sign character immediately prefixes the most significant digit of the formatted number.
Examples :

```
nbr = -23456.789
nbr_dig = 8
nbr_dp = 2
lead_char = '#'

p_str = "##-23456.78"
```

(c)

1. If the integer value of the number to format is zero and
2. the number of digits to format is greater than one
3. OR the number is NOT negative,
4. but NO leading character available;
5. then one digit of '0' value is formatted.
   This is NOT a leading character; but a single integer digit of '0' value.

See also Note #2b4B.

4. (a) NULL-character terminate option DISABLED prevents overwriting previous character array formatting.
   (b) WARNING: Unless ' `p_str` ' character array is pre-/post-terminated, NULL-character terminate option DISABLED will cause character string run-on.
5. (a) Format buffer size is NOT validated; buffer overruns MUST be prevented by caller.
   (b) To prevent character buffer overrun :
   `Character array size MUST be \>= ('nbr_dig' + 'nbr_dp' + 1 negative sign + 1 decimal point + 1 'NUL' terminator) characters`
6. String format terminates when :
   (a) Format string pointer is passed a NULL pointer.
      1. No string formatted; NULL pointer is returned.
   (b) Total number of digits to format (' `nbr_dig` + `nbr_dp` ') is zero.
      1. NULL string formatted (see Note #7a); NULL pointer is returned.
   (c) Number of digits to format (' `nbr_dig` ') is less than number of significant integer digits of the number to format (' `nbr` '), including possible negative sign.
      1. Invalid string formatted (see Note #7); NULL pointer is returned.
   (d) Lead character is NOT a valid, printable character (see Note #3a).
      1. Invalid string formatted (see Note #7); NULL pointer is returned.
   (e) Number successfully formatted into character string array.
7. For any unsuccessful string format or error(s), an invalid string of question marks ('?') will be formatted, where the number of question marks is determined by the number of digits (' `nbr_dig` ') & number of decimal point digits (' `nbr_dp` ') to format :

```
               { (a) 0 (NULL string)           , if 'nbr_dig' = 0 AND
               {                                  'nbr_dp'  = 0
               {
               { (b) 'nbr_dig'                 , if 'nbr_dig' > 0 AND
               {                                  'nbr_dp'  = 0
Invalid string's  {
  number of =     { (c) ['nbr_dp'          +   , if 'nbr_dig' = 0 AND
 question marks   {      1 (for decimal point) +      'nbr_dp'  > 0
               {      1 (for negative sign) ]
               {
               { (d) ['nbr_dig'         +   , if 'nbr_dig' > 0 AND
               {     'nbr_dp'           +      'nbr_dp'  > 0
               {      1 (for decimal point) ]
```

Str_ParseNbr_Int32U()

Description

Parses a 32-bit unsigned integer from a string.

Files

`lib_str.h/lib_str.c`

Prototype

```
CPU_INT32U Str_ParseNbr_Int32U (const CPU_CHAR   *p_str,
                                      CPU_CHAR   *p_str_next,
                                      CPU_INT08U  nbr_base)
```

Arguments

`p_str`

Pointer to the string (see Notes #1 and #2a).

`p_str_next`

Optional pointer to a variable to :

(a) Return a pointer to first character following the integer string, if NO error(s) [see Note #2a2B2];

(b) Return a pointer to ' `p_str` ', if any errors occur (see Note #2a2A2).

`nbr_base`

Base of number to parse (see Notes #2a1B1 and #2a2B1).

Returned Value

- Parsed integer, if integer parsed with NO overflow (see Note #2a3A).
- `DEF_INT_32U_MAX_VAL` , if integer parsed but overflowed (see Note #2a3A1).
- 0, otherwise (see Note #2a3B).

Notes / Warnings

1. String buffer NOT modified.
2. (a) IEEE Std 1003.1, 2004 Edition, Section ' `strtoul()` : DESCRIPTION' states that "these functions shall convert the initial portion of the string pointed to by ' `str` ' (' `p_str` ') to a type unsigned long ... representation" :
   1. "First, they decompose the input string into three parts" :
      (A) "An initial, possibly empty, sequence of white-space characters [as specified by isspace()]."

1. "The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character that is of the expected form. The subject sequence shall contain no characters if the input string is empty or consists entirely of white-space characters."

(B)

1. "A subject sequence interpreted as an integer represented in some radix determined by the value of 'base' (' `nbr_base` ')" :

   (a) "If the value of 'base' (' `nbr_base` ') is 0, the expected form of the subject sequence is that of a decimal constant, octal constant, or hexadecimal constant" :

   1. "A decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits."
   2. "An octal constant consists of the prefix '0' optionally followed by a sequence of the digits '0' to '7' only."
   3. "A hexadecimal constant consists of the prefix '0x' or '0X' followed by a sequence of the decimal digits and letters 'a' (or 'A') to 'f' (or 'F') with values 10 to 15 respectively."

   (b) "If the value of 'base' (' `nbr_base` ') is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by 'base' (' `nbr_base` ')" :

   1. (A) "The letters from 'a' (or 'A') to 'z' (or 'Z') inclusive are ascribed the values 10 to 35"; ...
      (B) "only letters whose ascribed values are less than that of base are permitted."
   2. (A) "If the value of 'base' (' `nbr_base` ') is 16, the characters '0x' or '0X' may optionally precede the sequence of letters and digits."
      (B) Although NO specification states that "if the value of 'base' (' `nbr_base` ') is" 8, the '0' character "may optionally precede the sequence of letters and digits"; it seems reasonable to allow the '0' character to be optionally parsed.

2. "A subject sequence .... may be preceded by a '+' or '-' sign."

   (a) However, it does NOT seem reasonable to parse & convert a negative number integer string into an unsigned integer.

(C)

1. (a) "A final string of one or more unrecognized characters," ...
   (b) "including the terminating null byte of the input string" ...
2. "other than a sign or a permissible letter or digit."

2. Second, "they shall attempt to convert the subject sequence to an unsigned integer" :

   (A) "If the subject sequence is empty or does not have the expected form" :

   1. "no conversion [is] performed"; ...
   2. "the value of ' `str` ' (' `p_str` ') [is] stored in the object pointed to by ' `endptr` ' (' `p_str_next` '), provided that ' `endptr` ' (' `p_str_next` ') is not a null pointer."

   (B) "If the subject sequence has the expected form" :

   1. (a) "and the value of 'base' (' `nbr_base` ') is 0, the sequence of characters starting with the first digit shall be interpreted as an integer constant."
      (b) "and the value of 'base' (' `nbr_base` ') is between 2 and 36, it shall be used as the base for conversion, ascribing to each letter its value as given above" (see Note #2a1B1b1A).
   2. "A pointer to the final string shall be stored in the object pointed to by ' `endptr` ' (' `p_str_next` '), provided that ' `endptr` ' (' `p_str_next` ') is not a null pointer."

3. Lastly, IEEE Std 1003.1, 2004 Edition, Section ' `strtoul()` : RETURN VALUE' states that :

   (A) "Upon successful completion, these functions shall return the converted value."

   1. "If the correct value is outside the range of representable values, {ULONG_MAX} ... shall be returned."

   (B) "If no conversion could be performed, 0 shall be returned."

(b)

1. IEEE Std 1003.1, 2004 Edition, Section ' `strtoul()` : ERRORS' states that "these functions shall fail if" :
   (A) "[EINVAL] - The value of 'base' (' `nbr_base` ') is not supported."
   (B) "[ERANGE] - The value to be returned is not representable."
2. IEEE Std 1003.1, 2004 Edition, Section ' `strtoul()` : ERRORS' states that "these functions may fail if" :
   (A) "[EINVAL] - No conversion could be performed."

3. Return integer value and next string pointer should be used to diagnose parse success or failure :

   (a) Valid parse string integer :

```
p_str = " ABCDE xyz"
nbr_base = 16

nbr = 703710
p_str_next = " xyz"
```

(b) Invalid parse string integer :

```
p_str = " ABCDE"
nbr_base = 10

nbr = 0
p_str_next = p_str = " ABCDE"
```

(c) Valid hexadecimal parse string integer :

```
p_str = " 0xGABCDE"
nbr_base = 16

nbr = 0
p_str_next = "xGABCDE"
```

(d) Valid decimal parse string integer ('0x' prefix ignored following invalid hexadecimal characters) :

```
p_str = " 0xGABCDE"
nbr_base = 0

nbr = 0
p_str_next = "xGABCDE"
```

(e) Valid decimal parse string integer ('0' prefix ignored following invalid octal characters) :

```
p_str = " 0GABCDE"
nbr_base = 0

nbr = 0
p_str_next = "GABCDE"
```

(f) Parse string integer overflow :

```
p_str = " 12345678901234567890*123456"
nbr_base = 10

nbr = DEF_INT_32U_MAX_VAL
p_str_next = "*123456"
```

(g) Invalid negative unsigned parse string :

```
p_str = " -12345678901234567890*123456"
nbr_base = 10

nbr = 0
p_str_next = p_str = " -12345678901234567890*123456"
```

Str_ParseNbr_Int32S()

Description

Parses a 32-bit signed integer from a string.

Files

`lib_str.h/lib_str.c`

Prototype

```
CPU_INT32S Str_ParseNbr_Int32S (const CPU_CHAR   *p_str,
                                CPU_CHAR   *p_str_next,
                                CPU_INT08U  nbr_base)
```

Arguments

`p_str`

Pointer to the string (see Notes #1 and #2a).

`p_str_next`

Optional pointer to a variable to ... :

(a) Return a pointer to first character following the integer string, if NO error(s) [see Note #2a2B2];

(b) Return a pointer to 'p_str', otherwise (see Note #2a2A2).

`nbr_base`

Base of number to parse (see Notes #2a1B1 and #2a2B1).

Returned Value

Parsed integer, if integer parsed with NO over- or underflow (see Note #2a3A).

- `DEF_INT_32S_MIN_VAL` , if integer parsed but negatively underflowed (see Note #2a3A1a).
- `DEF_INT_32U_MAX_VAL` , if integer parsed but positively overflowed (see Note #2a3A1b).

0, otherwise (see Note #2a3B).

Notes / Warnings

1. String buffer NOT modified.
2. (a) IEEE Std 1003.1, 2004 Edition, Section ' `strtol()` : DESCRIPTION' states that "these functions shall convert the initial portion of the string pointed to by 'str' (' `p_str` ') to a type long ... representation" :
    1. "First, they decompose the input string into three parts" :
       (A) "An initial, possibly empty, sequence of white-space characters [as specified by][as specified by `isspace()`]."
        1. "The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character that is of the expected form. The subject sequence shall contain no characters if the input string is empty or consists entirely of white-space characters."

       (B)
        1. "A subject sequence interpreted as an integer represented in some radix determined by the value of 'base' (' `nbr_base` ')" :
           (a) "If the value of 'base' (' `nbr_base` ') is 0, the expected form of the subject sequence is that of a decimal constant, octal constant, or hexadecimal constant" :
            1. "A decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits."
            2. "An octal constant consists of the prefix '0' optionally followed by a sequence of the digits '0' to '7' only."
            3. "A hexadecimal constant consists of the prefix '0x' or '0X' followed by a sequence of the decimal digits and letters 'a' (or 'A') to 'f' (or 'F') with values 10 to 15 respectively."

           (b) "If the value of 'base' (' `nbr_base` ') is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by 'base' (' `nbr_base` ')" :
            1. (A) "The letters from 'a' (or 'A') to 'z' (or 'Z') inclusive are ascribed the values 10 to 35"; ...
               (B) "only letters whose ascribed values are less than that of base are permitted."
            2. (A) "If the value of 'base' (' `nbr_base` ') is 16, the characters '0x' or '0X' may optionally precede the sequence of letters and digits."
               (B) Although NO specification states that "if the value of 'base' (' `nbr_base` ') is" 8, the '0' character "may optionally precede the sequence of letters and digits"; it seems reasonable to allow the '0' character to be optionally parsed.
        2. "A subject sequence .... may be preceded by a '+' or '-' sign."

(a) However, it does NOT seem reasonable to parse & convert a negative number integer string into an unsigned integer.

(C)

1. (a) "A final string of one or more unrecognized characters," ...
   (b) "including the terminating null byte of the input string" ...
2. "other than a sign or a permissible letter or digit."

2. Second, "they shall attempt to convert the subject sequence to an integer" :
   (A) "If the subject sequence is empty or does not have the expected form" :
   1. "no conversion is performed"; ...
   2. "the value of 'str' (' `p_str` ') is stored in the object pointed to by 'endptr' (' `p_str_next` '), provided that 'endptr' (' `p_str_next` ') is not a null pointer."

   (B) "If the subject sequence has the expected form" :
   1. (a) "and the value of 'base' (' `nbr_base` ') is 0, the sequence of characters starting with the first digit shall be interpreted as an integer constant."
      (b) "and the value of 'base' (' `nbr_base` ') is between 2 and 36, it shall be used as the base for conversion, ascribing to each letter its value as given above" (see Note #2a1B1b1A).
   2. "A pointer to the final string shall be stored in the object pointed to by 'endptr' (' `p_str_next` '), provided that 'endptr' (' `p_str_next` ') is not a null pointer."

3. Lastly, IEEE Std 1003.1, 2004 Edition, Section ' `strtol()` : RETURN VALUE' states that :
   (A) "Upon successful completion, these functions shall return the converted value."
   1. "If the correct value is outside the range of representable values", either of the following "shall be returned" :
      (a) "{LONG_MIN}" or ...
      (b) "{LONG_MAX}"

   (B) "If no conversion could be performed, 0 shall be returned."

(b)

1. IEEE Std 1003.1, 2004 Edition, Section ' `strtoul()` : ERRORS' states that "these functions shall fail if" :
   (A) "[EINVAL] - The value of 'base' ('nbr_base') is not supported."
   (B) "[ERANGE] - The value to be returned is not representable."
2. IEEE Std 1003.1, 2004 Edition, Section ' `strtoul()` : ERRORS' states that "these functions may fail if" :
   (A) "[EINVAL] - No conversion could be performed."

3. Return integer value and next string pointer should be used to diagnose parse success or failure :
   (a) Valid parse string integer :

```
p_str = " ABCDE xyz"
nbr_base = 16

nbr = 703710
p_str_next = " xyz"
```

(b) Invalid parse string integer :

```
p_str = " ABCDE"
nbr_base = 10

nbr = 0
p_str_next = p_str = " ABCDE"
```

(c) Valid hexadecimal parse string integer :

```
p_str = " 0xGABCDE"
nbr_base = 16

nbr = 0
p_str_next = "xGABCDE"
```

(d) Valid decimal parse string integer ('0x' prefix ignored following invalid hexadecimal characters) :

```
p_str = " 0xGABCDE"
nbr_base = 0

nbr = 0
p_str_next = "xGABCDE"
```

(e) Valid decimal parse string integer ('0' prefix ignored following invalid octal characters) :

```
p_str = " 0GABCDE"
nbr_base = 0

nbr = 0
p_str_next = "GABCDE"
```

(f) Parse string integer overflow :

```
p_str = " 12345678901234567890\*123456"
nbr_base = 10

nbr = DEF_INT_32S_MAX_VAL
p_str_next = "\*123456"
```

(g) Parse string integer underflow :

```
p_str = " -12345678901234567890\*123456"
nbr_base = 10

nbr = DEF_INT_32S_MIN_VAL
p_str_next = "\*123456"
```

API LIB Utilities

- DEF_BIT()
- DEF_BITxx()
- DEF_BIT_MASK()
- DEF_BIT_MASK_xx()
- DEF_BIT_FIELD()
- DEF_BIT_FIELD_xx()
- DEF_BIT_SET()
- DEF_BIT_SET_xx() (deprecated)
- DEF_BIT_CLR()
- DEF_BIT_CLR_xx() (deprecated)
- DEF_BIT_TOGGLE()
- DEF_BIT_FIELD_RD()
- DEF_BIT_FIELD_ENC()
- DEF_BIT_FIELD_WR()
- DEF_BIT_IS_SET()
- DEF_BIT_IS_CLR()
- DEF_BIT_IS_SET_ANY()
- DEF_BIT_IS_CLR_ANY()
- DEF_GET_U_MAX_VAL()
- CONTAINER_OF()

DEF_BIT()

Description

Create bit mask with single, specified bit set.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
DEF_BIT (bit)
```

Arguments

`bit`

Bit number of bit to set.

Returned Value

Bit mask with single, specified bit set.

Notes / Warnings

1. ' `bit` ' SHOULD be a non-negative integer.
2. (a) ' `bit` ' values that overflow the target CPU &/or compiler environment (e.g., negative or greater-than-CPU-data-size values) MAY generate compiler warnings &/or errors.

DEF_BITxx()

Description

Create bit mask of specified bit size with single, specified bit set.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
DEF_BIT08 (bit)

DEF_BIT16 (bit)

DEF_BIT32 (bit)

DEF_BIT64 (bit)
```

Arguments

`bit`

Bit number of bit to set.

Returned Value

Bit mask with single, specified bit set.

Notes / Warnings

1. ' `bit` ' SHOULD be a non-negative integer.
2. (a) ' `bit` ' values that overflow the target CPU &/or compiler environment (e.g., negative or greater-than-CPU-data-size values) MAY generate compiler warnings &/or errors.
   (b) To avoid overflowing any target CPU &/or compiler's integer data type, unsigned bit constant '1' is cast to specified integer data type size.

3. Ideally, `DEF_BITxx()` macro's should be named `DEF_BIT_xx()` ; however, these names already previously-released for bit constant #define's (see 'STANDARD DEFIN BIT DEFINES').

DEF_BIT_MASK()

Description

Shift a bit mask.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
DEF_BIT_MASK (bit_mask, bit_shift)
```

Arguments

`bit_mask`

Bit mask to shift.

`bit_shift`

Number of bit positions to left-shift bit mask.

Returned Value

Shifted bit mask.

Notes / Warnings

1. (a) '`bit_mask`' SHOULD be an unsigned integer.
   (b) '`bit_shift`' SHOULD be a non-negative integer.
2. '`bit_shift`' values that overflow the target CPU &/or compiler environment (e.g., negative or greater-than-CPU-data-size values) MAY generate compiler warnings &/or errors.

DEF_BIT_MASK_xx()

Description

Shift a bit mask of specified bit size.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
DEF_BIT_MASK_08 (bit_mask, bit_shift)

DEF_BIT_MASK_16 (bit_mask, bit_shift)

DEF_BIT_MASK_32 (bit_mask, bit_shift)

DEF_BIT_MASK_64 (bit_mask, bit_shift)
```

Arguments

`bit_mask`

Bit mask to shift.

`bit_shift`

Number of bit positions to left-shift bit mask.

Returned Value

Shifted bit mask.

Notes / Warnings

1. (a) ' `bit_mask` ' SHOULD be an unsigned integer.
   (b) ' `bit_shift` ' SHOULD be a non-negative integer.
2. ' `bit_shift` ' values that overflow the target CPU &/or compiler environment (e.g., negative or greater-than-CPU-data-size values) MAY generate compiler warnings &/or errors.

DEF_BIT_FIELD()

Description

Create & shift a contiguous bit field.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
DEF_BIT_FIELD (bit_field, bit_shift)
```

Arguments

`bit_field`

Number of contiguous bits to set in the bit field.

`bit_shift`

Number of bit positions to left-shift bit field.

Returned Value

Shifted bit field.

Notes / Warnings

1. ' `bit_field` ' & ' `bit_shift` ' SHOULD be non-negative integers.
2. (a) ' `bit_field` '/' `bit_shift` ' values that overflow the target CPU &/or compiler environment (e.g., negative or greater-than-CPU-data-size values) MAY generate compiler warnings &/or errors.
   (b) To avoid overflowing any target CPU &/or compiler's integer data type, unsigned bit constant '1' is suffixed with 'L'ong integer modifier.

DEF_BIT_FIELD_xx()

Description

Create & shift a contiguous bit field of specified bit size.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
DEF_BIT_FIELD_08 (bit_field, bit_shift)

DEF_BIT_FIELD_16 (bit_field, bit_shift)

DEF_BIT_FIELD_32 (bit_field, bit_shift)

DEF_BIT_FIELD_64 (bit_field, bit_shift)
```

Arguments

`bit_field`

Number of contiguous bits to set in the bit field.

`bit_shift`

Number of bit positions to left-shift bit field.

Returned Value

Shifted bit field.

Notes / Warnings

1. ' `bit_field` ' & ' `bit_shift` ' SHOULD be non-negative integers.
2. (a) ' `bit_field` '/' `bit_shift` ' values that overflow the target CPU &/or compiler environment (e.g., negative or greater-than-CPU-data-size values) MAY generate compiler warnings &/or errors.
   (b) To avoid overflowing any target CPU &/or compiler's integer data type, unsigned bit constant '1' is cast to specified integer data type size.

DEF_BIT_SET()

Description

Set specified bit(s) in a value.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
DEF_BIT_SET (val, mask)
```

Arguments

`val`

Value to modify by setting specified bit(s).

`mask`

Mask of bits to set.

Returned Value

Modified value with specified bit(s) set.

Notes / Warnings

1. ' `val` ' & ' `mask` ' SHOULD be unsigned integers.

DEF_BIT_SET_xx() (deprecated)

Description

Set specified bit(s) in a value of specified bit size.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
DEF_BIT_SET_08 (val, mask)

DEF_BIT_SET_16 (val, mask)

DEF_BIT_SET_32 (val, mask)

DEF_BIT_SET_64 (val, mask)
```

Arguments

`val`

Value to modify by setting specified bit(s).

`mask`

Mask of bits to set.

Returned Value

Modified value with specified bit(s) set.

Notes / Warnings

1. ' `val` ' & ' `mask` ' SHOULD be unsigned integers.
2. These macros are deprecated and should be replaced by the `DEF_BIT_SET` macro.

DEF_BIT_CLR()

Description

Clear specified bit(s) in a value.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
DEF_BIT_CLR (val, mask)
```

Arguments

`val`

Value to modify by clearing specified bit(s).

`mask`

Mask of bits to clear.

Returned Value

Modified value with specified bit(s) clear.

Notes / Warnings

1. ' `val` ' & ' `mask` ' SHOULD be unsigned integers.

DEF_BIT_CLR_xx() (deprecated)

Description

Clear specified bit(s) in a value of specified bit size.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
DEF_BIT_CLR_08 (val, mask)
DEF_BIT_CLR_16 (val, mask)
DEF_BIT_CLR_32 (val, mask)
DEF_BIT_CLR_64 (val, mask)
```

Arguments

`val`

Value to modify by clearing specified bit(s).

`mask`

Mask of bits to clear.

Returned Value

Modified value with specified bit(s) clear.

Notes / Warnings

1. ' `val` ' & ' `mask` ' SHOULD be unsigned integers.
2. These macros are deprecated and should be replaced by the `DEF_BIT_CLR` macro.

DEF_BIT_TOGGLE()

Description

Toggles specified bit(s) in a value.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
DEF_BIT_TOGGLE (val, mask)
```

Arguments

`val`

Value to modify by toggling specified bit(s).

`mask`

Mask of bits to toggle.

Returned Value

Modified value with specified bit(s) toggled.

Notes / Warnings

1. ' `val` ' & ' `mask` ' SHOULD be unsigned integers.

DEF_BIT_FIELD_RD()

Description

Reads a ' `val` ' field, masked and shifted, given by mask ' `field_mask` '.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
DEF_BIT_FIELD_RD (val, field_mask)
```

Arguments

`val`

Value to read from.

`field_mask`

Mask of field to read. See note #1, #2 and #3.

Returned Value

Field value, masked and right-shifted to bit position 0.

Notes / Warnings

1. ' `field_mask` ' argument must NOT be 0.
2. ' `field_mask` ' argument must contain a mask with contiguous set bits.
3. ' `val` ' & ' `field_mask` ' SHOULD be unsigned integers.

DEF_BIT_FIELD_ENC()

Description

Encodes given ' `field_val` ' at position given by mask ' `field_mask` '.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
DEF_BIT_FIELD_ENC (field_val, field_mask)
```

Arguments

`field_val`

Value to encode.

`field_mask`

Mask of field to read. See note #1 and #2.

Returned Value

Field value, masked and left-shifted to field position.

Notes / Warnings

1. ' `field_mask` ' argument must contain a mask with contiguous set bits.
2. ' `field_val` ' & ' `field_mask` ' SHOULD be unsigned integers.

DEF_BIT_FIELD_WR()

Description

Writes ' `field_val` ' field at position given by mask ' `field_mask` ' in variable ' `var` '.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
DEF_BIT_FIELD_WR (var, field_val, field_mask)
```

Arguments

`var`

Variable to write field to. See note #2.

`field_val`

Desired value for field. See note #2.

`field_mask`

Mask of field to write to. See note #1 and #2.

Returned Value

None.

Notes / Warnings

1. ' `field_mask` ' argument must contain a mask with contiguous set bits.
2. ' `var` ', ' `field_val` ' & ' `field_mask` ' SHOULD be unsigned integers.

DEF_BIT_IS_SET()

Description

Determine if specified bit(s) in a value are set.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
DEF_BIT_IS_SET (val, mask)
```

Arguments

`val`

Value to check for specified bit(s) set.

`mask`

Mask of bits to check if set (see Note #2).

Returned Value

`DEF_YES` , if ALL specified bit(s) are set in value.

Notes / Warnings

1. ' `val` ' & ' `mask` ' SHOULD be unsigned integers.
2. NULL ' `mask` ' allowed; returns ' `DEF_NO` ' since NO mask bits specified.

DEF_BIT_IS_CLR()

Description

Determine if specified bit(s) in a value are clear.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
DEF_BIT_IS_CLR (val, mask)
```

Arguments

`val`

Value to check for specified bit(s) clear.

`mask`

Mask of bits to check if clear (see Note #2).

Returned Value

`DEF_YES` , if ALL specified bit(s) are clear in value.

Notes / Warnings

1. ' `val` ' & ' `mask` ' SHOULD be unsigned integers.
2. NULL ' `mask` ' allowed; returns ' `DEF_NO` ' since NO mask bits specified.

DEF_BIT_IS_SET_ANY()

Description

Determine if any specified bit(s) in a value are set.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
DEF_BIT_IS_SET_ANY (val, mask)
```

Arguments

`val`

Value to check for specified bit(s) set.

`mask`

Mask of bits to check if set (see Note #2).

Returned Value

`DEF_YES` , if ANY specified bit(s) are set in value.

Notes / Warnings

1. ' `val` ' & ' `mask` ' SHOULD be unsigned integers.

2. NULL ' `mask` ' allowed; returns ' `DEF_NO` ' since NO mask bits specified.

DEF_BIT_IS_CLR_ANY()

Description

Determine if any specified bit(s) in a value are clear.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
DEF_BIT_IS_CLR_ANY (val, mask)
```

Arguments

`val`

Value to check for specified bit(s) clear.

`mask`

Mask of bits to check if clear (see Note #2).

Returned Value

DEF_YES, if ANY specified bit(s) are clear in value.

Notes / Warnings

1. ' `val` ' & ' `mask` ' SHOULD be unsigned integers.
2. NULL ' `mask` ' allowed; returns ' `DEF_NO` ' since NO mask bits specified.

DEF_GET_U_MAX_VAL()

Description

Get the maximum unsigned value that can be represented in an unsigned integer variable of the same data type size as an object.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
DEF_GET_U_MAX_VAL (obj)
```

Arguments

`obj`

Object or data type to return maximum unsigned value (see Note #1).

Returned Value

Maximum unsigned integer value that can be represented by the object, if NO error(s).

Notes / Warnings

1. ' `obj` ' SHOULD be an integer object or data type but COULD also be a character or pointer object or data type.

CONTAINER_OF()

Description

Find pointer to structure type ' `parent_type` ', containing ' `p_member` '.

Files

`lib_utils.h`

Prototype

```
CONTAINER_OF(p_member, parent_type, member)
```

Arguments

`p_member`

Pointer to member of structure of which the pointer to container structure is found.

`parent_type`

Name of the parent structure data type.

`member`

Name of the member field, in the parent structure data type.

Returned Value

Pointer to structure containing ' `p_member` '.

Notes / Warnings

1. ' `p_member` ' SHOULD NOT be `DEF_NULL` .

Logging API

Logging API

1. Log_DataIsAvail()
2. Log_Output()

Log_DataIsAvail()

Description

Indicates if there is data present to be logged or not.

Files

`logging.h/logging.c`

Prototype

```
CPU_BOOLEAN  Log_DataIsAvail (void)
```

Arguments

None.

Returned Value

- DEF_YES, if there is data that is available to be logged.
- DEF_FAIL, otherwise.

Notes / Warnings

None.

Log_Output()

Description

Outputs data accumulated in the ring buffer.

Files

`logging.h/logging.c`

Prototype

```
void  Log_Output (void)
```

Arguments

None.

Returned Value

None.

Notes / Warnings

None.

RTOS Err API

RTOS_ERR API

- RTOS_ERR_CODE_GET()
- RTOS_ERR_STR_GET()
- RTOS_ERR_DESC_STR_GET()
- RTOS_ERR_SET()
- RTOS_ERR_COPY()

RTOS_ERR_CODE_GET()

Description

Obtains the error code from an error variable, no matter how RTOS_ERR_CFG_LEGACY_EN is configured.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
RTOS_ERR_CODE_GET (err_var)
```

Arguments

`err_var`

Error variable from which to obtain error code.

Returned Value

Error code in passed variable.

Notes / Warnings

None.

RTOS_ERR_STR_GET()

Description

Obtains the description string from an error code. Will return a string saying strings are unavailable if RTOS_ERR_CFG_STR_EN is set to DEF_DISABLED.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
RTOS_ERR_STR_GET (err_code)
```

Arguments

`err_code`

Error code for which to obtain the string.

Returned Value

The requested error description string.

Notes / Warnings

None.

RTOS_ERR_DESC_STR_GET()

Description

Obtains the error string from an error code. Will return a string saying strings are unavailable if RTOS_ERR_CFG_STR_EN is set to DEF_DISABLED.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
RTOS_ERR_DESC_STR_GET (err_code)
```

Arguments

`err_code`

Error code for which to obtain the string.

Returned Value

The requested error string.

Notes / Warnings

None.

RTOS_ERR_SET()

Description

Sets a given error variable to an error value. Also sets the extended error, if enabled.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
RTOS_ERR_SET(err_var, err_code)
```

Arguments

`err_var`

Error variable to set to 'err'.

`err_code`

Error value to which to set 'err_var'.

Returned Value

None.

Notes / Warnings

None.

RTOS_ERR_COPY()

Description

Copies every field of an error variable to another one.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
RTOS_ERR_COPY (err_dst, err_src)
```

Arguments

`err_dst`

Destination error variable.

`err_src`

Source error variable.

Returned Value

None.

Notes / Warnings

1. This wrapper is present if other fields require particular manipulations to copy, even if a simple assignation could have been enough for the moment.

Toolchain Abstraction API

Toolchain Abstraction API

- PP_UNUSED_PARAM()
- PP_ALIGN()
- PP_ISR_DECL()
- PP_ISR_DEF()

PP_UNUSED_PARAM()

Description

Removes warning associated to a function parameter being present but not referenced in a given function.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
PP_UNUSED_PARAM (param)
```

Arguments

`param`

Parameter that is unused.

Returned Value

None.

Notes / Warnings

1. This macro can be overriden by defining it first in the compiler options.

PP_ALIGN()

This macro is deprecated and will be removed in an upcoming release. Use CMSIS __ALIGN instead.

Description

Forces variable to be aligned on specific memory multiple.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
PP_ALIGN (_variable, _align)
```

Arguments

`_variable`

Variable to align.

`_align`

Alignment required, in bytes.

Returned Value

None.

Notes / Warnings

1. This macro can be overriden by defining it first in the compiler options.

PP_ISR_DECL()

> This macro is deprecated and will be removed in an upcoming release.

Description

Declare a function, indicating to the compiler it is an ISR.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
PP_ISR_DECL (_isr)
```

Arguments

`_isr`

The ISR function's name.

Returned Value

None.

Notes / Warnings

1. This macro can be overriden by defining it first in the compiler options.
2. Usage is as follows: `PP_ISR_DECL(My_ISR_Function);`

PP_ISR_DEF()

> This macro is deprecated and will be removed in an upcoming release.

Description

Define a function, indicating to the compiler that it is an ISR.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
PP_ISR_DEF (_isr)
```

Arguments

`_isr`

The ISR function's name.

Returned Value

None.

Notes / Warnings

1. This macro can be overriden by defining it first in the compiler options.
2. Usage is as follows: `PP_ISR_DEF(My_ISR_Function) { [...] }`

Utilities API

Utilities API

- APP_RTOS_ASSERT_CRITICAL()
- APP_RTOS_ASSERT_DBG()
- APP_RTOS_ASSERT_CRITICAL_FAIL()
- APP_RTOS_ASSERT_DBG_FAIL()
- RTOS_Version()

APP_RTOS_ASSERT_CRITICAL()

Description

Assert given expression. In case of failure, calls `RTOS_CFG_RTOS_ASSERT_CRITICAL_FAILED_END_CALL(ret_val)` if defined, `CPU_SW_EXCEPTION(ret_val)` if not.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
APP_RTOS_ASSERT_CRITICAL (expr, ret_val)
```

Arguments

`expr`

Expression to assert. If expression is false, the assert fail call will be made.

`ret_val`

Value that would be returned from the function, ';' if void.

Returned Value

None.

Notes / Warnings

1. Usage of assert is as follows: `APP_RTOS_ASSERT_CRITICAL((p_buf != DEF_NULL), ;);`

APP_RTOS_ASSERT_DBG()

Description

Assert given expression. In case of failure, calls `RTOS_CFG_RTOS_ASSERT_DBG_FAILED_END_CALL(ret_val)` if defined, `CPU_SW_EXCEPTION(ret_val)` if not.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
APP_RTOS_ASSERT_DBG (expr, ret_val)
```

Arguments

`expr`

Expression to assert. If expression is false, the assert fail call will be made.

`ret_val`

Value that would be returned from the function, ';' if void.

Returned Value

None.

Notes / Warnings

1. Usage of assert is as follows: `APP_RTOS_ASSERT_DBG((p_buf != DEF_NULL), DEF_NULL);`

APP_RTOS_ASSERT_CRITICAL_FAIL()

Description

Calls END_CALL as configured by user. No check is made.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
APP_RTOS_ASSERT_CRITICAL_FAIL (ret_val)
```

Arguments

`ret_val`

Value that would be returned from the function, ';' if void.

Returned Value

None.

Notes / Warnings

None.

APP_RTOS_ASSERT_DBG_FAIL()

Description

Checks if assert is enabled for current module. Calls END_CALL as configured by user. No check is made.

Files

`lib_mem.h/lib_mem.c`

Prototype

```
APP_RTOS_ASSERT_DBG_FAIL (ret_val)
```

Arguments

`ret_val`

Value that would be returned from the function, ';' if void.

Returned Value

None.

Notes / Warnings

None.

RTOS_Version()

Description

Obtain current version of Micrium OS.

Files

`rtos_version.h/rtos_utils.c`

Prototype

```
CPU_INT32U  RTOS_Version (void)
```

Arguments

None.

Returned Value

Version number of Micrium OS (Vx.yy.zz) multiplied by 10000.

Notes / Warnings

None.

Shell API

Shell API

- Shell_ConfigureCmdUsage()
- Shell_ConfigureMemSeg()
- Shell_Init()
- Shell_Exec()
- Shell_CmdTblAdd()
- Shell_CmdTblRem()
- Shell_Scanner()
- Shell_OptParse()

Shell_ConfigureCmdUsage()

Description

Configure the properties of the commands used by the Shell sub-module.

Files

`shell.h/shell.c`

Prototype

```
void  Shell_ConfigureCmdUsage (SHELL_CFG_CMD_USAGE  *p_cfg)
```

Arguments

`p_cfg`

Pointer to the structure containing the command usage parameters.

Returned Value

None.

Notes / Warnings

1. This function is optional. If it is called, it must be called before `Shell_Init()`. If it is not called, default values will be used to initialize the module.

Shell_ConfigureMemSeg()

Description

Configure the memory segment that will be used to allocate internal data needed by Shell instead of the default memory segment.

Files

`shell.h/shell.c`

Prototype

```
void Shell_ConfigureMemSeg (MEM_SEG *p_mem_seg)
```

Arguments

`p_mem_seg`

- Pointer to the memory segment from which the internal data will be allocated.
- If `DEF_NULL`, the internal data will be allocated from the global Heap.

Returned Value

None.

Notes / Warnings

1. This function is optional. If it is called, it must be called before `Shell_Init()`. If it is not called, default values will be used to initialize the module.

Shell_Init()

Description

Initializes the shell module.

Files

`shell.h/shell.c`

Prototype

```
void Shell_Init (RTOS_ERR *p_err)
```

Arguments

`p_err`

Pointer to variable that receives the following return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_OWNERSHIP`
- `RTOS_ERR_ALREADY_EXISTS`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_NO_MORE_RSRC`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

None.

Notes / Warnings

1. `Shell_Init()` MUST be called before the other Shell functions are invoked.
2. `Shell_Init()` MUST ONLY be called ONCE from the product's application.
3. Module command pools MUST be initialized before initializing the pool with pointers to module commands.
4. The functions `Shell_Configure...()` can be used to configure more specific properties of the Shell sub-module, when `RTOS_CFG_EXTERNALIZE_OPTIONAL_CFG_EN` is set to 1. If set to 1, the structure Shell_InitCfg needs to be declared and filled by the application to configure these specific properties for the module.

Shell_Exec()

Description

Parses and executes a command passed in the following parameters :

(a) Parse input string

(b) Search command

(c) Execute command

Files

`shell.h/shell.c`

Prototype

```
CPU_INT16S  Shell_Exec (CPU_CHAR         *in,
                        SHELL_OUT_FNCT    out_fnct,
                        SHELL_CMD_PARAM  *p_cmd_param,
                        RTOS_ERR         *p_err)
```

Arguments

`in`

Pointer to a CPU_CHAR string holding a complete command and its argument(s). This string MUST be editable, it cannot be 'const'.

`out_fnct`

Pointer to the 'output' function used by command (see Note #1).

`p_cmd_param`

Pointer to the command additional parameters.

`p_err`

Pointer to variable that receives the following return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_SHELL_CMD_EXEC`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_CMD_EMPTY`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_NO_MORE_RSRC`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

Command specific return value.

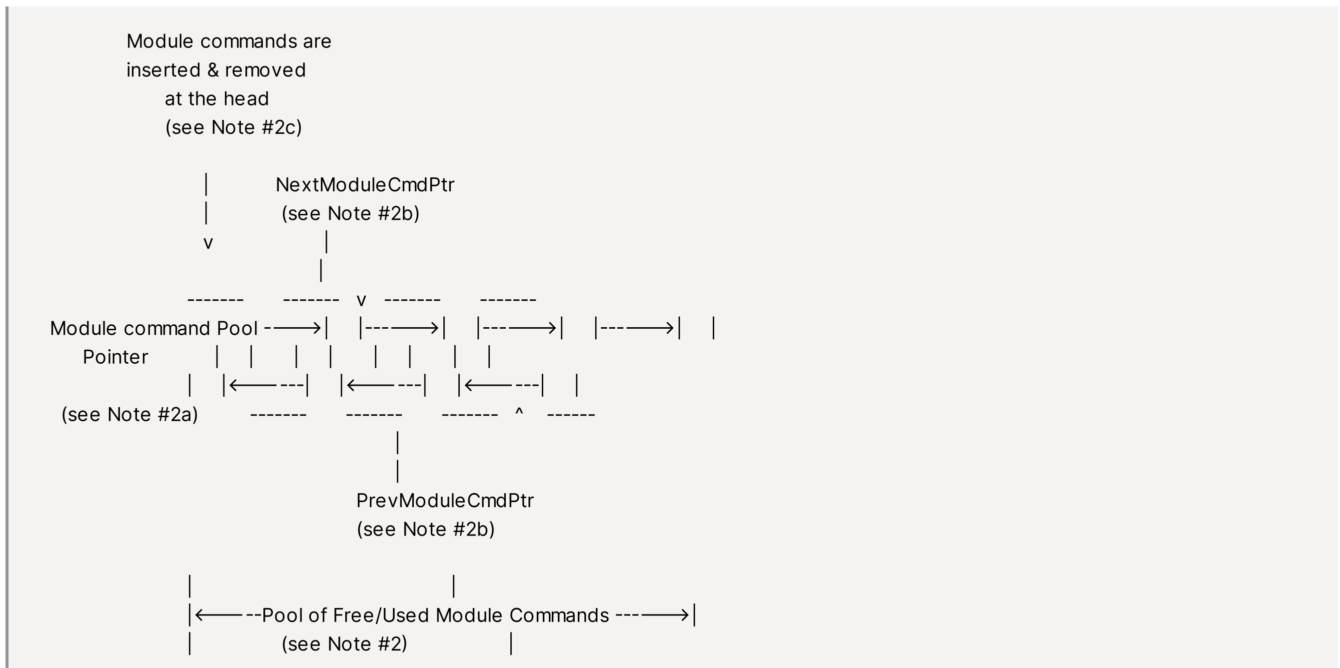
Notes / Warnings

1. The command may generate some output that should be transmitted to some devices (socket, RS-232 link, ...). The caller of this function is responsible for the implementation of such functions, if output is desired.

Shell_CmdTblAdd()

Description

1. Allocates and initializes a module command as follows :
   (a) Validate module command
   (b) Get a free module command
   (c) Initialize module command
   (d) Add to module command used pool.
2. The module command pools are implemented as doubly-linked lists :
   (a) 'Shell_ModuleCmdUsedPoolPtr' and 'Shell_ModuleCmdFreePoolPtr' points to the head of the module command pool.
   (b) Module command NextModuleCmdPtr's and PrevModuleCmdPtr's links each command to form the module command pool doubly-linked list.
   (c) Module command are inserted and removed at the head of the module command pool lists.



Files

`shell.h/shell.c`

Prototype

```
void  Shell_CmdTblAdd (CPU_CHAR   *cmd_tbl_name,
                       SHELL_CMD   cmd_tbl[],
                       RTOS_ERR   *p_err)
```

Arguments

`cmd_tbl_name`

Pointer to character string representing the name of the command table.

`cmd_tbl`

Command table to add.

`p_err`

Pointer to variable that receives the following return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_OWNERSHIP`
- `RTOS_ERR_ALREADY_EXISTS`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_NO_MORE_RSRC`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

None.

Notes / Warnings

1. The 'cmd_tbl_name' parameter is not mandatory in the current implementation. Although you could pass a 'NULL' value for this parameter, it is recommended to provide it to allow the removal of 'cmd_tbl' from the Shell_CmdTblRem().

However, passing NULL for this parameter will result in the first command prefix to be extracted and used as the command table name.

1. If an empty character array is passed in the cmd_tbl_name parameter, the function will extract the first command prefix to use as the command table name.

Shell_CmdTblRem()

Description

Removes a module command as follows :

(a) Search module command (b) Remove module command (c) Update module command pools

Files

`shell.h/shell.c`

Prototype

```
void Shell_CmdTblRem (CPU_CHAR  *cmd_tbl_name,                RTOS_ERR  *p_err)
```

Arguments

`cmd_tbl_name`

Pointer to character string representing the name of the command table.

p_err

Pointer to variable that receives the following return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_POOL_FULL
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_OWNERSHIP
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_NOT_FOUND
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_IS_OWNER
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_ABORT
- RTOS_ERR_TIMEOUT

Returned Value

None.

Notes / Warnings

None.

Shell_Scanner()

Description

Scans and parses the command line.

Files

shell.h/shell.c

Prototype

```
CPU_INT16U  Shell_Scanner (CPU_CHAR    *in,
                           CPU_CHAR    *arg_tbl[],
                           CPU_INT16U   arg_tbl_size,
                           RTOS_ERR    *p_err)
```

Arguments

in

Pointer to a NUL terminated string holding a complete command and its argument(s).

arg_tbl

Array of pointer that will receive pointers to token.

arg_tbl_size

Size of arg_tbl array.

p_err

Pointer to variable that receives the following return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_NO_MORE_RSRC

Returned Value

Number of token(s) (command name and argument(s)).

Notes / Warnings

1. The first token is always the command name itself.
2. This function modify the 'in' arguments by replacing token's delimiter characters by termination character ('\0').

Shell_OptParse()

Description

Parses optional command line arguments.

Files

`shell.h/shell.c`

Prototype

```
SHELL_OPTPARSE_STATE  Shell_OptParse (CPU_INT16U           argc,
                                      CPU_CHAR            *argv[],
                                      SHELL_OPTPARSE_STATE  state,
                                      CPU_CHAR            *opt_str)
```

Arguments

`argc`

Argument count.

`argv`

Pointer to argument table.

`state`

Argument parsing state structure.

`opt_str`

Argument string to parse.

Returned Value

SHELL_OPTPARSE_STATE.

Notes / Warnings

None.

RTOS Task Cfg

RTOS_TASK_CFG

Description

This structure is used to configure some of Micrium OS internal tasks.

File(s)

rtos_types.h

Fields

Name	Type	Description
.Prio	RTOS_TASK_PRIO	Priority of the task.
.StkSizeElements	CPU_STK_SIZE	Size of the task's stack, in number of elements.
.StkPtr	CPU_STK	Pointer to the base of the stack's task. If set to DEF_NULL, stack will be automatically allocated from common's memory segment .

Notes / Warnings

None.

CPU API

CPU API

- CPU Core API
- CPU Cache Management API

CPU Core API

CPU Core API

- CPU_BREAK()
- CPU_CntLeadZeros()
- CPU_CntLeadZeros08()
- CPU_CntLeadZeros16()
- CPU_CntLeadZeros32()
- CPU_CntLeadZeros64()
- CPU_CntTrailZeros()
- CPU_CntTrailZeros08()
- CPU_CntTrailZeros16()
- CPU_CntTrailZeros32()
- CPU_CntTrailZeros64()
- CPU_Init()
- CPU_NameClr()
- CPU_NameGet()
- CPU_NameSet()
- CPU_PopCnt32()
- CPU_SW_EXCEPTION()
- CPU_TS32_to_uSec()
- CPU_TS64_to_uSec()
- CPU_TS_Get32()
- CPU_TS_Get64()
- CPU_TS_TmrFreqGet()
- CPU_TS_TmrFreqSet()
- CPU_TS_Update()

CPU_BREAK()

Description

Creates a software-generated CPU break.

Files

`cpu.h/cpu_core.c`

Prototype

```
CPU_BREAK()
```

Arguments

None.

Returned Value

None.

Notes / Warnings

None.

CPU_CntLeadZeros()

Description

Counts the number of contiguous, most-significant, leading zero bits in a data value. Dispatches the call to the proper `CPU_CntLeadZeros??()` implementation according to the size of `CPU_DATA`.

Files

`cpu.h/cpu_core.c`

Prototype

```
CPU_DATA  CPU_CntLeadZeros (CPU_DATA  val)
```

Arguments

`val`

Data value to count the number of leading zero bits.

Returned Value

Number of contiguous, most-significant, leading zero bits in 'val', if NO error(s). `DEF_INT_CPU_U_MAX_VAL`, otherwise.

Notes / Warnings

1. For more information, see `CPU_CntLeadZeros()` function header comment block in `cpu_core.c`.

CPU_CntLeadZeros08()

Description

Counts the number of contiguous, most-significant, leading zero bits in an 8-bit data value.

Files

`cpu.h/cpu_core.c`

Prototype

```
CPU_DATA  CPU_CntLeadZeros08 (CPU_INT08U  val)
```

Arguments

`val`

Data value to count the number of leading zero bits.

Returned Value

Number of contiguous, most-significant, leading zero bits in 'val'.

Notes / Warnings

1. For more information, see `CPU_CntLeadZeros08()` function header comment block in `cpu_core.c`.

CPU_CntLeadZeros16()

Description

Counts the number of contiguous, most-significant, leading zero bits in a 16-bit data value.

Files

`cpu.h/cpu_core.c`

Prototype

```
CPU_DATA CPU_CntLeadZeros16 (CPU_INT16U val)
```

Arguments

`val`

Data value to count the number of leading zero bits.

Returned Value

Number of contiguous, most-significant, leading zero bits in 'val'.

Notes / Warnings

1. For more information, see `CPU_CntLeadZeros16()` function header comment block in `cpu_core.c` .

CPU_CntLeadZeros32()

Description

Counts the number of contiguous, most-significant, leading zero bits in a 32-bit data value.

Files

`cpu.h/cpu_core.c`

Prototype

```
CPU_DATA CPU_CntLeadZeros32 (CPU_INT32U val)
```

Arguments

`val`

Data value to count the number of leading zero bits.

Returned Value

Number of contiguous, most-significant, leading zero bits in ' `val` '.

Notes / Warnings

1. For more information, see `CPU_CntLeadZeros32()` function header comment block in `cpu_core.c` .

CPU_CntLeadZeros64()

Description

Counts the number of contiguous, most-significant, leading zero bits in a 64-bit data value.

Files

`cpu.h/cpu_core.c`

Prototype

```
CPU_DATA CPU_CntLeadZeros64 (CPU_INT64U val)
```

Arguments

`val`

Data value to count the number of leading zero bits.

Returned Value

Number of contiguous, most-significant, leading zero bits in ' `val` '.

Notes / Warnings

1. For more information, see `CPU_CntLeadZeros64()` function header comment block in `cpu_core.c` .

CPU_CntTrailZeros()

Description

Counts the number of contiguous, least-significant, trailing zero bits in a data value. This function dispatches the call to the proper `CPU_CntTrailZeros??()` implementation according to the size of `CPU_DATA` .

Files

`cpu.h/cpu_core.c`

Prototype

```
CPU_DATA  CPU_CntTrailZeros (CPU_DATA  val)
```

Arguments

`val`

Data value to count the number of trailing zero bits.

Returned Value

Number of contiguous, least-significant, trailing zero bits in ' `val` '.

Notes / Warnings

1. For more information, see `CPU_CntTrailZeros()` function header comment block in `cpu_core.c` .

CPU_CntTrailZeros08()

Description

Counts the number of contiguous, least-significant, trailing zero bits in an 8-bit data value.

Files

`cpu.h/cpu_core.c`

Prototype

```
CPU_DATA  CPU_CntTrailZeros08 (CPU_INT08U  val)
```

Arguments

`val`

Data value to count the number of trailing zero bits.

Returned Value

Number of contiguous, least-significant, trailing zero bits in ' `val` '.

Notes / Warnings

1. For more information, see `CPU_CntTrailZeros08()` function header comment block in `cpu_core.c` .

CPU_CntTrailZeros16()

Description

Counts the number of contiguous, least-significant, trailing zero bits in a 16-bit data value.

Files

`cpu.h/cpu_core.c`

Prototype

```
CPU_DATA  CPU_CntTrailZeros16 (CPU_INT16U  val)
```

Arguments

`val`

Data value to count the number of trailing zero bits.

Returned Value

Number of contiguous, least-significant, trailing zero bits in ' `val` '.

Notes / Warnings

1. For more information, see `CPU_CntTrailZeros16()` function header comment block in `cpu_core.c` .

CPU_CntTrailZeros32()

Description

Counts the number of contiguous, least-significant, trailing zero bits in a 32-bit data value.

Files

`cpu.h/cpu_core.c`

Prototype

```
CPU_DATA  CPU_CntTrailZeros32 (CPU_INT32U  val)
```

Arguments

`val`

Data value to count the number of trailing zero bits.

Returned Value

Number of contiguous, least-significant, trailing zero bits in ' `val` '.

Notes / Warnings

1. For more information, see `CPU_CntTrailZeros32()` function header comment block in `cpu_core.c` .

CPU_CntTrailZeros64()

Description

Counts the number of contiguous, least-significant, trailing zero bits in a 64-bit data value.

Files

`cpu.h/cpu_core.c`

Prototype

```
CPU_DATA  CPU_CntTrailZeros64 (CPU_INT64U  val)
```

Arguments

`val`

Data value to count the number of trailing zero bits.

Returned Value

Number of contiguous, least-significant, trailing zero bits in ' `val` '.

Notes / Warnings

1. For more information, see `CPU_CntTrailZeros64()` function header comment block in `cpu_core.c` .

CPU_Init()

Description

Initializes the CPU module :

- Initialize the CPU timestamps.
- Initialize the CPU interrupts disabled time measurements.
- Initialize the CPU host name.
- Initialize the CPU cache management.
- Initialize the CPU interrupt management.

Files

`cpu.h/cpu_core.c`

Prototype

```
void  CPU_Init (void)
```

Arguments

None.

Returned Value

None.

Notes / Warnings

1. `CPU_Init()` MUST be called :
   a. ONLY ONCE from a product's application.
   b. BEFORE product's application calls any core CPU module function.
2. The following initialization functions MUST be sequenced as follows :
   a. `CPU_TS_Init()` SHOULD precede ALL calls to other CPU timestamp functions.
   b. `CPU_IntDisMeasInit()` SHOULD precede ALL calls to `CPU_CRITICAL_ENTER()` / `CPU_CRITICAL_EXIT()` and other CPU interrupts disabled time measurement functions.
   c. `CPU_IntInit()` SHOULD precede ALL calls to other CPU interrupt management functions.

CPU_NameClr()

Description

Clears the CPU Name.

Files

`cpu.h/cpu_core.c`

Prototype

```
void  CPU_NameClr (void)
```

Arguments

None.

Returned Value

None.

Notes / Warnings

None.

CPU_NameGet()

Description

Gets the CPU host name.

Files

`cpu.h/cpu_core.c`

Prototype

```
void  CPU_NameGet (CPU_CHAR  *p_name,
                   RTOS_ERR  *p_err)
```

Arguments

`p_name`

Pointer to an ASCII character array that will receive the return CPU host.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

1. The size of the ASCII character array that will receive the return CPU host name ASCII string :
   a. MUST be greater than or equal to the current CPU host name's ASCII string size, including the terminating NULL character.
   b. SHOULD be greater than or equal to `CPU_CFG_NAME_SIZE` .

CPU_NameSet()

Description

Sets the CPU host name.

Files

`cpu.h/cpu_core.c`

Prototype

```
void  CPU_NameSet (const  CPU_CHAR  *p_name,
                   RTOS_ERR  *p_err)
```

Arguments

`p_name`

Pointer to a CPU host name to set.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_ARG`

Returned Value

None.

Notes / Warnings

1. The 'p_name' ASCII string size, including the terminating NULL character, MUST be less than or equal to `CPU_CFG_NAME_SIZE` .

CPU_PopCnt32()

Description

Calculates the population count (hamming weight) for value (number of bits set).

Files

`cpu.h/cpu_core.c`

Prototype

```
CPU_INT08U  CPU_PopCnt32 (CPU_INT32U  value)
```

Arguments

`value`

Value upon which to calculate the population.

Returned Value

The value's population count.

Notes / Warnings

1. This algorithm is taken from the Hamming Weight. For more information on the Hamming Weight, see Wikipedia .

CPU_SW_EXCEPTION()

Description

Trap unrecoverable software exception.

Files

`cpu.h/cpu_core.c`

Prototype

```
CPU_SW_EXCEPTION (CPU_DATA  err_rtn_val)
```

Arguments

`err_rtn_val`

Error type and/or value of the calling function to return.

Returned Value

None.

Notes / Warnings

1. For more information, see `CPU_SW_EXCEPTION()` function header comment block in cpu.h.

CPU_TS32_to_uSec()

Description

Converts a 32-bit CPU timestamp from timer counts to microseconds.

Files

`cpu.h/cpu_core.c`

Prototype

```
CPU_INT64U  CPU_TS32_to_uSec(CPU_TS32  ts_cnts)
```

Arguments

`ts_cnts`

CPU timestamp (in timestamp timer counts).

Returned Value

Converted CPU timestamp (in microseconds).

Notes / Warnings

1. For more information, see `CPU_TS32_to_uSec()` function header comment block in `cpu_core.c`.

CPU_TS64_to_uSec()

Description

Converts a 64-bit CPU timestamp from timer counts to microseconds.

Files

`cpu.h/cpu_core.c`

Prototype

```
CPU_INT64U  CPU_TS64_to_uSec(CPU_TS64  ts_cnts)
```

Arguments

`ts_cnts`

CPU timestamp (in timestamp timer counts).

Returned Value

Converted CPU timestamp (in microseconds).

Notes / Warnings

1. For more information, see `CPU_TS64_to_uSec()` function header comment block in cpu_core.c.

CPU_TS_Get32()

Description

Gets the current 32-bit CPU timestamp.

Files

`cpu.h/cpu_core.c`

Prototype

```
CPU_TS32  CPU_TS_Get32 (void)
```

Arguments

None.

Returned Value

Current 32-bit CPU timestamp (in timestamp timer counts).

Notes / Warnings

1. For more information, see `CPU_TS_Get32()` function header comment block in `cpu_core.c` .

CPU_TS_Get64()

Description

Gets the current 64-bit CPU timestamp.

Files

`cpu.h/cpu_core.c`

Prototype

```
CPU_TS64  CPU_TS_Get64 (void)
```

Arguments

None.

Returned Value

Current 64-bit CPU timestamp (in timestamp timer counts).

Notes / Warnings

1. For more information, see `CPU_TS_Get64()` function header comment block in `cpu_core.c` .

CPU_TS_TmrFreqGet()

Description

Gets the CPU timestamp's timer frequency.

Files

`cpu.h/cpu_core.c`

Prototype

```
CPU_TS_TMR_FREQ  CPU_TS_TmrFreqGet (RTOS_ERR  *p_err)
```

Arguments

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_CFG`

Returned Value

CPU timestamp's timer frequency (in Hertz), if NO error(s). 0, otherwise.

Notes / Warnings

None.

CPU_TS_TmrFreqSet()

Description

Sets the CPU timestamp's timer frequency.

Files

`cpu.h/cpu_core.c`

Prototype

```
void  CPU_TS_TmrFreqSet (CPU_TS_TMR_FREQ  freq_hz)
```

Arguments

`freq_hz`

Frequency to set for CPU timestamp's timer (in Hertz).

Returned Value

None.

Notes / Warnings

1. For more information, see `CPU_TS_TmrFreqSet()` function header comment block in `cpu_core.c` .

CPU_TS_Update()

Description

Updates the current CPU timestamp(s).

Files

`cpu.h/cpu_core.c`

Prototype

```
void  CPU_TS_Update (void)
```

Arguments

None.

Returned Value

None.

Notes / Warnings

1. For more information, see `CPU_TS_Update()` function header comment block in `cpu_core.c`.

CPU Cache Management API

CPU Cache Management API

- CPU_DCACHE_RANGE_FLUSH()
- CPU_DCACHE_RANGE_INV()
- CPU_MB()
- CPU_RMB()
- CPU_WMB()

CPU_DCACHE_RANGE_FLUSH()

Description

Flush (clean) a range of data cache.

Files

`cpu_cache.h/cpu_cache_*.c`

Prototype

```
CPU_DCACHE_RANGE_FLUSH (void      *addr_start,
                        CPU_ADDR   len)
```

Arguments

`addr_start`

Start address of the region to flush.

`len`

Size of the region to flush in bytes.

Returned Value

None.

Notes / Warnings

None.

CPU_DCACHE_RANGE_INV()

Description

Invalidate a range of data cache.

Files

`cpu_cache.h/cpu_cache_*.c`

Prototype

```
CPU_DCACHE_RANGE_INV (void      *addr_start,
                      CPU_ADDR   len)
```

Arguments

`addr_start`

Start address of the region to invalidate.

`len`

Size of the region to invalidate in bytes.

Returned Value

None.

Notes / Warnings

None.

CPU_MB()

Description

Create a Full (read and write) memory barrier.

Files

`cpu_port.h`

Prototype

```
CPU_MB()
```

Arguments

None.

Returned Value

None.

Notes / Warnings

None.

CPU_RMB()

Description

Create a Read (load) memory barrier.

Files

`cpu_port.h`

Prototype

```
CPU_RMB()
```

Arguments

None.

Returned Value

None.

Notes / Warnings

None.

CPU_WMB()

Description

Create a Write (store) memory barrier.

Files

`cpu_port.h`

Prototype

```
CPU_WMB()
```

Arguments

None.

Returned Value

None.

Notes / Warnings

None.

Kernel API

Kernel API

- Kernel Core API
- Kernel Event Flag API
- Kernel Message Queue API
- Kernel Monitor API
- Kernel Mutex API
- Kernel Port Hooks API
- Kernel Semaphore API
- Kernel Statistic API
- Kernel Task Management API
- Kernel Time Management API
- Kernel Timer API

Kernel Core API

Kernel Core API

- OSInit()
- OSIntEnter()
- OSIntExit()
- OSSchedRoundRobinCfg()
- OSSchedRoundRobinYield()
- OSSched()
- OSSchedLock()
- OSSchedUnlock()
- OSStart()
- OSVersion()
- OS_ConfigureISRStk()
- OS_ConfigureMemSeg()
- OS_ConfigureMsgPoolSize()
- OS_ConfigureStkLimit()
- OS_ConfigureStatTask()
- OS_ConfigureTmrTask()

OSInit()

Description

Initializes the internals of the Kernel and MUST be called before creating any Kernel object and before calling `OSStart()` .

Files

`os.h/os_core.c`

Prototype

```
void  OSInit (RTOS_ERR  *p_err)
```

Arguments

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`
- `RTOS_ERR_SEG_OVF`

Returned Value

None.

Notes / Warnings

1. This function MUST be called AFTER Common's `Mem_Init()` .

OSIntEnter()

Description

Used in an interrupt service routine (ISR) to notify the Kernel that you are about to service an interrupt. This allows the Kernel to keep track of interrupt nesting and only performs rescheduling at the last nested ISR.

Files

`os.h/os_core.c`

Prototype

```
void OSIntEnter (void)
```

Arguments

None.

Returned Value

None.

Notes / Warnings

1. This function MUST be called with interrupts already disabled.
2. Your ISR can directly increment ' `OSIntNestingCtr` ' without calling this function because `OSIntNestingCtr` has been declared 'global'. The port is actually considered part of the OS and is allowed to access the Kernel's variables.
3. You MUST still call `OSIntExit()` even though you can increment ' `OSIntNestingCtr` ' directly.
4. You MUST invoke `OSIntEnter()` and `OSIntExit()` in pairs. In other words, for every call to `OSIntEnter()` (or direct increment to `OSIntNestingCtr`) at the beginning of the ISR you MUST have a call to `OSIntExit()` at the end of the ISR.
5. You are allowed to nest interrupts up to 250 levels deep.

OSIntExit()

Description

Notifies the Kernel that you have completed servicing an ISR. When the last nested ISR has completed, the Kernel will call the scheduler to determine whether a new, high-priority task is ready to run.

Files

`os.h/os_core.c`

Prototype

```
void OSIntExit (void)
```

Arguments

None.

Returned Value

None.

Notes / Warnings

1. You MUST invoke `OSIntEnter()` and `OSIntExit()` in pairs. In other words, for every call to `OSIntEnter()` (or direct increment to `OSIntNestingCtr`) at the beginning of the ISR, you MUST have a call to `OSIntExit()` at the end of the ISR.
2. Rescheduling is prevented when the scheduler is locked (see `OSSchedLock()`).

OSSchedRoundRobinCfg()

Description

Changes the round-robin scheduling parameters.

Files

`os.h/os_core.c`

Prototype

```
void  OSSchedRoundRobinCfg (CPU_BOOLEAN   en,
                            OS_TICK       dflt_time_quanta,
                            RTOS_ERR     *p_err)
```

Arguments

`en`

Determines if the round-robin will be used:

- `DEF_ENABLED` Round-robin scheduling is enabled.
- `DEF_DISABLED` Round-robin scheduling is disabled.

`dflt_time_quanta`

Default number of ticks between time slices. A value of 0 assumes `OSCfg_TickRate_Hz` / 10.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

None.

OSSchedRoundRobinYield()

Description

Gives up the CPU when a task is finished its execution before its time slice expires.

Files

`os.h/os_core.c`

Prototype

```
void  OSSchedRoundRobinYield (RTOS_ERR  *p_err)
```

Arguments

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_NONE_WAITING`
- `RTOS_ERR_OS_SCHED_LOCKED`

Returned Value

None.

Notes / Warnings

1. This function MUST be called from a task.

OSSched()

Description

This function is called by other kernel services to determine whether a new, high priority task has been made ready to run. This function is invoked by TASK level code and is not used to reschedule tasks from ISRs (see `OSIntExit()` for ISR rescheduling).

Files

`os.h/os_core.c`

Prototype

```
void OSSched (void)
```

Arguments

None.

Returned Value

None.

Notes / Warnings

1. Rescheduling is prevented when the scheduler is locked (see `OSSchedLock()`).

OSSchedLock()

Description

Prevents rescheduling from taking place, allowing your application to prevent context switches until you are ready to permit context switching.

Files

`os.h/os_core.c`

Prototype

```
void OSSchedLock (RTOS_ERR *p_err)
```

Arguments

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_WOULD_OVF`

Returned Value

None.

Notes / Warnings

1. You MUST invoke `OSSchedLock()` and `OSSchedUnlock()` in pairs. In other words, for every call to `OSSchedLock()`, you MUST have a call to `OSSchedUnlock()`.

OSSchedUnlock()

Description

Re-allows rescheduling.

Files

`os.h/os_core.c`

Prototype

```
void OSSchedUnlock (RTOS_ERR *p_err)
```

Arguments

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_OS_SCHED_LOCKED`

Returned Value

None.

Notes / Warnings

1. You MUST invoke `OSSchedLock()` and `OSSchedUnlock()` in pairs. In other words, for every call to `OSSchedLock()`, you MUST have a call to `OSSchedUnlock()`.

OSStart()

Description

Starts the multitasking process which lets the Kernel manage the tasks that you created. Before you can call `OSStart()`, you MUST have called `OSInit()` and you MUST have created at least one application task.

Files

`os.h/os_core.c`

Prototype

```
void OSStart (RTOS_ERR *p_err)
```

Arguments

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

1. `OSStartHighRdy()` MUST:
   a) Call `OSTaskSwHook()` .
   b) Load the context of the task pointed to by `OSTCBHighRdyPtr` .
   c) Execute the task.
2. `OSStart()` is not supposed to return. If it does, that would be considered a fatal error.

OSVersion()

Description

Returns the version number of the Kernel. The returned value is the Kernel's version number multiplied by 10000. In other words, version 3.01.02 would be returned as 30102.

Files

`os.h/os_core.c`

Prototype

```
CPU_INT16U  OSVersion (RTOS_ERR  *p_err)
```

Arguments

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

The version number of the Kernel multiplied by 10000.

Notes / Warnings

1. This function is `DEPRECATED` and will be removed in a future version of this product. Instead, use the `RTOS_VERSION` define. which can be found in `rtos/common/include/version.h` .

OS_ConfigureISRStk()

Description

Configure the stack used for ISRs, if available.

Files

`os.h/os_core.c`

Prototype

```
void  OS_ConfigureISRStk (CPU_STK       *p_stk_base_ptr,
                          CPU_STK_SIZE   stk_size)
```

Arguments

`p_stk_base_ptr`

Pointer to the base of the buffer used as the stack.

`stk_size`

Size of the stack, in CPU_STK elements.

Returned Value

None.

Notes / Warnings

1. This function is optional. If it is called, it must be called before `OSInit()` . If it is not called, default values will be used.

OS_ConfigureMemSeg()

Description

Configure the memory segment used by the kernel.

Files

`os.h/os_core.c`

Prototype

```
void OS_ConfigureMemSeg (MEM_SEG *p_mem_seg)
```

Arguments

`p_mem_seg`

Pointer to the memory segment in which the kernel data will be allocated.

Returned Value

None.

Notes / Warnings

1. This function is optional. If it is called, it must be called before `OSInit()` . If it is not called, default values will be used.

OS_ConfigureMsgPoolSize()

Description

Configure the kernel message pool size.

Files

`os.h/os_core.c`

Prototype

```
void OS_ConfigureMsgPoolSize (OS_MSG_SIZE msg_pool_size)
```

Arguments

`msg_pool_size`

Number of messages the kernel will manage. Shared between task message queues and regular message queues.

Returned Value

None.

Notes / Warnings

1. This function is optional. If it is called, it must be called before `OSInit()` . If it is not called, default values will be used.

OS_ConfigureStkLimit()

Description

Configure the application stack limit.

Files

`os.h/os_core.c`

Prototype

```
void OS_ConfigureStkLimit (CPU_STK_SIZE task_stk_limit)
```

Arguments

`task_stk_limit`

Stack limit in percentage to empty.

Returned Value

None.

Notes / Warnings

1. This function is optional. If it is called, it must be called before `OSInit()`. If it is not called, default values will be used.

OS_ConfigureStatTask()

Description

If enabled, configure the Statistics Task.

Files

`os.h/os_core.c`

Prototype

```
void OS_ConfigureStatTask (OS_TASK_CFG *p_stat_task_cfg)
```

Arguments

`p_stat_task_cfg`

Pointer to the Statistics Task configuration.

Returned Value

None.

Notes / Warnings

1. This function is optional. If it is called, it must be called before `OSInit()`. If it is not called, default values will be used.

OS_ConfigureTmrTask()

Description

If enabled, configure the Timer Management Task.

Files

`os.h/os_core.c`

Prototype

```
void OS_ConfigureTmrTask (OS_TASK_CFG *p_tmr_task_cfg)
```

Arguments

`p_tmr_task_cfg`

Pointer to the Timer Management Task configuration.

Returned Value

None.

Notes / Warnings

1. This function is optional. If it is called, it must be called before `OSInit()` . If it is not called, default values will be used.

Kernel Event Flag API

Kernel Event Flag API

- OSFlagCreate()
- OSFlagDel()
- OSFlagPend()
- OSFlagPendAbort()
- OSFlagPendGetFlagsRdy()
- OSFlagPost()

OSFlagCreate()

Description

This function is called to create an event flag group.

Files

`os.h/os_flag.c`

Prototype

```
void  OSFlagCreate (OS_FLAG_GRP  *p_grp,
                    CPU_CHAR     *p_name,
                    OS_FLAGS      flags,
                    RTOS_ERR     *p_err)
```

Arguments

`p_grp`

Pointer to the event flag group to create. Your application is responsible for allocating storage for the flag group.

`p_name`

The name of the event flag group.

`flags`

Contains the initial value to store in the event flag group (typically 0).

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`

Returned Value

None.

Notes / Warnings

None.

OSFlagDel()

Description

This function deletes an event flag group and readies all tasks pending on the event flag group.

Files

`os.h/os_flag.c`

Prototype

```
OS_OBJ_QTY  OSFlagDel (OS_FLAG_GRP  *p_grp,
                       OS_OPT        opt,
                       RTOS_ERR     *p_err)
```

Arguments

`p_grp`

Pointer to the event flag group.

`opt`

Determines delete options as follows:

- `OS_OPT_DEL_NO_PEND` Deletes the event flag group ONLY if no task is pending.
- `OS_OPT_DEL_ALWAYS` Deletes the event flag group even if tasks are waiting. In this case, all the pending tasks will be made ready.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`
- `RTOS_ERR_OS_TASK_WAITING`

Returned Value

- == 0 If no tasks were waiting on the event flag group, or upon error.
- > 0 If one or more tasks waiting on the event flag group are now ready and informed.

Notes / Warnings

1. Use this function with care. Tasks that would normally expect the presence of the event flag group MUST check the return code of `OSFlagPost()` and `OSFlagPend()` .

OSFlagPend()

Description

This function is called to wait for a combination of bits to be set in an event flag group. Your application can wait for either ANY bit to be set or ALL bits to be set.

Files

`os.h/os_flag.c`

Prototype

```
OS_FLAGS  OSFlagPend (OS_FLAG_GRP  *p_grp,
                      OS_FLAGS      flags,
                      OS_TICK       timeout,
                      OS_OPT        opt,
                      CPU_TS       *p_ts,
                      RTOS_ERR     *p_err)
```

Arguments

`p_grp`

Pointer to the event flag group.

`flags`

Bit pattern that indicates which bit(s) (i.e., flags) to wait for. The bits you want are specified by setting the corresponding bits in 'flags' (e.g., if your application waits for bits 0 and 1, then the 'flags' would contain 0x03).

`timeout`

Optional timeout (in clock ticks) that your task will wait for the desired bit combination. If you specify 0, the task will wait forever at the specified event flag group, or until a message arrives.

`opt`

Specifies whether you want ALL bits to be set or ANY of the bits to be set. You can specify the 'ONE' of the following arguments:

- `OS_OPT_PEND_FLAG_CLR_ALL` Wait for ALL bits in 'flags' to be clear. (0)
- `OS_OPT_PEND_FLAG_CLR_ANY` Wait for ANY bit in 'flags' to be clear. (0)
- `OS_OPT_PEND_FLAG_SET_ALL` Wait for ALL bits in 'flags' to be set. 1.
- `OS_OPT_PEND_FLAG_SET_ANY` Wait for ANY bit in 'flags' to be set. 1.

You can 'ADD' `OS_OPT_PEND_FLAG_CONSUME` if you want the event flag to be 'consumed' by the call. For example, to wait for any flag in a group AND clear the flags that are present, set 'wait_opt' to:

`OS_OPT_PEND_FLAG_SET_ANY + OS_OPT_PEND_FLAG_CONSUME`

You can also 'ADD' the type of pend with 'ONE' of the two options:

- `OS_OPT_PEND_BLOCKING` Task will block if flags are not available.
- `OS_OPT_PEND_NON_BLOCKING` Task will NOT block if flags are not available.

`p_ts`

Pointer to a variable that receives the `timestamp` of when the event flag group was posted, aborted, or deleted. If you pass a NULL pointer (i.e. (CPU_TS *)0) then you will not get the `timestamp`. In other words, passing a NULL pointer is valid and indicates that you don't need the `timestamp`.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

The flags in the event flag group that made the task ready or, 0 if a timeout or an error occurred.

Notes / Warnings

None.

OSFlagPendAbort()

Description

This function aborts and prepares any tasks currently waiting on an event flag group. Rather than posting to the event flag group with `OSFlagPost()` , you should use this function to fault-abort the wait on the event flag group.

Files

`os.h/os_flag.c`

Prototype

```
OS_OBJ_QTY  OSFlagPendAbort (OS_FLAG_GRP  *p_grp,
                             OS_OPT        opt,
                             RTOS_ERR     *p_err)
```

Arguments

`p_grp`

Pointer to the event flag group.

`opt`

Determines the type of ABORT performed:

- `OS_OPT_PEND_ABORT_1` ABORT wait for a single task (HPT) waiting on the event flag.
- `OS_OPT_PEND_ABORT_ALL` ABORT wait for ALL tasks that are waiting on the event flag.
- `OS_OPT_POST_NO_SCHED` Do not call the scheduler.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NONE_WAITING`

Returned Value

- == 0 If no tasks were waiting on the event flag group, or upon error.
- > 0 If one or more tasks waiting on the event flag group are now ready and informed.

Notes / Warnings

None.

OSFlagPendGetFlagsRdy()

Description

This function is called to obtain the flags that caused the task to be ready. In other words, this function allows you to reveal "Who done it!"

Files

`os.h/os_flag.c`

Prototype

```
OS_FLAGS  OSFlagPendGetFlagsRdy (RTOS_ERR  *p_err)
```

Arguments

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

The flags that caused the task to be ready.

Notes / Warnings

None.

OSFlagPost()

Description

This function is called to set or clear some bits in an event flag group. The bits to set or clear are specified by a 'bit mask'.

Files

`os.h/os_flag.c`

Prototype

```
OS_FLAGS  OSFlagPost (OS_FLAG_GRP  *p_grp,
                      OS_FLAGS      flags,
                      OS_OPT        opt,
                      RTOS_ERR     *p_err)
```

Arguments

`p_grp`

Pointer to the event flag group.

`flags`

If 'opt' (see below) is `OS_OPT_POST_FLAG_SET`, each bit set in 'flags' will be SET to the corresponding bit in the event flag group. For example, to set bits 0, 4, and 5, you would set 'flags' to:

`0x31` (note, bit 0 is least significant bit)

- If ' `opt` ' (see below) is `OS_OPT_POST_FLAG_CLR`, each bit set in 'flags' will CLEAR the corresponding bit in the event flag group. For example, to clear bits 0, 4, and 5, you would specify 'flags' as:
  `0x31` (note, bit 0 is least significant bit)

`opt`

Indicates whether the flags will be Set or Cleared :

- `OS_OPT_POST_FLAG_SET` Set.
- `OS_OPT_POST_FLAG_CLR` Cleared.

You can also 'ADD' `OS_OPT_POST_NO_SCHED` to prevent the scheduler from being called.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

The new value of the event flags bits that are still set.

Notes / Warnings

1. The execution time of this function depends on the number of tasks waiting on the event flag group.

Kernel Message Queue API

Kernel Message Queue API

- OSQCreate()
- OSQDel()
- OSQFlush()
- OSQPend()
- OSQPendAbort()
- OSQPost()

OSQCreate()

Description

Called by your application to create a message queue. Message queues MUST be created before they can be used.

Files

`os.h/os_q.c`

Prototype

```
void  OSQCreate (OS_Q         *p_q,
                 CPU_CHAR     *p_name,
                 OS_MSG_QTY   max_qty,
                 RTOS_ERR     *p_err)
```

Arguments

`p_q`

Pointer to the message queue.

`p_name`

Pointer to an ASCII string used to name the message queue.

`max_qty`

Indicates the maximum size of the message queue (must be non-zero). Note that it is not possible to have a size higher than the maximum number of `OS_MSGs` available.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`

Returned Value

None.

Notes / Warnings

None.

OSQDel()

Description

This function deletes a message queue and readies all tasks pending on the queue.

Files

`os.h/os_q.c`

Prototype

```
OS_OBJ_QTY  OSQDel (OS_Q      *p_q,
            OS_OPT     opt,
            RTOS_ERR  *p_err)
```

Arguments

`p_q`

Pointer to the message queue to delete.

`opt`

Determines delete options as follows:

- `OS_OPT_DEL_NO_PEND` Deletes the queue ONLY if no task is pending.
- `OS_OPT_DEL_ALWAYS` Deletes the queue even if tasks are waiting. In this case, all pending tasks will be readied.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`
- `RTOS_ERR_OS_TASK_WAITING`

Returned Value

- == 0 If no tasks were waiting on the queue, or upon error.
- > 0 If one or more tasks waiting on the queue are now readied and informed.

Notes / Warnings

(1) Use this function with care. Tasks that would normally expect the presence of the queue MUST check the return code of `OSQPend()`.

(2) Because ALL tasks pending on the queue will be readied, you MUST be careful handling resources in applications where the queue is used for mutual exclusion because these resource will no longer be guarded by the queue.

OSQFlush()

Description

Flushes the contents of the message queue.

Files

`os.h/os_q.c`

Prototype

```
OS_MSG_QTY  OSQFlush (OS_Q      *p_q,
             RTOS_ERR  *p_err)
```

Arguments

`p_q`

Pointer to the message queue to flush.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

- == 0 If no entries were freed, or upon error.
- > 0 The number of freed entries.

Notes / Warnings

(1) Use great care with this function because when you flush the queue, you LOSE the references to what the queue entries are pointing, potentially causing 'memory leaks'. In other words, the data to which you are pointing that are being referenced by the queue entries should, most likely, be de-allocated (i.e., freed).

OSQPend()

Description

Waits for a message to be sent to a queue.

Files

`os.h/os_q.c`

Prototype

```
void  *OSQPend (OS_Q         *p_q,
                OS_TICK       timeout,
                OS_OPT        opt,
                OS_MSG_SIZE  *p_msg_size,
                CPU_TS       *p_ts,
                RTOS_ERR     *p_err)
```

Arguments

`p_q`

Pointer to the message queue.

`timeout`

Optional timeout period (in clock ticks). If non-zero, your task waits for a message to arrive at the queue up to the amount of time specified by this argument. However, if you specify 0, your task will wait forever at the specified queue or until a message arrives.

`opt`

Determines whether the user wants to block if the queue is empty or not:

- `OS_OPT_PEND_BLOCKING` Task will block.
- `OS_OPT_PEND_NON_BLOCKING` Task will NOT block.

`p_msg_size`

Pointer to a variable that receives the size of the message.

`p_ts`

Pointer to a variable that receives the timestamp of when the message was received, pend aborted, or the message queue was deleted, If you pass a NULL pointer (i.e., (CPU_TS *)0), you will not get the timestamp. In other words, passing a NULL pointer is valid and indicates that you do not need the timestamp.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

- != `(void *)0` Pointer to the message received.
- == `(void *)0` If you received a NULL pointer message, or if no message was received, or if '`p_q`' is a NULL pointer, or if you didn't pass a pointer to a queue.

Notes / Warnings

None.

OSQPendAbort()

Description

Aborts and readies any tasks currently waiting on a queue. Use this function to fault-abort the wait on the queue, rather than the normal signaling of the queue via `OSQPost()`.

Files

`os.h/os_q.c`

Prototype

```
OS_OBJ_QTY  OSQPendAbort (OS_Q      *p_q,
                          OS_OPT     opt,
                          RTOS_ERR  *p_err)
```

Arguments

`p_q`

Pointer to the message queue.

`opt`

Determines the type of ABORT performed:

- `OS_OPT_PEND_ABORT_1` ABORT wait for a single task (HPT) waiting on the message queue.
- `OS_OPT_PEND_ABORT_ALL` ABORT wait for ALL tasks that are waiting on the message queue.
- `OS_OPT_POST_NO_SCHED` Do not call the scheduler.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_NONE_WAITING

Returned Value

- == 0 If no tasks were waiting on the queue, or upon error.
- > 0 If one or more tasks waiting on the queue are now readied and informed.

Notes / Warnings

None.

OSQPost()

Description

Sends a message to a queue. With the 'opt' argument, you can specify if the message is broadcast to all waiting tasks and/or if you post the message to the front of the queue (LIFO) or normally (FIFO) at the end of the queue.

Files

`os.h/os_q.c`

Prototype

```
void  OSQPost (OS_Q         *p_q,
               void         *p_void,
               OS_MSG_SIZE   msg_size,
               OS_OPT        opt,
               RTOS_ERR     *p_err)
```

Arguments

`p_q`

Pointer to a message queue.

`p_void`

Pointer to the message to send.

`msg_size`

Specifies the size of the message (in bytes).

`opt`

Determines the type of POST performed:

- `OS_OPT_POST_ALL` POST to ALL tasks that are waiting on the queue. This option can be added to either `OS_OPT_POST_FIFO` or `OS_OPT_POST_LIFO` .
- `OS_OPT_POST_FIFO` POST message to end of queue (FIFO) and wake up a single waiting task.
- `OS_OPT_POST_LIFO` POST message to the front of the queue (LIFO) and wake up a single waiting task.
- `OS_OPT_POST_NO_SCHED` Do not call the scheduler.
- `OS_OPT_POST_NO_SCHED` can be added (OR'd) with other options.
- `OS_OPT_POST_ALL` can be added (OR'd) with other options.
  The possible combinations of options are:
- `OS_OPT_POST_FIFO`
- `OS_OPT_POST_LIFO`
- `OS_OPT_POST_FIFO + OS_OPT_POST_ALL`
- `OS_OPT_POST_LIFO + OS_OPT_POST_ALL`
- `OS_OPT_POST_FIFO + OS_OPT_POST_NO_SCHED`

- `OS_OPT_POST_LIFO + OS_OPT_POST_NO_SCHED`
- `OS_OPT_POST_FIFO + OS_OPT_POST_ALL + OS_OPT_POST_NO_SCHED`
- `OS_OPT_POST_LIFO + OS_OPT_POST_ALL + OS_OPT_POST_NO_SCHED`

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_NO_MORE_RSRC`

Returned Value

None.

Notes / Warnings

None.

Kernel Monitor API

Kernel Monitor API

- OSMonCreate()
- OSMonDel()
- OSMonOp()

OSMonCreate()

Description

Creates a monitor.

Files

os.h/os_mon.c

Prototype

```
void  OSMonCreate (OS_MON     *p_mon,
                   CPU_CHAR   *p_name,
                   void       *p_mon_data,
                   RTOS_ERR   *p_err)
```

Arguments

p_mon

Pointer to the monitor to initialize. Your application is responsible for allocating storage space for the monitor.

p_name

Pointer to the name to assign to this monitor.

p_mon_data

Pointer to the monitor's global data.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_OS_ILLEGAL_RUN_TIME

Returned Value

None.

Notes / Warnings

None.

OSMonDel()

Description

This function deletes a monitor.

Files

`os.h/os_mon.c`

Prototype

```
OS_OBJ_QTY  OSMonDel (OS_MON    *p_mon,
                      OS_OPT     opt,
                      RTOS_ERR  *p_err)
```

Arguments

`p_mon`

Pointer to the monitor to delete.

`opt`

Determines delete options as follows:

- `OS_OPT_DEL_NO_PEND` Deletes the monitor ONLY if there are no tasks pending.
- `OS_OPT_DEL_ALWAYS` Deletes the monitor even if there are tasks waiting. In this case, all pending tasks will be ready.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`
- `RTOS_ERR_OS_TASK_WAITING`

Returned Value

- == 0 If there were no tasks waiting on the monitor, or upon error.
- > 0 If one or more tasks are waiting on the monitor are now ready and informed.

Notes / Warnings

(1) Use this function with care. Tasks that would normally expect the presence of the monitor MUST check the return code of `OSMonOp()` .

(2) Because ALL tasks pending on the monitor will be ready, be careful in applications where the monitor is used for mutual exclusion because the resource(s) will no longer be guarded by the monitor.

OSMonOp()

Description

Performs an operation on a monitor.

Files

`os.h/os_mon.c`

Prototype

```
void  OSMonOp (OS_MON                *p_mon,
               OS_TICK                timeout,
               void                  *p_arg,
               OS_MON_ON_ENTER_PTR    p_on_enter,
               OS_MON_ON_EVAL_PTR     p_on_eval,
               OS_OPT                 opt,
               RTOS_ERR              *p_err)
```

Arguments

`p_mon`

Pointer to the monitor.

`timeout`

Optional timeout to be applied if the monitor blocks (pending).

`p_arg`

Argument of the monitor.

`p_on_enter`

Callback called at the entry of `OSMonOp()` .

`p_on_eval`

Callback to be registered as the monitor's evaluation function.

`opt`

Possible option :

OS_OPT_POST_NO_SCHED Do not call the scheduler.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

None.

Notes / Warnings

None.

Kernel Mutex API

Kernel Mutex API

- OSMutexCreate()
- OSMutexDel()
- OSMutexPend()
- OSMutexPendAbort()
- OSMutexPost()

OSMutexCreate()

Description

Creates a mutex so that multiple program threads can take turns sharing the same resource.

Files

`os.h/os_mutex.c`

Prototype

```
void  OSMutexCreate (OS_MUTEX  *p_mutex,
                     CPU_CHAR  *p_name,
                     RTOS_ERR  *p_err)
```

Arguments

`p_mutex`

Pointer to the mutex to initialize. Your application is responsible for allocating storage for the mutex.

`p_name`

Pointer to the name you would like to give the mutex.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`

Returned Value

None.

Notes / Warnings

None.

OSMutexDel()

Description

Deletes a mutex and readies all tasks pending on the mutex.

Files

`os.h/os_mutex.c`

Prototype

```
OS_OBJ_QTY  OSMutexDel (OS_MUTEX  *p_mutex,
                        OS_OPT     opt,
                        RTOS_ERR  *p_err)
```

Arguments

`p_mutex`

Pointer to the mutex to delete.

`opt`

Determines delete options as follows:

`OS_OPT_DEL_NO_PEND` Deletes the mutex ONLY if no tasks are pending. `OS_OPT_DEL_ALWAYS` Deletes the mutex even if tasks are waiting. In this case, all pending tasks will be readied.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`
- `RTOS_ERR_OS_TASK_WAITING`

Returned Value

- == 0 If there are no tasks waiting on the mutex, or upon error.
- > 0 If there are one or more tasks waiting on the mutex that are now readied and informed.

Notes / Warnings

1. Use this function with care. Tasks that would normally expect the presence of the mutex MUST check the return code of `OSMutexPend()` .
2. Because ALL tasks pending on the mutex will be readied, be careful in applications where the mutex is used for mutual exclusion because the resource(s) will no longer be guarded by the mutex.

OSMutexPend()

Description

This function waits for a mutex.

Files

`os.h/os_mutex.c`

Prototype

```
void  OSMutexPend (OS_MUTEX  *p_mutex,
                   OS_TICK    timeout,
                   OS_OPT     opt,
                   CPU_TS    *p_ts,
                   RTOS_ERR  *p_err)
```

Arguments

`p_mutex`

Pointer to the mutex.

`timeout`

Optional timeout period (in clock ticks). If non-zero, the task will wait for the resource up to the amount of time (in 'ticks') specified by this argument. If you specify 0, the task will wait forever at the specified mutex, or until the resource becomes available.

`opt`

Determines whether the feature to block if the mutex is available or not:

- `OS_OPT_PEND_BLOCKING` Task will block.
- `OS_OPT_PEND_NON_BLOCKING` Task will NOT block.

`p_ts`

Pointer to a variable that will receive the timestamp of when the mutex was posted or pend aborted or the mutex deleted. If you pass a NULL pointer (i.e., (CPU_TS *)0), you will not get the timestamp. In other words, passing a NULL pointer is valid and indicates that you don't need the timestamp.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_IS_OWNER` (see note (1))
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

None.

Notes / Warnings

1. A mutex can be nested, so `RTOS_ERR_IS_OWNER` error can be used as an indicator that you are nesting the mutex. If the correct number of Post is done the mutex will be released.

OSMutexPendAbort()

Description

Aborts and readies any tasks currently waiting on a mutex. Rather than signal the mutex via `OSMutexPost()`, use this function to fault-abort the wait on the mutex.

Files

`os.h/os_mutex.c`

Prototype

```
OS_OBJ_QTY  OSMutexPendAbort (OS_MUTEX  *p_mutex,
                              OS_OPT     opt,
                              RTOS_ERR  *p_err)
```

Arguments

`p_mutex`

Pointer to the mutex.

`opt`

Determines the type of ABORT performed:

- `OS_OPT_PEND_ABORT_1` ABORT wait for a single task (HPT) waiting on the mutex.
- `OS_OPT_PEND_ABORT_ALL` ABORT wait for ALL tasks that are waiting on the mutex.
- `OS_OPT_POST_NO_SCHED` Do not call the scheduler.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NONE_WAITING`

Returned Value

- == 0 If there were no tasks waiting on the mutex, or upon error.
- > 0 If there were one or more tasks waiting on the mutex are now ready and informed.

Notes / Warnings

None.

OSMutexPost()

Description

Signals a mutex.

Files

`os.h/os_mutex.c`

Prototype

```
void  OSMutexPost (OS_MUTEX  *p_mutex,
                   OS_OPT     opt,
                   RTOS_ERR  *p_err)
```

Arguments

`p_mutex`

Pointer to the mutex.

`opt`

Option that alters the behavior of the post. The choices are:

- `OS_OPT_POST_NONE` No special option selected.
- `OS_OPT_POST_NO_SCHED` If you don't want the scheduler to be called after the post.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_OWNERSHIP`
- `RTOS_ERR_IS_OWNER`

Returned Value

None.

Notes / Warnings

None.

Kernel Port Hooks API

Kernel Port Hooks API

- OSInitHook()
- OSRedzoneHitHook()
- OSStatTaskHook()
- OSTaskCreateHook()
- OSTaskDelHook()
- OSTaskReturnHook()
- OSTaskStkInit()
- OSTaskSwHook()

OSInitHook()

Description

Called by `OSInit()` at the beginning of `OSInit()` .

Files

`os.h/os_cpu_c.c`

Prototype

```
void  OSInitHook (void)
```

Arguments

None.

Returned Value

None.

Notes / Warnings

None.

OSRedzoneHitHook()

Description

Called when a task's stack has overflowed.

Files

`os.h/os_cpu_c.c`

Prototype

```
void  OSRedzoneHitHook (OS_TCB  *p_tcb)
```

Arguments

`p_tcb`

- Pointer to the TCB of the offending task.

- NULL if ISR.

Returned Value

None.

Notes / Warnings

None.

OSStatTaskHook()

Description

Allows your application to add functionality to the statistics task. This function is called every second by the Kernel's statistics task.

Files

`os.h/os_cpu_c.c`

Prototype

```
void  OSStatTaskHook (void)
```

Arguments

None.

Returned Value

None.

Notes / Warnings

None.

OSTaskCreateHook()

Description

Called when a task is created.

Files

`os.h/os_cpu_c.c`

Prototype

```
void  OSTaskCreateHook (OS_TCB  *p_tcb)
```

Arguments

`p_tcb`

Pointer to the TCB of the task being created.

Returned Value

None.

Notes / Warnings

None.

OSTaskDelHook()

Description

Called when a task is deleted.

Files

`os.h/os_cpu_c.c`

Prototype

```
void OSTaskDelHook (OS_TCB *p_tcb)
```

Arguments

`p_tcb`

Pointer to the TCB of the task being deleted.

Returned Value

None.

Notes / Warnings

None.

OSTaskReturnHook()

Description

Called if a task accidentally returns. In other words, a task should either be an infinite loop or delete itself when done.

Files

`os.h/os_cpu_c.c`

Prototype

```
void OSTaskReturnHook (OS_TCB *p_tcb)
```

Arguments

`p_tcb`

Pointer to the TCB of the task that is returning.

Returned Value

None.

Notes / Warnings

None.

OSTaskStkInit()

Description

Initializes the stack frame of the task being created as if it had been already switched-out. This function is called by `OSTaskCreate()` and is highly processor specific.

Files

`os.h/os_cpu_c.c`

Prototype

```
CPU_STK  *OSTaskStkInit (OS_TASK_PTR    p_task,
                         void          *p_arg,
                         CPU_STK       *p_stk_base,
                         CPU_STK       *p_stk_limit,
                         CPU_STK_SIZE   stk_size,
                         OS_OPT         opt)
```

Arguments

`p_task`

Pointer to the task entry point address.

`p_arg`

Pointer to a user-supplied data area that will be passed to the task when the task first executes.

`p_stk_base`

Pointer to the base address of the stack.

`p_stk_limit`

Pointer to the element to set as the 'watermark' limit of the stack.

`stk_size`

Size of the stack (measured as number of `CPU_STK` elements).

`opt`

Options used to alter the behavior of `OSTaskStkInit()` . See OS.H for `OS_TASK_OPT_xxx` .

Returned Value

Always returns the location of the new top-of-stack once the processor registers have been placed on the stack in the proper order.

Notes / Warnings

None.

OSTaskSwHook()

Description

Allows you to perform other operations during a context switch. This function is called when a task switch is performed.

Files

`os.h/os_cpu_c.c`

Prototype

```
void  OSTaskSwHook (void)
```

Arguments

None.

Returned Value

None.

Notes / Warnings

1. Interrupts are disabled during this call.
2. It is assumed that the global pointer '`OSTCBHighRdyPtr`' points to the TCB of the task that will be 'switched in' (i.e., the highest priority task) and, '`OSTCBCurPtr`' points to the task being switched out (i.e., the preempted task).

Kernel Semaphore API

Kernel Semaphore API

- OSSemCreate()
- OSSemDel()
- OSSemPend()
- OSSemPendAbort()
- OSSemPost()
- OSSemSet()

OSSemCreate()

Description

Creates a semaphore.

Files

`os.h/os_sem.c`

Prototype

```
void  OSSemCreate (OS_SEM      *p_sem,
                   CPU_CHAR    *p_name,
                   OS_SEM_CTR   cnt,
                   RTOS_ERR    *p_err)
```

Arguments

`p_sem`

Pointer to the semaphore to initialize. Your application is responsible for allocating storage for the semaphore.

`p_name`

Pointer to the name to assign to the semaphore.

`cnt`

The initial value for the semaphore.

- If used to share resources, you should initialize to the number of resources available.
- If used to signal the occurrence of event(s), you should initialize to 0.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`

Returned Value

None.

Notes / Warnings

None.

OSSemDel()

Description

Deletes a semaphore.

Files

`os.h/os_sem.c`

Prototype

```
OS_OBJ_QTY  OSSemDel (OS_SEM    *p_sem,
                      OS_OPT     opt,
                      RTOS_ERR  *p_err)
```

Arguments

`p_sem`

Pointer to the semaphore to delete.

`opt`

Determines delete options as follows:

- `OS_OPT_DEL_NO_PEND` Deletes the semaphore ONLY if there are no pending tasks.
- `OS_OPT_DEL_ALWAYS` Deletes the semaphore even if tasks are waiting. In this case, all the pending tasks will be readied.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`
- `RTOS_ERR_OS_TASK_WAITING`

Returned Value

- == 0 If there were no tasks waiting on the semaphore, or upon error.
- > 0 If one or more tasks waiting on the semaphore that are now readied and informed.

Notes / Warnings

1. Use this function with care. Tasks that would normally expect the presence of the semaphore MUST check the return code of `OSSemPend()` .
2. Because ALL tasks pending on the semaphore will be readied, be careful with applications where the semaphore is used for mutual exclusion because the resource(s) will no longer be guarded by the semaphore.

OSSemPend()

Description

Waits for a semaphore.

Files

`os.h/os_sem.c`

Prototype

```
OS_SEM_CTR  OSSemPend (OS_SEM    *p_sem,
                       OS_TICK    timeout,
                       OS_OPT     opt,
                       CPU_TS    *p_ts,
                       RTOS_ERR  *p_err)
```

Arguments

`p_sem`

Pointer to the semaphore.

`timeout`

Optional timeout period (in clock ticks). If non-zero, your task will wait for the resource up to the amount of time (in 'ticks') specified by this argument. If you enter 0, your task will wait forever at the specified semaphore, or until the resource becomes available (or the event occurs).

`opt`

Determines whether the user wants to block if the semaphore is available or not:

- `OS_OPT_PEND_BLOCKING` Task will block.
- `OS_OPT_PEND_NON_BLOCKING` Task will NOT block.

`p_ts`

Pointer to a variable that receives the timestamp of when the semaphore was posted or pending aborted or the semaphore deleted. If you pass a NULL pointer (i.e. (CPU_TS*)0), you will not get the timestamp. In other words, passing a NULL pointer is valid and indicates that you don't need the timestamp.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

The current value of the semaphore counter, or 0 if not available.

Notes / Warnings

None.

OSSemPendAbort()

Description

Aborts and readies any tasks currently waiting on a semaphore. Rather than signal the semaphore via OSSemPost(), use this function to fault-abort the wait on the semaphore.

Files

`os.h/os_sem.c`

Prototype

```
OS_OBJ_QTY  OSSemPendAbort (OS_SEM    *p_sem,
                            OS_OPT     opt,
                            RTOS_ERR  *p_err)
```

Arguments

`p_sem`

Pointer to the semaphore.

`opt`

Determines the type of ABORT performed:

- `OS_OPT_PEND_ABORT_1` ABORT waits for a single task (HPT) waiting on the semaphore.
- `OS_OPT_PEND_ABORT_ALL` ABORT waits for ALL tasks that are waiting on the semaphore.
- `OS_OPT_POST_NO_SCHED` Do not call the scheduler.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NONE_WAITING`

Returned Value

- == 0 If no tasks were waiting on the semaphore, or upon error.
- > 0 If one or more tasks waiting on the semaphore are now readied and informed.

Notes / Warnings

None.

OSSemPost()

Description

Signals a semaphore.

Files

`os.h/os_sem.c`

Prototype

```
OS_SEM_CTR  OSSemPost (OS_SEM    *p_sem,
                       OS_OPT     opt,
                       RTOS_ERR  *p_err)
```

Arguments

`p_sem`

Pointer to the semaphore.

`opt`

Determines the type of POST performed:

- `OS_OPT_POST_1` POST and ready only the highest priority task waiting on semaphore (if tasks are waiting).
- `OS_OPT_POST_ALL` POST to ALL tasks that are waiting on the semaphore.
- `OS_OPT_POST_NO_SCHED` Do not call the scheduler.

Note(s):

1. `OS_OPT_POST_NO_SCHED` can be added with one of the other options.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_WOULD_OVF`

Returned Value

The current value of the semaphore counter or 0 upon error.

Notes / Warnings

None.

OSSemSet()

Description

Sets the semaphore count to the value specified as an argument. Typically this value would be 0, but you can set the semaphore to any value.

Files

`os.h/os_sem.c`

Prototype

```
void  OSSemSet (OS_SEM       *p_sem,
                OS_SEM_CTR   cnt,
                RTOS_ERR    *p_err)
```

Arguments

`p_sem`

Pointer to the semaphore.

`cnt`

The new value for the semaphore count. You would pass 0 to reset the semaphore count.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_OS_TASK_WAITING`

Returned Value

None.

Notes / Warnings

None.

Kernel Statistic API

Kernel Statistic API

- OSStatReset()
- OSStatTaskCPUUsageInit()

OSStatReset()

Description

Called by your application to reset the statistics.

Files

`os.h/os_stat.c`

Prototype

```
void  OSStatReset (RTOS_ERR  *p_err)
```

Arguments

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

None.

OSStatTaskCPUUsageInit()

Description

Establishes CPU usage by first determining how high a 32-bit counter would count to in 1/10 of a second if no other tasks were ready to execute during that time. CPU usage is determined by a low priority task which keeps track of this 32-bit counter every second, but this time, with other tasks running. CPU usage is determined by:

`OS_Stat_IdleCtrCPU Usage (%) = 100 * (1 - ------------------) OS_Stat_IdleCtrMax`

Files

`os.h/os_stat.c`

Prototype

```
void  OSStatTaskCPUUsageInit (RTOS_ERR  *p_err)
```

Arguments

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_NOT_READY`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_OS_SCHED_LOCKED`

Returned Value

None.

Notes / Warnings

None.

Kernel Task Management API

Kernel Task Management API

- OSTaskChangePrio()
- OSTaskCreate()
- OSTaskDel()
- OSTaskQFlush()
- OSTaskQPend()
- OSTaskQPendAbort()
- OSTaskQPost()
- OSTaskRegGet()
- OSTaskRegGetID()
- OSTaskRegSet()
- OSTaskResume()
- OSTaskSuspend()
- OSTaskSemPend()
- OSTaskSemPendAbort()
- OSTaskSemPost()
- OSTaskSemSet()
- OSTaskStkChk()
- OSTaskStkRedzoneChk()
- OSTaskTimeQuantaSet()

OSTaskChangePrio()

Description

Allows you to dynamically change the priority of a task. Note that the new priority MUST be available.

Files

`os.h/os_task.c`

Prototype

```
void  OSTaskChangePrio (OS_TCB    *p_tcb,
                        OS_PRIO    prio_new,
                        RTOS_ERR  *p_err)
```

Arguments

`p_tcb`

Pointer to the TCB of the task for which to change the priority.

`prio_new`

The new priority.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_ARG

Returned Value

None.

Notes / Warnings

None.

OSTaskCreate()

Description

Allows the the Kernel to manage the execution of a task. Tasks can either be created prior to the start of multitasking or by a running task. A task cannot be created by an ISR.

Files

os.h/os_task.c

Prototype

```
void  OSTaskCreate (OS_TCB         *p_tcb,
                    CPU_CHAR       *p_name,
                    OS_TASK_PTR     p_task,
                    void           *p_arg,
                    OS_PRIO         prio,
                    CPU_STK        *p_stk_base,
                    CPU_STK_SIZE    stk_limit,
                    CPU_STK_SIZE    stk_size,
                    OS_MSG_QTY      q_size,
                    OS_TICK         time_quanta,
                    void           *p_ext,
                    OS_OPT          opt,
                    RTOS_ERR       *p_err)
```

Arguments

p_tcb

Pointer to the task's TCB.

p_name

Pointer to an ASCII string that provides a name for the task.

p_task

Pointer to the task's code.

p_arg

Pointer to an optional data area which can pass parameters to the task when the task executes. For the task, it believes it was invoked and passed the argument ' p_arg ' as follows:

`void Task (void p_arg) { for (;;) { Task code; } }`

prio

The task's priority. A unique priority MUST be assigned to each task. The lower the number, the higher the priority.

p_stk_base

Pointer to the base address of the stack (i.e., low address).

`stk_limit`

The number of stack elements to set as 'watermark' limits for the stack. This value represents the number of `CPU_STK` entries left before the stack is full. For example, specifying 10% of the '`stk_size`' value indicates that the stack limit will be reached when the stack reaches 90% full.

`stk_size`

The size of the stack in number of elements. If `CPU_STK` is set to `CPU_INT08U`, the '`stk_size`' corresponds to the number of bytes available. If `CPU_STK` is set to `CPU_INT16U`, the '`stk_size`' contains the number of 16-bit entries available. Finally, if `CPU_STK` is set to `CPU_INT32U`, the '`stk_size`' contains the number of 32-bit entries available on the stack.

`q_size`

The maximum number of messages that can be sent to the task.

`time_quanta`

Amount of time (in ticks) for a time slice when the round-robin between tasks. Set to 0 to use the default.

`p_ext`

Pointer to a user-supplied memory location which is used as a TCB extension. For example, this user memory can hold the contents of floating-point registers during a context switch, the time each task takes to execute, the number of times the task has been switched-in, etc.

`opt`

Contains additional information (or options) about the behavior of the task. See OS_OPT_TASK_xxx in OS.H. Current choices are:

- `OS_OPT_TASK_NONE` No option selected.
- `OS_OPT_TASK_STK_CHK` Stack checking to be allowed for the task.
- `OS_OPT_TASK_STK_CLR` Clear the stack when the task is created.
- `OS_OPT_TASK_SAVE_FP` If the CPU has floating-point registers, save them during a context switch.
- `OS_OPT_TASK_NO_TLS` If the caller does not want or need TLS (Thread Local Storage) support for the task. If you do not include this option, TLS will be supported by default.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`

Returned Value

None.

Notes / Warnings

1. `OSTaskCreate()` triggers a critical assert when a stack overflow is detected during stack initialization. In this case, some memory may have been corrupted and should be treated as a fatal error.

OSTaskDel()

Description

Allows you to delete a task. The calling task can delete itself by specifying a NULL pointer for '`p_tcb`'. The deleted task is returned to the dormant state and can be re-activated by creating the deleted task again.

Files

`os.h/os_task.c`

Prototype

```
void  OSTaskDel (OS_TCB   *p_tcb,
           RTOS_ERR  *p_err)
```

Arguments

`p_tcb`

Pointer to the TCB of the task to delete.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_INVALID_STATE`

Returned Value

None.

Notes / Warnings

1. '`p_err`' is set to `RTOS_ERR_NONE` before `OSSched()` to allow the returned err or code to be monitored even for a task that is deleting itself. In this case, '`p_err`' MUST point to a global variable that can be accessed by another task.

OSTaskQFlush()

Description

Flushes the task's internal message queue.

Files

`os.h/os_task.c`

Prototype

```
OS_MSG_QTY  OSTaskQFlush (OS_TCB   *p_tcb,
                   RTOS_ERR  *p_err)
```

Arguments

`p_tcb`

Pointer to the task's TCB. Specifying a NULL pointer indicates that you wish to flush the message queue of the calling task.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

The number of entries freed from the queue.

Notes / Warnings

1. Use this function with great care. When you flush the queue, you lose the references to what the queue entries are pointing, which can cause 'memory leaks'. In other words, the data being pointed to that is referenced by the queue entries

should, most likely, need to be de-allocated (i.e., freed).

OSTaskQPend()

Description

This function causes the current task to wait for a message to be posted to it.

Files

`os.h/os_task.c`

Prototype

```
void  *OSTaskQPend (OS_TICK       timeout,
                    OS_OPT        opt,
                    OS_MSG_SIZE  *p_msg_size,
                    CPU_TS       *p_ts,
                    RTOS_ERR     *p_err)
```

Arguments

`timeout`

Optional timeout period (in clock ticks). If non-zero, your task will wait for a message to arrive up to the amount of time specified by this argument. If you specify 0, your task will wait forever, or until a message arrives.

`opt`

Determines if the user wants to block if the task's queue is empty or not:

- `OS_OPT_PEND_BLOCKING` Task will block.
- `OS_OPT_PEND_NON_BLOCKING` Task will NOT block.

`p_msg_size`

Pointer to a variable that will receive the size of the message.

`p_ts`

Pointer to a variable that will receive the timestamp of when the message was received. If you pass a NULL pointer (i.e., (CPU_TS *)0), you will not get the timestamp. In other words, passing a NULL pointer is valid and indicates that you don't need the timestamp.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

A pointer to the message received or a NULL pointer upon error.

Notes / Warnings

It is possible to receive a NULL pointer when there are no errors.

OSTaskQPendAbort()

Description

Aborts and readies the task specified. Use this function to fault-abort the wait for a message, rather than to normally post the message to the task via `OSTaskQPost()` .

Files

`os.h/os_task.c`

Prototype

```
CPU_BOOLEAN  OSTaskQPendAbort (OS_TCB    *p_tcb,
                               OS_OPT     opt,
                               RTOS_ERR  *p_err)
```

Arguments

`p_tcb`

Pointer to the TCB of the task to pend abort.

`opt`

Provides options for this function:

- `OS_OPT_POST_NONE` No option specified.
- `OS_OPT_POST_NO_SCHED` Indicates that the scheduler will not be called.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NONE_WAITING`

Returned Value

- == `DEF_FALSE` If task was not waiting for a message, or upon error.
- == `DEF_TRUE` If task was waiting for a message and was readied and informed.

Notes / Warnings

None.

OSTaskQPost()

Description

Sends a message to a task.

Files

`os.h/os_task.c`

Prototype

```
void  OSTaskQPost (OS_TCB       *p_tcb,
                   void         *p_void,
                   OS_MSG_SIZE   msg_size,
                   OS_OPT        opt,
                   RTOS_ERR     *p_err)
```

Arguments

`p_tcb`

Pointer to the TCB of the task receiving a message. If you specify a NULL pointer, the message will be posted to the task's queue of the calling task. In other words, you'd be posting a message to yourself.

`p_void`

Pointer to the message to send.

`msg_size`

The size of the message sent (in bytes).

`opt`

Specifies whether the post will be FIFO or LIFO:

- `OS_OPT_POST_FIFO` Post at the end of the queue.
- `OS_OPT_POST_LIFO` Post at the front of the queue.
- `OS_OPT_POST_NO_SCHED` Do not run the scheduler after the post.

Note(s):

1. `OS_OPT_POST_NO_SCHED` can be added with one of the other options.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_NO_MORE_RSRC`
- `RTOS_ERR_INVALID_STATE`

Returned Value

None.

Notes / Warnings

None.

OSTaskRegGet()

Description

Obtains the current value of a task register. Task registers are application specific and can be used to store task specific values such as 'error numbers' (i.e., errno), statistics, etc.

Files

`os.h/os_task.c`

Prototype

```
OS_REG  OSTaskRegGet (OS_TCB     *p_tcb,
                      OS_REG_ID   id,
                      RTOS_ERR   *p_err)
```

Arguments

`p_tcb`

Pointer to the TCB of the task from which you want to read the register. If ' `p_tcb` ' is a NULL pointer, you will get the register of the current task.

`id`

The 'id' of the desired task variable. Note that the 'id' must be less than `OS_CFG_TASK_REG_TBL_SIZE` .

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

The current value of the task's register, or 0 if an error is detected.

Notes / Warnings

None.

OSTaskRegGetID()

Description

This function obtains a task register ID. This function allows task register IDs to be allocated dynamically instead of statically.

Files

`os.h/os_task.c`

Prototype

```
OS_REG_ID  OSTaskRegGetID (RTOS_ERR  *p_err)
```

Arguments

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NO_MORE_RSRC`

Returned Value

The next available task register 'id' or `OS_CFG_TASK_REG_TBL_SIZE` if an error is detected.

Notes / Warnings

None.

OSTaskRegSet()

Description

Changes the current value of a task register. Task registers are application specific and can be used to store task specific values such as error numbers (i.e., errno), statistics, etc.

Files

`os.h/os_task.c`

Prototype

```
void OSTaskRegSet (OS_TCB     *p_tcb,
                   OS_REG_ID   id,
                   OS_REG      value,
                   RTOS_ERR   *p_err)
```

Arguments

`p_tcb`

Pointer to the TCB of the task for which you want to set the register. If 'p_tcb' is a NULL pointer, change the register of the current task.

`id`

The ' `id` ' of the desired task register. Note that the ' `id` ' must be less than `OS_CFG_TASK_REG_TBL_SIZE` .

`value`

The desired value for the task register.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

None.

OSTaskResume()

Description

Resumes a previously suspended task. This is the only call that removes an explicit task suspension.

Files

`os.h/os_task.c`

Prototype

```
void OSTaskResume (OS_TCB    *p_tcb,
                   RTOS_ERR  *p_err)
```

Arguments

`p_tcb`

Pointer to the TCB of the task to resume.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_STATE`

Returned Value

None.

Notes / Warnings

None.

OSTaskSuspend()

Description

This function is called to suspend a task. The task can be the calling task if 'p_tcb' is a NULL pointer or the pointer to the TCB of the calling task.

Files

`os.h/os_task.c`

Prototype

```
void  OSTaskSuspend (OS_TCB   *p_tcb,
                     RTOS_ERR  *p_err)
```

Arguments

`p_tcb`

Pointer to the TCB of the task to resume. If p_tcb is a NULL pointer, suspend the current task.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS`
- `RTOS_ERR_NOT_READY`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_OS_SCHED_LOCKED`

Returned Value

None.

Notes / Warnings

Use this function with great care. If you suspend a task that is waiting for an event (i.e., a message, a semaphore, a queue, etc.) you will prevent this task from running when the event arrives.

OSTaskSemPend()

Description

Blocks the current task until a signal is sent by another task or ISR.

Files

`os.h/os_task.c`

Prototype

```
OS_SEM_CTR  OSTaskSemPend (OS_TICK    timeout,
                           OS_OPT     opt,
                           CPU_TS    *p_ts,
                           RTOS_ERR  *p_err)
```

Arguments

`timeout`

The amount of time you will wait for the signal.

`opt`

Determines if the user wants to block if a semaphore post was not received:

- `OS_OPT_PEND_BLOCKING` Task will block.
- `OS_OPT_PEND_NON_BLOCKING` Task will NOT block.

`p_ts`

Pointer to a variable that will receive the timestamp of when the semaphore was posted or pend aborted. If you pass a NULL pointer (i.e., (CPU_TS *)0), you will not get the timestamp. In other words, passing a NULL pointer is valid and indicates that you don't need the timestamp.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

The current count of signals the task received, 0 if none.

Notes / Warnings

None.

OSTaskSemPendAbort()

Description

Aborts and readies the task specified. This function should be used to fault-abort the wait for a signal, rather than to normally post the signal to the task via `OSTaskSemPost()` .

Files

`os.h/os_task.c`

Prototype

```
CPU_BOOLEAN  OSTaskSemPendAbort (OS_TCB    *p_tcb,
                                 OS_OPT     opt,
                                 RTOS_ERR  *p_err)
```

Arguments

`p_tcb`

Pointer to the TCB of the task to pend abort.

`opt`

Provides options for this function:

- `OS_OPT_POST_NONE` No option selected.
- `OS_OPT_POST_NO_SCHED` Indicates that the scheduler will not be called.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NONE_WAITING`

Returned Value

- == `DEF_FALSE` If task was not waiting for a message, or upon error.
- == `DEF_TRUE` If task was waiting for a message and was readied and informed.

Notes / Warnings

None.

OSTaskSemPost()

Description

Signals a task waiting for a signal.

Files

`os.h/os_task.c`

Prototype

```
OS_SEM_CTR  OSTaskSemPost (OS_TCB    *p_tcb,
                           OS_OPT     opt,
                           RTOS_ERR  *p_err)
```

Arguments

`p_tcb`

The pointer to the TCB of the task to signal. A NULL pointer indicates that you are sending a signal to yourself.

`opt`

Determines the type of POST performed:

- `OS_OPT_POST_NONE` No option.
- `OS_OPT_POST_NO_SCHED` Do not call the scheduler.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_INVALID_STATE`

Returned Value

The current value of the task's signal counter, or 0 if called from an ISR.

Notes / Warnings

None.

OSTaskSemSet()

Description

Clears the signal counter.

Files

`os.h/os_task.c`

Prototype

```
OS_SEM_CTR  OSTaskSemSet (OS_TCB      *p_tcb,
                          OS_SEM_CTR   cnt,
                          RTOS_ERR    *p_err)
```

Arguments

`p_tcb`

Pointer to the TCB of the task to clear the counter. If you specify a NULL pointer, the signal counter of the current task will be cleared.

`cnt`

The desired value of the semaphore counter.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_OS_TASK_WAITING`

Returned Value

The value of the signal counter before being set, or 0 on error.

Notes / Warnings

None.

OSTaskStkChk()

Description

Calculates the amount of free memory left on the specified task's stack.

Files

`os.h/os_task.c`

Prototype

```
void  OSTaskStkChk (OS_TCB        *p_tcb,
                    CPU_STK_SIZE  *p_free,
                    CPU_STK_SIZE  *p_used,
                    RTOS_ERR      *p_err)
```

Arguments

`p_tcb`

Pointer to the TCB of the task to check. If you specify a NULL pointer, you are specifying that you want to check the stack of the current task.

`p_free`

Pointer to a variable that will receive the number of free 'entries' on the task's stack.

`p_used`

Pointer to a variable that will receive the number of used 'entries' on the task's stack.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_NOT_SUPPORTED`

Returned Value

None.

Notes / Warnings

None.

OSTaskStkRedzoneChk()

Description

Verifies a task's stack redzone.

Files

`os.h/os_task.c`

Prototype

```
CPU_BOOLEAN  OSTaskStkRedzoneChk (OS_TCB  *p_tcb)
```

Arguments

`p_tcb`

Pointer to the TCB of the task to check or null for the current task.

Returned Value

- `DEF_FAIL` If the stack is corrupted.
- `DEF_OK` If the stack is NOT corrupted.

Notes / Warnings

(1) This function is INTERNAL to Micrium OS Kernel and your application CAN call it.

OSTaskTimeQuantaSet()

Description

Changes the value of the task's specific time slice.

Files

`os.h/os_task.c`

Prototype

```
void  OSTaskTimeQuantaSet (OS_TCB    *p_tcb,
                           OS_TICK    time_quanta,
                           RTOS_ERR  *p_err)
```

Arguments

`p_tcb`

Pointer to the TCB of the task to change. If you specify a NULL pointer, the current task is assumed.

`time_quanta`

The number of ticks before the CPU is taken away when round-robin scheduling is enabled.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

None.

Kernel Time Management API

Kernel Time Management API

1. OSTimeTickRateHzGet()
2. OSTimeDly()
3. OSTimeDlyHMSM()
4. OSTimeDlyResume()
5. OSTimeGet()

OSTimeTickRateHzGet()

Description

Gets kernel tick rate, in Hertz.

Files

`os.h/os_time.c`

Prototype

```
OS_RATE_HZ OSTimeTickRateHzGet (RTOS_ERR *p_err)
```

Arguments

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

Kernel tick rate, in Hertz.

OSTimeDly()

Description

Delays the execution of the currently running task until the specified number of system ticks expires. This directly equates to delaying the current task for some time to expire. No delay will result if the specified delay is 0. If the specified delay is greater than 0, this results in a context switch.

Files

`os.h/os_time.c`

Prototype

```
void  OSTimeDly (OS_TICK    dly,
                 OS_OPT     opt,
                 RTOS_ERR  *p_err)
```

Arguments

dly

Value in 'clock ticks' that the task for which will either delay, the target matches the value of the tick counter (OSTickCtr). Note that setting this to 0 means that no delay will be applied to the task.

Depending on the option argument, the task will wake up when OSTickCtr reaches:

```
OS_OPT_TIME_DLY       OSTimeGet() + dly
OS_OPT_TIME_TIMEOUT   OSTimeGet() + dly
OS_OPT_TIME_PERIODIC  OSTCBCurPtr.TickCtrPrev + dly
```

opt

Specifies whether 'dly' represents absolute or relative time; default option is OS_OPT_TIME_DLY :

- OS_OPT_TIME_DLY Specifies a relative time from the current count tick retrieved with OSTimeGet().
- OS_OPT_TIME_TIMEOUT Same as OS_OPT_TIME_DLY .
- OS_OPT_TIME_PERIODIC Indicates that 'dly' specifies the periodic value that current tick count (retrieved with OSTimeGet()) must reach before the task will be resumed.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_ARG
- RTOS_ERR_OS_SCHED_LOCKED

Returned Value

None.

Notes / Warnings

None.

OSTimeDlyHMSM()

Description

Delay execution of the currently running task until some time expires. This call allows you to specify the delay time in HOURS, MINUTES, SECONDS, and MILLISECONDS instead of ticks.

Files

os.h/os_time.c

Prototype

```
void  OSTimeDlyHMSM (CPU_INT16U   hours,
                     CPU_INT16U   minutes,
                     CPU_INT16U   seconds,
                     CPU_INT32U   milli,
                     OS_OPT       opt,
                     RTOS_ERR    *p_err)
```

Arguments

hours

Specifies the number of hours that the task will be delayed (max. is 999 if the tick rate is 1000 Hz or less otherwise, a higher value would overflow a 32-bit unsigned counter). (max. 99 if ' opt ' is OS_OPT_TIME_HMSM_STRICT)

`minutes`

Specifies the number of minutes. (max. 59 if 'opt' is `OS_OPT_TIME_HMSM_STRICT`)

`seconds`

Specifies the number of seconds. (max. 59 if 'opt' is `OS_OPT_TIME_HMSM_STRICT`)

`milli`

Specifies the number of milliseconds. (max. 999 if 'opt' is `OS_OPT_TIME_HMSM_STRICT`)

`opt`

Specifies time delay bit-field options logically OR'd; default options marked with *** :

- *** `OS_OPT_TIME_DLY` Specifies a relative time from the current time returned by `OSTimeGet()` .
- `OS_OPT_TIME_TIMEOUT` Same as `OS_OPT_TIME_DLY` .
- `OS_OPT_TIME_PERIODIC` Indicates that the delay specifies the periodic value that the current count tick (retrieved with `OSTimeGet()`) must reach before the task will be resumed.
- *** `OS_OPT_TIME_HMSM_STRICT` Strictly allows only
  - hours (0...99)
  - minutes (0...59)
  - seconds (0...59)
  - milliseconds (0...999)
- `OS_OPT_TIME_HMSM_NON_STRICT` Allows any value of
  - hours (0...999)
  - minutes (0...9999)
  - seconds (0...65535)
  - milliseconds (0...4294967295)

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_OS_SCHED_LOCKED`

Returned Value

None.

Notes / Warnings

1. The resolution of milliseconds depends on the tick rate. For example, you cannot do a 10 mS delay if the ticker interrupts every 100 mS. In this case, the delay would be set to 0. The actual delay is rounded to the nearest tick.
2. Although this function allows you to delay a task for many hours, it is not recommended to put a task to sleep for that long.

OSTimeDlyResume()

Description

Resumes a task that has been delayed through a call to either `OSTimeDly()` or `OSTimeDlyHMSM()` . Note that you cannot call this function to resume a task that is waiting for an event with timeout.

Files

`os.h/os_time.c`

Prototype

```
void  OSTimeDlyResume (OS_TCB    *p_tcb,
                       RTOS_ERR  *p_err)
```

Arguments

`p_tcb`

Pointer to the TCB of the task to resume.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_OS_TASK_SUSPENDED`
- `RTOS_ERR_INVALID_STATE`

Returned Value

None.

Notes / Warnings

None.

OSTimeGet()

Description

Used by your application to obtain the current value of the counter to keep track of the number of clock ticks.

Files

`os.h/os_time.c`

Prototype

```
OS_TICK  OSTimeGet (RTOS_ERR  *p_err)
```

Arguments

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

The current tick count.

Notes / Warnings

None.

Kernel Timer API

Kernel Timer API

- OSTmrCreate()
- OSTmrDel()
- OSTmrRemainGet()
- OSTmrSet()
- OSTmrStart()
- OSTmrStateGet()
- OSTmrStop()

OSTmrCreate()

Description

Called by your application code to create a timer.

Files

`os.h/os_tmr.c`

Prototype

```
void  OSTmrCreate (OS_TMR               *p_tmr,
                   CPU_CHAR             *p_name,
                   OS_TICK               dly,
                   OS_TICK               period,
                   OS_OPT                opt,
                   OS_TMR_CALLBACK_PTR   p_callback,
                   void                 *p_callback_arg,
                   RTOS_ERR             *p_err)
```

Arguments

`p_tmr`

Pointer to the timer to create. Your application is responsible for allocating storage for the timer.

`p_name`

Pointer to an ASCII string that names the timer (useful for debugging)

`dly`

Initial delay.

- If the timer is configured for `ONE-SHOT` mode, this is the timeout used.
- If the timer is configured for `PERIODIC` mode, this is the first timeout to wait for before the timer starts entering periodic mode.

`period`

The 'period' being repeated for the timer. If you specified '`OS_OPT_TMR_PERIODIC`' as an option, when the timer expires, it will automatically restart with the same period.

`opt`

Specifies either:

- `OS_OPT_TMR_ONE_SHOT` The timer counts down only once.
- `OS_OPT_TMR_PERIODIC` The timer counts down and then reloads itself.

`p_callback`

Pointer to a callback function that will be called when the timer expires. The callback function must be declared as follows:

```
void MyCallback (OS_TMR p_tmr, void p_arg);
```

`p_callback_arg`

Pointer to an argument that is passed to the callback function when it is called.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`

Returned Value

None.

Notes / Warnings

1. This function only creates the timer. In other words, the timer is not started when created. To start the timer, call `OSTmrStart()` .

OSTmrDel()

Description

Called by your application code to delete a timer.

Files

`os.h/os_tmr.c`

Prototype

```
CPU_BOOLEAN  OSTmrDel (OS_TMR    *p_tmr,
                       RTOS_ERR  *p_err)
```

Arguments

`p_tmr`

Pointer to the timer to stop and delete.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`
- `RTOS_ERR_NOT_INIT`

Returned Value

- == DEF_TRUE If the timer was deleted.
- == DEF_FALSE If the timer was not deleted or upon an error.

Notes / Warnings

None.

OSTmrRemainGet()

Description

Called to get the number of ticks before a timer times out.

Files

`os.h/os_tmr.c`

Prototype

```
OS_TICK  OSTmrRemainGet (OS_TMR    *p_tmr,
                         RTOS_ERR  *p_err)
```

Arguments

`p_tmr`

Pointer to the timer to obtain the remaining time from.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_INIT`

Returned Value

The time remaining for the timer to expire. The time represents 'timer' increments. In other words, if `OS_TmrTask()` is signaled every 1/10 of a second, the returned value represents the number of 1/10 of a second remaining before the timer expires.

Notes / Warnings

None.

OSTmrSet()

Description

Called by your application code to set a timer.

Files

`os.h/os_tmr.c`

Prototype

```
void  OSTmrSet (OS_TMR              *p_tmr,
                OS_TICK              dly,
                OS_TICK              period,
                OS_TMR_CALLBACK_PTR  p_callback,
                void                *p_callback_arg,
                RTOS_ERR            *p_err)
```

Arguments

`p_tmr`

Pointer to the timer to set.

`dly`

Initial delay. If the timer is configured for ONE-SHOT mode, this is the timeout used. If the timer is configured for PERIODIC mode, this is the first timeout to wait for before the timer starts entering periodic mode.

`period`

The 'period' being repeated for the timer. If you specified ' `OS_OPT_TMR_PERIODIC` ' as an option, when the timer expires, it will automatically restart with the same period.

`p_callback`

Pointer to a callback function that will be called when the timer expires. The callback function must be declared as follows:

```
void MyCallback (OS_TMR *p_tmr, void *p_arg);
```

`p_callback_arg`

Pointer to an argument that is passed to the callback function when it is called.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

1. This function can be called on a running timer. The change to the delay and period will only take effect after the current period or delay has passed. Change to the callback will take effect immediately.

OSTmrStart()

Description

Called by your application code to start a timer.

Files

`os.h/os_tmr.c`

Prototype

```
CPU_BOOLEAN  OSTmrStart (OS_TMR    *p_tmr,
                         RTOS_ERR  *p_err)
```

Arguments

`p_tmr`

Pointer to the timer to start.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_NOT_INIT

Returned Value

- DEF_TRUE If the timer was started.
- DEF_FALSE If the timer was not started or upon an error.

Notes / Warnings

1. When starting/restarting a timer, regardless if it is in PERIODIC or ONE-SHOT mode, the timer is linked to the timer list with the OS_OPT_LINK_DLY option. This option sets the initial expiration time for the timer. For timers in PERIODIC mode, subsequent expiration times are handled by the OS_TmrTask() .

OSTmrStateGet()

Description

Called to determine what state the timer is in:

Files

os.h/os_tmr.c

Prototype

```
OS_STATE  OSTmrStateGet (OS_TMR    *p_tmr,
                         RTOS_ERR  *p_err)
```

Arguments

p_tmr

Pointer to the timer.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE

Returned Value

The current state of the timer (see description).

Notes / Warnings

None.

OSTmrStop()

Description

Called by your application code to stop a timer.

Files

os.h/os_tmr.c

Prototype

```
CPU_BOOLEAN  OSTmrStop (OS_TMR    *p_tmr,
                        OS_OPT     opt,
                        void      *p_callback_arg,
                        RTOS_ERR  *p_err)
```

Arguments

`p_tmr`

Pointer to the timer to stop.

`opt`

Allows you to specify an option to this functions which can be:

- `OS_OPT_TMR_NONE` Stop the timer.
- `OS_OPT_TMR_CALLBACK` Stop the timer and execute the callback function, pass it the callback argument specified when the timer was created.
- `OS_OPT_TMR_CALLBACK_ARG` Stop the timer and execute the callback specified in THIS function call.

`p_callback_arg`

Pointer to a 'new' callback argument that can be passed to the callback function instead of the timer's callback argument. In other words, use 'callback_arg' passed in THIS function INSTEAD of `p_tmr->OSTmrCallbackArg`

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_INIT`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_INVALID_STATE`

Returned Value

- `DEF_TRUE` If we stop the timer (if the timer is already stopped, it returns as `DEF_TRUE`).
- `DEF_FALSE` If the timer is not stopped.

Notes / Warnings

None.

SILICON LABS

IO API

IO API

- IO Core API
- Serial API
- SPI API
- SD API

IO Core API

IO Core API

- IO_Init()

IO_Init()

Description

Initializes the IO submodules.

Files

`io.h/io.c`

Prototype

```
void IO_Init (RTOS_ERR *p_err)
```

Arguments

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`

Returned Value

None.

Notes / Warnings

1. This function is NOT thread-safe and should be called ONCE during system initialization, before multi-threading has begun.
2. This function will not return an error -- any error conditions will trigger a failed assertion.

Serial API

Serial API

- Serial_ConfigureHandleQty()
- Serial_ConfigureMemSeg()
- IO_SERIAL_CTRLR_REG()

Serial_ConfigureHandleQty()

Description

Configure maximum number of serial handles.

Files

`serial.h/serial.c`

Prototype

```
void Serial_ConfigureHandleQty (CPU_SIZE_T  handle_qty)
```

Arguments

`handle_qty`

Maximum number of serial handles.

Returned Value

None.

Notes / Warnings

None.

Serial_ConfigureMemSeg()

Description

Configures memory segment to use for internal data allocations.

Files

`serial.h/serial.c`

Prototype

```
void Serial_ConfigureMemSeg (MEM_SEG  *p_seg)
```

Arguments

`p_seg`

Pointer to memory segment.

Returned Value

None.

Notes / Warnings

None.

IO_SERIAL_CTRLR_REG()

Description

Registers a IO serial controller to the platform manager.

Files

`serial.h`

Prototype

```
IO_SERIAL_CTRLR_REG(name, p_drv_info)
```

Arguments

`name`

Unique name for the IO serial controller. It is recommended to follow the standard "serX" or "spiX" where X is a digit.

`p_drv_info`

Pointer to the serial device hardware information structure of type `SERIAL_CTRLR_DRV_INFO` .

Returned Value

None.

Notes / Warnings

1. This macro should normally be called from the BSP.

SPI API

SPI API

- SPI_ConfigureMemSeg()
- SPI_ConfigureSlaveHandleQty()
- SPI_BusAdd()
- SPI_BusHandleGetFromName()
- SPI_BusStart()
- SPI_BusStop()
- SPI_BusLoopBackEn()
- SPI_SlaveOpen()
- SPI_SlaveClose()
- SPI_SlaveSel()
- SPI_SlaveDesel()
- SPI_SlaveRx()
- SPI_SlaveTx()
- SPI_SlaveXfer()

SPI_ConfigureMemSeg()

Description

Sets the memory segment where file system internal data structures will be allocated.

Files

`spi.h/spi.c`

Prototype

```
void SPI_ConfigureMemSeg (MEM_SEG *p_seg)
```

Arguments

`p_seg`

Pointer to a memory segment.

Returned Value

None.

Notes / Warnings

None.

SPI_ConfigureSlaveHandleQty()

Description

Configure the maximum number of SPI slave handles for all the SPI busses.

Files

`spi.h/spi.c`

Prototype

```
void SPI_ConfigureSlaveHandleQty (CPU_SIZE_T handle_qty)
```

Arguments

`handle_qty`

Max number of SPI slave handles.

Returned Value

None.

Notes / Warnings

None.

SPI_BusAdd()

Description

Creates/Adds SPI bus.

Files

`spi_bus.h/spi_bus.c`

Prototype

```
SPI_BUS_HANDLE SPI_BusAdd (const CPU_CHAR *name,
                           RTOS_ERR *p_err)
```

Arguments

`name`

Name to give to the new bus. Must be unique, constant, and in scope for the remainder of the program.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

Handle to SPI bus.

Notes / Warnings

None.

SPI_BusHandleGetFromName()

Description

Gets SPI bus handle from serial controller name.

Files

`spi_bus.h/spi_bus.c`

Prototype

```
SPI_BUS_HANDLE  SPI_BusHandleGetFromName (const  CPU_CHAR  *name)
```

Arguments

`name`

Serial controller name.

Returned Value

Bas handle, if exists. SPI_BusHandleNull, otherwise.

Notes / Warnings

None.

SPI_BusStart()

Description

Starts given SPI bus/driver.

Files

`spi_bus.h/spi_bus.c`

Prototype

```
void  SPI_BusStart (SPI_BUS_HANDLE   bus_handle,
                    RTOS_ERR        *p_err)
```

Arguments

`bus_handle`

Handle to SPI bus.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_ABORT`

- `RTOS_ERR_TIMEOUT`

Returned Value

None.

Notes / Warnings

None.

SPI_BusStop()

Description

Stops given SPI bus/driver.

Files

`spi_bus.h/spi_bus.c`

Prototype

```
void  SPI_BusStop (SPI_BUS_HANDLE   bus_handle,
                   RTOS_ERR        *p_err)
```

Arguments

`bus_handle`

Handle to SPI bus.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

None.

Notes / Warnings

None.

SPI_BusLoopBackEn()

Description

Enables loopback on SPI bus via IO Ctrl.

Files

`spi_bus.h/spi_bus.c`

Prototype

```
void SPI_BusLoopBackEn (SPI_BUS_HANDLE   bus_handle,
                        CPU_BOOLEAN      en,
                        RTOS_ERR        *p_err)
```

Arguments

`bus_handle`

Handle to SPI bus.

`en`

Flag indicating if loopback mode should be enabled or disabled.

- `DEF_ENABLED`
- `DEF_DISABLED`

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

None.

SPI_SlaveOpen()

Description

Opens/creates a slave on an SPI bus.

Files

`spi_slave.h/spi_slave.c`

Prototype

```
SPI_SLAVE_HANDLE  SPI_SlaveOpen (      SPI_BUS_HANDLE   bus_handle,
                                 const SPI_SLAVE_INFO  *p_slave_info,
                                       RTOS_ERR        *p_err)
```

Arguments

`bus_handle`

Handle on SPI bus.

`p_slave_info`

Pointer to slave information structure.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`

- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_BLK_ALLOC_CALLBACK
- RTOS_ERR_SEG_OVF

Returned Value

SPI slave handle

Notes / Warnings

None.

SPI_SlaveClose()

Description

Closes an SPI slave.

Files

spi_slave.h/spi_slave.c

Prototype

```
void  SPI_SlaveClose (SPI_SLAVE_HANDLE   slave_handle,
                      RTOS_ERR          *p_err)
```

Arguments

slave_handle

Handle on SPI slave.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_NOT_INIT
- RTOS_ERR_POOL_FULL
- RTOS_ERR_OWNERSHIP
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_IS_OWNER
- RTOS_ERR_INVALID_STATE
- RTOS_ERR_ABORT
- RTOS_ERR_TIMEOUT

Returned Value

None.

Notes / Warnings

None.

SPI_SlaveSel()

Description

Enables slave-select for the slave associated with 'handle'

Files

`spi_slave.h/spi_slave.c`

Prototype

```
void  SPI_SlaveSel (SPI_SLAVE_HANDLE   slave_handle,
                    CPU_INT32U         timeout_ms,
                    SPI_OPT            opt,
                    RTOS_ERR          *p_err)
```

Arguments

`slave_handle`

Handle on SPI slave.

`timeout_ms`

Timeout, in milliseconds.

`opt`

Options flags for slave-select. May be one of the following:

- `SPI_OPT_NONE` Block until the bus is available
- `SPI_OPT_NON_BLOCKING` Return immediately if bus is unavailable

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_NOT_INIT`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

None.

Notes / Warnings

1. Enabling slave-select will lock the handle's SPI bus to the calling task, preventing any other task from accessing it.
2. Calls to `SPI_SlaveSel()` cannot be nested.

SPI_SlaveDesel()

Description

Disables slave-select for the slave associated with 'slave_handle'.

Files

`spi_slave.h/spi_slave.c`

Prototype

```
void  SPI_SlaveDesel (SPI_SLAVE_HANDLE   slave_handle,
                      RTOS_ERR          *p_err)
```

Arguments

`slave_handle`

Handle on an SPI slave.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_NOT_INIT`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OWNERSHIP`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

None.

Notes / Warnings

None.

SPI_SlaveRx()

Description

Receives data from SPI slave.

Files

`spi_slave.h/spi_slave.c`

Prototype

```
void  SPI_SlaveRx (SPI_SLAVE_HANDLE   slave_handle,
                   CPU_INT08U        *p_buf,
                   CPU_INT32U         buf_len,
                   CPU_INT32U         timeout_ms,
                   RTOS_ERR          *p_err)
```

Arguments

`slave_handle`

Handle on an SPI slave.

`p_buf`

Pointer to receive buffer.

`buf_len`

Buffer length, in octets,

`timeout_ms`

Timeout, in milliseconds.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_NOT_INIT`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

None.

Notes / Warnings

None.

SPI_SlaveTx()

Description

Transmits data to SPI slave.

Files

`spi_slave.h/spi_slave.c`

Prototype

```
void  SPI_SlaveTx (SPI_SLAVE_HANDLE   slave_handle,
                   CPU_INT08U        *p_buf,
                   CPU_INT32U         buf_len,
                   CPU_INT32U         timeout_ms,
                   RTOS_ERR          *p_err)
```

Arguments

`slave_handle`

Handle on an SPI slave.

`p_buf`

Pointer to transmit buffer.

buf_len

Buffer length, in octets,

timeout_ms

Timeout, in milliseconds.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_NOT_INIT
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_IS_OWNER
- RTOS_ERR_INVALID_STATE
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_ABORT
- RTOS_ERR_TIMEOUT

Returned Value

None.

Notes / Warnings

None.

SPI_SlaveXfer()

Description

Transmits and receives data to/from SPI slave.

Files

spi_slave.h/spi_slave.c

Prototype

```
void  SPI_SlaveXfer (SPI_SLAVE_HANDLE   slave_handle,
                     CPU_INT08U        *p_buf_rx,
                     CPU_INT08U        *p_buf_tx,
                     CPU_INT32U         buf_len,
                     CPU_INT32U         timeout_ms,
                     RTOS_ERR          *p_err)
```

Arguments

slave_handle

Handle on an SPI slave.

p_buf_rx

Pointer to receive buffer.

p_buf_tx

Pointer to transmit buffer.

`buf_len`

Buffer length, in octets,

`timeout_ms`

Timeout, in milliseconds.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_NOT_INIT`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

None.

Notes / Warnings

None.

SD API

SD API

- SD_ConfigureMemSeg()
- SD_ConfigureIO_FnctHandleQty()
- SD_ConfigureEventQty()
- SD_ConfigureXferQty()
- SD_ConfigureEventFncts()
- SD_ConfigureCoreTaskStk()
- SD_ConfigureAsyncTaskStk()
- SD_CoreTaskPrioSet()
- SD_AsyncTaskPrioSet()
- SD_OperationsTimeoutSet()
- SD_BusAdd()
- SD_BusHandleGetFromName()
- SD_BusStart()
- SD_BusStop()
- SD_AlignReqGet()
- SD_CardTypeGet()
- SD_BSP_BusCardDetectEvent()
- SD_BSP_BusCardRemoveEvent()
- IO_SD_CARD_CTRLR_REG()

SD_ConfigureMemSeg()

Description

Configures the memory segment where SD module data structures will be allocated.

Files

`sd.h/sd.c`

Prototype

```
void SD_ConfigureMemSeg (MEM_SEG  *p_seg                MEM_SEG  *p_seg_buf)
```

Arguments

`p_seg`

Pointer to memory segment to use when allocating control data. Can be the same segment used for `p_seg_buf`. `DEF_NULL` means general purpose heap segment.

`p_seg_buf`

Pointer to memory segment to use when allocating data buffers. Can be the same segment used for `p_seg`. `DEF_NULL` means general purpose heap segment.

Returned Value

None.

Notes / Warnings

1. Calling this function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the IO SD module is initialized via the `IO_Init()` function.

SD_ConfigureIO_FnctHandleQty()

Description

Configures the maximum number of SD IO Functions for all the SD buses.

Files

`sd.h/sd.c`

Prototype

```
void SD_ConfigureIO_FnctHandleQty (CPU_SIZE_T  fnct_handle_qty)
```

Arguments

`fnct_handle_qty`

Max number of SD IO function handles.

Returned Value

None.

Notes / Warnings

1. Calling this function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the IO SD module is initialized via the `IO_Init()` function.

SD_ConfigureEventQty()

Description

Configures the maximum number of events for all the SD buses.

Files

`sd.h/sd.c`

Prototype

```
void  SD_ConfigureEventQty (CPU_SIZE_T  event_qty)
```

Arguments

`event_qty`

Quantity of events.

Returned Value

None.

Notes / Warnings

1. Calling this function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the IO SD module is initialized via the `IO_Init()` function.

SD_ConfigureXferQty()

Description

Configures the maximum number of simultaneous transfers for all the SD buses.

Files

`sd.h/sd.c`

Prototype

```
void  SD_ConfigureXferQty (CPU_SIZE_T  xfer_qty)
```

Arguments

`xfer_qty`

Quantity of transfers.

Returned Value

None.

Notes / Warnings

1. Calling this function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the IO SD module is initialized via the `IO_Init()` function.

SD_ConfigureEventFncts()

Description

Sets the structure of callback that will be used by the IO SD module to notify the application of certain events.

Files

`sd.h/sd.c`

Prototype

```
void SD_ConfigureEventFncts (const SD_EVENT_FNCTS  *p_event_fncts)
```

Arguments

`p_event_fncts`

Pointer to a structure containing the event functions to call. [Content MUST be persistent]

Returned Value

None.

Notes / Warnings

1. Calling this function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the IO SD module is initialized via the `IO_Init()` function.

SD_ConfigureCoreTaskStk()

Description

Configures the SD core task's stack.

Files

`sd.h/sd.c`

Prototype

```
void SD_ConfigureCoreTaskStk (CPU_INT32U    stk_size_elements,
                              void         *p_stk)
```

Arguments

`stk_size_elements`

Size, in stack elements, of the task's stack.

`p_stk`

Pointer to base of the task's stack. If `DEF_NULL` , stack will be allocated from KAL's memory segment.

Returned Value

None.

Notes / Warnings

1. Calling this function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the IO SD module is initialized via the `IO_Init()` function.
3. In order to change the priority of the IO SD core task, use the function `SD_CoreTaskPrioSet()` .

SD_ConfigureAsyncTaskStk()

Description

Configures the SD async task's stack.

Files

`sd.h/sd.c`

Prototype

```
void SD_ConfigureAsyncTaskStk (CPU_INT32U    stk_size_elements,
                               void         *p_stk)
```

Arguments

`stk_size_elements`

Size, in stack elements, of the task's stack.

`p_stk`

Pointer to base of the task's stack. If `DEF_NULL` , stack will be allocated from KAL's memory segment.

Returned Value

None.

Notes / Warnings

1. Calling this function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the IO SD module is initialized via the `IO_Init()` function.
3. In order to change the priority of the IO SD core task, use the function `SD_AsyncTaskPrioSet()` .

SD_CoreTaskPrioSet()

Description

Assigns a new priority to the IO SD core task.

Files

`sd.h/sd.c`

Prototype

```
void SD_CoreTaskPrioSet (CPU_INT08U   prio,
                         RTOS_ERR    *p_err)
```

Arguments

`prio`

New priority of the the core task.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_ARG`

Returned Value

None.

Notes / Warnings

1. This function cannot be called before the IO SD module has been initialized via the `IO_Init()` function.

SD_AsyncTaskPrioSet()

Description

Assigns a new priority to the IO SD async task.

Files

`sd.h/sd.c`

Prototype

```
void SD_AsyncTaskPrioSet (CPU_INT08U   prio,
                          RTOS_ERR    *p_err)
```

Arguments

`prio`

New priority of the the async task.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_ARG`

Returned Value

None.

Notes / Warnings

1. This function cannot be called before the IO SD module has been initialized via the `IO_Init()` function.

SD_OperationsTimeoutSet()

Description

Assigns a new timeout value for SD operations.

Files

`sd.h/sd.c`

Prototype

```
void SD_OperationsTimeoutSet (CPU_INT32U   timeout_ms,
                              RTOS_ERR    *p_err)
```

Arguments

`timout_ms`

New timeout value, in ms.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

1. This function cannot be called before the IO SD module has been initialized via the `IO_Init()` function.

SD_BusAdd()

Description

Adds a SD bus.

Files

`sd.h/sd.c`

Prototype

```
SD_BUS_HANDLE SD_BusAdd (const CPU_CHAR  *name,
                         RTOS_ERR  *p_err)
```

Arguments

`name`

Name of SD bus controller.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_NOT_FOUND
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_IS_OWNER
- RTOS_ERR_SEG_OVF
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_ABORT
- RTOS_ERR_TIMEOUT

Returned Value

None.

Notes / Warnings

None.

SD_BusHandleGetFromName()

Description

Gets SD bus handle from its name.

Files

sd.h/sd.c

Prototype

```
SD_BUS_HANDLE SD_BusHandleGetFromName (const CPU_CHAR   *name)
```

Arguments

name

Name of SD bus controller.

Returned Value

Handle to SD bus.

Notes / Warnings

None.

SD_BusStart()

Description

Starts SD bus controller.

Files

sd.h/sd.c

Prototype

```
void SD_BusStart (SD_BUS_HANDLE    bus_handle,
                  RTOS_ERR         *p_err)
```

Arguments

`bus_handle`

Handle to SD bus.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

None.

Notes / Warnings

None.

SD_BusStop()

Description

Stops SD bus controller.

Files

`sd.h/sd.c`

Prototype

```
void SD_BusStop (SD_BUS_HANDLE   bus_handle,
                 RTOS_ERR       *p_err)
```

Arguments

`bus_handle`

Handle to SD bus.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

None.

Notes / Warnings

None.

SD_AlignReqGet()

Description

Gets required buffer alignment.

Files

`sd.h/sd.c`

Prototype

```
CPU_SIZE_T SD_AlignReqGet (SD_BUS_HANDLE   bus_handle,
                           RTOS_ERR       *p_err)
```

Arguments

`bus_handle`

Handle to SD bus.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

Required alignment, in octets.

Notes / Warnings

None.

SD_CardTypeGet()

Description

Gets type of SD card currently connected.

Files

`sd.h/sd.c`

Prototype

```
SD_CARDTYPE SD_CardTypeGet (SD_BUS_HANDLE   bus_handle,
                            RTOS_ERR       *p_err)
```

Arguments

`bus_handle`

Handle to SD bus.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

Type of SD card.

Notes / Warnings

None.

SD_BSP_BusCardDetectEvent()

Description

Reports card detect event to IO-SD core task.

Files

sd.h/sd.c

Prototype

```
void SD_BSP_BusCardDetectEvent (SD_BUS_HANDLE bus_handle)
```

Arguments

bus_handle

Handle to SD bus.

Returned Value

None.

Notes / Warnings

None.

SD_BSP_BusCardRemoveEvent()

Description

Reports card remove event to IO-SD core task.

Files

sd.h/sd.c

Prototype

```
void SD_BSP_BusCardRemoveEvent (SD_BUS_HANDLE bus_handle)
```

Arguments

bus_handle

Handle to SD bus.

Returned Value

None.

Notes / Warnings

None.

IO_SD_CARD_CTRLR_REG()

Description

Registers a IO SD controller to the platform manager.

Files

sd.h

Prototype

```
IO_SD_CARD_CTRLR_REG (name                    p_drv_info)
```

Arguments

name

Unique name for the IO SD controller. It is recommended to follow the standard "sdX", where X is a digit.

`p_drv_info`

Pointer to the SD Bus driver hardware information structure of type `SD_CARD_CTRLR_DRV_INFO`.

Returned Value

None.

Notes / Warnings

1. This macro should normally be called from the BSP.

File System API

File System API

- File System Core API
- File System Storage API
- File System POSIX API

File System Core API

File System Core API

- FSCore_ConfigureMemSeg()
- FSCore_ConfigureMaxFatObjCnt()
- FSCore_ConfigureMaxObjCnt()
- FSCore_Init()
- FSCache_Create()
- FSCache_Assign()
- FSCache_DfltAssign()
- FSCache_Get()
- FSCache_MaxBlkSizeGet()
- FSCache_MinBlkSizeGet()
- FS_FAT_Fmt()
- FSDir_Open()
- FSDir_Close()
- FSDir_Rd()
- FSDir_Query()
- FSDir_PathGet()
- FSDir_VolGet()
- FSEntry_Create()
- FSEntry_Del()
- FSEntry_Rename()
- FSEntry_AttribSet()
- FSEntry_TimeSet()
- FSEntry_Query()
- FSFile_Open()
- FSFile_Close()
- FSFile_Rd()
- FSFile_Wr()
- FSFile_Copy()
- FSFile_Truncate()
- FSFile_PosSet()
- FSFile_PosGet()
- FSFile_Query()
- FSFile_PathGet()
- FSFile_VolGet()
- FSFile_BufAssign()
- FSFile_BufFlush()
- FSFile_ErrClr()
- FSFile_IsEOF()
- FSFile_IsErr()
- FSFile_Lock()
- FSFile_TryLock()
- FSFile_Unlock()
- FSPartition_Init()
- FSPartition_Add()
- FSPartition_CntGet()
- FSPartition_Query()
- FSVol_Open()
- FSVol_Close()

- FSVol_Sync()
- FSVol_Query()
- FSVol_PartitionNbrGet()
- FSVol_LabelSet()
- FSVol_LabelGet()
- FSVol_NameGet()
- FSVol_Get()
- FSVol_BlkDevGet()
- FSWrkDir_Open()
- FSWrkDir_Close()
- FSWrkDir_TaskBind()
- FSWrkDir_TaskUnbind()
- FSWrkDir_Get()
- FSWrkDir_VolGet()
- FSWrkDir_PathGet()

FSCore_ConfigureMemSeg()

Description

Set the memory segment where file system internal data structures will be allocated.

Files

`fs_core.h/fs_core.c`

Prototype

```
void  FSCore_ConfigureMemSeg (MEM_SEG  *p_seg)
```

Arguments

`p_seg`

Pointer to a memory segment.

Returned Value

none.

Notes / Warnings

None.

FSCore_ConfigureMaxFatObjCnt()

Description

Set the maximum number of file system FAT objects.

Files

`fs_core.h/fs_core.c`

Prototype

```
void  FSCore_ConfigureMaxFatObjCnt (FS_CORE_CFG_MAX_FAT_OBJ_CNT  max_cnt)
```

Arguments

`max_cnt`

Structure containing the maximum number of each FAT object.

Returned Value

none.

Notes / Warnings

None.

FSCore_ConfigureMaxObjCnt()

Description

Set the maximum number of file system core objects.

Files

`fs_core.h/fs_core.c`

Prototype

```
void FSCore_ConfigureMaxObjCnt (FS_CORE_CFG_MAX_OBJ_CNT  max_cnt)
```

Arguments

`max_cnt`

Structure containing the maximum number of each core object.

Returned Value

none.

Notes / Warnings

None.

FSCore_Init()

Description

Initialize file system.

Files

`fs_core.h/fs_core.c`

Prototype

```
void FSCore_Init (RTOS_ERR  *p_err)
```

Arguments

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_ALREADY_INIT`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_SEG_OVF`

Returned Value

none.

Notes / Warnings

None.

FSCache_Create()

Description

Create a cache instance.

Files

`fs_core_cache.h/fs_core_cache.c`

Prototype

```
FS_CACHE  *FSCache_Create (const FS_CACHE_CFG  *p_cache_cfg,
                           RTOS_ERR      *p_err)
```

Arguments

`p_cache_cfg`

Pointer to a cache configuration structure.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_INIT`
- `RTOS_ERR_SEG_OVF`

Returned Value

Pointer to a cache instance.

Notes / Warnings

(1) The cache configuration structure has the following fields:

```
typedef  struct fs_cache_cfg {
   CPU_SIZE_T   Align;                  /* Buffer alignment requirement.                 */
   CPU_SIZE_T   MinBlkCnt;              /* Number of buffers available to cache module.  */
   CPU_SIZE_T   MinLbSize;              /* Minimum logical block size to consider.       */
   CPU_SIZE_T   MaxLbSize;              /* Maximum logical block size to consider.       */
   MEM_SEG     *BlkMemSegPtr;           /* Memory segment where buffers will be allocated from. */
} FS_CACHE_CFG;
```

FSCache_Assign()

Description

Bind a block device to a cache instance.

Files

`fs_core_cache.h/fs_core_cache.c`

Prototype

```
void  FSCache_Assign (FS_BLK_DEV_HANDLE   blk_dev_handle,
                      FS_CACHE           *p_cache,
                      RTOS_ERR           *p_err)
```

Arguments

`blk_dev_handle`

Handle to a block device.

`p_cache`

Pointer to a cache instance.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_SIZE_INVALID`
- `RTOS_ERR_BLK_DEV_CLOSED`

Returned Value

none.

Notes / Warnings

None.

FSCache_DfltAssign()

Description

Create a cache instance with default parameters and bind it to a block device.

Files

`fs_core_cache.h/fs_core_cache.c`

Prototype

```
FS_CACHE  *FSCache_DfltAssign (FS_BLK_DEV_HANDLE   blk_dev_handle,
                               CPU_SIZE_T          cache_blk_cnt,
                               RTOS_ERR           *p_err)
```

Arguments

`blk_dev_handle`

Handle to a block device.

`blk_dev_handle`

Number of cache blocks to allocate.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_INIT RTOS_ERR_SEG_OVF`
- `RTOS_ERR_BLK_DEV_CLOSED`

Returned Value

Pointer to the created default cache.

Notes / Warnings

(1) This function is a sort of wrapper that uses `FSCache_Create()` and `FSCache_Assign()` .

(2) The default cache parameters are:

```
FS_CACHE_CFG.BlkMemSegPtr = default core memory segment (that is heap region)
FS_CACHE_CFG.Align        = default alignment required by block device
FS_CACHE_CFG.MaxLbSize    = default block device sector size
FS_CACHE_CFG.MinLbSize    = default block device sector size
```

FSCache_Get()

Description

Get a cache instance from a block device.

Files

`fs_core_cache.h/fs_core_cache.c`

Prototype

```
FS_CACHE  *FSCache_Get (FS_BLK_DEV_HANDLE   blk_dev_handle)
```

Arguments

`blk_dev_handle`

Handle to a block device.

Returned Value

Pointer to the associated cache instance or `DEF_NULL` if cache has not been assigned.

Notes / Warnings

None.

FSCache_MaxBlkSizeGet()

Description

Get the maximum block size supported by the given cache instance.

Files

`fs_core_cache.h/fs_core_cache.c`

Prototype

```
FS_LB_SIZE  FSCache_MaxBlkSizeGet (FS_CACHE  *p_cache)
```

Arguments

`p_cache`

Pointer to a cache instance.

Returned Value

Maximum supported block size.

Notes / Warnings

None.

FSCache_MinBlkSizeGet()

Description

Get the minimum block size supported by the given cache instance.

Files

`fs_core_cache.h/fs_core_cache.c`

Prototype

```
FS_LB_SIZE  FSCache_MinBlkSizeGet (FS_CACHE  *p_cache)
```

Arguments

`p_cache`

Pointer to a cache instance.

Returned Value

Minimum supported block size.

Notes / Warnings

None.

FS_FAT_Fmt()

Description

Format a partition in FAT.

Files

`fs_fat.h/fs_fat.c`

Prototype

```
void  FS_FAT_Fmt (FS_BLK_DEV_HANDLE   blk_dev_handle,
                  FS_PARTITION_NBR    partition_nbr,
                  FS_FAT_VOL_CFG     *p_fat_vol_cfg,
                  RTOS_ERR           *p_err)
```

Arguments

`blk_dev_handle`

Handle to a block device.

`partition_nbr`

Number of the partition to be formatted. The first partition is partition number #1. FS_PARTITION_NBR_VOID to erase the MBR and use the whole media.

`p_fat_vol_cfg`

- Pointer to a FAT volume configuration structure (optional).
- `DEF_NULL` for default values (see Note #1).

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_NOT_FOUND
- RTOS_ERR_VOL_OPENED
- RTOS_ERR_PARTITION_INVALID
- RTOS_ERR_BLK_DEV_CORRUPTED
- RTOS_ERR_BLK_DEV_CLOSED
- RTOS_ERR_IO

Returned Value

none.

Notes / Warnings

(1) In general, the best option for the FAT volume configuration structure is `DEF_NULL`. In that case, the File System FAT layer will automatically determine the proper FAT configuration values based in the block device total size. If you want to specify the values, the FAT configuration structure is composed of:

- The cluster size expressed in number of sectors.
- The reserved area size expressed in number of sectors.
- The number of entries in the root directory table.
- The type of FAT: FAT 12, 16 or 32.
- The number of FAT tables.

You may use these typical values for the FAT configuration structure:

```
typedef struct fs_fat_vol_cfg {
  CPU_SIZE_T  ClusSizeInSec;
  CPU_SIZE_T  RsvdAreaSizeInSec;
  CPU_INT16U  RootDirEntryCnt;
  CPU_INT08U  FAT_Type;
  CPU_INT08U  NbrFATs;
} FS_FAT_VOL_CFG;
```

Configuration	Field	Typical Values
Cluster size	.ClusSizeInSec	1, 2, 4, 8, 16, 32, 64
Reserved area size	.RsvdAreaSizeInSec	FS_FAT_DFLT_RSVD_SEC_CNT_FAT12FS_FAT_DFLT_RSVD_SEC_CNT_FAT16FS_FAT_DFLT_RSVD_SE
Number of entries in the root directory table	.RootDirEntryCnt	FS_FAT_DFLT_ROOT_ENT_CNT_FAT12FS_FAT_DFLT_ROOT_ENT_CNT_FAT16FS_FAT_DFLT_ROOT_EN
Type of FAT	.FAT_Type	FS_FAT_TYPE_FAT12FS_FAT_TYPE_FAT16FS_FAT_TYPE_FAT32
Number of FAT tables	.NbrFATs	FS_FAT_DFLT_NBR_FATS_FAT12FS_FAT_DFLT_NBR_FATS_FAT16FS_FAT_DFLT_NBR_FATS_FAT32

FSDir_Open()

Description

Open a directory.

Files

`fs_core_dir.h/fs_core_dir.c`

Prototype

```
FS_DIR_HANDLE FSDir_Open ( FS_WRK_DIR_HANDLE wrk_dir_handle,
                           const CPU_CHAR     *path,
                           FS_FLAGS           mode,
                           RTOS_ERR           *p_err)
```

Arguments

`wrk_dir_handle`

Handle to a working directory (see Note #1).

`path`

Path of the directory relative to the give working directory.

`mode`

Directory access mode; valid bit-wise OR of one or more of the following

- `FS_DIR_ACCESS_MODE_EXCL` Directory will be opened if and only if it does not already exist.
- `FS_DIR_ACCESS_MODE_CREATE` Directory will be created, if necessary.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_INIT`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_ALREADY_EXISTS`
- `RTOS_ERR_TYPE_INVALID`
- `RTOS_ERR_NAME_INVALID`
- `RTOS_ERR_PARENT_NOT_DIR`
- `RTOS_ERR_MAX_DEPTH_EXCEEDED`
- `RTOS_ERR_DIR_FULL`
- `RTOS_ERR_WRK_DIR_CLOSED`
- `RTOS_ERR_VOL_CLOSED`
- `RTOS_ERR_BLK_DEV_CLOSED`
- `RTOS_ERR_VOL_FULL`
- `RTOS_ERR_VOL_CORRUPTED`
- `RTOS_ERR_BLK_DEV_CORRUPTED`
- `RTOS_ERR_IO`

Returned Value

Handle to the opened directory.

Notes / Warnings

1. If no working directory has been opened with `FSWrkDir_Open()`, you can pass a NULL working directory handle using the #define `FS_WRK_DIR_NULL`.

FSDir_Close()

Description

Close a directory.

Files

`fs_core_dir.h/fs_core_dir.c`

Prototype

```
void  FSDir_Close (FS_DIR_HANDLE   dir_handle,
                   RTOS_ERR       *p_err)
```

Arguments

`dir_handle`

Handle to a directory.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_ENTRY_CLOSED`

Returned Value

none.

Notes / Warnings

None.

FSDir_Rd()

Description

Read a directory entry from a directory table.

Files

`fs_core_dir.h/fs_core_dir.c`

Prototype

```
CPU_BOOLEAN  FSDir_Rd (FS_DIR_HANDLE   dir_handle,
                       FS_ENTRY_INFO  *p_entry_info,
                       CPU_CHAR       *p_buf,
                       CPU_SIZE_T      buf_size,
                       RTOS_ERR       *p_err)
```

Arguments

`dir_handle`

Handle to a directory.

`p_entry_info`

Pointer to a structure that will receive the entry information.

`p_buf`

Pointer to a buffer that will receive the entry name.

`buf_size`

Size of the given buffer.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_ENTRY_CLOSED
- RTOS_ERR_VOL_CLOSED
- RTOS_ERR_VOL_CORRUPTED
- RTOS_ERR_BLK_DEV_CLOSED
- RTOS_ERR_BLK_DEV_CORRUPTED
- RTOS_ERR_IO

Returned Value

DEF_NO , if there are other available entries.

DEF_YES , if there are no more entry.

Notes / Warnings

1. Entries for "dot" (current directory) and "dot-dot" (parent directory) will be returned, if present.
2. A subsequent call to FSDir_Rd() will return the next directory entry in the table until there is no more entry to read.

FSDir_Query()

Description

Get information about a directory.

Files

fs_core_dir.h/fs_core_dir.c

Prototype

```
void  FSDir_Query (FS_DIR_HANDLE   dir_handle,
                   FS_ENTRY_INFO  *p_entry_info,
                   RTOS_ERR       *p_err)
```

Arguments

dir_handle

Handle to a directory.

p_entry_info

Pointer to structure that will receive the entry information.

p_err

Pointer to variable that will receive the return error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_ENTRY_CLOSED
- RTOS_ERR_VOL_CLOSED
- RTOS_ERR_VOL_CORRUPTED
- RTOS_ERR_BLK_DEV_CLOSED
- RTOS_ERR_BLK_DEV_CORRUPTED
- RTOS_ERR_IO

Returned Value

none.

Notes / Warnings

1. If data is stored in the file buffer waiting to be written to the storage medium, the actual file size may need to be adjusted to account for that buffered data.
2. The entry information structure has the following fields:

```
typedef struct fs_entry_attrib {
    CPU_INT32U     Wr:1;          /* Bit indicating if entry has write access.          */
    CPU_INT32U     Rd:1;          /* Bit indicating if entry has read access.           */
    CPU_INT32U     Hidden:1;      /* Bit indicating if entry is hidden.                 */
    CPU_INT32U     IsDir:1;       /* Bit indicating if entry is a directory or a file.  */
    CPU_INT32U     IsRootDir:1;   /* Bit indicating if entry is root directory.         */
} FS_ENTRY_ATTRIB;

typedef struct fs_entry_info {
    FS_ENTRY_ATTRIB  Attrib;          /* Entry attributes.                        */
    FS_ID            NodeId;          /* Entry node id.                           */
    FS_ID            DevId;           /* Entry device id.                         */
    CPU_SIZE_T       Size;            /* File size in octets.                     */
    CLK_TS_SEC       DateAccess;      /* Date of last access.                     */
    FS_LB_QTY        BlkCnt;          /* Number of blocks allocated for file.     */
    FS_LB_SIZE       BlkSize;         /* Block size in octets.                    */
    CLK_TS_SEC       DateTimeWr;      /* Date/time of last write.                 */
    CLK_TS_SEC       DateTimeCreate;  /* Date/time of creation.                   */
} FS_ENTRY_INFO;
```

FSDir_PathGet()

Description

Get absolute path to an opened file.

Files

`fs_core_dir.h/fs_core_dir.c`

Prototype

```
void  FSDir_PathGet (FS_DIR_HANDLE   dir_handle,
                     CPU_CHAR       *p_buf,
                     CPU_SIZE_T      buf_size,
                     RTOS_ERR       *p_err)
```

Arguments

`dir_handle`

Handle to a directory.

`p_buf`

Pointer to a buffer that will receive the path.

`buf_size`

Size of the provided buffer.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_ENTRY_CLOSED`
- `RTOS_ERR_VOL_CLOSED`

- RTOS_ERR_VOL_CORRUPTED
- RTOS_ERR_BLK_DEV_CLOSED
- RTOS_ERR_BLK_DEV_CORRUPTED
- RTOS_ERR_IO

Returned Value

none.

Notes / Warnings

None.

FSDir_VolGet()

Description

Get volume containing the given directory.

Files

fs_core_dir.h/fs_core_dir.c

Prototype

```
FS_VOL_HANDLE FSDir_VolGet (FS_DIR_HANDLE dir_handle)
```

Arguments

dir_handle

Handle to a directory.

Returned Value

Handle to the parent volume or NULL handle if the directory is closed.

Notes / Warnings

None.

FSEntry_Create()

Description

Create a file or directory.

Files

fs_core_entry.h/fs_core_entry.c

Prototype

```
void FSEntry_Create (   FS_WRK_DIR_HANDLE   wrk_dir_handle,
                 const CPU_CHAR        *path,
                      FS_FLAGS          entry_type,
                      CPU_BOOLEAN       excl,
                      RTOS_ERR         *p_err)
```

Arguments

wrk_dir_handle

Handle to a working directory.

`p_path`

Entry path relative to the given working directory.

`entry_type`

Indicates whether the new entry is a directory / a file:

- `FS_ENTRY_TYPE_DIR` , if the entry can be a directory.
- `FS_ENTRY_TYPE_FILE` , if the entry can be a file.

`excl`

Indicates whether the creation of new entry will be exclusive :

- `DEF_YES` , if the entry will be created ONLY if the entry does not exist.
- `DEF_NO` , if the entry will be created even if the entry does exist.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_ALREADY_EXISTS`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_NAME_INVALID`
- `RTOS_ERR_ENTRY_OPENED`
- `RTOS_ERR_ENTRY_ROOT_DIR`
- `RTOS_ERR_ENTRY_PARENT_NOT_DIR`
- `RTOS_ERR_WRK_DIR_CLOSED`
- `RTOS_ERR_VOL_CLOSED`
- `RTOS_ERR_VOL_CORRUPTED`
- `RTOS_ERR_BLK_DEV_CLOSED`
- `RTOS_ERR_BLK_DEV_CORRUPTED`
- `RTOS_ERR_IO`

Returned Value

none.

Notes / Warnings

1. The function behavior is resumed in the table below.

Entry Type Argument	'excl' Argument	Entry Already Exists?	Entry Type on Disk	Behavior
X	X	no	-	Create entry
FS_ENTRY_TYPE_FILE	DEF_NO	yes	file	Truncate file
FS_ENTRY_TYPE_FILE	DEF_YES	yes	file	Error: already exists
FS_ENTRY_TYPE_FILE	X	yes	directory	Error: type mismatch
FS_ENTRY_TYPE_DIR	X	yes	file	Error: type mismatch
FS_ENTRY_TYPE_DIR	DEF_NO	yes	directory	Nothing is done
FS_ENTRY_TYPE_DIR	DEF_YES	yes	directory	Error: already exists

"X" means "don't care". "-" means "not relevant".

1. The root directory may NOT be created.

FSEntry_Del()

Description

Delete a file or directory.

Files

`fs_core_entry.h/fs_core_entry.c`

Prototype

```
void FSEntry_Del (    FS_WRK_DIR_HANDLE   wrk_dir_handle,
                 const CPU_CHAR         *path,
                      FS_FLAGS           entry_type,
                      RTOS_ERR          *p_err)
```

Arguments

`wrk_dir_handle`

Handle to a working directory.

`p_path`

Entry path relative to the given working directory.

`entry_type`

Indicates whether the entry to delete is a directory or a file :

- `FS_ENTRY_TYPE_DIR` , if the entry is a directory.
- `FS_ENTRY_TYPE_FILE` , if the entry is a file.
- `FS_ENTRY_TYPE_ANY` , if the entry is any type.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_ENTRY_OPENED`
- `RTOS_ERR_ENTRY_ROOT_DIR`
- `RTOS_ERR_INVALID_TYPE`
- `RTOS_ERR_NAME_INVALID`
- `RTOS_ERR_DIR_NOT_EMPTY`
- `RTOS_ERR_WRK_DIR_CLOSED`
- `RTOS_ERR_VOL_CLOSED`
- `RTOS_ERR_VOL_CORRUPTED`
- `RTOS_ERR_BLK_DEV_CLOSED`
- `RTOS_ERR_BLK_DEV_CORRUPTED`
- `RTOS_ERR_IO`

Returned Value

none.

Notes / Warnings

(1) When a file is removed, the space occupied by the file is freed and is no longer accessible.

(2) The root directory cannot be deleted.

(3) A directory can be removed only if it is an empty directory.

FSEntry_Rename()

Description

Rename a file or directory.

Files

`fs_core_entry.h/fs_core_entry.c`

Prototype

```
void FSEntry_Rename (   FS_WRK_DIR_HANDLE  src_wrk_dir_handle,
                 const CPU_CHAR        *p_src_path,
                    FS_WRK_DIR_HANDLE  dest_wrk_dir_handle,
                 const CPU_CHAR        *p_dest_path,
                    CPU_BOOLEAN        excl,
                    RTOS_ERR          *p_err)
```

Arguments

`src_wrk_dir_handle`

Handle to the source working directory.

`p_src_path`

Source entry path relative to the given source working directory.

`dest_wrk_dir_handle`

Handle to the destination working directory.

`p_dest_path`

Destination entry path relative to the given source working directory.

`excl`

Indicates whether creation of new entry should be exclusive (see Note #3):

- `DEF_YES` , if the entry will be renamed ONLY if the destination entry does not exist.
- `DEF_NO` , if the entry will be renamed even if the destination entry does exist.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_ENTRY_ROOT_DIR`
- `RTOS_ERR_ENTRY_PARENT_NOT_DIR`
- `RTOS_ERR_ENTRY_OPENED`
- `RTOS_ERR_ALREADY_EXISTS`
- `RTOS_ERR_NAME_INVALID`
- `RTOS_ERR_INVALID_TYPE`
- `RTOS_ERR_DIR_NOT_EMPTY`
- `RTOS_ERR_WRK_DIR_CLOSED`
- `RTOS_ERR_VOL_CLOSED`
- `RTOS_ERR_VOL_CORRUPTED`
- `RTOS_ERR_BLK_DEV_CLOSED`
- `RTOS_ERR_BLK_DEV_CORRUPTED`
- `RTOS_ERR_IO`

Returned Value

none.

Notes / Warnings

1. If source and destination entries reside on different volumes, then the source entry must be a file. If the source entry is a directory, an error will be returned.
2. (a) If the source and destination entries are the same entry, the volume will not be modified and no error will be returned.
   (b) If the source entry is a file:
   1. ... the destination entry must NOT be a directory.
   2. ... if 'excl' is `DEF_NO` and the destination entry is a file, the destination entry will be deleted.

   (c) If the source entry is a directory:
   1. ... the destination entry must NOT be a file.
   2. ... if 'excl' is `DEF_NO` and the destination entry is a directory, the target entry MUST be empty; if so, it will be deleted.
3. If 'excl' is `DEF_NO` , the destination entry must not exist.
4. The root directory may NOT be renamed.

FSEntry_AttribSet()

Description

Set a file or directory's attributes.

Files

`fs_core_entry.h/fs_core_entry.c`

Prototype

```
void  FSEntry_AttribSet (    FS_WRK_DIR_HANDLE   wrk_dir_handle,
                   const CPU_CHAR          *path,
                         FS_FLAGS          attrib,
                         RTOS_ERR          *p_err)
```

Arguments

`wrk_dir_handle`

Handle to a working directory.

`p_path`

Entry path relative to the given working directory.

`attrib`

Entry attributes to set (see Note #2).

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_ENTRY_OPENED`
- `RTOS_ERR_ENTRY_ROOT_DIR`
- `RTOS_ERR_WRK_DIR_CLOSED`
- `RTOS_ERR_VOL_CLOSED`
- `RTOS_ERR_VOL_CORRUPTED`
- `RTOS_ERR_BLK_DEV_CLOSED`
- `RTOS_ERR_BLK_DEV_CORRUPTED`
- `RTOS_ERR_IO`

Returned Value

none.

Notes / Warnings

1. If the entry does not exist, an error is returned.
2. Three entry attributes may be modified by this function :

- `FS_ENTRY_ATTRIB_RD` Entry is readable.
- `FS_ENTRY_ATTRIB_WR` Entry is writable.
- `FS_ENTRY_ATTRIB_HIDDEN` Entry is hidden from user-level processes.
  (a) An attribute will be cleared if its flag is not OR'd into 'attrib'. The attribute will be set otherwise.
  (b) If another flag besides these is set, then an error will be returned.

1. The attributes of the root directory may NOT be set.

FSEntry_TimeSet()

> This function is deprecated and will be replaced by sl_fs_entry_time_set() in a next release.

Description

Set a file or directory's date/time.

Files

`fs_core_entry.h/fs_core_entry.c`

Prototype

```
void FSEntry_TimeSet (   FS_WRK_DIR_HANDLE  wrk_dir_handle,
                  const CPU_CHAR       *path,
                      CLK_DATE_TIME    *p_time,
                      CPU_INT08U       time_type,
                      RTOS_ERR         *p_err)
```

Arguments

`wrk_dir_handle`

Handle to a working directory.

`p_path`

Entry path relative to the given working directory.

`p_time`

Pointer to date/time.

`time_type`

Flag to indicate which Date/Time should be set

- `FS_DATE_TIME_CREATE`
- `FS_DATE_TIME_MODIFY`
- `FS_DATE_TIME_ACCESS`
- `FS_DATE_TIME_ALL`

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`

- RTOS_ERR_NOT_FOUND
- RTOS_ERR_ENTRY_OPENED
- RTOS_ERR_ENTRY_ROOT_DIR
- RTOS_ERR_WRK_DIR_CLOSED
- RTOS_ERR_VOL_CLOSED
- RTOS_ERR_VOL_CORRUPTED
- RTOS_ERR_BLK_DEV_CLOSED
- RTOS_ERR_BLK_DEV_CORRUPTED
- RTOS_ERR_IO

Returned Value

none.

Notes / Warnings

1. The date/time of the root directory may NOT be set.
2. The pointer `p_time` points to a structure of type `CLK_DATE_TIME`.

FSEntry_Query()

Description

Get information about a file or directory.

Files

`fs_core_entry.h/fs_core_entry.c`

Prototype

```
void FSEntry_Query (    FS_WRK_DIR_HANDLE  wrk_dir_handle,
                        const CPU_CHAR        *path,
                        FS_ENTRY_INFO      *p_entry_info,
                        RTOS_ERR           *p_err)
```

Arguments

`wrk_dir_handle`

Handle to a working directory.

`p_path`

Entry path relative to the given working directory.

`p_entry_info`

Pointer to structure that will receive the entry information.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_NOT_FOUND
- RTOS_ERR_WRK_DIR_CLOSED
- RTOS_ERR_VOL_CLOSED
- RTOS_ERR_VOL_CORRUPTED
- RTOS_ERR_BLK_DEV_CLOSED
- RTOS_ERR_BLK_DEV_CORRUPTED
- RTOS_ERR_IO

Returned Value

none.

Notes / Warnings

1. The entry information structure has the following fields:

```
typedef struct fs_entry_attrib {
    CPU_INT32U    Wr:1;              /* Bit indicating if entry has write access.       */
    CPU_INT32U    Rd:1;              /* Bit indicating if entry has read access.        */
    CPU_INT32U    Hidden:1;          /* Bit indicating if entry is hidden.              */
    CPU_INT32U    IsDir:1;           /* Bit indicating if entry is a directory or a file. */
    CPU_INT32U    IsRootDir:1;       /* Bit indicating if entry is root directory.      */
} FS_ENTRY_ATTRIB;

typedef struct fs_entry_info {
    FS_ENTRY_ATTRIB  Attrib;         /* Entry attributes.                               */
    FS_ID          NodeId;           /* Entry node id.                                  */
    FS_ID          DevId;            /* Entry device id.                                */
    CPU_SIZE_T     Size;             /* File size in octets.                            */
    CLK_TS_SEC     DateAccess;       /* Date of last access.                            */
    FS_LB_QTY      BlkCnt;           /* Number of blocks allocated for file.            */
    FS_LB_SIZE     BlkSize;          /* Block size in octets.                           */
    CLK_TS_SEC     DateTimeWr;       /* Date/time of last write.                        */
    CLK_TS_SEC     DateTimeCreate;   /* Date/time of creation.                          */
} FS_ENTRY_INFO;
```

FSFile_Open()

Description

Open a file.

Files

`fs_core_file.h/fs_core_file.c`

Prototype

```
FS_FILE_HANDLE  FSFile_Open (      FS_WRK_DIR_HANDLE   wrk_dir_handle,
                             const CPU_CHAR           *path,
                                   FS_FLAGS            mode,
                                   RTOS_ERR           *p_err)
```

Arguments

`wrk_dir_handle`

Handle to a working directory (see Note #1).

`path`

File path relative to the given working directory.

`mode`

File access mode; valid bit-wise OR of one or more of the following (see Note #2):

- `FS_FILE_ACCESS_MODE_RD` File opened for reads.
- `FS_FILE_ACCESS_MODE_WR` File opened for writes.
- `FS_FILE_ACCESS_MODE_CREATE` File will be created, if necessary.
- `FS_FILE_ACCESS_MODE_TRUNCATE` File length will be truncated to 0.

- FS_FILE_ACCESS_MODE_APPEND All writes will be performed at EOF.
- FS_FILE_ACCESS_MODE_EXCL File will be opened if & only if it does not already exist.

p_err

Pointer to variable that will receive the return error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_NOT_INIT
- RTOS_ERR_NOT_FOUND
- RTOS_ERR_ALREADY_EXISTS
- RTOS_ERR_TYPE_INVALID
- RTOS_ERR_NAME_INVALID
- RTOS_ERR_PARENT_NOT_DIR
- RTOS_ERR_MAX_DEPTH_EXCEEDED
- RTOS_ERR_DIR_FULL
- RTOS_ERR_WRK_DIR_CLOSED
- RTOS_ERR_VOL_CLOSED
- RTOS_ERR_VOL_FULL
- RTOS_ERR_VOL_CORRUPTED
- RTOS_ERR_BLK_DEV_CLOSED
- RTOS_ERR_BLK_DEV_CORRUPTED
- RTOS_ERR_IO

Returned Value

Handle to the opened file descriptor.

Notes / Warnings

1. If no working directory has been opened with FSWrkDir_Open(), you can pass a NULL working directory handle using the #define FS_WRK_DIR_NULL.
2. Micrium OS File System native file access modes along with equivalent standard 'fopen()' access modes. Combinations not listed below will generate an error. The FS_FILE_ACCESS_MODE_EXCL flag is allowed wherever FS_FILE_ACCESS_MODE_CREATE is allowed.

Native Access Mode	Posix Access Mode
RD	"r" / "rb"
WR	
WR | APPEND	
WR | TRUNCATE	
WR | TRUNCATE | APPEND	
WR | CREATE	
WR | CREATE | APPEND	"a" / "ab"
WR | CREATE | TRUNCATE	"w" / "wb"
WR | CREATE | TRUNCATE | APPEND	
RD | WR	"r+" / "rb+" / "r+b"
RD | WR | APPEND	
RD | WR | TRUNCATE	
RD | WR | TRUNCATE | APPEND	
RD | WR | CREATE | TRUNCATE	"w+" / "wb+" / "w+b"
RD | WR | CREATE | APPEND	"a+" / "ab+" / "a+b"
RD | WR TRUNCATE | APPEND	
RD | WR | CREATE | TRUNCATE | APPEND	

3. The depth of the directory tree may NOT exceed 255, counting from the virtual file system root directory. The 'RTOS_ERR_WOULD_OVF' error is returned whenever this limit is exceeded. This can only happen when the 'FS_FILE_ACCESS_MODE_CREATE' access mode is used.
4. If the entry located at the specified path is a directory, the 'RTOS_ERR_INVALID_TYPE' is returned.

FSFile_Close()

Description

Close a file.

Files

`fs_core_file.h/fs_core_file.c`

Prototype

```
void  FSFile_Close (FS_FILE_HANDLE   file_handle,
                    RTOS_ERR        *p_err)
```

Arguments

`file_handle`

Handle to a file.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_ENTRY_CLOSED`
- `RTOS_ERR_VOL_CORRUPTED`
- `RTOS_ERR_BLK_DEV_CORRUPTED`
- `RTOS_ERR_IO`

Returned Value

none.

Notes / Warnings

None.

FSFile_Rd()

Description

Read from a file.

Files

`fs_core_file.h/fs_core_file.c`

Prototype

```
CPU_SIZE_T  FSFile_Rd (FS_FILE_HANDLE   file_handle,
                       void            *p_dest,
                       CPU_SIZE_T       size,
                       RTOS_ERR        *p_err)
```

Arguments

`file_handle`

Handle to a file.

`p_dest`

Pointer to destination buffer.

`size`

Number of octets to read.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_ENTRY_CLOSED`
- `RTOS_ERR_FILE_ERR_STATE`
- `RTOS_ERR_VOL_CLOSED`
- `RTOS_ERR_VOL_CORRUPTED`
- `RTOS_ERR_BLK_DEV_CLOSED`
- `RTOS_ERR_BLK_DEV_CORRUPTED`
- `RTOS_ERR_IO`

Returned Value

Number of bytes read (may be smaller than 'size' if the end of file is encountered).

Notes / Warnings

(1) All accesses to the file descriptor are atomically performed before the actual I/O's occur. Since `FSFile_Rd()` may execute concurrently, this means that there is no (guaranteed) relation between the order in which `FSFile_Rd()` calls return and the order of the returned data chunks inside the file.

(2) If an error occurred in the previous file access, the error indicator must be cleared (with `FSFile_ErrClr()`) before another access will be allowed. Otherwise, the ' `RTOS_ERR_FILE_ERR_STATE` ' error is set and the function returns.

FSFile_Wr()

Description

Write to a file.

Files

`fs_core_file.h/fs_core_file.c`

Prototype

```
CPU_SIZE_T  FSFile_Wr (     FS_FILE_HANDLE   file_handle,
                     const void          *p_src,
                          CPU_SIZE_T       size,
                          RTOS_ERR        *p_err)
```

Arguments

`file_handle`

Handle to a file.

`p_src`

Pointer to source buffer.

size

Number of octets to write.

p_err

Pointer to variable that will receive the return error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_ENTRY_CLOSED
- RTOS_ERR_FILE_ERR_STATE
- RTOS_ERR_FILE_ACCESS_MODE_INVALID
- RTOS_ERR_VOL_CLOSED
- RTOS_ERR_VOL_FULL
- RTOS_ERR_VOL_CORRUPTED
- RTOS_ERR_BLK_DEV_CLOSED
- RTOS_ERR_BLK_DEV_CORRUPTED
- RTOS_ERR_IO

Returned Value

Number of octets written.

Notes / Warnings

(1) The file MUST have been opened in write or update (read/write) mode.

(2) If the file was opened in append mode, all writes are forced to the EOF.

(3) If an error occurred in the previous file access, the error indicator must be cleared (with FSFile_ErrClr()) before another access will be allowed. Otherwise, the ' RTOS_ERR_FILE_ERR_STATE ' error will be set and the function will return.

FSFile_Copy()

Description

Copy a file.

Files

fs_core_file.h/fs_core_file.c

Prototype

```
void FSFile_Copy (     FS_WRK_DIR_HANDLE   src_wrk_dir_handle,
            const CPU_CHAR         *src_path,
                  FS_WRK_DIR_HANDLE   dest_wrk_dir_handle,
            const CPU_CHAR         *dest_path,
                  CPU_BOOLEAN        excl,
                  RTOS_ERR          *p_err)
```

Arguments

src_wrk_dir_handle

Handle to source working directory.

src_path

Source file path.

dest_wrk_dir_handle

Handle to destination working directory.

`dest_path`

Destination file path.

`excl`

Indicates whether creation of new entry should be exclusive :

- `DEF_YES` , if the entry will be copied ONLY if destination entry does not exist.
- `DEF_NO` , if the entry will be copied even if destination entry does exist.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_ENTRY_OPENED`
- `RTOS_ERR_ALREADY_EXISTS`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_VOL_CLOSED`
- `RTOS_ERR_VOL_FULL`
- `RTOS_ERR_VOL_CORRUPTED`
- `RTOS_ERR_BLK_DEV_CLOSED`
- `RTOS_ERR_BLK_DEV_CORRUPTED`
- `RTOS_ERR_IO`

Returned Value

none.

Notes / Warnings

1. The source file MUST exist, otherwise an error is returned.
2. If 'excl' is `DEF_NO` , the destination entry must either not exist or be an existing file; it may not be an existing directory. If 'excl' is `DEF_YES` , the destination entry must not exist.

FSFile_Truncate()

Description

Truncate or extend a file to a specified length.

Files

`fs_core_file.h/fs_core_file.c`

Prototype

```
void  FSFile_Truncate (FS_FILE_HANDLE   file_handle,
                       FS_FILE_SIZE     size,
                       RTOS_ERR        *p_err)
```

Arguments

`file_handle`

Handle to a file.

`size`

Size of file after truncation or extension.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_ENTRY_CLOSED
- RTOS_ERR_FILE_ERR_STATE
- RTOS_ERR_FILE_ACCESS_MODE_INVALID
- RTOS_ERR_VOL_CLOSED
- RTOS_ERR_VOL_FULL
- RTOS_ERR_VOL_CORRUPTED
- RTOS_ERR_BLK_DEV_CLOSED
- RTOS_ERR_BLK_DEV_CORRUPTED
- RTOS_ERR_IO

Returned Value

none.

Notes / Warnings

(1) The file MUST be opened in write or read/write mode.

FSFile_PosSet()

Description

Set file position indicator.

Files

fs_core_file.h/fs_core_file.c

Prototype

```
void FSFile_PosSet (FS_FILE_HANDLE  file_handle,
                    FS_FILE_OFFSET  offset,
                    FS_FLAGS        origin,
                    RTOS_ERR       *p_err)
```

Arguments

file_handle

Handle to a file.

offset

Offset from the file position specified by 'origin'.

origin

Reference position for offset :

FS_FILE_ORIGIN_START Offset is from the beginning of the file.

FS_FILE_ORIGIN_CUR Offset is from current file position.

FS_FILE_ORIGIN_END Offset is from the end of the file.

p_err

Pointer to variable that will receive the return error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_ENTRY_CLOSED

- RTOS_ERR_VOL_CLOSED

If a file buffer in write mode is in use, the following error codes may be triggered as well:

- RTOS_ERR_VOL_FULL
- RTOS_ERR_VOL_CORRUPTED
- RTOS_ERR_BLK_DEV_CLOSED
- RTOS_ERR_BLK_DEV_CORRUPTED
- RTOS_ERR_IO

Returned Value

none.

Notes / Warnings

1. (a) The new file position, measured in bytes from the beginning of the file is obtained by adding 'offset' to...
    1. ...0 (the beginning of the file) if 'origin' is FS_FILE_ORIGIN_START,
    2. ...the current file position if 'origin' is FS_FILE_ORIGIN_CUR.
    3. ...the file size if 'origin' is FS_FILE_ORIGIN_END.

   (b) The end-of-file indicator is cleared.
2. If the file position indicator is set beyond the file's current data...
   (a) ...and data is later written to that point, reads from the gap will read 0.
   (b) ...the file MUST be opened in write or read/write mode.
3. Locking is needed to protect the file buffer from being emptied and/or the current position from being changed while a write operation is being performed.

FSFile_PosGet()

Description

Get file position indicator.

Files

fs_core_file.h/fs_core_file.c

Prototype

```
FS_FILE_SIZE  FSFile_PosGet (FS_FILE_HANDLE   file_handle,
                             RTOS_ERR        *p_err)
```

Arguments

file_handle

Handle to a file.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_ENTRY_CLOSED

Returned Value

The current file position measured in bytes from the beginning of the file.

Notes / Warnings

None.

FSFile_Query()

Description

Get information about a file.

Files

`fs_core_file.h/fs_core_file.c`

Prototype

```
void  FSFile_Query (FS_FILE_HANDLE   file_handle,
                    FS_ENTRY_INFO   *p_entry_info,
                    RTOS_ERR        *p_err)
```

Arguments

`file_handle`

Handle to a file.

`p_entry_info`

Pointer to structure that will receive the file information.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_ENTRY_CLOSED`
- `RTOS_ERR_VOL_CLOSED`
- `RTOS_ERR_VOL_CORRUPTED`
- `RTOS_ERR_BLK_DEV_CLOSED`
- `RTOS_ERR_BLK_DEV_CORRUPTED`
- `RTOS_ERR_IO`

Returned Value

none.

Notes / Warnings

(1) The file information structure has the following fields:

```
typedef  struct  fs_entry_attrib {
    CPU_INT32U      Wr:1;                  /* Bit indicating if entry has write access.          */
    CPU_INT32U      Rd:1;                  /* Bit indicating if entry has read access.           */
    CPU_INT32U      Hidden:1;              /* Bit indicating if entry is hidden.                 */
    CPU_INT32U      IsDir:1;               /* Bit indicating if entry is a directory or a file.  */
    CPU_INT32U      IsRootDir:1;           /* Bit indicating if entry is root directory.         */
} FS_ENTRY_ATTRIB;

typedef  struct  fs_entry_info {
    FS_ENTRY_ATTRIB  Attrib;               /* Entry attributes.                                  */
    FS_ID            NodeId;               /* Entry node id.                                     */
    FS_ID            DevId;                /* Entry device id.                                   */
    CPU_SIZE_T       Size;                 /* File size in octets.                               */
    CLK_TS_SEC       DateAccess;           /* Date of last access.                               */
    FS_LB_QTY        BlkCnt;               /* Number of blocks allocated for file.               */
    FS_LB_SIZE       BlkSize;              /* Block size in octets.                              */
    CLK_TS_SEC       DateTimeWr;           /* Date/time of last write.                           */
    CLK_TS_SEC       DateTimeCreate;       /* Date/time of creation.                             */
} FS_ENTRY_INFO;
```

FSFile_PathGet()

Description

Get absolute path to a opened file.

Files

`fs_core_file.h/fs_core_file.c`

Prototype

```
void  FSFile_PathGet (FS_FILE_HANDLE   file_handle,
                      CPU_CHAR        *p_buf,
                      CPU_SIZE_T       buf_size,
                      RTOS_ERR        *p_err)
```

Arguments

`file_handle`

Handle to a file.

`buf`

Pointer to a buffer that will receive the file path.

`buf_size`

Size of the provided buffer.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_ENTRY_CLOSED`
- `RTOS_ERR_VOL_CLOSED`
- `RTOS_ERR_VOL_CORRUPTED`
- `RTOS_ERR_BLK_DEV_CLOSED`
- `RTOS_ERR_BLK_DEV_CORRUPTED`
- `RTOS_ERR_IO`

Returned Value

none.

Notes / Warnings

None.

FSFile_VolGet()

Description

Get handle to the volume in which the file is.

Files

`fs_core_file.h/fs_core_file.c`

Prototype

```
FS_VOL_HANDLE FSFile_VolGet (FS_FILE_HANDLE file_handle)
```

Arguments

`file_handle`

Handle to a file.

Returned Value

Handle to the volume or NULL handle if the file is closed.

Notes / Warnings

None.

FSFile_BufAssign()

Description

Assign buffer to a file.

Files

`fs_core_file.h/fs_core_file.c`

Prototype

```
void FSFile_BufAssign (FS_FILE_HANDLE  file_handle,
                       void           *p_buf,
                       FS_FLAGS        mode,
                       CPU_SIZE_T      size,
                       RTOS_ERR       *p_err)
```

Arguments

`file_handle`

Handle to a file.

`p_buf`

Pointer to buffer.

`mode`

Buffer mode :

- `FS_FILE_BUF_MODE_RD` data buffered for reads.
- `FS_FILE_BUF_MODE_WR` data buffered for writes.
- `FS_FILE_BUF_MODE_RD_WR` data buffered for reads & writes.

`size`

Size of the given buffer, in octets.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_SIZE_INVALID`
- `RTOS_ERR_FILE_ACCESS_MODE_INVALID`

- RTOS_ERR_ENTRY_CLOSED
- RTOS_ERR_VOL_CLOSED

Returned Value

none.

Notes / Warnings

1. Once a buffer is assigned to a file, a new buffer may not be assigned nor may the assigned buffer be removed. To change the buffer, the file should be closed and re-opened.
2. ' size ' MUST be more than or equal to the size of one sector; it will be rounded DOWN to the nearest size of a multiple of full sectors.
3. Upon power loss, any data stored in file buffers will be lost.

FSFile_BufFlush()

Description

Flush buffer contents to file.

Files

fs_core_file.h/fs_core_file.c

Prototype

```
void  FSFile_BufFlush (FS_FILE_HANDLE   file_handle,
                      RTOS_ERR        *p_err)
```

Arguments

file_handle

Handle to a file.

p_err

Pointer to variable that will receive the return error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_ENTRY_CLOSED
- RTOS_ERR_VOL_CLOSED
- RTOS_ERR_VOL_FULL
- RTOS_ERR_VOL_CORRUPTED
- RTOS_ERR_BLK_DEV_CLOSED
- RTOS_ERR_BLK_DEV_CORRUPTED
- RTOS_ERR_IO

Returned Value

none.

Notes / Warnings

1. The flush operation is as follows:
   (a) If the most recent operation is output (write), all unwritten data is written to the file.
   (b) If the most recent operation is input (read), all buffered data is cleared.
   (c) If a read or write error occurs, the error indicator is set.
2. If an error occurred in the previous file access, the error indicator must be cleared (with FSFile_ErrClr()) before another access will be allowed.

FSFile_ErrClr()

Description

Clear file descriptor error state.

Files

`fs_core_file.h/fs_core_file.c`

Prototype

```
void  FSFile_ErrClr (FS_FILE_HANDLE   file_handle,
                     RTOS_ERR        *p_err)
```

Arguments

`file_handle`

Handle to a file.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_ENTRY_CLOSED`

Returned Value

none.

Notes / Warnings

None.

FSFile_IsEOF()

Description

Test EOF indicator on a file.

Files

`fs_core_file.h/fs_core_file.c`

Prototype

```
CPU_BOOLEAN  FSFile_IsEOF (FS_FILE_HANDLE   file_handle,
                           RTOS_ERR        *p_err)
```

Arguments

`file_handle`

Handle to a file.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_ENTRY_CLOSED`

Returned Value

- `DEF_YES` , if EOF indicator is set.
- `DEF_NO` , otherwise.

Notes / Warnings

1. `FSFile_ErrClr()` can be used to clear the EOF indicator.

FSFile_IsErr()

Description

Check whether a file descriptor is in error state.

Files

`fs_core_file.h/fs_core_file.c`

Prototype

```
CPU_BOOLEAN  FSFile_IsErr (FS_FILE_HANDLE   file_handle,
                          RTOS_ERR        *p_err)
```

Arguments

`file_handle`

Handle to a file.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_ENTRY_CLOSED`

Returned Value

- `DEF_YES` , if the files descriptor is in error state.
- `DEF_NO` , otherwise.

Notes / Warnings

None.

FSFile_Lock()

Description

Acquire task ownership of a file.

Files

`fs_core_file.h/fs_core_file.c`

Prototype

```
void  FSFile_Lock (FS_FILE_HANDLE   file_handle,
                  RTOS_ERR        *p_err)
```

Arguments

`file_handle`

Handle to a file.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_ENTRY_CLOSED`

Returned Value

none.

Notes / Warnings

1. File locks may be nested.
2. The file descriptor must be acquired so that it is not freed while holding the lock.

FSFile_TryLock()

Description

Acquire task ownership of a file (if available).

Files

`fs_core_file.h/fs_core_file.c`

Prototype

```
void  FSFile_TryLock (FS_FILE_HANDLE   file_handle,
                      RTOS_ERR        *p_err)
```

Arguments

`file_handle`

Handle to a file.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_ENTRY_CLOSED`
- `RTOS_ERR_WOULD_BLOCK`

Returned Value

none.

Notes / Warnings

1. `FSFile_TryLock()` is the non-blocking version of `FSFile_Lock()` If the lock is not available, the function returns an error.
2. File locks may be nested.

FSFile_Unlock()

Description

Release task ownership of a file.

Files

`fs_core_file.h/fs_core_file.c`

Prototype

```
void  FSFile_Unlock (FS_FILE_HANDLE  file_handle)
```

Arguments

`file_handle`

Handle to a file.

Returned Value

none.

Notes / Warnings

None.

FSPartition_Init()

Description

Initialize the partition structure on a block device (see Note #1).

Files

`fs_core_partition.h/fs_core_partition.c`

Prototype

```
void  FSPartition_Init (FS_BLK_DEV_HANDLE   blk_dev_handle,
                        FS_LB_QTY           lb_cnt,
                        RTOS_ERR           *p_err)
```

Arguments

`blk_dev_handle`

Handle to a block device.

`lb_cnt`

Size (in logical blocks) of first partition. 0 if the partition will occupy the entire device (see Note #2).

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_SIZE_INVALID`
- `RTOS_ERR_VOL_OPENED`
- `RTOS_ERR_BLK_DEV_CLOSED`
- `RTOS_ERR_BLK_DEV_CORRUPTED`
- `RTOS_ERR_IO`

Returned Value

none.

Notes / Warnings

(1) This function creates the first partition with index 1. Any subsequent partition is created using the function `FSPartition_Add()` .

(2) If 0 is passed as the number of logical blocks composing the first partition, no MBR (Master Boot Record) sector is created. If non-zero is passed, a MBR sector is created allowing the creation of four partitions.

(2) Function returns an error if a volume is open on the device. All volumes (& files) MUST be closed prior to initializing the partition structure, since it will obliterate any existing file system.

FSPartition_Add()

Description

Add a partition to a device.

Files

`fs_core_partition.h/fs_core_partition.c`

Prototype

```
FS_PARTITION_NBR  FSPartition_Add (FS_BLK_DEV_HANDLE   blk_dev_handle,
                                   FS_LB_QTY           lb_cnt,
                                   RTOS_ERR           *p_err)
```

Arguments

`blk_dev_handle`

Handle to a block device.

`lb_cnt`

Size (in logical blocks) of the partition to add.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_SIZE_INVALID`
- `RTOS_ERR_PARTITION_MAX_EXCEEDED`
- `RTOS_ERR_BLK_DEV_CLOSED`

Returned Value

The index of the created partition.

Notes / Warnings

1. The first partition on the device has index 1.
2. If there is no valid partition on the device, the function will automatically initialize a MBR (Master Boot Record) sector and create the first partition.

FSPartition_CntGet()

Description

Get number of partitions on a block device.

Files

`fs_core_partition.h/fs_core_partition.c`

Prototype

```
FS_PARTITION_NBR  FSPartition_CntGet (FS_BLK_DEV_HANDLE   blk_dev_handle,
                                      RTOS_ERR           *p_err)
```

Arguments

`blk_dev_handle`

Handle to a block device.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_BLK_DEV_CLOSED`
- `RTOS_ERR_BLK_DEV_CORRUPTED`
- `RTOS_ERR_IO`

Returned Value

Number of partitions, if NO errors.

0, otherwise.

Notes / Warnings

1. If no Master Boot Record is present on the media's sector #0, a valid formatted partition can still be present. In that case, FSPartition_CntGet() will return 1 partition as the number of partitions since no partition table exists to describe other partitions.

FSPartition_Query()

Description

Query partition information.

Files

`fs_core_partition.h/fs_core_partition.c`

Prototype

```
void  FSPartition_Query (FS_BLK_DEV_HANDLE    blk_dev_handle,
                         FS_PARTITION_NBR     partition_nbr,
                         FS_PARTITION_INFO   *p_partition_info,
                         RTOS_ERR            *p_err)
```

Arguments

`blk_dev_handle`

Handle to a block device.

`partition_nbr`

Number of the partition to query (first partition is number 1).

`p_partition_info`

Pointer to variable that will receive the partition information.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_BLK_DEV_CLOSED`

- RTOS_ERR_BLK_DEV_CORRUPTED
- RTOS_ERR_IO

Returned Value

none.

Notes / Warnings

1. The partition information structure has the following fields:

```
typedef struct fs_partition_info {
    FS_LB_NBR   StartSec;                  /* Sector number where the partition starts.          */
    FS_LB_QTY   SecCnt;                    /* Number of sectors composing the partition.          */
    CPU_INT08U  Type;                      /* Partition type found in DOS partition table entry.   */
} FS_PARTITION_INFO;
```

The field .Type may have one of the following values listed below. The following page gives additional information about some possible partition types. Also you may refer to the Table 5.3 of the book "File System Forensic Analysis, Carrier Brian, 2005" that gives various values used in the partition type field of DOS partitions.

```
#define FS_PARTITION_TYPE_FAT12_CHS                0x01u
#define FS_PARTITION_TYPE_FAT16_16_32MB            0x04u
#define FS_PARTITION_TYPE_CHS_MICROSOFT_EXT        0x05u
#define FS_PARTITION_TYPE_FAT16_CHS_32MB_2GB       0x06u
#define FS_PARTITION_TYPE_FAT32_CHS                0x0Bu
#define FS_PARTITION_TYPE_FAT32_LBA                0x0Cu
#define FS_PARTITION_TYPE_FAT16_LBA_32MB_2GB       0x0Eu
#define FS_PARTITION_TYPE_LBA_MICROSOFT_EXT        0x0Fu
#define FS_PARTITION_TYPE_HID_FAT12_CHS            0x11u
#define FS_PARTITION_TYPE_HID_FAT16_16_32MB_CHS    0x14u
#define FS_PARTITION_TYPE_HID_FAT16_CHS_32MB_2GB   0x15u
#define FS_PARTITION_TYPE_HID_CHS_FAT32            0x1Bu
#define FS_PARTITION_TYPE_HID_LBA_FAT32            0x1Cu
#define FS_PARTITION_TYPE_HID_FAT16_LBA_32MB_2GB   0x1Eu
#define FS_PARTITION_TYPE_NTFS                     0x07u
#define FS_PARTITION_TYPE_MICROSOFT_MBR            0x42u
#define FS_PARTITION_TYPE_SOLARIS_X86              0x82u
#define FS_PARTITION_TYPE_LINUX_SWAP               0x82u
#define FS_PARTITION_TYPE_LINUX                    0x83u
#define FS_PARTITION_TYPE_HIBERNATION_A            0x84u
#define FS_PARTITION_TYPE_LINUX_EXT                0x85u
#define FS_PARTITION_TYPE_NTFS_VOLSETA             0x86u
#define FS_PARTITION_TYPE_NTFS_VOLSETB             0x87u
#define FS_PARTITION_TYPE_HIBERNATION_B            0xA0u
#define FS_PARTITION_TYPE_HIBERNATION_C            0xA1u
#define FS_PARTITION_TYPE_FREE_BSD                 0xA5u
#define FS_PARTITION_TYPE_OPEN_BSD                 0xA6u
#define FS_PARTITION_TYPE_MAX_OSX                  0xA8u
#define FS_PARTITION_TYPE_NET_BSD                  0xA9u
#define FS_PARTITION_TYPE_MAC_OSX_BOOT             0xABu
#define FS_PARTITION_TYPE_BSDI                     0xB7u
#define FS_PARTITION_TYPE_BSDI_SWAP                0xB8u
#define FS_PARTITION_TYPE_EFI_GPT_DISK             0xEEu
#define FS_PARTITION_TYPE_EFI_SYS_PART             0xEFu
#define FS_PARTITION_TYPE_VMWARE_FILE_SYS          0xFBu
#define FS_PARTITION_TYPE_VMWARE_SWAP              0xFCu
```

FSVol_Open()

Description

Open a volume & mount on the file system.

Files

`fs_core_vol.h/fs_core_vol.c`

Prototype

```
FS_VOL_HANDLE  FSVol_Open (      FS_BLK_DEV_HANDLE   blk_dev_handle,
                                 FS_PARTITION_NBR    partition_nbr,
                           const CPU_CHAR           *vol_name,
                                 FS_FLAGS            open_opt,
                                 RTOS_ERR           *p_err)
```

Arguments

`blk_dev_handle`

Block device handle.

`partition_nbr`

Number of the partition to be opened. Maximum of 4 partitions allowed per media.

`vol_name`

Volume name to be assigned to the opened volume (see Note #1).

`open_opt`

Open options. Any OR'd combination among:

- `FS_VOL_OPT_DFLT` Write operations allowed, auto sync disabled, default sys-specific options.
- `FS_VOL_OPT_ACCESS_MODE_RD_ONLY` Write operations disallowed.
- `FS_VOL_OPT_AUTO_SYNC` Auto sync enabled.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_INIT`
- `RTOS_ERR_VOL_OPENED`
- `RTOS_ERR_NAME_INVALID`
- `RTOS_ERR_VOL_FMT_INVALID`
- `RTOS_ERR_PARTITION_INVALID`
- `RTOS_ERR_BLK_DEV_CORRUPTED`
- `RTOS_ERR_BLK_DEV_CLOSED`
- `RTOS_ERR_IO`

Returned Value

Handle to the opened volume.

Notes / Warnings

(1) Volume name MUST be unique across all opened volumes.

FSVol_Close()

Description

Close a volume.

Files

`fs_core_vol.h/fs_core_vol.c`

Prototype

```
void  FSVol_Close (FS_VOL_HANDLE   vol_handle,
              RTOS_ERR       *p_err)
```

Arguments

`vol_handle`

Handle to a volume.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_VOL_CLOSED`

Returned Value

none.

Notes / Warnings

1. The volume is closed even when entries are opened.
2. During the volume close operation, an IO access could happen to the media. In the case of a removable media, if the close operation is performed after the media removal, the IO access will return an IO error. This error condition is expected in that case. Hence the IO error is considered as NO error. In the case of a fixed media or if the device close operation is done before a removable media is disconnected, an IO error means that something important did not work. This condition will be detected by the application when re-opening the volume.

FSVol_Sync()

Description

Ensure that all pending write operations reach the underlying physical media.

Files

`fs_core_vol.h/fs_core_vol.c`

Prototype

```
void  FSVol_Sync (FS_VOL_HANDLE   vol_handle,
             RTOS_ERR       *p_err)
```

Arguments

`vol_handle`

Handle to a volume.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_VOL_CLOSED`
- `RTOS_ERR_BLK_DEV_CLOSED`

- RTOS_ERR_BLK_DEV_CORRUPTED
- RTOS_ERR_IO

Returned Value

none.

Notes / Warnings

None.

FSVol_Query()

Description

Obtain information about a volume.

Files

fs_core_vol.h/fs_core_vol.c

Prototype

```
void  FSVol_Query (FS_VOL_HANDLE   vol_handle,
                   FS_VOL_INFO    *p_vol_info,
                   RTOS_ERR       *p_err)
```

Arguments

vol_handle

Volume handle.

p_vol_info

Pointer to structure that will receive volume information.

p_err

Pointer to variable that will receive the return error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_VOL_CLOSED
- RTOS_ERR_VOL_CORRUPTED
- RTOS_ERR_BLK_DEV_CLOSED
- RTOS_ERR_BLK_DEV_CORRUPTED
- RTOS_ERR_IO

Returned Value

none.

Notes / Warnings

(1) The volume information structure has the following fields:

```
typedef  struct  fs_vol_info {
    FS_LB_QTY  BadSecCnt;                  /* Number of bad sectors.          */
    FS_LB_QTY  FreeSecCnt;                 /* Number of free sectors.         */
    FS_LB_QTY  UsedSecCnt;                 /* Number of used sectors.         */
    FS_LB_QTY  TotSecCnt;                  /* Total number of sectors.        */
} FS_VOL_INFO;
```

FSVol_PartitionNbrGet()

Description

Get the partition number where the volume resides.

Files

`fs_core_vol.h/fs_core_vol.c`

Prototype

```
FS_PARTITION_NBR  FSVol_PartitionNbrGet (FS_VOL_HANDLE   vol_handle,
                                         RTOS_ERR       *p_err)
```

Arguments

`vol_handle`

Handle to a volume.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_VOL_CLOSED`

Returned Value

Partition number.

Notes / Warnings

None.

FSVol_LabelSet()

Description

Set volume label.

Files

`fs_core_vol.h/fs_core_vol.c`

Prototype

```
void  FSVol_LabelSet (      FS_VOL_HANDLE   vol_handle,
                      const CPU_CHAR       *p_label,
                            RTOS_ERR       *p_err)
```

Arguments

`vol_handle`

Handle to a volume.

`p_label`

Pointer to volume label.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_NAME_INVALID
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_VOL_CLOSED
- RTOS_ERR_VOL_CORRUPTED
- RTOS_ERR_BLK_DEV_CLOSED
- RTOS_ERR_BLK_DEV_CORRUPTED
- RTOS_ERR_IO

Returned Value

none.

Notes / Warnings

None.

FSVol_LabelGet()

Description

Get volume label.

Files

`fs_core_vol.h/fs_core_vol.c`

Prototype

```
void  FSVol_LabelGet (FS_VOL_HANDLE   vol_handle,
                      CPU_CHAR       *p_label,
                      CPU_SIZE_T      label_size,
                      RTOS_ERR       *p_err)
```

Arguments

`vol_handle`

Handle to a volume.

`p_label`

Pointer to a buffer that will receive volume label.

`label_size`

Size of the give string buffer (see Note #1).

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_VOL_CLOSED
- RTOS_ERR_VOL_CORRUPTED
- RTOS_ERR_BLK_DEV_CLOSED
- RTOS_ERR_BLK_DEV_CORRUPTED
- RTOS_ERR_IO

Returned Value

none.

Notes / Warnings

(1) 'label_size' is the maximum length string that can be stored in the buffer; it does NOT include the final NULL character. The buffer MUST be of at least 'label_size' + 1 characters.

FSVol_NameGet()

Description

Get a volume's name.

Files

fs_core_vol.h/fs_core_vol.c

Prototype

```
void  FSVol_NameGet (FS_VOL_HANDLE   vol_handle,
                     CPU_CHAR       *p_buf,
                     CPU_SIZE_T      buf_size,
                     RTOS_ERR       *p_err)
```

Arguments

vol_handle

Handle to a volume.

p_buf

Pointer to a buffer that will receive the volume name.

buf_size

Size of the provided buffer.

p_err

Pointer to variable that will receive the return error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_VOL_CLOSED
- RTOS_ERR_WOULD_OVF

Returned Value

none.

Notes / Warnings

None.

FSVol_Get()

Description

Get a volume by name.

Files

fs_core_vol.h/fs_core_vol.c

Prototype

```
FS_VOL_HANDLE FSVol_Get (const CPU_CHAR *p_vol_name)
```

Arguments

`p_vol_name`

Pointer to name of the volume.

Returned Value

Handle to the found volume

NULL handle if no volume is found.

Notes / Warnings

None.

FSVol_BlkDevGet()

Description

Get parent block device handle.

Files

`fs_core_vol.h/fs_core_vol.c`

Prototype

```
FS_BLK_DEV_HANDLE FSVol_BlkDevGet (FS_VOL_HANDLE vol_handle)
```

Arguments

`vol_handle`

Handle to a volume.

Returned Value

Handle to the parent block device

NULL handle if the volume closed.

Notes / Warnings

None.

FSWrkDir_Open()

Description

Open a working directory.

Files

`fs_core_working_dir.h/fs_core_working_dir.c`

Prototype

```
FS_WRK_DIR_HANDLE FSWrkDir_Open ( FS_WRK_DIR_HANDLE wrk_dir_handle,
                                  const CPU_CHAR    *p_path,
                                  RTOS_ERR          *p_err)
```

Arguments

`wrk_dir_handle`

Handle to the current working directory.

`p_path`

Pointer to a directory path relative to the given working directory.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_INIT`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_INVALID_TYPE`
- `RTOS_ERR_ENTRY_MAX_DEPTH_EXCEEDED`
- `RTOS_ERR_WRK_DIR_CLOSED`
- `RTOS_ERR_VOL_CLOSED`
- `RTOS_ERR_VOL_CORRUPTED`
- `RTOS_ERR_BLK_DEV_CLOSED`
- `RTOS_ERR_BLK_DEV_CORRUPTED`
- `RTOS_ERR_IO`

Returned Value

Handle to the opened working directory.

Notes / Warnings

None.

FSWrkDir_Close()

Description

Close a working directory.

Files

`fs_core_working_dir.h/fs_core_working_dir.c`

Prototype

```
void  FSWrkDir_Close (FS_WRK_DIR_HANDLE   wrk_dir_handle,
                      RTOS_ERR           *p_err)
```

Arguments

`working_dir_handle`

Handle to the working directory to be closed.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_WRK_DIR_CLOSED`

Returned Value

none.

Notes / Warnings

None.

FSWrkDir_TaskBind()

Description

Bind a working directory to the current task.

Files

`fs_core_working_dir.h/fs_core_working_dir.c`

Prototype

```
void  FSWrkDir_TaskBind (      FS_WRK_DIR_HANDLE   wrk_dir_handle,
                         const CPU_CHAR           *p_path,
                               RTOS_ERR           *p_err)
```

Arguments

`wrk_dir_handle`

Handle to a working directory.

`p_path`

Pointer to a directory path relative to the given working directory.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_INVALID_TYPE`
- `RTOS_ERR_ENTRY_MAX_DEPTH_EXCEEDED`
- `RTOS_ERR_WRK_DIR_CLOSED`
- `RTOS_ERR_VOL_CLOSED`
- `RTOS_ERR_VOL_CORRUPTED`
- `RTOS_ERR_BLK_DEV_CLOSED`
- `RTOS_ERR_BLK_DEV_CORRUPTED`
- `RTOS_ERR_IO`

Returned Value

none.

Notes / Warnings

None.

FSWrkDir_TaskUnbind()

Description

Unbind a working directory from the current task.

Files

fs_core_working_dir.h/fs_core_working_dir.c

Prototype

```
void FSWrkDir_TaskUnbind (RTOS_ERR *p_err)
```

Arguments

p_err

Pointer to variable that will receive the return error code(s) from this function:

- RTOS_ERR_NONE

Returned Value

none.

Notes / Warnings

None.

FSWrkDir_Get()

Description

Get working directory bound to the current task.

Files

fs_core_working_dir.h/fs_core_working_dir.c

Prototype

```
FS_WRK_DIR_HANDLE FSWrkDir_Get (void)
```

Arguments

None.

Returned Value

Working directory handle.

Notes / Warnings

None.

FSWrkDir_VolGet()

Description

Get volume handle associated to given working directory.

Files

fs_core_working_dir.h/fs_core_working_dir.c

Prototype

```
FS_VOL_HANDLE FSWrkDir_VolGet (FS_WRK_DIR_HANDLE wrk_dir_handle)
```

Arguments

wrk_dir_handle

Handle to a working directory.

Returned Value

Parent volume handle or NULL handle if working directory is closed.

Notes / Warnings

None.

FSWrkDir_PathGet()

Description

Get the absolute path of a working directory.

Files

`fs_core_working_dir.h/fs_core_working_dir.c`

Prototype

```
CPU_SIZE_T  FSWrkDir_PathGet (FS_WRK_DIR_HANDLE   wrk_dir_handle,
                              CPU_CHAR           *p_buf,
                              CPU_SIZE_T          buf_size,
                              RTOS_ERR           *p_err)
```

Arguments

`wrk_dir_handle`

Handle to a working directory.

`p_buf`

Pointer to a buffer that will receive the path.

`buf_size`

Size of the provided buffer.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_ENTRY_CLOSED`
- `RTOS_ERR_VOL_CLOSED`
- `RTOS_ERR_VOL_CORRUPTED`
- `RTOS_ERR_BLK_DEV_CLOSED`
- `RTOS_ERR_BLK_DEV_CORRUPTED`
- `RTOS_ERR_IO`

Returned Value

The total number of characters in the path.

Notes / Warnings

None.

File System Storage API

File System Storage API

- FSStorage_ConfigureMediaConnCallback()
- FSStorage_ConfigureMediaPollTaskStk()
- FSStorage_ConfigureMemSeg()
- FSStorage_ConfigureMaxSCSILogicalUnitCnt()
- FSStorage_Init()
- FSStorage_PollTaskPeriodSet()
- FSStorage_PollTaskPrioSet()
- FSBlkDev_Open()
- FSBlkDev_Close()
- FSBlkDev_Rd()
- FSBlkDev_Wr()
- FSBlkDev_Sync()
- FSBlkDev_Trim()
- FSBlkDev_AlignReqGet()
- FSBlkDev_LbCntGet()
- FSBlkDev_LbSizeGet()
- FSBlkDev_LbSizeLog2Get()
- FSBlkDev_Get()
- FSBlkDev_NameGet()
- FSBlkDev_MediaGet()
- FSMedia_Get()
- FSMedia_FirstGet()
- FSMedia_LowFmt()
- FSMedia_IsConn()
- FSMedia_NameGet()
- FSMedia_MaxCntGet()
- FSMedia_TypeGet()
- FSMedia_TypeStrGet()
- FSMedia_AlignReqGet()
- FS_NAND_Open()
- FS_NAND_Close()
- FS_NAND_BlkErase()
- FS_NAND_ChipErase()
- FS_NAND_Dump()
- FS_NAND_FTL_ConfigureLowParams()
- FS_NOR_Open()
- FS_NOR_Close()
- FS_NOR_Rd()
- FS_NOR_Wr()
- FS_NOR_BlkErase()
- FS_NOR_ChipErase()
- FS_NOR_XIP_Cfg()
- FS_NOR_Get()
- FS_NOR_BlkCntGet()
- FS_NOR_BlkSizeLog2Get()
- FS_NOR_FTL_ConfigureLowParams()
- FS_NOR_FTL_LowCompact()

- FS_NOR_FTL_LowDefrag()
- FS_RAM_Disk_Add()
- FS_SCSI_Open()
- FS_SCSI_Close()
- FS_SCSI_LU_InfoGet()
- FS_SD_Open()
- FS_SD_Close()
- FS_SD_CID_Rd()
- FS_SD_CSD_Rd()
- FS_SD_InfoGet()
- FS_NAND_HW_INFO_REG()
- FS_NOR_QUAD_SPI_HW_INFO_REG()
- FS_SD_CARD_HW_INFO_REG()
- FS_SD_SPI_HW_INFO_REG()

FSStorage_ConfigureMediaConnCallback()

Description

Set the media connection and disconnection callbacks.

Files

`fs_storage.h/fs_storage.c`

Prototype

```
void  FSStorage_ConfigureMediaConnCallback (void   (*on_conn)     (FS_MEDIA_HANDLE  media_handle),
                                            void   (*on_disconn)  (FS_MEDIA_HANDLE  media_handle))
```

Arguments

`on_conn`

Callback that will be called whenever a removable media (SD or SCSI) is connected.

`on_disconn`

Callback that will be called whenever a removable media (SD or SCSI) is disconnected.

Returned Value

none.

Notes / Warnings

1. If the connection callback is NOT provided, the poll task will not be started.

FSStorage_ConfigureMediaPollTaskStk()

Description

Set the media polling task stack start address and stack size.

Files

`fs_storage.h/fs_storage.c`

Prototype

```
void  FSStorage_ConfigureMediaPollTaskStk (void        *p_stk,
                                           CPU_INT32U   stk_size_elements)
```

Arguments

`p_stk`

Pointer to the start of the stack memory region.

`stk_size_elements`

Size of the stack in stack size elements.

Returned Value

none.

Notes / Warnings

None.

FSStorage_ConfigureMemSeg()

Description

Configure memory segment for the storage module.

Files

`fs_storage.h/fs_storage.c`

Prototype

```
void FSStorage_ConfigureMemSeg (MEM_SEG *p_seg)
```

Arguments

`p_seg`

Pointer to a user-defined memory segment.

Returned Value

none.

Notes / Warnings

None.

FSStorage_ConfigureMaxSCSILogicalUnitCnt()

Description

Configure the (fixed) quantity of SCSI logical units available.

Files

`fs_storage.h/fs_storage.c`

Prototype

```
void FSStorage_ConfigureMaxSCSILogicalUnitCnt (CPU_SIZE_T max_cnt)
```

Arguments

`max_cnt`

Maximum SCSI logical unit available quantity.

Returned Value

none.

Notes / Warnings

None.

FSStorage_Init()

Description

Initialize the file system storage sub-module.

Files

`fs_storage.h/fs_storage.c`

Prototype

```
void  FSStorage_Init (RTOS_ERR  *p_err)
```

Arguments

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_SEG_OVF`

Returned Value

none.

Notes / Warnings

None.

FSStorage_PollTaskPeriodSet()

Description

Set the media poll task period.

Files

`fs_storage.h/fs_storage.c`

Prototype

```
void  FSStorage_PollTaskPeriodSet (CPU_INT32U   period_ms,
                                   RTOS_ERR    *p_err)
```

Arguments

`period_ms`

Interval of time (in milliseconds) between two polling.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

none.

Notes / Warnings

None.

FSStorage_PollTaskPrioSet()

Description

Set the priority of the media poll task.

Files

`fs_storage.h/fs_storage.c`

Prototype

```
void  FSStorage_PollTaskPrioSet (RTOS_TASK_PRIO   prio,
                                 RTOS_ERR        *p_err)
```

Arguments

`prio`

New priority for the media poll task.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

none.

Notes / Warnings

None.

FSBlkDev_Open()

Description

Open a block device.

Files

`fs_blk_dev.h/fs_blk_dev.c`

Prototype

```
FS_BLK_DEV_HANDLE  FSBlkDev_Open (FS_MEDIA_HANDLE   media_handle,
                                  RTOS_ERR         *p_err)
```

Arguments

`media_handle`

Handle to a media.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`

- RTOS_ERR_NOT_INIT
- RTOS_ERR_BLK_DEV_OPENED
- RTOS_ERR_BLK_DEV_LOW_FMT_INCOMPATIBLE
- RTOS_ERR_BLK_DEV_LOW_FMT_INVALID
- RTOS_ERR_IO

Returned Value

Handle to the opened block device.

Notes / Warnings

None.

FSBlkDev_Close()

Description

Close a block device.

Files

fs_blk_dev.h/fs_blk_dev.c

Prototype

```
void  FSBlkDev_Close (FS_BLK_DEV_HANDLE   blk_dev_handle,
                      RTOS_ERR           *p_err)
```

Arguments

blk_dev_handle

Handle to a block device.

p_err

Pointer to variable that will receive the return error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_BLK_DEV_CLOSED

Returned Value

none.

Notes / Warnings

None.

FSBlkDev_Rd()

Description

Read blocks from a block device.

Files

fs_blk_dev.h/fs_blk_dev.c

Prototype

```
void  FSBlkDev_Rd (FS_BLK_DEV_HANDLE   blk_dev_handle,
                   void               *p_dest,
                   FS_LB_NBR           start_lb_nbr,
                   FS_LB_QTY           lb_cnt,
                   RTOS_ERR           *p_err)
```

Arguments

`blk_dev_handle`

Handle to a block device.

`p_dest`

Pointer to a destination buffer.

`start_lb_nbr`

Logical block number of the block to start reading from.

`lb_cnt`

Number of logical blocks to read.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_BLK_DEV_CLOSED`
- `RTOS_ERR_BLK_DEV_CORRUPTED`
- `RTOS_ERR_IO`

Returned Value

none.

Notes / Warnings

None.

FSBlkDev_Wr()

Description

Write blocks to a block device.

Files

`fs_blk_dev.h/fs_blk_dev.c`

Prototype

```
void  FSBlkDev_Wr (FS_BLK_DEV_HANDLE   blk_dev_handle,
                   void               *p_src,
                   FS_LB_NBR           start_lb_nbr,
                   FS_LB_QTY           lb_cnt,
                   RTOS_ERR           *p_err)
```

Arguments

`blk_dev_handle`

Handle to a block device.

`p_src`

Pointer to a source buffer.

`start_lb_nbr`

Logical block number of the block to start writing to.

`lb_cnt`

Number of logical blocks to write.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_BLK_DEV_CLOSED`
- `RTOS_ERR_BLK_DEV_CORRUPTED`
- `RTOS_ERR_IO`

Returned Value

none.

Notes / Warnings

None.

FSBlkDev_Sync()

Description

Sync a block device.

Files

`fs_blk_dev.h/fs_blk_dev.c`

Prototype

```
void  FSBlkDev_Sync (FS_BLK_DEV_HANDLE   blk_dev_handle,
                     RTOS_ERR           *p_err)
```

Arguments

`blk_dev_handle`

Handle to a block device.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_BLK_DEV_CLOSED`
- `RTOS_ERR_BLK_DEV_CORRUPTED`
- `RTOS_ERR_IO`

Returned Value

none.

Notes / Warnings

None.

FSBlkDev_Trim()

Description

Trim a block device.

Files

`fs_blk_dev.h/fs_blk_dev.c`

Prototype

```
void  FSBlkDev_Trim (FS_BLK_DEV_HANDLE   blk_dev_handle,
                     FS_LB_NBR           start_lb_nbr,
                     FS_LB_QTY           lb_cnt,
                     RTOS_ERR           *p_err)
```

Arguments

`blk_dev_handle`

Handle to a block device.

`start_lb_nbr`

Logical block number of the block to start trimming from.

`lb_cnt`

Number of logical blocks to trim.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_BLK_DEV_CLOSED`
- `RTOS_ERR_BLK_DEV_CORRUPTED`
- `RTOS_ERR_IO`

Returned Value

none.

Notes / Warnings

None.

FSBlkDev_AlignReqGet()

Description

Get the alignment requirement.

Files

`fs_blk_dev.h/fs_blk_dev.c`

Prototype

```
CPU_SIZE_T  FSBlkDev_AlignReqGet (FS_BLK_DEV_HANDLE   blk_dev_handle,
                                  RTOS_ERR           *p_err)
```

Arguments

`blk_dev_handle`

Handle to a block device.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_BLK_DEV_CLOSED`

Returned Value

- Alignment requirement in bytes, if NO error(s).
- 0, otherwise.

Notes / Warnings

None.

FSBlkDev_LbCntGet()

Description

Get the number of logical blocks on a block device.

Files

`fs_blk_dev.h/fs_blk_dev.c`

Prototype

```
FS_LB_QTY  FSBlkDev_LbCntGet (FS_BLK_DEV_HANDLE   blk_dev_handle,
                              RTOS_ERR            *p_err)
```

Arguments

`blk_dev_handle`

Handle to a block device.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_BLK_DEV_CLOSED`

Returned Value

- Number of logical blocks, if NO error(s).
- 0, otherwise.

Notes / Warnings

None.

FSBlkDev_LbSizeGet()

Description

Get logical block size.

Files

fs_blk_dev.h/fs_blk_dev.c

Prototype

```
FS_LB_SIZE  FSBlkDev_LbSizeGet (FS_BLK_DEV_HANDLE   blk_dev_handle,
                                RTOS_ERR           *p_err)
```

Arguments

`blk_dev_handle`

Handle to a block device.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_BLK_DEV_CLOSED`

Returned Value

Logical block size.

Notes / Warnings

None.

FSBlkDev_LbSizeLog2Get()

Description

Get base-2 logarithm of the logical block size.

Files

fs_blk_dev.h/fs_blk_dev.c

Prototype

```
CPU_INT08U  FSBlkDev_LbSizeLog2Get (FS_BLK_DEV_HANDLE   blk_dev_handle,
                                    RTOS_ERR           *p_err)
```

Arguments

`blk_dev_handle`

Handle to a block device.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_BLK_DEV_CLOSED`

Returned Value

Base-2 logarithm of the logical block size.

Notes / Warnings

None.

FSBlkDev_Get()

Description

Get a block device handle from the media handle.

Files

`fs_blk_dev.h/fs_blk_dev.c`

Prototype

```
FS_BLK_DEV_HANDLE  FSBlkDev_Get (FS_MEDIA_HANDLE  media_handle)
```

Arguments

`media_handle`

Handle to a media.

Returned Value

Handle to the block device.

Notes / Warnings

(1) If a block device has no been open, the returned handle is NULL. You may check for NULL handle using the macro FS_BLK_DEV_HANDLE_IS_NULL().

FSBlkDev_NameGet()

Description

Get block device name.

Files

`fs_blk_dev.h/fs_blk_dev.c`

Prototype

```
void  FSBlkDev_NameGet (FS_BLK_DEV_HANDLE   blk_dev_handle,
                        CPU_CHAR           *p_buf,
                        CPU_SIZE_T          buf_size,
                        RTOS_ERR           *p_err)
```

Arguments

`blk_dev_handle`

Handle to a block device.

`p_buf`

Pointer to a buffer that will receive the block device name.

`buf_size`

Size of the given buffer.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`

- `RTOS_ERR_BLK_DEV_CLOSED`

Returned Value

none.

Notes / Warnings

None.

FSBlkDev_MediaGet()

Description

Get parent media.

Files

`fs_blk_dev.h/fs_blk_dev.c`

Prototype

```
FS_MEDIA_HANDLE FSBlkDev_MediaGet (FS_BLK_DEV_HANDLE blk_dev_handle)
```

Arguments

`blk_dev_handle`

Handle to a block device.

Returned Value

Handle to the parent media or NULL handle if the block device handle is invalid.

Notes / Warnings

None.

FSMedia_Get()

Description

Get a handle to a media by name.

Files

`fs_media.h/fs_media.c`

Prototype

```
FS_MEDIA_HANDLE FSMedia_Get (const CPU_CHAR *p_name)
```

Arguments

`p_name`

Pointer to media name.

Returned Value

Handle to a media

NULL handle if no media is found with the given name.

Notes / Warnings

1. You may check for NULL handle using the macro `FS_MEDIA_HANDLE_IS_NULL()` .

FSMedia_FirstGet()

Description

Get the first added media.

Files

`fs_media.h/fs_media.c`

Prototype

```
FS_MEDIA_HANDLE  FSMedia_FirstGet (void)
```

Arguments

Returned Value

Handle to a media.

Notes / Warnings

None.

FSMedia_LowFmt()

Description

Low-level format a media so that it can be opened as a block device.

Files

`fs_media.h/fs_media.c`

Prototype

```
void  FSMedia_LowFmt (FS_MEDIA_HANDLE   media_handle,
                      RTOS_ERR         *p_err)
```

Arguments

`media_handle`

Handle to a media.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_INIT`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_BLK_DEV_OPENED`
- `RTOS_ERR_IO`

Returned Value

Notes / Warnings

1. This function will write flash translation layer metadata to NOR and NAND flash memories. For other types of media, this function does nothing.

FSMedia_IsConn()

Description

Check whether a media is connected.

Files

`fs_media.h/fs_media.c`

Prototype

```
CPU_BOOLEAN  FSMedia_IsConn (FS_MEDIA_HANDLE  media_handle)
```

Arguments

`media_handle`

Handle to a media.

Returned Value

- `DEF_YES` , if the media is connected.
- `DEF_NO` , otherwise.

Notes / Warnings

None.

FSMedia_NameGet()

Description

Get a media name.

Files

`fs_media.h/fs_media.c`

Prototype

```
void  FSMedia_NameGet (FS_MEDIA_HANDLE   media_handle,
                       CPU_CHAR         *p_buf,
                       CPU_SIZE_T        buf_size,
                       RTOS_ERR         *p_err)
```

Arguments

`media_handle`

Handle to a media.

`p_buf`

Pointer to a buffer that will receive the media name.

`buf_size`

Size of the given buffer.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_IO`

Returned Value

none.

Notes / Warnings

None.

FSMedia_MaxCntGet()

Description

Get the maximum number of co-existing media.

Files

`fs_media.h/fs_media.c`

Prototype

```
CPU_SIZE_T  FSMedia_MaxCntGet (void)
```

Arguments

Returned Value

Maximum number of media.

Notes / Warnings

None.

FSMedia_TypeGet()

Description

Get the type of a media.

Files

`fs_media.h/fs_media.c`

Prototype

```
FS_MEDIA_TYPE  FSMedia_TypeGet (FS_MEDIA_HANDLE  media_handle,
                               RTOS_ERR        *p_err)
```

Arguments

`media_handle`

Handle to a media.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_IO`

Returned Value

Type of the media. The possible types can be:

- `FS_MEDIA_TYPE_RAM_DISK`

- `FS_MEDIA_TYPE_NOR`
- `FS_MEDIA_TYPE_NAND`
- `FS_MEDIA_TYPE_SD_SPI`
- `FS_MEDIA_TYPE_SD_CARD`
- `FS_MEDIA_TYPE_SCSI`

Notes / Warnings

None.

FSMedia_TypeStrGet()

Description

Get a string corresponding to the given media type.

Files

`fs_media.h/fs_media.c`

Prototype

```
const CPU_CHAR *FSMedia_TypeStrGet (FS_MEDIA_TYPE type)
```

Arguments

`type`

Media type.

- `FS_MEDIA_TYPE_RAM_DISK`
- `FS_MEDIA_TYPE_NOR`
- `FS_MEDIA_TYPE_NAND`
- `FS_MEDIA_TYPE_SD_SPI`
- `FS_MEDIA_TYPE_SD_CARD`
- `FS_MEDIA_TYPE_SCSI`

Returned Value

Pointer to a string literal corresponding to the given media type.

Notes / Warnings

None.

FSMedia_AlignReqGet()

Description

Get the read/write buffer alignment requirement for a media.

Files

`fs_media.h/fs_media.c`

Prototype

```
CPU_SIZE_T  FSMedia_AlignReqGet (FS_MEDIA_HANDLE   media_handle,
                                 RTOS_ERR         *p_err)
```

Arguments

media_handle

Handle to a media.

p_err

Pointer to variable that will receive the return error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_IO

Returned Value

Alignment requirement.

Notes / Warnings

None.

FS_NAND_Open()

Description

Open a NAND media.

Files

fs_nand.h/fs_nand.c

Prototype

```
FS_NAND_HANDLE  FS_NAND_Open (FS_MEDIA_HANDLE   media_handle,
                              RTOS_ERR         *p_err)
```

Arguments

media_handle

Handle to a media.

p_err

Pointer to variable that will receive the return error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_NOT_INIT
- RTOS_ERR_IO
- RTOS_ERR_NOT_FOUND

Returned Value

Handle to the opened NAND.

Notes / Warnings

None.

FS_NAND_Close()

Description

Close a NAND media.

Files

fs_nand.h/fs_nand.c

Prototype

```
void  FS_NAND_Close (FS_NAND_HANDLE   nand_handle,
                     RTOS_ERR        *p_err)
```

Arguments

`nand_handle`

Handle to a NAND.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

none.

Notes / Warnings

None.

FS_NAND_BlkErase()

Description

Erase a physical block.

Files

`fs_nand.h/fs_nand.c`

Prototype

```
void   FS_NAND_BlkErase (FS_NAND_HANDLE    nand_handle,
                         FS_NAND_BLK_QTY   blk_ix,
                         RTOS_ERR         *p_err)
```

Arguments

`nand_handle`

Handle to a NAND.

`blk_ix`

Index of the block to be erased.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_IO`

Returned Value

none.

Notes / Warnings

None.

FS_NAND_ChipErase()

Description

Erase an entire NAND chip. Erase counts for each block will also be erased, affecting wear leveling mechanism.

Files

`fs_nand.h/fs_nand.c`

Prototype

```
void  FS_NAND_ChipErase (FS_NAND_HANDLE   nand_handle,
                         RTOS_ERR        *p_err)
```

Arguments

`nand_handle`

Handle to a NAND.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_IO`

Returned Value

none.

Notes / Warnings

1. WARNING: this function will reset erase counts, affecting wear leveling algorithms. Use this function only in very specific cases when a low-level format does not suffice.

FS_NAND_Dump()

Description

Dumps raw NAND contents in multiple data chunks through a user-supplied callback function.

Files

`fs_nand.h/fs_nand.c`

Prototype

```
void  FS_NAND_Dump (FS_NAND_HANDLE      nand_handle,
                    FS_NAND_DUMP_FUNC   dump_func,
                    void               *p_buf,
                    FS_NAND_BLK_QTY     first_blk_ix,
                    FS_NAND_BLK_QTY     blk_cnt,
                    RTOS_ERR           *p_err)
```

Arguments

`nand_handle`

Handle to a NAND.

`dump_fnct`

Callback that will return the read raw NAND data in chunks.

`p_buf`

Pointer to a buffer long enough to contain a NAND page.

`first_blk_ix`

Index of the block to start dumping at.

`blk_cnt`

Number of blocks to dump.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_IO`

Returned Value

none.

Notes / Warnings

None.

FS_NAND_FTL_ConfigureLowParams()

Description

Configure NAND FTL parameters.

Files

`fs_nand_ftl.h/fs_nand_ftl.c`

Prototype

```
void FS_NAND_FTL_ConfigureLowParams (    FS_MEDIA_HANDLE   media_handle,
                                const FS_NAND_FTL_CFG  *p_nand_cfg,
                                    RTOS_ERR         *p_err)
```

Arguments

`media_handle`

Handle to a media.

`p_nand_cfg`

Pointer to a NAND FTL configuration structure.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_INVALID_CFG`

Returned Value

none.

Notes / Warnings

1. `FS_NAND_FTL_ConfigureLowParams()` must be called prior to `FSBlkDev_Open()` or `FSMedia_LowFmt()`. It will configure a NAND FTL instance using the user specific parameters and allocate all needed resources by the FTL instance. Then `FSBlkDev_Open()` or `FSMedia_LowFmt()` can be called and the NAND FTL instance can be obtained from the internal NAND FTL list. Thus avoiding to reallocate resources once more time for the same NAND instance.
2. The NAND FTL configuration structure has the following fields:

```
typedef struct fs_nand_ftl_cfg {
    FS_LB_SIZE        SecSize;                /* Sec size in octets.                    */
    FS_NAND_BLK_QTY   BlkCnt;                 /* Total blk cnt.                         */
    FS_NAND_BLK_QTY   BlkIxFirst;             /* Ix of first blk to be used.            */
    FS_NAND_UB_QTY    UB_CntMax;              /* Max nbr of Update Blocks.              */
    CPU_INT08U        RUB_MaxAssoc;           /* Max assoc of Random Update Blocks.     */
    CPU_INT08U        AvailBlkTblEntryCntMax; /* Nbr of entries in avail blk tbl.       */
} FS_NAND_FTL_CFG;
```

FS_NOR_Open()

Description

Open a NOR for raw access.

Files

`fs_nor.h/fs_nor.c`

Prototype

```
FS_NOR_HANDLE  FS_NOR_Open (FS_MEDIA_HANDLE  media_handle,
                            RTOS_ERR        *p_err)
```

Arguments

`media_handle`

Handle to a media.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_INIT`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_MEDIA_CLOSED`
- `RTOS_ERR_IO`

Returned Value

Handle to the opened NOR.

Notes / Warnings

None.

FS_NOR_Close()

Description

Close a NOR.

Files

`fs_nor.h/fs_nor.c`

Prototype

```
void  FS_NOR_Close (FS_NOR_HANDLE   nor_handle,
                    RTOS_ERR       *p_err)
```

Arguments

`nor_handle`

Handle to a NOR.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_IO`

Returned Value

none.

Notes / Warnings

None.

FS_NOR_Rd()

Description

Read from a NOR.

Files

`fs_nor.h/fs_nor.c`

Prototype

```
void  FS_NOR_Rd (FS_NOR_HANDLE   nor_handle,
                 void           *p_dest,
                 CPU_INT32U      start_addr,
                 CPU_INT32U      cnt,
                 RTOS_ERR       *p_err)
```

Arguments

`nor_handle`

Handle to a NOR media instance.

`p_dest`

Pointer to destination buffer.

`start_addr`

Start address of read (see Note #1).

`cnt`

Number of octets to read.

`p_err`

Pointer to variable that will receive return the error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_IO`

Returned Value

none.

Notes / Warnings

1. Start address represents the flash device physical address from which the read must start. For instance if the flash device has a size of 32Mb, the physical address range is 0x00000000-0x003FFFFF. In this example, the start address must be in this range. Note that the start address is relative to the absolute start of the flash device. There is no offset to account for.

FS_NOR_Wr()

Description

Write to a NOR.

Files

`fs_nor.h/fs_nor.c`

Prototype

```
void  FS_NOR_Wr (FS_NOR_HANDLE   nor_handle,
                 void           *p_src,
                 CPU_INT32U      start_addr,
                 CPU_INT32U      cnt,
                 RTOS_ERR       *p_err)
```

Arguments

`nor_handle`

Handle to a NOR.

`p_src`

Pointer to source buffer.

`start_addr`

Start address of write (see Note #1).

`cnt`

Number of octets to write.

`p_err`

Pointer to variable that will receive return the error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_IO`

Returned Value

none.

Notes / Warnings

1. Start address represents the flash device physical address from which the write must start. For instance if the flash device has a size of 32Mb, the physical address range is 0x00000000-0x003FFFFF. In this example, the start address must be in this range. Note that the start address is relative to the absolute start of the flash device. There is no offset to account for.
2. Care should be taken if this function is used while a file system exists on the device, or if the device is low-level formatted. The octet location(s) modified are NOT verified as being outside any existing file system or low-level format information.
3. During a program operation, only '1' bits can be changed; a '0' bit cannot be changed to a '1'. The application MUST know that the octets being programmed have not already been programmed.
4. The success of high-level write operations can be checked by volume write check functionality. The check here covers write operations for low-level data (e.g., block & sector headers).

FS_NOR_BlkErase()

Description

Erase a block on a NOR.

Files

`fs_nor.h/fs_nor.c`

Prototype

```
void  FS_NOR_BlkErase (FS_NOR_HANDLE   nor_handle,
                       CPU_INT32U      blk_ix,
                       RTOS_ERR       *p_err)
```

Arguments

`nor_handle`

Handle to a NOR.

`blk_ix`

Index of the block to be erased.

`p_err`

Pointer to variable that will receive return the error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_IO`

Returned Value

none.

Notes / Warnings

1. Care should be taken if this function is used while a file system exists on the device, or if the device is low-level formatted. The erased block is NOT verified as being outside any existing file system or low-level format information.

FS_NOR_ChipErase()

Description

Erase an entire NOR.

Files

`fs_nor.h/fs_nor.c`

Prototype

```
void  FS_NOR_ChipErase (FS_NOR_HANDLE   nor_handle,
                        RTOS_ERR       *p_err)
```

Arguments

`nor_handle`

Handle to a NOR.

`p_err`

Pointer to variable that will receive return the error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_IO`

Returned Value

none.

Notes / Warnings

1. This function should NOT be used while a file system exists on the device, or if the device is low-level formatted, unless the intent is to destroy all existing information.

FS_NOR_XIP_Cfg()

Description

Configure NOR flash and (Quad) SPI controller in XIP (eXecute-In-Place) mode.

Files

`fs_nor.h/fs_nor.c`

Prototype

```
void  FS_NOR_XIP_Cfg (FS_NOR_HANDLE nor_handle,
                      CPU_BOOLEAN   xip_en,
                      RTOS_ERR     *p_err)
```

Arguments

`nor_handle`

Handle to a NOR.

`xip_en`

XIP mode enable/disable flag.

`p_err`

Pointer to variable that will receive return the error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_IO`

Returned Value

none.

Notes / Warnings

none.

FS_NOR_Get()

Description

Get an NOR handle from a given media handle.

Files

`fs_nor.h/fs_nor.c`

Prototype

```
FS_NOR_HANDLE  FS_NOR_Get (FS_MEDIA_HANDLE  media_handle)
```

Arguments

`media_handle`

Handle to a media.

Returned Value

Handle to a NOR.

Notes / Warnings

None.

FS_NOR_BlkCntGet()

Description

Get the number of blocks on a NOR.

Files

`fs_nor.h/fs_nor.c`

Prototype

```
CPU_INT32U  FS_NOR_BlkCntGet (FS_NOR_HANDLE   nor_handle,
                              RTOS_ERR       *p_err)
```

Arguments

`nor_handle`

Handle to a NOR.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

The number of blocks.

Notes / Warnings

None.

FS_NOR_BlkSizeLog2Get()

Description

Get the base-2 logarithm of a NOR block size.

Files

`fs_nor.h/fs_nor.c`

Prototype

```
CPU_INT08U  FS_NOR_BlkSizeLog2Get (FS_NOR_HANDLE   nor_handle,
                                   RTOS_ERR       *p_err)
```

Arguments

`nor_handle`

Handle to a NOR.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

Base-2 logarithm of the block size.

Notes / Warnings

None.

FS_NOR_FTL_ConfigureLowParams()

Description

Configure NOR FTL parameters.

Files

`fs_nor_ftl.h/fs_nor_ftl.c`

Prototype

```
void  FS_NOR_FTL_ConfigureLowParams (      FS_MEDIA_HANDLE   media_handle,
                                     const FS_NOR_FTL_CFG   *p_cfg,
                                           RTOS_ERR         *p_err)
```

Arguments

`media_handle`

Handle to a media.

`p_cfg`

Pointer to a NOR FTL configuration structure.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_SEG_OVF`

Returned Value

none.

Notes / Warnings

1. `FS_NOR_FTL_ConfigureLowParams()` must be called prior to `FSBlkDev_Open()` or `FSMedia_LowFmt()` . It will configure a NOR FTL instance using the user specific parameters and allocate all needed resources by the FTL instance. Then `FSBlkDev_Open()` or `FSMedia_LowFmt()` can be called and the NOR FTL instance can be obtained from the internal NOR FTL list. Thus avoiding to reallocate resources once more time for the same NOR instance.
2. The NOR FTL configuration structure has the following fields:

```
typedef struct fs_nor_ftl_cfg {
    CPU_INT08U  RegionNbr;                 /* Block region within flash.                */
    CPU_ADDR    StartOffset;               /* Start address of data within flash.       */
    CPU_INT08U  PctRsvd;                   /* Percentage of device area reserved.       */
    CPU_INT16U  EraseCntDiffTh;            /* Erase count difference threshold.         */
    CPU_INT32U  DevSize;                   /* Size of flash, in octets.                 */
    FS_LB_SIZE  SecSize;                   /* Sector size of low-level formatted flash. */
} FS_NOR_FTL_CFG;
```

FS_NOR_FTL_LowCompact()

Description

Low-level compact a NOR device.

Files

`fs_nor_ftl.h/fs_nor_ftl.c`

Prototype

```
void  FS_NOR_FTL_LowCompact (FS_BLK_DEV_HANDLE   blk_dev_handle,
                             RTOS_ERR           *p_err)
```

Arguments

`blk_dev_handle`

Handle to a NOR-based block device.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_BLK_DEV_CLOSED`
- `RTOS_ERR_BLK_DEV_CORRUPTED`
- `RTOS_ERR_IO`

Returned Value

none.

Notes / Warnings

1. Compacting groups sectors containing high-level data into as few blocks as possible. If an image of a file system is to be formed for deployment, to be burned into chips for production, then it should be compacted after all files & directories are created.

FS_NOR_FTL_LowDefrag()

Description

Low-level defragment a NOR device.

Files

`fs_nor_ftl.h/fs_nor_ftl.c`

Prototype

```
void  FS_NOR_FTL_LowDefrag (FS_BLK_DEV_HANDLE   blk_dev_handle,
                            RTOS_ERR           *p_err)
```

Arguments

`blk_dev_handle`

Handle to a NOR-based block device.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_BLK_DEV_CLOSED`
- `RTOS_ERR_BLK_DEV_CORRUPTED`
- `RTOS_ERR_IO`

Returned Value

none.

Notes / Warnings

1. Defragmentation groups sectors containing high-level data into as few blocks as possible, in order of logical sector. A defragmented file system should have near-optimal access speeds in a read-only environment. See also '`FS_NOR_FTL_LowCompact()` Note #1'.

FS_RAM_Disk_Add()

Description

Add a RAM disk instance.

Files

`fs_ramdisk.h/fs_ramdisk.c`

Prototype

```
void  FS_RAM_Disk_Add (const  CPU_CHAR          *name,
                       const  FS_RAM_DISK_CFG   *p_cfg,
                              RTOS_ERR          *p_err)
```

Arguments

`name`

Media name to be assigned to the created RAM disk.

`p_cfg`

Pointer to a RAM disk configuration structure.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_INIT`
- `RTOS_ERR_SEG_OVF`

Returned Value

none.

Notes / Warnings

(1) The RAM disk configuration structure has the following fields:

```
typedef struct fs_ram_disk_cfg {
    void        *DiskPtr;                          /* Pointer to region simulating a RAM disk.           */
    FS_LB_QTY    LbCnt;                            /* Number of logical blocks composing RAM disk region. */
    FS_LB_SIZE   LbSize;                           /* Logical block size in bytes.                       */
} FS_RAM_DISK_CFG;
```

FS_SCSI_Open()

Description

Open a SCSI device.

Files

`fs_scsi.h/fs_scsi.c`

Prototype

```
FS_SCSI_HANDLE FS_SCSI_Open (FS_MEDIA_HANDLE   media_handle,
                             RTOS_ERR         *p_err)
```

Arguments

`media_handle`

Handle to a media.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_INIT`
- `RTOS_ERR_IO`

Returned Value

Handle to a SCSI device.

Notes / Warnings

None.

FS_SCSI_Close()

Description

Close a SCSI device.

Files

`fs_scsi.h/fs_scsi.c`

Prototype

```
void  FS_SCSI_Close (FS_SCSI_HANDLE   scsi_handle,
                     RTOS_ERR        *p_err)
```

Arguments

`sd_handle`

Handle to a SCSI device.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

none.

Notes / Warnings

None.

FS_SCSI_LU_InfoGet()

Description

Get SCSI logical unit information.

Files

`fs_scsi.h/fs_scsi.c`

Prototype

```
void FS_SCSI_LU_InfoGet (FS_SCSI_HANDLE    scsi_handle,
                         FS_SCSI_LU_INFO  *p_lu_info,
                         RTOS_ERR         *p_err)
```

Arguments

`ssci_handle`

Handle to a SCSI device.

`p_lu_info`

Pointer to logical unit information structure to be populated.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

none.

Notes / Warnings

(1) The SCSI logical unit information structure has the following fields:

```
typedef struct fs_scsi_lu_info {
    CPU_BOOLEAN  Removable;                          /* Flag indicating if LU is removable.              */
    FS_LB_SIZE   SecDfltSize;                        /* Default size of a sector.                        */
    FS_LB_QTY    SecCnt;                             /* Number of sectors composing the device.          */
    CPU_CHAR     VendorID_StrTbl[FS_SCSI_CMD_INQUIRY_VID_FIELD_LEN + 1u];
    CPU_CHAR     ProductID_StrTbl[FS_SCSI_CMD_INQUIRY_PID_FIELD_LEN + 1u];
    CPU_CHAR     ProdRevLevelStrTbl[FS_SCSI_CMD_INQUIRY_PROD_REV_LEVEL_FIELD_LEN + 1u];
} FS_SCSI_LU_INFO;
```

FS_SD_Open()

Description

Open a SD Card or SPI.

Files

`fs_sd.h/fs_sd.c`

Prototype

```
FS_SD_HANDLE  FS_SD_Open (FS_MEDIA_HANDLE   media_handle,
                          RTOS_ERR         *p_err)
```

Arguments

`media_handle`

Handle to a media.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_INIT`
- `RTOS_ERR_IO`

Returned Value

Handle to a SD Card or SPI.

Notes / Warnings

None.

FS_SD_Close()

Description

Close a SD Card or SPI.

Files

`fs_sd.h/fs_sd.c`

Prototype

```
void  FS_SD_Close (FS_SD_HANDLE   sd_handle,
                   RTOS_ERR      *p_err)
```

Arguments

`sd_handle`

Handle to a SD Card or SPI.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_IO`

Returned Value

none.

Notes / Warnings

None.

FS_SD_CID_Rd()

Description

Read SD CID (Card IDentification number) register.

Files

`fs_sd.h/fs_sd.c`

Prototype

```
void  FS_SD_CID_Rd (FS_SD_HANDLE   sd_handle,
                    CPU_INT08U    *p_dest,
                    RTOS_ERR      *p_err)
```

Arguments

`sd_handle`

Handle to a SD card.

`p_dest`

Pointer to 16-byte buffer that will receive SD/MMC Card ID register.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_IO`

Returned Value

none.

Notes / Warnings

1. Refer to specification "Physical Layer Simplified Specification Version 4.10", section '5.2 CID register' for more details about CID fields.

FS_SD_CSD_Rd()

Description

Read SD CSD (Card Specific Data) register.

Files

`fs_sd.h/fs_sd.c`

Prototype

```
void FS_SD_CSD_Rd (FS_SD_HANDLE   sd_handle,
                   CPU_INT08U    *p_dest,
                   RTOS_ERR      *p_err)
```

Arguments

`sd_handle`

Handle to a SD card.

`p_dest`

Pointer to 16-byte buffer that will receive SD/MMC Card Specific Data register.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_IO`

Returned Value

none.

Notes / Warnings

1. Refer to specification "Physical Layer Simplified Specification Version 4.10", section '5.3.2 CSD Register' for more details about CSD fields.

FS_SD_InfoGet()

Description

Get SD information.

Files

`fs_sd.h/fs_sd.c`

Prototype

```
void  FS_SD_InfoGet (FS_SD_HANDLE   sd_handle,
                     FS_SD_INFO    *p_sd_info,
                     RTOS_ERR      *p_err)
```

Arguments

`sd_handle`

Handle to a SD card.

`p_sd_info`

Pointer to a SD information structure to be populated.

`p_err`

Pointer to variable that will receive the return error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_IO`

Returned Value

none.

Notes / Warnings

(1) The SD information structure has the following fields:

```
typedef struct fs_sd_info {
    CPU_INT32U  BlkSize;                /* Block size, in octets (typically 512).      */
    CPU_INT32U  NbrBlks;                /* Capacity of device, in blocks.              */
    CPU_INT32U  ClkFreq;                /* Max supported clock freq, in Hz.            */
    CPU_INT32U  Timeout;                /* Communication timeout, in cycles.           */
    CPU_INT08U  CardType;               /* Card type.                                  */
    CPU_BOOLEAN HighCapacity;           /* Standard capacity/high capacity.            */
    CPU_INT08U  ManufID;                /* Manufacturer ID.                            */
    CPU_INT16U  OEM_ID;                 /* OEM/Application ID.                         */
    CPU_INT32U  ProdSN;                 /* Product serial number.                      */
    CPU_CHAR    ProdName[7];            /* Product name.                               */
    CPU_INT16U  ProdRev;                /* Product revision.                           */
    CPU_INT16U  Date;                   /* Date.                                       */
} FS_SD_INFO;
```

FS_NAND_HW_INFO_REG()

Description

Registers a NAND memory controller to the platform manager.

Files

`fs_nand.h`

Prototype

```
FS_NAND_HW_INFO_REG(name, p_hw_info)
```

Arguments

`name`

Unique name for the NAND memory controller. It is recommended to follow the standard "nandX" where X is a digit.

`p_hw_info`

Pointer to the NAND hardware information structure of type `FS_NAND_HW_INFO`.

Returned Value

None.

Notes / Warnings

1. This macro should normally be called from the BSP.

FS_NOR_QUAD_SPI_HW_INFO_REG()

Description

Registers a Quad SPI controller to the platform manager. A Quad SPI controller interfaces only to NOR flash devices.

Files

`fs_nor.h`

Prototype

```
FS_NOR_QUAD_SPI_HW_INFO_REG(p_name, p_hw_info)
```

Arguments

`name`

Unique name for the Quad SPI controller. It is recommended to follow the standard "norX" where X is a digit.

`p_hw_info`

Pointer to the NOR Quad SPI hardware information structure of type `FS_NOR_QUAD_SPI_HW_INFO` .

Returned Value

None.

Notes / Warnings

1. This macro should normally be called from the BSP.

FS_SD_CARD_HW_INFO_REG()

Description

Registers a SD Card (also called SD Host Controller [SDHC]) controller to the platform manager. The SDHC controller is a peripheral found on a microcontroller.

Files

`fs_sd_card.h`

Prototype

```
FS_SD_CARD_HW_INFO_REG(p_name, p_hw_info)
```

Arguments

`name`

Unique name for the SD Card controller. It is recommended to follow the standard "sdX" where X is a digit.

`p_hw_info`

Pointer to the SD Card hardware information structure of type FS_SD_CARD_HW_INFO.

Returned Value

None.

Notes / Warnings

1. This macro should normally be called from the BSP.

FS_SD_SPI_HW_INFO_REG()

Description

Registers a SD SPI controller to the platform manager. A SD SPI controller refers here to the controller embedded in an external SD card. This SD card is connected to the microcontroller via a SPI interface and thus communicates with a SPI controller located inside the microcontroller. This macro requires that a SPI controller is registered to the platform manager with the macro `IO_SERIAL_CTRLR_REG()`.

Files

`fs_sd_spi.h`

Prototype

```
FS_SD_SPI_HW_INFO_REG(p_name, p_ctrlr_name, slave_id)
```

Arguments

`p_name`

Unique name for the SD SPI controller. It is recommended to follow the standard "sdX" where X is a digit.

`p_ctrlr_name`

Unique name identifying a SPI controller registered with `IO_SERIAL_CTRLR_REG()`. The name should be "spiX" where X is a digit.

`slave_id`

Slave ID of the SD card on the SPI bus. The same SPI bus can be shared among several slaves (for instance, a Wi-Fi device, a NOR flash device, etc.). In that case, the slave ID allows to select the SD card when transferring data.

Returned Value

None.

Notes / Warnings

1. This macro should normally be called from the BSP.

File System Posix API

File System POSIX API

- _fs_errno()
- fs_perror()
- fs_chdir()
- fs_getcwd()
- fs_opendir()
- fs_closedir()
- fs_readdir_r()
- fs_mkdir()
- fs_remove()
- fs_rename()
- fs_rmdir()
- fs_stat()
- fs_fopen()
- fs_fclose()
- fs_fread()
- fs_fwrite()
- fs_ftruncate()
- fs_feof()
- fs_ferror()
- fs_clearerr()
- fs_fgetpos()
- fs_fsetpos()
- fs_fseek()
- fs_ftell()
- fs_rewind()
- fs_fileno()
- fs_fstat()
- fs_flockfile()
- fs_ftrylockfile()
- fs_funlockfile()
- fs_setbuf()
- fs_setvbuf()
- fs_fflush()
- fs_asctime_r()
- fs_ctime_r()
- fs_localtime_r()
- fs_mktime()

_fs_errno()

Description

Converts file system error code into Posix error code.

Files

`fs_core_posix.h/fs_core_posix.c`

Prototype

```
int _fs_errno (void)
```

Arguments

Returned Value

Posix error code.

Notes / Warnings

None.

fs_perror()

Description

Print POSIX error code along with user's output string.

Files

`fs_core_posix.h/fs_core_posix.c`

Prototype

```
void fs_perror (const char *p_err_desc)
```

Arguments

`p_err_desc`

Pointer to the user output string.

Returned Value

none.

Notes / Warnings

None.

fs_chdir()

Description

Set the working directory for the current task.

Files

`fs_core_posix.h/fs_core_posix.c`

Prototype

```
int fs_chdir (const char *path_dir)
```

Arguments

`path_dir`

String that specifies EITHER the absolute working directory path to set OR a relative path that will be applied to the current working directory.

Returned Value

- 0, if no error occurs.
- -1, otherwise.

Notes / Warnings

None.

fs_getcwd()

Description

Get the working directory for the current task.

Files

fs_core_posix.h/fs_core_posix.c

Prototype

```
char *fs_getcwd (char      *path_dir,
             fs_size_t   size)
```

Arguments

path_dir

String buffer that will receive the working directory path.

size

Size of string buffer.

Returned Value

- Pointer to working directory path, if no error occurs.
- Pointer to NULL, otherwise.

Notes / Warnings

None.

fs_opendir()

Description

Open a directory.

Files

fs_core_posix.h/fs_core_posix.c

Prototype

```
FS_DIR *fs_opendir (const char *name_full)
```

Arguments

name_full

Name of the directory.

Returned Value

- Pointer to a directory, if NO errors.
- Pointer to NULL, otherwise.

Notes / Warnings

None.

fs_closedir()

Description

Close and free a directory.

Files

`fs_core_posix.h/fs_core_posix.c`

Prototype

```
int  fs_closedir (FS_DIR  *p_dir)
```

Arguments

`p_dir`

Pointer to a directory.

Returned Value

- 0, if directory is successfully closed.
- -1, if any error was encountered.

Notes / Warnings

1. After a directory is closed, the application MUST cease from accessing its directory pointer. This could cause file system corruption, since this handle may be re-used for a different directory.

fs_readdir_r()

Description

Read a directory entry from a directory.

Files

`fs_core_posix.h/fs_core_posix.c`

Prototype

```
int  fs_readdir_r (FS_DIR            *p_dir,
              struct  fs_dirent   *p_dir_entry,
              struct  fs_dirent  **pp_result)
```

Arguments

`p_dir`

Pointer to a directory.

`p_dir_entry`

Pointer to variable that will receive directory entry information.

`pp_result`

1. Pointer to variable that will receive :
   (a) ... 'p_dir_entry' if NO error occurs AND directory does not encounter EOF.
   (b) ... pointer to NULL if an error occurs OR directory encounters EOF.

Returned Value

- 1, if an error occurs.
- 0, if no error occurs.

Notes / Warnings

1. IEEE Std 1003.1, 2004 Edition, Section '`readdir()` : DESCRIPTION' states that :
   (a) "The '`readdir()`' function shall not return directory entries containing empty names. If entries for dot or dot-dot exist, one entry shall be returned for dot and one entry shall be returned for dot-dot; otherwise, they shall not be returned."
   (b) "If a file is removed from or added to the directory after the most recent call to 'opendir()' or '`rewinddir()`', whether a subsequent call to '`readdir()`' returns an entry for that file is unspecified."
2. IEEE Std 1003.1, 2004 Edition, Section 'readdir() : RETURN VALUE' states that "[i]f successful, the '`readdir_r()`' function shall return zero; otherwise, an error shall be returned to indicate the error".
3. The directory entry information has the following fields:

```
struct fs_dirent {
   fs_ino_t  d_ino;                    /* File serial number.                    */
   char      d_name[256];              /* Buffer receiving directory entry name.      */
};
```

fs_mkdir()

Description

Create a directory.

Files

`fs_core_posix.h/fs_core_posix.c`

Prototype

```
int fs_mkdir (const char *name_full)
```

Arguments

`name_full`

Name of the directory.

Returned Value

- -1, if an error occurs.
- 0, if no error occurs.

Notes / Warnings

None.

fs_remove()

Description

Delete a file or directory.

Files

`fs_core_posix.h/fs_core_posix.c`

Prototype

```
int fs_remove (const char *name_full)
```

Arguments

`name_full`

Name of the entry.

Returned Value

- 0, if the entry is removed.
- -1, if the entry is NOT removed.

Notes / Warnings

1. IEEE Std 1003.1, 2004 Edition, Section '`remove()` : DESCRIPTION' states that :
   (a) "If '`path`' does not name a directory, '`remove(path)`' shall be equivalent to '`unlink(path)`'."
   (b) "If path names a directory, `remove(path)` shall be equivalent to `rmdir(path)`."
      1. See '`fs_rmdir()` Note(s)'.

fs_rename()

Description

Rename a file or directory.

Files

`fs_core_posix.h/fs_core_posix.c`

Prototype

```
int fs_rename (const char *name_full_old,
          const char *name_full_new)
```

Arguments

`name_full_old`

Old path of the entry.

`name_full_new`

New path of the entry.

Returned Value

0, if the entry is renamed.

-1, if the entry is NOT renamed.

Notes / Warnings

1. IEEE Std 1003.1, 2004 Edition, Section 'rename() : DESCRIPTION' states that :
   (a) "If the 'old' argument and the 'new' argument resolve to the same existing file, 'rename()' shall return successfully and perform no other action."
   (b) "If the 'old' argument points to the pathname of a file that is not a directory, the 'new' argument shall not point to the pathname of a directory. If the link named by the 'new' argument exists, it shall be removed and 'old' renamed to 'new'."
   (c) "If the 'old' argument points to the pathname of a directory, the 'new' argument shall not point to the pathname of a file that is not a directory. If the directory named by the 'new' argument exists, it shall be removed and 'old' renamed to 'new'."
      1. "If 'new' names an existing directory, it shall be required to be an empty directory."
   (d) "The 'new' pathname shall not contain a path prefix that names 'old'."
2. IEEE Std 1003.1, 2004 Edition, Section '`rename()` : RETURN VALUE' states that "[u]pon successful completion, '`rename()`' shall return 0; otherwise, -1 shall be returned".

fs_rmdir()

Description

Delete a directory.

Files

`fs_core_posix.h/fs_core_posix.c`

Prototype

```
int fs_rmdir (const char *name_full)
```

Arguments

`name_full`

Name of the directory.

Returned Value

0, if the directory is removed.

-1, if the directory is NOT removed.

Notes / Warnings

1. IEEE Std 1003.1, 2004 Edition, Section '`rmdir()` : DESCRIPTION' states that :
   (a) "The '`rmdir()`' function shall remove a directory whose name is given by path. The directory shall be removed only if it is an empty directory."
   (b) "If the directory is the root directory or the current working directory of any process, it is unspecified whether the function succeeds, or whether it shall fail"
2. IEEE Std 1003.1, 2004 Edition, Section '`rmdir()` : RETURN VALUE' states that "[u]pon successful completion, the function '`rmdir()`' shall return 0. Otherwise, -1 shall be returned".
3. The root directory CANNOT be removed.

fs_stat()

Description

Get information about a file or directory.

Files

`fs_core_posix.h/fs_core_posix.c`

Prototype

```
int fs_stat (const char          *name_full,
             struct fs_stat  *p_info)
```

Arguments

`name_full`

Name of the entry.

`p_info`

Pointer to structure that will receive the entry information.

Returned Value

- 0, if the function succeeds.
- -1, otherwise.

Notes / Warnings

1. The entry information structure has the following fields:

```
struct  fs_stat {
    fs_dev_t      st_dev;                 /* Device ID of device containing file.              */
    fs_ino_t      st_ino;                 /* File serial number.                               */
    fs_mode_t     st_mode;                /* Mode of file.                                     */
    fs_nlink_t    st_nlink;               /* Number of hard links to the file.                 */
    fs_uid_t      st_uid;                 /* User ID of file.                                  */
    fs_gid_t      st_gid;                 /* Group ID of file.                                 */
    fs_off_t      st_size;                /* File size in bytes.                               */
    fs_time_t     st_atime;               /* Time of last access.                              */
    fs_time_t     st_mtime;               /* Time of last data modification.                   */
    fs_time_t     st_ctime;               /* Time of last status change.                       */
    fs_blksize_t  st_blksize;             /* Preferred I/O block size for file.                */
    fs_blkcnt_t   st_blocks;              /* Number of blocks allocated for file.              */
};
```

fs_fopen()

Description

Open a file.

Files

`fs_core_posix.h/fs_core_posix.c`

Prototype

```
FS_FILE  *fs_fopen (const  char  *name_full,
                   const  char  *str_mode)
```

Arguments

`name_full`

Name of the file.

`str_mode`

Access mode of the file (see Note #1a).

Returned Value

Pointer to a file, if NO errors.

Pointer to NULL, otherwise.

Notes / Warnings

1. IEEE Std 1003.1, 2004 Edition, Section 'fopen() : DESCRIPTION' states that :
   (a) "If ['str_mode'] is one of the following, the file is open in the indicated mode." :

"r or rb"	Open file for reading
"w or wb"	Truncate to zero length or create file for writing
"a or ab"	Append; open and create file for writing at end-of-file
"r+ or rb+ or r+b"	Open file for update (reading and writing)
"w+ or wb+ or w+b"	Truncate to zero length or create file for update

"r or rb"	Open file for reading
"a+ or ab+ or a+b"	Append; open or create for update, writing at end-of-file

(b) "The character 'b' shall have no effect"
(c) "Opening a file with read mode ... shall fail if the file does not exist or cannot be read"
(d) "Opening a file with append mode ... shall cause all subsequent writes to the file to be forced to the then current end-of-file"
(e) "When a file is opened with update mode ... both input and output may be performed.... However, the application shall ensure that output is not directly followed by input without an intervening call to '`fflush()`' or to a file positioning function ('`fseek()`', '`fsetpos()`', or '`rewind()`'), and input is not directly followed by output without an intervening call to a file positioning function, unless the input operation encounters end-of-file."

2. IEEE Std 1003.1, 2004 Edition, Section '`fopen()` : RETURN VALUE' states that "[u]pon successful completion '`fopen()`' shall return a pointer to the object controlling the stream. Otherwise a null pointer shall be returned'.

fs_fclose()

Description

Close and free a file.

Files

`fs_core_posix.h/fs_core_posix.c`

Prototype

```
int fs_fclose (FS_FILE *p_file)
```

Arguments

`p_file`

Pointer to a file.

Returned Value

- 0, if the file was successfully closed.
- EOF, otherwise.

Notes / Warnings

1. After a file is closed, the application MUST cease from accessing its file pointer. This could cause file system corruption, since this handle may be re-used for a different file.
2.

(a) If the most recent operation is output (write), all unwritten data is written to the file.

(b) Any buffer assigned with fs_setbuf() or fs_setvbuf() shall no longer be accessed by the file system and may be re-used by the application.

fs_fread()

Description

Read from a file.

Files

`fs_core_posix.h/fs_core_posix.c`

Prototype

```
fs_size_t fs_fread (void      *p_dest,
                    fs_size_t  size,
```

```
            fs_size_t   nitems,
            FS_FILE    *p_file)
```

Arguments

`p_dest`

Pointer to destination buffer.

`size`

Size of each item to read.

`nitems`

Number of items to read.

`p_file`

Pointer to a file.

Returned Value

Number of items read.

Notes / Warnings

1. IEEE Std 1003.1, 2004 Edition, Section 'fread() : DESCRIPTION' states that :
   (a) "The ' `fread()` ' function shall read into the array pointed to by ' `ptr` ' up to ' `nitems` ' elements whose size is specified by 'size' in bytes"
   (b) "The file position indicator for the stream ... shall be advanced by the number of bytes successfully read"
2. IEEE Std 1003.1, 2004 Edition, Section ' `fread()` : RETURN VALUE' states that "[u]pon completion, ' `fread()` ' shall return the number of elements which is less than 'nitems' only if a read error or end-of-file is encountered".
3. See ' `fs_fopen()` Note #1e'.
4. The file MUST have been opened in read or update (read/write) mode.
5. If an error occurs while reading from the file, a value less than ' `nitems` ' will be returned. To determine whether the premature return was caused by reaching the end-of-file, the ' `fs_feof()` ' function should be used :

```
rtn = fs_fread(pbuf, 1, 1000, pfile);
if (rtn \< 1000) {
   eof = fs_feof();
   if (eof != 0) {
      // File has reached EOF
   } else {
      // Error has occurred
   }
}
```

fs_fwrite()

Description

Write to a file.

Files

`fs_core_posix.h/fs_core_posix.c`

Prototype

```
fs_size_t  fs_fwrite (const  void      *p_src,
                      fs_size_t   size,
                      fs_size_t   nitems,
                      FS_FILE    *p_file)
```

Arguments

`p_src`

Pointer to source buffer.

`size`

Size of each item to write.

`nitems`

Number of items to write.

`p_file`

Pointer to a file.

Returned Value

Number of items written.

Notes / Warnings

1. IEEE Std 1003.1, 2004 Edition, Section ' `fwrite()` : DESCRIPTION' states that :
   (a) "The ' `fwrite()` ' function shall write, from the array pointed to by 'ptr', up to ' `nitems` ' elements whose size is specified by ' `size` ', to the stream pointed to by ' `stream` '"
   (b) "The file position indicator for the stream ... shall be advanced by the number of bytes successfully written"
2. IEEE Std 1003.1, 2004 Edition, Section ' `fwrite()` : RETURN VALUE' states that "' `fwrite()` ' shall return the number of elements successfully written, which may be less than ' `nitems` ' if a write error is encountered".
3. See ' `fs_fopen()` Notes #1d & #1e'.
4. The file MUST have been opened in write or update (read/write) mode.

fs_ftruncate()

Description

Truncate a file.

Files

`fs_core_posix.h/fs_core_posix.c`

Prototype

```
int  fs_ftruncate (int       file_desc,
                   fs_off_t  size)
```

Arguments

`p_file`

Pointer to a file.

`size`

Length of file after truncation.

Returned Value

- 0, if the function succeeds.
- -1, otherwise.

Notes / Warnings

1. IEEE Std 1003.1, 2004 Edition, Section 'ftruncate() : DESCRIPTION' states that :
   (a) "If 'fildes' is not a valid file descriptor open for writing, the 'ftruncate()' function shall fail."
   (b) "[The] '`ftruncate()`' function shall cause the size of the file to be truncated to 'length'."
   1. "If the size of the file previously exceeded length, the extra data shall no longer be available to reads on the file."
   2. "If the file previously was smaller than this size, '`ftruncate`' shall either increase the size of the file or fail." This implementation increases the size of the file.
2. IEEE Std 1003.1, 2004 Edition, Section '`ftruncate()` : DESCRIPTION' states that [u]pon successful completion, '`ftruncate()`' shall return 0; otherwise -1 shall be returned and 'errno' set to indicate the error".
3. If the file position indicator before the call to '`fs_ftruncate()`' lay in the extra data destroyed by the function, then the file position will be set to the end-of-file.

fs_feof()

Description

Test EOF (End Of File) indicator on a file.

Files

`fs_core_posix.h/fs_core_posix.c`

Prototype

```
int fs_feof (FS_FILE *p_file)
```

Arguments

`p_file`

Pointer to a file.

Returned Value

- 0, if EOF indicator is NOT set or if an error occurred.
- Non-zero value, if EOF indicator is set.

Notes / Warnings

1. The return value from this function should ALWAYS be tested against 0 :

```
rtn = fs_feof(pfile);
if (rtn == 0) {
   // EOF indicator is NOT set
} else {
  // EOF indicator is    set
}
```

1. If the end-of-file indicator is set, that is `fs_feof()` returns a non-zero value, `fs_clearerr()` can be used to clear that indicator.

fs_ferror()

Description

Test error indicator on a file.

Files

`fs_core_posix.h/fs_core_posix.c`

Prototype

```
int  fs_ferror (FS_FILE  *p_file)
```

Arguments

`p_file`

Pointer to a file.

Returned Value

- 0, if error indicator is NOT set or if an error occurred.
- Non-zero value, if error indicator is set.

Notes / Warnings

1. The return value from this function should ALWAYS be tested against 0 :

```
rtn = fs_ferror(pfile);
if (rtn == 0) {
  // Error indicator is NOT set
} else {
  // Error indicator is     set
}
```

1. If the error indicator is set, that is fs_ferror() returns a non-zero value, fs_clearerr() can be used to clear that indicator.

fs_clearerr()

Description

Clear EOF (End Of File) and error indicators on a file.

Files

`fs_core_posix.h/fs_core_posix.c`

Prototype

```
void  fs_clearerr (FS_FILE  *p_file)
```

Arguments

`p_file`

Pointer to a file.

Returned Value

none.

Notes / Warnings

None.

fs_fgetpos()

Description

Get file position indicator.

Files

`fs_core_posix.h/fs_core_posix.c`

Prototype

```
int  fs_fgetpos (FS_FILE    *p_file,
             fs_fpos_t  *p_pos)
```

Arguments

`p_file`

Pointer to a file.

`p_pos`

Pointer to variable that will receive the file position indicator.

Returned Value

- 0, if no error occurs.
- Non-zero value, otherwise.

Notes / Warnings

1. The return value should be tested against 0 :

```
rtn = fs_fgetpos(pfile, &pos);
if (rtn == 0) {
   // No error occurred.
} else {
   // Handle error.
}
```

1. The value placed in ' `p_pos` ' should be passed to `fs_fsetpos()` to reposition the file to its position at the time when this function was called.

fs_fsetpos()

Description

Set file position indicator.

Files

`fs_core_posix.h/fs_core_posix.c`

Prototype

```
int  fs_fsetpos (       FS_FILE    *p_file,
             const  fs_fpos_t  *p_pos)
```

Arguments

`p_file`

Pointer to a file.

`p_pos`

Pointer to variable holding the file position.

Returned Value

- 0, if the function succeeds.
- Non-zero value, otherwise.

Notes / Warnings

1. The return value should be tested against 0 :

```
rtn = fs_fsetpos(pfile, &pos);
if (rtn == 0) {
   // No error occurred.
} else {
   // Handle error.
}
```

1. IEEE Std 1003.1, 2004 Edition, Section ' `fsetpos()` : DESCRIPTION' states that :
   (a) "If a read or write error occurs, the error indicator for the stream is set"
   (b) "The ' `fsetpos()` ' function shall set the file position and state indicators for the stream pointed to by stream according to the value of the object pointed to by 'pos', which the application shall ensure is a value obtained from an earlier call to 'fgetpos()' on the same stream."
2. IEEE Std 1003.1, 2004 Edition, Section ' `fsetpos()` : RETURN VALUE' states that "[t]he ' `fsetpos()` ' function shall return 0 if it succeeds; otherwise, it shall return a non-zero value".
3. No attempt is made to verify that the value stored in ' `p_pos` ' was returned from ' `fs_fgetpos()` '.
4. See also ' `fs_fseek()` Note #1d'.

fs_fseek()

Description

Set file position indicator.

Files

`fs_core_posix.h/fs_core_posix.c`

Prototype

```
int  fs_fseek (FS_FILE    *p_file,
             long  int   offset,
             int         origin)
```

Arguments

`p_file`

Pointer to a file.

`offset`

Offset from file position specified by 'origin'.

`origin`

Reference position for offset :

`SEEK_SET` Offset is from the beginning of the file.

`SEEK_CUR` Offset is from current file position.

`SEEK_END` Offset is from the end of the file.

Returned Value

0, if the function succeeds.

-1, otherwise.

Notes / Warnings

1. IEEE Std 1003.1, 2004 Edition, Section 'fread() : DESCRIPTION' states that :
   (a) "If a read or write error occurs, the error indicator for the stream shall be set"
   (b) "The new position measured in bytes from the beginning of the file, shall be obtained by adding 'offset' to the position specified by 'whence'. The specified point is ..."
      1. "... the beginning of the file for `SEEK_SET` "
      2. "... the current value of the file-position indicator for `SEEK_CUR` "
      3. "... end-of-file for `SEEK_END` "
   (c) "A successful call to ' `fseek()` ' shall clear the end-of-file indicator"
   (d) "The ' `fseek()` ' function shall allow the file-position indicator to be set beyond the end of existing data in the file. If data is later written at this point, subsequent reads of data in the gap shall return bytes with the value 0 until data is actually written into the gap."
2. IEEE Std 1003.1, 2004 Edition, Section ' `fread()` : RETURN VALUE' states that "[t]he ' `fseek()` ' and ' `fseeko()` ' functions shall return 0 if they succeeds. Otherwise, they shall return -1".
3. If the file position indicator is set beyond the file's current data, the file MUST be opened in write or read/write mode.

fs_ftell()

Description

Get file position indicator.

Files

`fs_core_posix.h/fs_core_posix.c`

Prototype

```
long int fs_ftell (FS_FILE *p_file)
```

Arguments

`p_file`

Pointer to a file.

Returned Value

- The current file position, if the function succeeds.
- -1, otherwise.

Notes / Warnings

1. IEEE Std 1003.1, 2004 Edition, Section 'ftell() : RETURN VALUE' states that :
   (a) "Upon successful completion, ' `ftell()` ' and ' `ftello()` ' shall return the current value of the file-position indicator for the stream measured in bytes from the beginning of the file."
   (b) "Otherwise, ' `ftell()` ' and ' `ftello()` ' shall return -1, cast to ' `long` ' and ' `off_t` ' respectively, and set errno to indicate the error."

fs_rewind()

Description

Reset file position indicator of a file.

Files

`fs_core_posix.h/fs_core_posix.c`

Prototype

```
void fs_rewind (FS_FILE *p_file)
```

Arguments

`p_file`

Pointer to a file.

Returned Value

none.

Notes / Warnings

1. IEEE Std 1003.1, 2004 Edition, Section '`rewind()`' : DESCRIPTION' states that :

"[T]he call '`rewind(stream)`' shall be equivalent to '`(void)fseek(stream, 0L, SEEK_SET)`' except that '`rewind()`' shall also clear the error indicator."

fs_fileno()

Description

Get file descriptor integer associated to file.

Files

`fs_core_posix.h/fs_core_posix.c`

Prototype

```
int fs_fileno (FS_FILE *p_file)
```

Arguments

`p_file`

Pointer to a file.

Returned Value

- File descriptor integer, if NO errors.
- -1, otherwise.

Notes / Warnings

None.

fs_fstat()

Description

Get information about a file.

Files

`fs_core_posix.h/fs_core_posix.c`

Prototype

```
int  fs_fstat (int            file_desc,
          struct  fs_stat  *p_info)
```

Arguments

`p_file`

Pointer to a file.

`p_info`

Pointer to structure that will receive the file information.

Returned Value

- 0, if the function succeeds.
- -1, otherwise.

Notes / Warnings

1. The file information structure has the following fields:

```
struct  fs_stat {
   fs_dev_t      st_dev;                    /* Device ID of device containing file.               */
   fs_ino_t      st_ino;                    /* File serial number.                                */
   fs_mode_t     st_mode;                   /* Mode of file.                                      */
   fs_nlink_t    st_nlink;                  /* Number of hard links to the file.                  */
   fs_uid_t      st_uid;                    /* User ID of file.                                   */
   fs_gid_t      st_gid;                    /* Group ID of file.                                  */
   fs_off_t      st_size;                   /* File size in bytes.                                */
   fs_time_t     st_atime;                  /* Time of last access.                               */
   fs_time_t     st_mtime;                  /* Time of last data modification.                    */
   fs_time_t     st_ctime;                  /* Time of last status change.                        */
   fs_blksize_t  st_blksize;                /* Preferred I/O block size for file.                 */
   fs_blkcnt_t   st_blocks;                 /* Number of blocks allocated for file.               */
};
```

fs_flockfile()

Description

Acquire task ownership of a file.

Files

`fs_core_posix.h/fs_core_posix.c`

Prototype

```
void  fs_flockfile (FS_FILE  *p_file)
```

Arguments

`p_file`

Pointer to a file.

Returned Value

none.

Notes / Warnings

1. IEEE Std 1003.1, 2004 Edition, Section '`flockfile()`, `ftrylockfile()`, `funlockfile()` : DESCRIPTION' states that :
   (a) "The '`flockfile()`' function shall acquire thread ownership of a `(FILE *)` object."

(b) "The functions shall behave as if there is a lock count associated with each `(FILE *)` object."

1. "The `(FILE *)` object is unlocked when the count is zero."
2. "When the count is positive, a single thread owns the `(FILE *)` object."
3. "When the '`flockfile()`' function is called, if the count is zero or if the count is positive and the caller owns the `(FILE *)`, the count shall be incremented. Otherwise, the calling thread shall be suspended, waiting for the count to return to zero."
4. "Each call to '`funlockfile()`' shall decrement the count."
5. "This allows matching calls to '`flockfile()`' (or successful calls to '`ftrylockfile()`') and '`funlockfile()`' to be nested."

fs_ftrylockfile()

Description

Acquire task ownership of a file (if available).

Files

`fs_core_posix.h/fs_core_posix.c`

Prototype

```
int fs_ftrylockfile (FS_FILE *p_file)
```

Arguments

`p_file`

Pointer to a file.

Returned Value

- 0, if no error occurs and the file lock is acquired.
- Non-zero value, otherwise.

Notes / Warnings

1. IEEE Std 1003.1, 2004 Edition, Section '`flockfile()`, `ftrylockfile()`, `funlockfile()` : DESCRIPTION' states that :
   (a) See '`fs_flockfile()` Note(s)'.
   (b) "The '`ftrylockfile()` function shall acquire for a thread ownership of a `(FILE *)` object if the object is available; '`ftrylockfile()`' is a non-blocking version of '`flockfile()`'."
2. IEEE Std 1003.1, 2004 Edition, Section '`flockfile()`, `ftrylockfile()`, `funlockfile()` : RETURN VALUES' states that "[t]he '`ftrylockfile()`' function shall return zero for success and non-zero to indicate that the lock cannot be acquired".

fs_funlockfile()

Description

Release task ownership of a file.

Files

`fs_core_posix.h/fs_core_posix.c`

Prototype

```
void fs_funlockfile (FS_FILE *p_file)
```

Arguments

`p_file`

Pointer to a file.

Returned Value

none.

Notes / Warnings

1. IEEE Std 1003.1, 2004 Edition, Section ' flockfile() , ftrylockfile() , funlockfile() : DESCRIPTION' states that :
   (a) See ' fs_flockfile() Note(s)'.

fs_setbuf()

Description

Assign buffer to a file.

Files

fs_core_posix.h/fs_core_posix.c

Prototype

```
void fs_setbuf (FS_FILE *p_file,
           char    *p_buf)
```

Arguments

p_file

Pointer to a file.

p_buf

Pointer to buffer.

Returned Value

none.

Notes / Warnings

1. IEEE Std 1003.1, 2004 Edition, Section ' setbuf() : DESCRIPTION' states that :

"Except that it returns no value, the function call: ' setbuf(stream, buf) ' shall be equivalent to: ' setvbuf(stream, buf, _IOFBF, BUFSIZ) ' if 'buf' is not a null pointer"

1. See ' fs_setvbuf() Note(s)'.

fs_setvbuf()

Description

Assign buffer to a file.

Files

fs_core_posix.h/fs_core_posix.c

Prototype

```
int fs_setvbuf (FS_FILE   *p_file,
           char      *p_buf,
           int       mode,
           fs_size_t  size)
```

Arguments

`p_file`

Pointer to a file.

`p_buf`

Pointer to buffer.

`mode`

Buffer mode.

- `_IOFBF` Data buffered for reads & writes.
- `_IONBR` Data unbuffered for reads & writes.

`size`

Size of buffer, in octets.

Returned Value

- -1, if an error occurs.
- 0, if no error occurs.

Notes / Warnings

1. IEEE Std 1003.1, 2004 Edition, Section ' `setvbuf()` : DESCRIPTION' states that :
   (a) "The `setvbuf()` function may be used after the stream pointed to by stream is associated with an open file but before any other operation (other than an unsuccessful call to `setvbuf()`) is performed on the stream."
   (b) "The argument 'mode' determines how 'stream' will be buffered ... "
   1. ... `FS__IOFBF` "causes input/output to be fully buffered".
   2. ... `FS__IONBF` "causes input/output to be unbuffered".
   3. No equivalent to ' `_IOLBF` ' is supported.

   (c) "If ' `buf` ' is not a null pointer, the array it points to may be used instead of a buffer allocated by the 'setvbuf' function and the argument ' `size` ' specifies the size of the array ...." This implementation REQUIRES that ' `buf` ' not be a null pointer; the array ' `buf` ' points to will always be used.
   (d) The function "returns zero on success, or nonzero if an invalid value is given for 'mode' or if the request cannot be honored".
2. 'size' MUST be more than or equal to the size of one sector and will be rounded DOWN to the size of a number of full sectors.
3. Once a buffer is assigned to a file, a new buffer may not be assigned nor may the assigned buffer be removed. To change the buffer, the file should be closed & re-opened.
4. Upon power loss, any data stored in file buffers will be lost.

fs_fflush()

Description

Flush buffer contents to file.

Files

`fs_core_posix.h/fs_core_posix.c`

Prototype

```
int fs_fflush (FS_FILE *p_file)
```

Arguments

`p_file`

Pointer to a file.

Returned Value

- 0, if flushing succeeds.
- EOF, otherwise.

Notes / Warnings

1. IEEE Std 1003.1, 2004 Edition, Section '`fflush()` : DESCRIPTION' states that :
   (a) "If '`stream`' points to an output stream or an update stream in which the most recent operation was not input, '`fflush()`' shall cause any unwritten data for that stream to be written to the file."
   (b) "If 'stream' is a null pointer, the '`fflush`' function performs this flushing action on all streams ...." #### Currently unimplemented.
   (c) "Upon successful completion, `fflush()` shall return 0; otherwise, it shall set the error indicator for the stream, return EOF."
2. IEEE Std 1003.1, 2004 Edition, Section '`fflush()`' defines no behavior for an input stream or update stream in which the most recent operation was input.
   (a) In this implementation, if the most recent operation is input, `fs_fflush()` clears all buffered input data.

fs_asctime_r()

Description

Convert date/time to string.

Files

`fs_core_posix.h/fs_core_posix.c`

Prototype

```
char *fs_asctime_r (const struct fs_tm *p_time,
                    char         *p_str_time)
```

Arguments

`p_time`

Pointer to date/time to format.

`p_str_time`

String buffer that will receive the date/time string (see Note #1).

Returned Value

Pointer to date/time string, if NO errors.

Pointer to NULL, otherwise.

Notes / Warnings

1. String buffer MUST be at least 26 characters long. Buffer overruns MUST be prevented by caller.
2. IEEE Std 1003.1, 2004 Edition, Section '`asctime()` : DESCRIPTION' states that :
   (a) "The '`asctime()`' function shall convert the broken-down time in the structure pointed to by '`timeptr`' into a string in the form: Sun Sep 16 01:03:52 1973\n\0.

fs_ctime_r()

Description

Convert timestamp to string.

Files

`fs_core_posix.h/fs_core_posix.c`

Prototype

```
char *fs_ctime_r (const fs_time_t *p_ts,
                  char        *p_str_time)
```

Arguments

`p_ts`

Pointer to timestamp to format.

`str_time`

String buffer that will receive the timestamp string (see Note #1).

Returned Value

- Pointer to timestamp buffer, if NO errors.
- Pointer to NULL, otherwise.

Notes / Warnings

1. The timestamp buffer MUST be at least 26 characters long. buffer overruns MUST be prevented by caller.
2. IEEE Std 1003.1, 2004 Edition, Section '`ctime()` : DESCRIPTION' states that :
   (a) '`ctime`' "shall be equivalent to: '`asctime(localtime(clock))`'".

fs_localtime_r()

Description

Convert timestamp to date/time.

Files

`fs_core_posix.h/fs_core_posix.c`

Prototype

```
struct fs_tm *fs_localtime_r (const      fs_time_t *p_ts,
                              struct fs_tm     *p_time)
```

Arguments

`p_ts`

Pointer to timestamp to convert.

`p_time`

Pointer to variable that will receive the date/time.

Returned Value

- Pointer to date/time, if NO errors.
- Pointer to NULL, otherwise.

Notes / Warnings

1. IEEE Std 1003.1, 2004 Edition, Section '4.14 Seconds Since the Epoch()' states that

(a) "If the year is <1970 or the value is negative, the relationship is undefined. If the year is >=1970 and the value is non-negative, the value is related to coordinated universal time name according to the C-language expression, where `tm_sec`, `tm_min`, `tm_hour`, `tm_yday`, and `tm_year` are all integer types:
`tm_sec + tm_min*60 + tm_hour*3600 + tm_yday*86400 + (tm_year-70)*31536000 + ((tm_year-69)/4)*86400 - ((tm_year-1)/100)*86400 + ((tm_year+299)/400)*86400`
(b) "The relationship between the actual time of day and the current value for seconds since the Epoch is unspecified."
2. The expression for the time value can be rewritten :

```
time_val = tm_sec + 60 * (tm_min +
        60 * (tm_hour +
        24 * (tm_yday + ((tm_year-69)/4) - ((tm_year-1)/100) + ((tm_year+299)/400) +
        365 * (tm_year - 70))))
```

fs_mktime()

Description

Convert date/time to timestamp.

Files

`fs_core_posix.h/fs_core_posix.c`

Prototype

```
fs_time_t  fs_mktime (struct  fs_tm  *p_time)
```

Arguments

`p_time`

Pointer to date/time to convert.

Returned Value

- Time value, if NO errors.
- (`fs_time_t`)-1, otherwise.

Notes / Warnings

1. See ' `fs_localtime_r()` Note #1'.
2. IEEE Std 1003.1, 2004 Edition, Section ' `mktime()` : DESCRIPTION' states that :
   (a) "The ' `mktime()` ' function shall convert the broken-down time, expressed as local time, in the structure pointed to by ' `timeptr` ', into a time since the Epoch"
   (b) "The original values of ' `tm_wday` ' and ' `tm_yday` ' components of the structure are ignored, and the original values of the other components are not restricted to the ranges described in `<time.h>` " (see also Note #3)
   (c) "Upon successful completion, the values of the ' `tm_wday` ' and ' `tm_yday` ' components of the structure shall be set appropriately, and the other components set to represent the specified time since the Epoch, but with their values forced to the ranges indicated in the `<time.h>` entry"
3. Even though strict range checking is NOT performed, the broken-down date/time components are restricted to positive values, and the month value MUST be between 0 & 11 (otherwise, the day of year cannot be determined).

Network API

Network API

- Network Core API
- HTTP Client API
- HTTP Server API
- MQTT Client API
- SMTP Client API
- SNTP Client API
- FTP Client API
- TFTP Client API
- TFTP Server API
- Telnet Server API
- IPerf API
- Mocana nanoSSL Certificate API

Network Core API

Network Core API

The application programming interfaces (APIs) to the whole Network core stack are described in this section.

Network Initialization

Function Name	Description
Net_Init()	Initializes Network stack
Net_ConfigureCoreTaskStk()	Configures the Network core task's stack.
Net_ConfigureCoreSvcTaskStk()	Configures the Network service task's stack. The service task is used for core modules like DHCP.
Net_ConfigureMemSeg()	Configures the memory segment to use when allocating control data for the network core.
Net_ConfigureDNS_Client()	Configure the DNS Client module parameters.
Net_CoreTaskPrioSet()	Assigns a new priority to the Network core task.
Net_CoreSvcTaskPrioSet()	Assigns a new priority to the Network core service task.

General Network Utilities

Function Name	Description
NET_UTIL_HOST_TO_NET_16()	Convert 16-bit integer values from CPU host-order to network-order.
NET_UTIL_HOST_TO_NET_32()	Convert 32-bit integer values from CPU host-order to network-order.
NET_UTIL_NET_TO_HOST_16()	Convert 16-bit integer values from network-order to CPU host- order.
NET_UTIL_NET_TO_HOST_32()	Convert 32-bit integer values from network-order to CPU host- order.
NetUtil_TS_Get_ms()	Get current millisecond timestamp.
NetUtil_TS_Get()	Get the current Internet Timestamp.

ASCII Functions

Function Name	Description
NetASCII_IPv4_to_Str()	Convert an IPv4 address in host-order into an IPv4 dotted-decimal notation ASCII string.
NetASCII_IPv6_to_Str()	Convert an IPv6 address into an IPv6 colon-decimal notation ASCII string.
NetASCII_MAC_to_Str()	Convert a Media Access Control (MAC) address into a hexadecimal address string.
NetASCII_Str_to_IP()	Convert a string of an IPv4 or IPv6 address in their respective decimal notation to an IPv4 or IPv6 address.
NetASCII_Str_to_IPv4()	Convert a string of an IPv4 address in dotted-decimal notation to an IPv4 address in host-order.
NetASCII_Str_to_IPv6()	Convert a string of an IPv6 address in common-decimal notation to an IPv6 address.
NetASCII_Str_to_MAC()	Convert an hexadecimal address string to a Media Access Control (MAC) address.

Network Buffer

Function Name	Description
NetBuf_PoolStatGet()	Get an interface's Network Buffers' statistics pool.
NetBuf_PoolStatResetMaxUsed()	Reset an interface's Network Buffers' statistics pool's maximum number of entries used.
NetBuf_RxLargePoolStatGet()	Get an interface's large receive buffers' statistics pool.
NetBuf_RxLargePoolStatResetMaxUsed()	Reset an interface's large receive buffers' statistics pool's maximum number of entries used.
NetBuf_TxLargePoolStatGet()	Get an interface's large transmit buffers' statistics pool.
NetBuf_TxLargePoolStatResetMaxUsed()	Reset an interface's large transmit buffers' statistics pool's maximum number of entries used.
NetBuf_TxSmallPoolStatGet()	Get an interface's small transmit buffers' statistics pool.
NetBuf_TxSmallPoolStatResetMaxUsed()	Reset an interface's small transmit buffers' statistics pool's maximum number of entries used.

Network Connection

Function Name	Description
NetConn_CfgAccessedTh()	Configure network connection access promotion threshold.
NetConn_PoolStatGet()	Get Network Connections' statistics pool.
NetConn_PoolStatResetMaxUsed()	Reset Network Connections' statistics pool's maximum number of entries used.

Network Timer

Function Name	Description
NetTmr_PoolStatGet()	Gets the network timer statistics pool.
NetTmr_PoolStatResetMaxUsed()	Resets the network timer statistics pool's maximum number of entries used.

Network Interface

Function Name	Description
NetIF_Add()	Add a network device and hardware as a network interface.
NetIF_AddrHW_Get()	Get network interface's hardware address.
NetIF_AddrHW_IsValid()	Validate a network interface hardware address.
NetIF_AddrHW_Set()	Set network interface's hardware address.
NetIF_CfgPerfMonPeriod()	Configure the network interface Performance Monitor Handler timeout.
NetIF_CfgPhyLinkPeriod()	Configure network interface Physical Link State Handler timeout.
NetIF_GetExtAvailCtr()	Returns the number of external interface configured.
NetIF_GetNbrBaseCfgd()	Gets the interface base number (first interface ID).
NetIF_GetRxDataAlignPtr()	Get an aligned pointer into a receive application data buffer.
NetIF_GetTxDataAlignPtr()	Get an aligned pointer into a transmit application data buffer.
NetIF_TypeGet()	Get the network interface type.
NetIF_IO_Ctrl()	Handle network interface and/or device specific (I/O) control(s).
NetIF_IsEn()	Validate network interface as enabled.
NetIF_IsEnCfgd()	Validate configured network interface as enabled.

Function Name	Description
NetIF_ISR_Handler()	Handle a network interface's device interrupts.
NetIF_IsValid()	Validate network interface number.
NetIF_IsValidCfgd()	Validate configured network interface number.
NetIF_LinkStateGet()	Get network interface's last known physical link state.
NetIF_LinkStateSubscribe()	Subscribe to get notified when an interface link state changes.
NetIF_LinkStateUnsubscribe()	Unsubscribe to get notified when interface link state changes.
NetIF_LinkStateWaitUntilUp()	Wait for a network interface's physical link state to be UP .
NetIF_MTU_Get()	Get network interface's MTU.
NetIF_MTU_Set()	Set network interface's MTU.
NetIF_Start()	Start a network interface.
NetIF_Stop()	Stop a network interface.
NetIF_TxSuspendTimeoutGet_ms()	Gets the network interface transmit suspend timeout value.
NetIF_TxSuspendTimeoutSet()	Sets the network interface transmit suspend timeout value.
NetIF_WaitSetupReady()	Wait for the network interface setup to be complete.

Ethernet Network Interface

Macro Name	Description
NET_CTRLR_ETHER_REG()	Registers an Ethernet controller to the platform manager.

Function Name	Description
NetIF_Ether_Add()	Add & initialize a specific instance of a network Ethernet interface.
NetIF_Ether_Start()	Start an Ethernet type interface

Wireless Network Interface

Macro Name	Description
NET_CTRLR_WIFI_SPI_REG()	Registers an external WiFi controller connected via SPI to the platform manager.

Function Name	Description
NetIF_WiFi_Add()	Add & initialize a specific instance of a network WiFi interface.
NetIF_WiFi_CreateAP()	Create a wireless ad-hoc access point.
NetIF_WiFi_GetPeerInfo()	Gets the peer info connected to the access point (when acting as an access point).
NetIF_WiFi_Join()	Join a wireless access point.
NetIF_WiFi_Leave()	Leave the access point previously joined.
NetIF_WiFi_Scan()	Scan available wireless access point.
NetIF_WiFi_Start()	Start a Wi-Fi-type interface.

DHCP Client

Function Name	Description
DHCPc_IF_Add()	Attaches an interface to the DHCP module and starts the DHCP process on this interface.
DHCPc_IF_Reboot	Reboots the DHCP Client process.
DHCPc_IF_Remove	Stops and removes the DHCP operation on the given network interface.

DNS Client

Function Name	Description
DNSc_CacheClrAll()	Flushes the DNS cache.
DNSc_CacheClrHost()	Removes a host from the cache.
DNSc_CfgServerByAddr()	Configures the DNS server to use by default with an address structure.
DNSc_CfgServerByStr()	Configures the DNS server to use by default using a string.
DNSc_GetServerByAddr()	Gets the default DNS server in an address object format.
DNSc_GetServerByStr()	Gets the default DNS server in string format.
DNSc_GetHost()	Converts a string representation of a host name to its corresponding IP address using DNS service.
DNSc GetHostAddrs()	Get a list of IP addresses assigned to a host.
DNSc FreeHostAddrs()	Free Host addresses allocated by DNS_HostAddrsGet().

ARP

Function Name	Description
NetARP_CacheGetAddrHW()	Get the hardware address corresponding to a specific ARP cache's protocol address.
NetARP_CachePoolStatGet()	Get ARP caches' statistics pool.
NetARP_CachePoolStatResetMaxUsed()	Reset ARP caches' statistics pool's maximum number of entries used.
NetARP_CacheProbeAddrOnNet()	Transmit an ARP request to probe the local network for a specific protocol address.
NetARP_CfgAddrFilterEn()	Configures the ARP address filter feature
NetARP_CfgCacheAccessedTh()	Configure ARP cache access promotion threshold.
NetARP_CfgCacheTimeout()	Configure ARP cache timeout for ARP Cache List.
NetARP_CfgCacheTxQ_MaxTh()	Configures the ARP cache maximum number of transmit packet buffers to enqueue.
NetARP_CfgPendReqMaxRetries()	Configure maximum number of ARP request retries for ARP cache in PEND state.
NetARP_CfgPendReqTimeout()	Configure timeout between ARP request timeouts for ARP cache in PEND state.
NetARP_CfgRenewReqMaxRetries()	Configure maximum number of ARP request retries for ARP cache in RENEW state.
NetARP_CfgRenewReqTimeout()	Configure timeout between ARP request timeouts for ARP cache in RENEW state.
NetARP_IsAddrProtocolConflict()	Check interface's protocol address conflict status between this interface's ARP host protocol address(es) and any other host(s) on the local network.
NetARP_TxReqGratuitous()	Prepares and transmits an unrequested ARP Request into the local network.

NDP

Function Name	Description
NetNDP_CfgNeighborCacheTimeout()	Configure NDP neighbor cache timeout for NDP Neighbor Cache List.

Function Name	Description
NetNDP_CfgReachabilityTimeout()	Configure one of the NDP neighbor timeouts associated with the Neighbors Unreachability Detection.
NetNDP_CfgSolicitMaxNbr()	Configure one of the NDP neighbor maximum solicitations number.

IGMP

Function Name	Description
NetIGMP_HostGrpJoin()	Join a host group.
NetIGMP_HostGrpLeave()	Leave a host group.

MLDP

Function Name	Description
NetMLDP_HostGrpJoin()	Join a MLDP Multicast host group.
NetMLDP_HostGrpLeave()	Leave a MLDP host group.

ICMP

Function Name	Description
NetICMP_TxEchoReq()	Send ICMPv4 or ICMPv6 Echo Request message to a network host.

IPv4

Function Name	Description
NetIPv4_AddrLinkLocalCfg()	Start the IPv4 Link Local process on the given interface.
NetIPv4_AddrLinkLocalCfgRemove()	Stop the IPv4 Link Local process on the given interface and remove the link local address if one has already been configured.
NetIPv4_CfgAddrAdd()	Add a statically-configured IPv4 host address, subnet mask, and default gateway to an interface.
NetIPv4_CfgAddrAddDynamic()	Add a dynamically-configured IPv4 host address, subnet mask, and default gateway to an interface.
NetIPv4_CfgAddrAddDynamicStart()	Start dynamic IPv4 address configuration for an interface.
NetIPv4_CfgAddrAddDynamicStop()	Stop dynamic IPv4 address configuration for an interface.
NetIPv4_CfgAddrRemove()	Remove a configured IPv4 host address from an interface.
NetIPv4_CfgAddrRemoveAll()	Remove all configured IPv4 host address(es) from an interface.
NetIPv4_CfgFragReasmTimeout()	Configure IPv4 fragment reassembly timeout.
NetIPv4_GetAddrDfltGateway()	Get the default gateway IPv4 address for a host's configured IPv4 address.
NetIPv4_GetAddrHost()	Get an interface's configured IPv4 host address(es).
NetIPv4_GetAddrSrc()	Get corresponding configured IPv4 host address for a remote IPv4 address to use as source address.
NetIPv4_GetAddrSubnetMask()	Get the IPv4 address subnet mask for a host's configured IPv4 address.
NetIPv4_IsAddrBroadcast()	Validate an IPv4 address as the limited broadcast IPv4 address.
NetIPv4_IsAddrClassA()	Validate an IPv4 address as a Class-A IPv4 address.
NetIPv4_IsAddrClassB()	Validate an IPv4 address as a Class-B IPv4 address.

Function Name	Description
NetIPv4_IsAddrClassC()	Validate an IPv4 address as a Class-C IPv4 address.
NetIPv4_IsAddrHost()	Validate an IPv4 address as one the host's IPv4 address(es).
NetIPv4_IsAddrHostCfgd()	Validate an IPv4 address as one the host's configured IPv4 address(es).
NetIPv4_IsAddrLocalHost()	Validate an IPv4 address as a Localhost IPv4 address.
NetIPv4_IsAddrLocalLink()	Validate an IPv4 address as a link-local IPv4 address.
NetIPv4_IsAddrMulticast()	Validate an IPv4 address as a multicast IP address.
NetIPv4_IsAddrsCfgdOnIF()	Check if any IPv4 address(es) are configured on an interface.
NetIPv4_IsAddrThisHost()	Validate an IPv4 address as the 'This Host' initialization IPv4 address.
NetIPv4_IsValidAddrHost()	Validate an IPv4 address as a valid IPv4 host address.
NetIPv4_IsValidAddrHostCfgd()	Validate an IPv4 address as a valid, configurable IPv4 host address.
NetIPv4_IsValidAddrSubnetMask()	Validate an IPv4 address subnet mask.

IPv6

Function Name	Description
NetIPv6_AddrAutoCfgDis()	Disables the IPv6 Stateless Address Auto-Configuration procedure.
NetIPv6_AddrAutoCfgEn()	Enables the IPv6 Stateless Address Auto-Configuation procedure.
NetIPv6_AddrMask()	Applies IPv6 mask on an address.
NetIPv6_AddrMaskByPrefixLen()	Gets the IPv6 address masked with the prefix length.
NetIPv6_AddrSubscribe()	Configures the IPv6 Address Configuration hook function.
NetIPv6_AddrUnsubscribe()	Removes a configured IPv6 host address and multicast solicited mode address from an interface.
NetIPv6_CfgAddrAdd()	Adds a statically-configured IPv6 host address to an interface.
NetIPv6_AddrTypeValidate()	Validates the type of an IPv6 address.
NetIPv6_CfgAddrRemove()	Removes a configured IPv6 host address and multicast solicited mode address from an interface.
NetIPv6_CfgAddrRemoveAll()	Removes all configured IPv6 host address(s) from an interface.
NetIPv6_CfgFragReasmTimeout()	Configures the IPv6 fragment reassembly timeout.
NetIPv6_CreateAddrFromID()	Creates an IPv6 address from a prefix and an identifier.
NetIPv6_CreateIF_ID()	Creates an IPv6 interface identifier.
NetIPv6_GetAddrHost()	Gets an interface's IPv6 host address(es).
NetIPv6_GetAddrSrc()	Finds the best matched source address in the IPv6 configured host addresses for the specified destination address.
NetIPv6_GetAddrMatchingLen()	Computes the number of identical most significant bits of two IPv6 addresses.
NetIPv6_GetAddrScope()	Gets the scope of a specific IPv6 address.
NetIPv6_IsAddrHostCfgd()	Validates an IPv6 address as a configured IPv6 host address on an enabled interface.
NetIPv6_IsAddrsCfgdOnIF()	Checks if any IPv6 host addresses are configured on a specific interface.
NetIPv6_IsValidAddrHost()	Validates an IPv6 host address.
NetIPv6_IsAddrLinkLocal()	Validates an IPv6 address as a link-local IPv6 address.
NetIPv6_IsAddrSiteLocal()	Validates an IPv6 address as a site-local address.
NetIPv6_IsAddrMcast()	Validates an IPv6 address as a multicast address.
NetIPv6_IsAddrMcastAllRouters()	Validates an IPv6 address as the all routers multicast address.

Function Name	Description
NetIPv6_IsAddrMcastAllNodes()	Validates an IPv6 address as the all nodes multicast address.
NetIPv6_IsAddrMcastSolNode()	Validates an IPv6 address as a solicited node multicast address.
NetIPv6_IsAddrMcastRsvd()	Validates an IPv6 address as a reserved multicast IPv6 address.
NetIPv6_IsAddrUnspecified()	Validates an IPv6 address as the unspecified IPv6 address.
NetIPv6_IsAddrLoopback()	Validates an IPv6 address as the IPv6 loopback address.

TCP

Function Name	Description
NetTCP_ConnCfgIdleTimeout()	Configures the TCP connection's idle timeout.
NetTCP_ConnCfgMaxSegSizeLocal()	Configures the TCP connection's local maximum segment size.
NetTCP_ConnCfgMSL_Timeout()	Configures the TCP connection's maximum segment lifetime (MSL) timeout.
NetTCP_ConnCfgReTxMaxTh()	Configures the TCP connection's maximum number of same segment retransmissions.
NetTCP_ConnCfgReTxMaxTimeout()	Configures the TCP connection's maximum retransmission timeout.
NetTCP_ConnCfgRxWinSize()	Configures the TCP connection's receive window size.
NetTCP_ConnCfgTxAckDlyTimeout()	Configures the TCP connection's transmit acknowledgment delay timeout.
NetTCP_ConnCfgTxAckImmedRxdPushEn()	Configures the TCP connection's transmit immediate acknowledgment for received and pushed TCP segments.
NetTCP_ConnCfgTxKeepAliveEn()	Configures the TCP connection's transmit keep-alive enable.
NetTCP_ConnCfgTxKeepAliveRetryTimeout()	Configures the TCP connection's transmit keep-alive retry timeout.
NetTCP_ConnCfgTxKeepAliveTh()	Configures the TCP connection's maximum number of consecutive keep-alives to transmit.
NetTCP_ConnCfgTxNagleEn()	Configures the TCP connection's transmit Nagle algorithm enable.
NetTCP_ConnCfgTxWinSize()	Configures the TCP connection's transmit window size.
NetTCP_ConnPoolStatGet()	Gets the TCP connections' statistics pool.
NetTCP_ConnPoolStatResetMaxUsed()	Resets the TCP connections' statistics pool's maximum number of entries used.
NetTCP_ConnStateGet()	Retrieves the TCP Connection State.

Socket Functions

Function Name	Description
NET_SOCK_DESC_CLR()	Remove a socket file descriptor ID as a member of a file descriptor set.
NET_SOCK_DESC_COPY()	Copy a file descriptor set to another file descriptor set.
NET_SOCK_DESC_INIT()	Initialize/zero-clear a file descriptor set.
NET_SOCK_DESC_IS_SET()	Check if a socket file descriptor ID is a member of a file descriptor set.
NET_SOCK_DESC_SET()	Add a socket file descriptor ID as a member of a file descriptor set.
NetSock_Accept()	Wait for new socket connections on a listening server socket.
NetSock_Bind()	Assign network addresses to sockets.
NetSock_CfgBlock()	Configure a socket's blocking mode.
NetSock_CfgConnChildQ_SizeGet()	Get socket's connection child queue size value.
NetSock_CfgConnChildQ_SizeSet()	Configure socket's child connection queue size.

Function Name	Description
NetSock_CfgIF()	Configure the interface that must be used by the socket.
NetSock_CfgRxQ_Size()	Configure socket's receive queue size.
NetSock_CfgSecure()	Configure a socket's secure mode.
NetSock_CfgSecureClientCertKeyInstall()	Install certificate and key that must be used by a client for mutual authentication.
NetSock_CfgSecureClientCommonName()	Configure client socket's common name.
NetSock_CfgSecureClientTrustCallBack()	Configure client socket's trust call back function.
NetSock_CfgSecureServerCertKeyInstall()	Install certificate (CERT) and private key (KEY) from a buffer which must be used by a server.
NetSock_CfgTimeoutConnAcceptDflt()	Set socket's connection accept timeout to configured-default value.
NetSock_CfgTimeoutConnAcceptGet_ms()	Get socket's connection accept timeout value.
NetSock_CfgTimeoutConnAcceptSet()	Set socket's connection accept timeout value.
NetSock_CfgTimeoutConnCloseDflt()	Set socket's connection close timeout to configured-default value.
NetSock_CfgTimeoutConnCloseGet_ms()	Get socket's connection close timeout value.
NetSock_CfgTimeoutConnCloseSet()	Set socket's connection close timeout value.
NetSock_CfgTimeoutConnReqDflt()	Set socket's connection request timeout to configured-default value.
NetSock_CfgTimeoutConnReqGet_ms()	Get socket's connection request timeout value.
NetSock_CfgTimeoutConnReqSet()	Set socket's connection request timeout value.
NetSock_CfgTimeoutRxQ_Dflt()	Set socket's connection receive queue timeout to configured-default value.
NetSock_CfgTimeoutRxQ_Get_ms()	Get socket's receive queue timeout value.
NetSock_CfgTimeoutRxQ_Set()	Set socket's connection receive queue timeout value.
NetSock_CfgTimeoutTxQ_Dflt()	Set socket's connection transmit queue timeout to configured-default value.
NetSock_CfgTimeoutTxQ_Get_ms()	Get socket's transmit queue timeout value.
NetSock_CfgTimeoutTxQ_Set()	Set socket's connection transmit queue timeout value.
NetSock_CfgTxIP_TOS() (IPv4 only)	Configure socket's transmit IPv4 Type of Service (TOS).
NetSock_CfgTxIP_TTL_Multicast() (IPv4 only)	Configure socket's transmit IPv4 multicast Time to live (TTL).
NetSock_CfgTxIP_TTL() (IPv4 only)	Configure socket's transmit IPv4 Time to Live (TTL).
NetSock_CfgTxQ_Size()	Configure socket's transmit queue size.
NetSock_Close()	Terminate communication and free a socket.
NetSock_Conn()	Connect a local socket to a remote socket address.
NetSock_GetConnTransportID()	Gets a socket's transport layer connection handle ID (e.g., TCP connection ID) if available.
NetSock_GetLocalIPAddr()	Gets the local IP address used in the socket connection.
NetSock_IsConn()	Check if a socket is connected to a remote socket.
NetSock_Listen()	Set a socket to accept incoming connections.
NetSock_Open()	Create a datagram (i.e., UDP) or stream (i.e., TCP) type socket.
NetSock_OptGet()	Get the specified socket option from the sock_id socket.
NetSock_OptSet()	Set the specified socket option to the sock_id socket.
NetSock_PoolStatGet()	Get Network Sockets' statistics pool.
NetSock_PoolStatResetMaxUsed()	Reset Network Sockets' statistics pool's maximum number of entries used.

Function Name	Description
NetSock_RxData() / NetSock_RxDataFrom()	Copy up to a specified number of bytes received from a remote socket into an application memory buffer.
NetSock_Sel()	Check if any sockets are ready for available read or write operations or error conditions.
NetSock_SelAbort()	Abort any tasks that are pending on a socket using the select functionality.
NetSock_TxData() / NetSock_TxDataTo()	Copy bytes from an application memory buffer into a socket to send to a remote socket.

BSD Sockets Functions

Function Name	Description
accept()	Wait for new socket connections on a listening server socket.
bind()	Assign network addresses to sockets.
close()	Terminate communication and free a socket.
connect()	Connect a local socket to a remote socket address.
FD_CLR()	Remove a socket file descriptor ID as a member of a file descriptor set.
FD_ISSET()	Check if a socket file descriptor ID is a member of a file descriptor set.
FD_SET()	Add a socket file descriptor ID as a member of a file descriptor set.
FD_ZERO()	Initialize/zero-clear a file descriptor set.
getsockopt()	Get a specific option value on a specific TCP socket.
htonl()	Convert 32-bit integer values from CPU host-order to network-order.
htons()	Convert 16-bit integer values from CPU host-order to network-order.
inet_addr()	Convert a string of an IPv4 address in dotted-decimal notation to an IPv4 address in host-order.
inet_aton()	Convert an IPv4 address in ASCII dotted-decimal notation to a network protocol IPv4 address in network-order.
inet_ntoa()	Convert an IPv4 address in host-order into an IPv4 dotted-decimal notation ASCII strin
inet_ntop()	Converts an IPv4 or IPv6 Internet network address into a string in Internet standard format.
inet_pton()	Converts an IPv4 or IPv6 Internet network address in its standard text presentation form into its numeric binary form.
getaddrinfo()	Converts human-readable text strings representing hostnames or IP addresses into a dynamically allocated linked list of struct addrinfo structures.
freeaddrinfo()	Frees addrinfo structures information that getaddrinfo() has allocated.
listen()	Set a socket to accept incoming connections.
ntohl()	Convert 32-bit integer values from network-order to CPU host-order.
ntohs()	Convert 16-bit integer values from network-order to CPU host-order.
recv() / recvfrom()	Copy up to a specified number of bytes received from a remote socket into an application memory buffer.
select()	Check if any sockets are ready for available read or write operations or error conditions.
send() / sentto()	Copy bytes from an application memory buffer into a socket to send to a remote socket.
setsockopt()	Set a specific option on a specific TCP socket.
socket()	Create a datagram (i.e., UDP) or stream (i.e., TCP) type socket.

Network Application Interface

Function Name	Description
Function Name	Description
NetApp_ClientDatagramOpen()	Open a UDP datagram using IPv4 or IPv6 address.
NetApp_ClientDatagramOpenByHostname()	Open a UDP datagram to a server using the server's hostname (select remote address using DNS)
NetApp_ClientStreamOpen()	Open and connect a TCP Stream to a server
NetApp_ClientStreamOpenByHostname()	Open and connect a TCP Stream to a server using the server's hostname (select remote address using DNS)
NetApp_SockOpen()	Open an application socket.
NetApp_SockClose()	Close an application socket.
NetApp_SockBind()	Bind an application socket to a local address.
NetApp_SockConn()	Connect an application socket to a remote address.
NetApp_SockListen()	Set an application socket to listen for connection requests.
NetApp_SockAccept()	Return a new application socket accepted from a listen application socket.
NetApp_SockRx()	Receive application data via socket.
NetApp_SockTx()	Transmit application data via socket.
NetApp_SetSockAddr()	Setup a socket address from an IPv4 or an IPv6 address.
NetApp_TimeDly_ms()	Delay for specified time, in milliseconds.

Network Initialization API

Function Name	Description
Net_Init()	Initializes Network stack
Net_ConfigureCoreTaskStk()	Configures the Network core task's stack.
Net_ConfigureCoreSvcTaskStk()	Configures the Network service task's stack. The service task is used for core modules like DHCP.
Net_ConfigureMemSeg()	Configures the memory segment to use when allocating control data for the network core.
Net_ConfigureDNS_Client()	Configure the DNS Client module parameters.
Net_CoreTaskPrioSet()	Assigns a new priority to the Network core task.
Net_CoreSvcTaskPrioSet()	Assigns a new priority to the Network core service task.

Net_Init()

Description

Initializes Network stack.

Files

`net.h/net.c`

Prototype

```
void Net_Init (RTOS_ERR  *p_err)
```

Arguments

p_err

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_OS_ILLEGAL_RUN_TIME
- RTOS_ERR_SEG_OVF
- RTOS_ERR_BLK_ALLOC_CALLBACK
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_NOT_AVAIL

Returned Value

None.

Notes / Warnings

1. Net_Init() must be called:
   (a) Only once from a product's application.
   (b) After product's OS has been initialized.

Net_ConfigureCoreTaskStk()

Description

Configures the Network core task's stack.

Files

net.h/net.c

Prototype

```
void  Net_ConfigureCoreTaskStk (CPU_INT32U   stk_size_elements,
                                void        *p_stk)
```

Arguments

stk_size_elements

Size, in stack elements, of the task's stack.

p_stk

Pointer to base of the task's stack. If DEF_NULL , stack will be allocated from Common's memory segment.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the Network module is initialized via the Net_Init() function.
3. In order to change the priority of the Network core task, use the function Net_CoreTaskPrioSet() .

Net_ConfigureCoreSvcTaskStk()

Description

Configures the Network service task's stack. The service task is used for core modules like DHCP.

Files

`net.h/net.c`

Prototype

```
void  Net_ConfigureCoreSvcTaskStk (CPU_INT32U   stk_size_elements,
                                   void        *p_stk)
```

Arguments

`stk_size_elements`

Size, in stack elements, of the task's stack.

`p_stk`

Pointer to base of the task's stack. If `DEF_NULL` , stack will be allocated from Common's memory segment.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the Network module is initialized via the `Net_Init()` function.
3. In order to change the priority of the Network core task, use the function `Net_SvcTaskPrioSet()` .

Net_ConfigureMemSeg()

Description

Configures the memory segment to use when allocating control data for the network core.

Files

`net.h/net.c`

Prototype

```
void  Net_ConfigureMemSeg (MEM_SEG  *p_mem_seg)
```

Arguments

`p_mem_seg`

Pointer to memory segment to use when allocating control data. DEF_NULL means general purpose heap segment.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the Network module is initialized via the `Net_Init()` function.

Net_ConfigureDNS_Client()

Description

Configure the DNS Client module parameters.

Files

`net.h/net.c`

Prototype

```
void Net_ConfigureDNS_Client(DNSc_CFG  *p_cfg)
```

Arguments

`p_cfg`

Pointer to the structure containing the new DNS client parameters.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the Network module is initialized via the `Net_Init()` function.

Net_CoreTaskPrioSet()

Description

Assigns a new priority to the Network core task.

Files

`net.h/net.c`

Prototype

```
void Net_CoreTaskPrioSet(CPU_INT08U   prio,
                         RTOS_ERR    *p_err)
```

Arguments

`prio`

New priority of the the Network core task.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_ARG

Returned Value

None.

Notes / Warnings

1. This function cannot be called before the Network module has been initialized via the Net_Init() function.

Net_CoreSvcTaskPrioSet()

Description

Assigns a new priority to the Network core service task.

Files

net.h/net.c

Prototype

```
void Net_CoreSvcTaskPrioSet(CPU_INT08U  prio,
                            RTOS_ERR   *p_err)
```

Arguments

prio

New priority of the the Network core service task.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_ARG

Returned Value

None.

Notes / Warnings

1. This function cannot be called before the Network module has been initialized via the Net_Init() function.

General Network Utilities API

Function Name	Description
NET_UTIL_HOST_TO_NET_16()	Convert 16-bit integer values from CPU host-order to network-order.
NET_UTIL_HOST_TO_NET_32()	Convert 32-bit integer values from CPU host-order to network-order.
NET_UTIL_NET_TO_HOST_16()	Convert 16-bit integer values from network-order to CPU host- order.
NET_UTIL_NET_TO_HOST_32()	Convert 32-bit integer values from network-order to CPU host- order.

Function Name	Description
NetUtil_TS_Get_ms()	Get current millisecond timestamp.
NetUtil_TS_Get()	Get the current Internet Timestamp.

NetUtil_TS_Get()

Description

Gets the current Internet Timestamp.

Files

`net_util.h/net_util.c`

Prototype

```
NET_TS  NetUtil_TS_Get (void)
```

Arguments

Returned Value

Internet Timestamp.

Notes / Warnings

1. "The Timestamp is a right-justified, 32-bit timestamp in milliseconds since midnight UT [Universal Time]" (RFC #791, Section 3.1 'Options : Internet Timestamp').
2. The developer is responsible for providing a real-time clock with correct time-zone configuration to implement the Internet Timestamp, if possible.

NetUtil_TS_Get_ms()

Description

Gets the current millisecond timestamp.

Files

`net_util.h/net_util.c`

Prototype

```
NET_TS_MS  NetUtil_TS_Get_ms (void)
```

Arguments

Returned Value

Timestamp (in milliseconds).

Notes / Warnings

1. (a)
    1. Although RFC #2988, Section 4 states that "there is no requirement for the clock granularity G used for computing [TCP] RTT measurements ... experience has shown that finer clock granularities (<= 100 msec) perform somewhat

better than more coarse granularities".

2. (a) RFC #2988, Section 2.4 states that "whenever RTO is computed, if it is less than 1 second then the RTO SHOULD be rounded up to 1 second".
   (b) RFC #1122, Section 4.2.3.1 states that "the recommended ... RTO ... upper bound should be 2*MSL" where RFC #793, Section 3.3 'Sequence Numbers : Knowing When to Keep Quiet' states that "the Maximum Segment Lifetime (MSL) is ... to be 2 minutes".
   (c) Therefore, the developer is responsible for providing a timestamp clock with adequate resolution to satisfy the clock granularity (see Note #1(a)) & adequate range to satisfy the minimum/maximum TCP RTO values (see Note #1(b)). Therefore, the required upper bound is :
   2 * MSL = 2 * 2 minutes = 4 minutes = 240 seconds

(b) Therefore, the developer is responsible for providing a timestamp clock with adequate resolution to satisfy the clock granularity (see Note #1(a)) & adequate range to satisfy the minimum/maximum TCP RTO values (see Note #1(b)).

NET_UTIL_HOST_TO_NET_16()

Convert 16-bit integer values from CPU host-order to network-order.

Files

`net_util.h`

Prototype

```
NET_UTIL_HOST_TO_NET_16(val);
```

Arguments

`val`

16-bit integer data value to convert.

Returned Value

16-bit integer value in network-order.

Required Configuration

None.

Notes / Warnings

For microprocessors that require data access to be aligned to appropriate word boundaries, val and any variable to receive the returned 16-bit integer *must* start on appropriately-aligned CPU addresses. This means that all 16-bit words *must* start on addresses that are multiples of 2 bytes.

NET_UTIL_HOST_TO_NET_32()

Convert 32-bit integer values from CPU host-order to network-order.

Files

`net_util.h`

Prototype

```
NET_UTIL_HOST_TO_NET_32(val);
```

Arguments

`val`

32-bit integer data value to convert.

Returned Value

32-bit integer value in network-order.

Required Configuration

None.

Notes / Warnings

For microprocessors that require data access to be aligned to appropriate word boundaries, val and any variable to receive the returned 32-bit integer *must* start on appropriately-aligned CPU addresses. This means that all 32-bit words *must* start on addresses that are multiples of 4 bytes.

NET_UTIL_NET_TO_HOST_16()

Convert 16-bit integer values from network-order to CPU host- order.

Files

`net_util.h`

Prototype

```
NET_UTIL_NET_TO_HOST_16(val);
```

Arguments

`val`

16-bit integer data value to convert.

Returned Value

16-bit integer value in CPU host-order.

Required Configuration

None.

Notes / Warnings

For microprocessors that require data access to be aligned to appropriate word boundaries, val and any variable to receive the returned 16-bit integer *must* start on appropriately-aligned CPU addresses. This means that all 16-bit words *must* start on addresses that are multiples of 2 bytes.

NET_UTIL_NET_TO_HOST_32()

Convert 32-bit integer values from network-order to CPU host- order.

Files

`net_util.h`

Prototype

```
NET_UTIL_NET_TO_HOST_32(val);
```

Arguments

`val`

32-bit integer data value to convert.

Returned Value

32-bit integer value in CPU host-order.

Required Configuration

None.

Notes / Warnings

For microprocessors that require data access to be aligned to appropriate word boundaries, val and any variable to receive the returned 32-bit integer *must* start on appropriately-aligned CPU addresses. This means that all 32-bit words *must* start on addresses that are multiples of 4 bytes.

ASCII Functions API

Function Name	Description
NetASCII_IPv4_to_Str()	Convert an IPv4 address in host-order into an IPv4 dotted-decimal notation ASCII string.
NetASCII_IPv6_to_Str()	Convert an IPv6 address into an IPv6 colon-decimal notation ASCII string.
NetASCII_MAC_to_Str()	Convert a Media Access Control (MAC) address into a hexadecimal address string.
NetASCII_Str_to_IP()	Convert a string of an IPv4 or IPv6 address in their respective decimal notation to an IPv4 or IPv6 address.
NetASCII_Str_to_IPv4()	Convert a string of an IPv4 address in dotted-decimal notation to an IPv4 address in host-order.
NetASCII_Str_to_IPv6()	Convert a string of an IPv6 address in common-decimal notation to an IPv6 address.
NetASCII_Str_to_MAC()	Convert an hexadecimal address string to a Media Access Control (MAC) address.

NetASCII_IPv4_to_Str()

Description

Converts a network protocol IPv4 address in host-order into an IPv4 address ASCII string in dotted-decimal notation.

Files

`net_ascii.h/net_ascii.c`

Prototype

```
void  NetASCII_IPv4_to_Str (NET_IPv4_ADDR   addr_ip,
                            CPU_CHAR       *p_addr_ip_ascii,
                            CPU_BOOLEAN     lead_zeros,
                            RTOS_ERR       *p_err)
```

Arguments

`addr_ip`

IPv4 address.

`p_addr_ip_ascii`

Pointer to an ASCII character array that will receive the return IPv4.

`lead_zeros`

- `DEF_NO` Do NOT prepend leading zeros to each decimal octet value.
- `DEF_YES` Prepend leading zeros to each decimal octet value.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

1. (a)
    1. RFC #1983 states that "dotted decimal notation ... refers [to] IPv4 addresses of the form A.B.C.D; where each letter represents (in decimal) one byte of a four byte IP address."
    2. In other words, the dotted-decimal IP address notation separates four decimal octet values by the dot, or period, character ('.'). Each decimal value represents one octet of the IP address starting with the most significant octet in network-order.
       IP Address Examples : 192.168.1.64 = 0xC0A80140

   (b)
    1. The return dotted-decimal IPv4 address ASCII string formats EXACTLY four decimal values separated by EXACTLY three dot characters and terminated with the NULL character.
    2. The size of the ASCII character array that will receive the returned IP address ASCII string SHOULD be greater than or equal to `NET_ASCII_LEN_MAX_ADDR_IP`.
2. (a) Leading zeros option prepends leading '0's prior to the first non-zero digit in each decimal octet value. The number of leading zeros is such that the decimal octet's number of decimal digits is equal to the maximum number of digits (3).
   (b) If leading zeros option DISABLED and the decimal value of the octet is zero, then one digit of '0' value is formatted.

NetASCII_IPv6_to_Str()

Description

Converts a network protocol IPv6 address in host-order into an IPv6 address ASCII string in dotted-decimal notation.

Files

`net_ascii.h/net_ascii.c`

Prototype

```
void  NetASCII_IPv6_to_Str (NET_IPv6_ADDR  *p_addr_ip,
                            CPU_CHAR       *p_addr_ip_ascii,
                            CPU_BOOLEAN     hex_lower_case,
                            CPU_BOOLEAN     lead_zeros,
                            RTOS_ERR       *p_err)
```

Arguments

`p_addr_ip`

Pointer to IPv6 address.

`p_addr_ip_ascii`

Pointer to an ASCII character array that will receive the return IPv6 address.

`hex_lower_case`

- `DEF_YES` , hexadecimal value will be in lower case
- `DEF_NO` , otherwise.

`lead_zeros`

Prepend leading zeros option (see Note #2) :

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

None.

NetASCII_MAC_to_Str()

Description

Converts an Ethernet MAC address into an Ethernet MAC address ASCII string.

Files

`net_ascii.h/net_ascii.c`

Prototype

```
void  NetASCII_MAC_to_Str (CPU_INT08U   *p_addr_mac,
                           CPU_CHAR     *p_addr_mac_ascii,
                           CPU_BOOLEAN   hex_lower_case,
                           CPU_BOOLEAN   hex_colon_sep,
                           RTOS_ERR     *p_err)
```

Arguments

`p_addr_mac`

Pointer to a memory buffer that contains the MAC address.

`p_addr_mac_ascii`

Pointer to an ASCII character array that will receive the return MAC

`hex_lower_case`

- `DEF_NO` Format alphabetic hexadecimal characters in upper case.
- `DEF_YES` Format alphabetic hexadecimal characters in lower case.

`hex_colon_sep`

- `DEF_NO` Separate hexadecimal values with a hyphen character.
- `DEF_YES` Separate hexadecimal values with a colon character.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

1. (a)
    1. RFC #1700, Section 'ETHERNET VENDOR ADDRESS COMPONENTS' states that "Ethernet addresses ... should be written hyphenated by octets (e.g., 12-34-56-78-9A-BC)".
    2. In other words, the (Ethernet) MAC address notation separates six hexadecimal octet values by the hyphen character ('-') or by the colon character (':'). Each hexadecimal value represents one octet of the MAC address starting with the most significant octet in network-order.

   (b)
    1. The return MAC address ASCII string formats EXACTLY six hexadecimal values separated by EXACTLY five hyphen characters or colon characters and terminated with the NULL character.
    2. The size of the ASCII character array that will receive the returned MAC address ASCII string MUST be greater than or equal to `NET_ASCII_LEN_MAX_ADDR_MAC`.
2. (a) The size of the memory buffer that contains the MAC address SHOULD be greater than or equal to `NET_ASCII_NBR_OCTET_ADDR_MAC`.

NetASCII_Str_to_IP()

Description

Converts a string representation of an IP address (IPv4 or IPv6) to its TCP/IP stack intern representation.

Files

`net_ascii.h/net_ascii.c`

Prototype

```
NET_IP_ADDR_FAMILY  NetASCII_Str_to_IP (CPU_CHAR    *p_addr_ip_ascii,
                                        void        *p_addr,
                                        CPU_INT08U   addr_max_len,
                                        RTOS_ERR    *p_err)
```

Arguments

`p_addr_ip_ascii`

Pointer to an ASCII string that contains a decimal IP address.

`p_addr`

Pointer to the variable that will received the converted IP address.

`addr_max_len`

Size of the variable that will received the converted IP address:

- `NET_IPv4_ADDR_SIZE`
- `NET_IPv6_ADDR_SIZE`

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NET_STR_ADDR_INVALID`

Returned Value

The IP family of the converted address, if no errors:

- `NET_IP_ADDR_FAMILY_IPv4`
- `NET_IP_ADDR_FAMILY_IPv6`

Otherwise,

`NET_IP_ADDR_FAMILY_UNKNOWN`

Notes / Warnings

None.

NetASCII_Str_to_IPv4()

Description

Converts an IPv4 address ASCII string in dotted-decimal notation to a network protocol IPv4 address in host-order.

Files

`net_ascii.h/net_ascii.c`

Prototype

```
NET_IPv4_ADDR  NetASCII_Str_to_IPv4 (CPU_CHAR  *p_addr_ip_ascii,
                                     RTOS_ERR  *p_err)
```

Arguments

`p_addr_ip_ascii`

Pointer to an ASCII string that contains a dotted-decimal IPv4 address.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE

Returned Value

Host-order IPv4 address represented by ASCII string, if NO error(s). NET_IPv4_ADDR_NONE , otherwise.

Notes / Warnings

1. (a)
    1. RFC #1983 states that "dotted decimal notation ... refers [to] IP addresses of the form A.B.C.D; where each letter represents, in decimal, one byte of a four byte IP address".
    2. In other words, the dotted-decimal IP address notation separates four decimal octet values by the dot (or period) character ('.'). Each decimal value represents one octet of the IP address starting with the most significant octet in network-order.
       IP Address Examples : 192.168.1.64 = 0xC0A80140

   (b) Therefore, the dotted-decimal IP address ASCII string MUST :
    1. Include ONLY decimal values and the dot, or period, character ('.') ; ALL other characters are trapped as invalid, including any leading or trailing characters.
    2. Include UP TO four decimal values separated by UP TO three dot characters and MUST be terminated with the NULL character.
    3. Ensure that each decimal value's number of decimal digits, including leading zeros, does NOT exceed the maximum number of digits(10).
       (A) However, any decimal value's number of decimal digits, including leading zeros, MAY be less than the maximum number of digits.
    4. Ensure that each decimal value does NOT exceed the maximum value for its form:
        1. a.b.c.d - 255.255.255.255
        2. a.b.c - 255.255.65535
        3. a.b - 255.16777215
        4. a - 4294967295
2. To avoid possible integer arithmetic overflow, the IP address octet arithmetic result MUST be declared as an integer data type with a greater resolution (i.e., greater number of bits) than the IP address octet data type(s).

NetASCII_Str_to_IPv6()

Description

Converts an IPv6 address ASCII string in common-decimal notation to a network protocol IPv6 address in host-order.

Files

net_ascii.h/net_ascii.c

Prototype

```
NET_IPv6_ADDR  NetASCII_Str_to_IPv6 (CPU_CHAR  *p_addr_ip_ascii,
                                     RTOS_ERR  *p_err)
```

Arguments

p_addr_ip_ascii

Pointer to an ASCII string that contains a common-decimal IPv6 address.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NET_STR_ADDR_INVALID`

Returned Value

Host-order IPv6 address represented by ASCII string, if NO error(s). `NET_IPv6_ADDR_NONE` , otherwise.

Notes / Warnings

None.

NetASCII_Str_to_MAC()

Description

Converts an Ethernet MAC address ASCII string to an Ethernet MAC address.

Files

`net_ascii.h/net_ascii.c`

Prototype

```
void  NetASCII_Str_to_MAC (CPU_CHAR    *p_addr_mac_ascii,
                           CPU_INT08U  *p_addr_mac,
                           RTOS_ERR    *p_err)
```

Arguments

`p_addr_mac_ascii`

Pointer to an ASCII string that contains a MAC address.

`p_addr_mac`

Pointer to a memory buffer that will receive the converted MAC address.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NET_STR_ADDR_INVALID`

Returned Value

None.

Notes / Warnings

1. (a)
    1. RFC #1700, Section 'ETHERNET VENDOR ADDRESS COMPONENTS' states that "Ethernet addresses ... should be written hyphenated by octets (e.g., 12-34-56-78-9A-BC)".
    2. In other words, the (Ethernet) MAC address notation separates six hexadecimal octet values by the hyphen character ('-') or by the colon character (':'). Each hexadecimal value represents one octet of the MAC address starting with the most significant octet in network-order.

(b) Therefore, the MAC address ASCII string MUST :
1. Include ONLY hexadecimal values and the hyphen character ('-') or the colon character (':') ; ALL other characters are trapped as invalid, including any leading or trailing characters.
2. Include EXACTLY six hexadecimal values separated by EXACTLY five hyphen characters or colon characters and MUST be terminated with the NULL character.
3. Ensure that each hexadecimal value's number of digits does NOT exceed the maximum number of digits (2.).
(A) However, any hexadecimal value's number of digits MAY be less than the maximum number of digits.

2. (a) The size of the memory buffer that will receive the converted MAC address MUST be greater than or equal to `NET_ASCII_NBR_OCTET_ADDR_MAC` .
(b) MAC address memory buffer array accessed by octets.
(c) MAC address memory buffer array cleared in case of any error(s).

Network Buffer API

Function Name	Description
NetBuf_PoolStatGet()	Get an interface's Network Buffers' statistics pool.
NetBuf_PoolStatResetMaxUsed()	Reset an interface's Network Buffers' statistics pool's maximum number of entries used.
NetBuf_RxLargePoolStatGet()	Get an interface's large receive buffers' statistics pool.
NetBuf_RxLargePoolStatResetMaxUsed()	Reset an interface's large receive buffers' statistics pool's maximum number of entries used.
NetBuf_TxLargePoolStatGet()	Get an interface's large transmit buffers' statistics pool.
NetBuf_TxLargePoolStatResetMaxUsed()	Reset an interface's large transmit buffers' statistics pool's maximum number of entries used.
NetBuf_TxSmallPoolStatGet()	Get an interface's small transmit buffers' statistics pool.
NetBuf_TxSmallPoolStatResetMaxUsed()	Reset an interface's small transmit buffers' statistics pool's maximum number of entries used.

NetBuf_PoolStatGet()

Description

Gets the network buffer statistics pool.

Files

`net_buf.h/net_buf.c`

Prototype

```
NET_STAT_POOL  NetBuf_PoolStatGet (NET_IF_NBR  if_nbr)
```

Arguments

`if_nbr`

Interface number to get network buffer statistics.

Returned Value

- Network buffer statistics pool, if NO error(s).
- NULL statistics pool, otherwise.

Notes / Warnings

None.

NetBuf_PoolStatResetMaxUsed()

Description

Resets the network buffer statistics pool's maximum number of used entries.

Files

`net_buf.h/net_buf.c`

Prototype

```
void NetBuf_PoolStatResetMaxUsed (NET_IF_NBR if_nbr)
```

Arguments

`if_nbr`

Interface number to reset network buffer statistics.

Returned Value

None.

Notes / Warnings

None.

NetBuf_RxLargePoolStatGet()

Description

Gets the large receive network buffer statistics pool.

Files

`net_buf.h/net_buf.c`

Prototype

```
NET_STAT_POOL NetBuf_RxLargePoolStatGet (NET_IF_NBR if_nbr)
```

Arguments

`if_nbr`

Interface number to get network buffer statistics.

Returned Value

Large receive network buffer statistics pool, if NO error(s). NULL statistics pool, otherwise.

Notes / Warnings

None.

NetBuf_RxLargePoolStatResetMaxUsed()

Description

Resets the large receive network buffer statistics pool's maximum number of entries used.

Files

`net_buf.h/net_buf.c`

Prototype

```
void NetBuf_RxLargePoolStatResetMaxUsed (NET_IF_NBR if_nbr)
```

Arguments

`if_nbr`

Interface number to reset network buffer statistics.

Returned Value

None.

Notes / Warnings

None.

NetBuf_TxLargePoolStatGet()

Description

Gets the large transmit network buffer statistics pool.

Files

`net_buf.h/net_buf.c`

Prototype

```
NET_STAT_POOL NetBuf_TxLargePoolStatGet (NET_IF_NBR if_nbr)
```

Arguments

`if_nbr`

Interface number to get network buffer statistics.

Returned Value

- Large transmit network buffer statistics pool, if NO error(s).
- NULL statistics pool, otherwise.

Notes / Warnings

None.

NetBuf_TxLargePoolStatResetMaxUsed()

Description

Resets the large receive network buffer statistics pool's maximum number of entries used.

Files

`net_buf.h/net_buf.c`

Prototype

```
void NetBuf_TxLargePoolStatResetMaxUsed (NET_IF_NBR if_nbr)
```

Arguments

`if_nbr`

Interface number to reset network buffer statistics.

Returned Value

None.

Notes / Warnings

None.

NetBuf_TxSmallPoolStatGet()

Description

Gets the small transmit network buffer statistics pool.

Files

`net_buf.h/net_buf.c`

Prototype

```
NET_STAT_POOL NetBuf_TxSmallPoolStatGet (NET_IF_NBR if_nbr)
```

Arguments

`if_nbr`

Interface number to get network buffer statistics.

Returned Value

- Small transmit network buffer statistics pool, if NO error(s).
- NULL statistics pool, otherwise.

Notes / Warnings

None.

NetBuf_TxSmallPoolStatResetMaxUsed()

Description

Resets the small transmit network buffer statistics pool's maximum number of entries used.

Files

`net_buf.h/net_buf.c`

Prototype

```
void NetBuf_TxSmallPoolStatResetMaxUsed (NET_IF_NBR if_nbr)
```

Arguments

`if_nbr`

Interface number to reset network buffer statistics.

Returned Value

None.

Notes / Warnings

None.

Network Connection API

Function Name	Description
NetConn_CfgAccessedTh()	Configure network connection access promotion threshold.
NetConn_PoolStatGet()	Get Network Connections' statistics pool.
NetConn_PoolStatResetMaxUsed()	Reset Network Connections' statistics pool's maximum number of entries used.

NetConn_CfgAccessedTh()

Description

Configures the network connection access promotion threshold.

Files

`net_conn.h/net_conn.c`

Prototype

```
CPU_BOOLEAN NetConn_CfgAccessedTh (CPU_INT16U nbr_access)
```

Arguments

`nbr_access`

Desired number of accesses before network connection is promoted.

Returned Value

- `DEF_OK` , network connection access promotion threshold configured.
- `DEF_FAIL` , otherwise.

Notes / Warnings

None.

NetConn_PoolStatGet()

Description

Gets the network connection statistics pool.

Files

`net_conn.h/net_conn.c`

Prototype

```
NET_STAT_POOL  NetConn_PoolStatGet (void)
```

Arguments

Returned Value

- Network connection statistics pool, if NO error(s).
- NULL statistics pool, otherwise.

Notes / Warnings

None.

NetConn_PoolStatResetMaxUsed()

Description

Resets the network connection statistics pool's maximum number of entries used.

Files

`net_conn.h/net_conn.c`

Prototype

```
void  NetConn_PoolStatResetMaxUsed (void)
```

Arguments

Returned Value

None.

Notes / Warnings

None.

Network Timer API

Function Name	Description
NetTmr_PoolStatGet()	Gets the network timer statistics pool.
NetTmr_PoolStatResetMaxUsed()	Resets the network timer statistics pool's maximum number of entries used.

NetTmr_PoolStatGet()

Description

Gets the network timer statistics pool.

Files

net_tmr.h/net_tmr.c

Prototype

```
NET_STAT_POOL  NetTmr_PoolStatGet (void)
```

Arguments

Returned Value

Network timer statistics pool, if NO error(s). NULL statistics pool, otherwise.

Notes / Warnings

1. NetTmr_PoolStatGet() blocked until network initialization completes.
2. ' NetTmr_PoolStat ' MUST ALWAYS be accessed exclusively in critical sections.

NetTmr_PoolStatResetMaxUsed()

Description

Resets the network timer statistics pool's maximum number of entries used.

Files

net_tmr.h/net_tmr.c

Prototype

```
void  NetTmr_PoolStatResetMaxUsed (void)
```

Arguments

Returned Value

None.

Notes / Warnings

1. NetTmr_PoolStatResetMaxUsed() blocked until network initialization completes.
   However, since ' NetTmr_PoolStat ' is reset when network initialization completes; NO error is returned.

Network Interface API

Function Name	Description
NetIF_Add()	Add a network device and hardware as a network interface.
NetIF_AddrHW_Get()	Get network interface's hardware address.
NetIF_AddrHW_IsValid()	Validate a network interface hardware address.
NetIF_AddrHW_Set()	Set network interface's hardware address.
NetIF_CfgPerfMonPeriod()	Configure the network interface Performance Monitor Handler timeout.
NetIF_CfgPhyLinkPeriod()	Configure network interface Physical Link State Handler timeout.
NetIF_GetExtAvailCtr()	Returns the number of external interface configured.
NetIF_GetNbrBaseCfgd()	Gets the interface base number (first interface ID).
NetIF_GetRxDataAlignPtr()	Get an aligned pointer into a receive application data buffer.
NetIF_GetTxDataAlignPtr()	Get an aligned pointer into a transmit application data buffer.
NetIF_TypeGet()	Get the network interface type.
NetIF_IO_Ctrl()	Handle network interface and/or device specific (I/O) control(s).
NetIF_IsEn()	Validate network interface as enabled.
NetIF_IsEnCfgd()	Validate configured network interface as enabled.
NetIF_ISR_Handler()	Handle a network interface's device interrupts.
NetIF_IsValid()	Validate network interface number.
NetIF_IsValidCfgd()	Validate configured network interface number.
NetIF_LinkStateGet()	Get network interface's last known physical link state.
NetIF_LinkStateSubscribe()	Subscribe to get notified when an interface link state changes.
NetIF_LinkStateUnsubscribe()	Unsubscribe to get notified when interface link state changes.
NetIF_LinkStateWaitUntilUp()	Wait for a network interface's physical link state to be UP .
NetIF_MTU_Get()	Get network interface's MTU.
NetIF_MTU_Set()	Set network interface's MTU.
NetIF_Start()	Start a network interface.
NetIF_Stop()	Stop a network interface.
NetIF_TxSuspendTimeoutGet_ms()	Gets the network interface transmit suspend timeout value.
NetIF_TxSuspendTimeoutSet()	Sets the network interface transmit suspend timeout value.
NetIF_WaitSetupReady()	Wait for the network interface setup to be complete.

NetIF_Add()

This function is DEPRECATED and will be removed in a future version of this product. Instead, use NetIF_Ether_Add() or NetIF_WiFi_Add().

Description

Add & initialize a specific instance of a network interface.

Files

net_if.h/net_if.c

Prototype

```
NET_IF_NBR  NetIF_Add (void      *p_if_api,
                       void      *p_dev_api,
                       void      *p_dev_bsp,
```

```
            void       *p_dev_api,
            void       *p_dev_bsp,
            void       *p_dev_cfg,
            void       *p_ext_api,
            void       *p_ext_cfg,
            RTOS_ERR  *p_err)
```

Arguments

`p_if_api`

Pointer to specific network interface API.

`p_dev_api`

Pointer to specific network device driver API.

`p_dev_bsp`

Pointer to specific network device board-specific API.

`p_dev_cfg`

Pointer to specific network device hardware configuration.

`p_ext_api`

Pointer to specific network extension layer API.

`p_ext_cfg`

Pointer to specific network extension layer configuration.

`p_err`

Pointer to variable that will receive the return error code from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_NO_MORE_RSRC`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`

Returned Value

- Interface number of the added interface, if NO error(s).
- `NET_IF_NBR_NONE` , otherwise.

Notes / Warnings

The first network interface added and started is the default interface used for all default communication.

When a PHY layer is present, the PHY API and configuration need to be indicate in the `p_ext_api` and `p_ext_cfg` arguments.

NetIF_AddrHW_Get()

Description

Gets the network interface's hardware address.

Files

`net_if.h/net_if.c`

Prototype

```
void  NetIF_AddrHW_Get (NET_IF_NBR   if_nbr,
                        CPU_INT08U  *p_addr_hw,
                        CPU_INT08U  *p_addr_len,
                        RTOS_ERR    *p_err)
```

Arguments

`if_nbr`

Network interface number to get the hardware address.

`p_addr_hw`

Pointer to a variable that will receive the hardware address.

`p_addr_len`

Pointer to a variable that will receive the address length.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

None.

Notes / Warnings

The hardware address is returned in network-order; i.e., the pointer to the hardware address points to the highest-order octet.

NetIF_AddrHW_IsValid()

Description

Validates the network interface's hardware address.

Files

`net_if.h/net_if.c`

Prototype

```
CPU_BOOLEAN  NetIF_AddrHW_IsValid (NET_IF_NBR   if_nbr,
                                   CPU_INT08U  *p_addr_hw,
                                   RTOS_ERR    *p_err)
```

Arguments

`if_nbr`

Interface number to validate the hardware address.

`p_addr_hw`

Pointer to an interface's hardware address.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- `DEF_YES` , if hardware address is valid.
- `DEF_NO` , otherwise.

Notes / Warnings

The hardware address MUST be in network-order; i.e., the pointer to the hardware address MUST point to the highest-order octet.

NetIF_AddrHW_Set()

Description

Sets the network interface's hardware address.

Files

`net_if.h/net_if.c`

Prototype

```
void  NetIF_AddrHW_Set (NET_IF_NBR   if_nbr,
                 CPU_INT08U  *p_addr_hw,
                 CPU_INT08U   addr_len,
                 RTOS_ERR    *p_err)
```

Arguments

`if_nbr`

Network interface number to set the hardware address.

`p_addr_hw`

Pointer to the hardware address.

`addr_len`

Length of the hardware address.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_INVALID_STATE

Returned Value

None.

Notes / Warnings

1. The hardware address MUST be in network-order; i.e., the pointer to the hardware address MUST point to the highest-order octet.
2. The interface MUST be stopped BEFORE setting a new hardware address, which does NOT take effect until the interface is re-started.

NetIF_CfgPerfMonPeriod()

Description

Configures the Network Interface Performance Monitor Handler time (i.e., scheduling period).

Files

`net_if.h/net_if.c`

Prototype

```
CPU_BOOLEAN  NetIF_CfgPerfMonPeriod (CPU_INT16U  time_ms)
```

Arguments

`time_ms`

Desired value for Network Interface Performance Monitor Handler time (in milliseconds).

Returned Value

- `DEF_OK` , Network Interface Performance Monitor Handler time is configured.
- `DEF_FAIL` , otherwise.

Notes / Warnings

Configured time does NOT reschedule the next performance monitor handling; it configures the scheduling of all subsequent performance monitor handling.

NetIF_CfgPhyLinkPeriod()

Description

Configures the Network Interface Physical Link State Handler time (i.e., scheduling period).

Files

`net_if.h/net_if.c`

Prototype

```
CPU_BOOLEAN  NetIF_CfgPhyLinkPeriod (CPU_INT16U  time_ms)
```

Arguments

`time_ms`

Desired value for Network Interface Physical Link State Handler time (in milliseconds).

Returned Value

- `DEF_OK` , Network Interface Physical Link State Handler timeout configured.
- `DEF_FAIL` , otherwise.

Notes / Warnings

Configured time does NOT reschedule the next physical link state handling; it configures the scheduling of all subsequent physical link state handling.

NetIF_GetExtAvailCtr()

Description

Returns the number of external interface configured.

Files

`net_if.h/net_if.c`

Prototype

```
CPU_INT08U  NetIF_GetExtAvailCtr (RTOS_ERR  *p_err)
```

Arguments

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

Number of external interfaces available.

Notes / Warnings

None.

NetIF_GetNbrBaseCfgd()

Description

Gets the interface base number (first interface ID).

Files

`net_if.h/net_if.c`

Prototype

```
NET_IF_NBR  NetIF_GetNbrBaseCfgd (void)
```

Arguments

None.

Returned Value

Interface base number.

Notes / Warnings

None.

NetIF_GetRxDataAlignPtr()

Description

Gets and places aligned pointer into a receive application data buffer.

Files

`net_if.h/net_if.c`

Prototype

```
void  *NetIF_GetRxDataAlignPtr (NET_IF_NBR   if_nbr,
                                void        *p_data,
                                RTOS_ERR    *p_err)
```

Arguments

`if_nbr`

Network interface number to get a receive application buffer's aligned data pointer.

`p_data`

Pointer to receive application data buffer to get an aligned pointer into.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- Pointer to aligned receive application data buffer address, if NO error(s).
- Pointer to NULL, otherwise.

Notes / Warnings

1. (a)
   1. (A) Optimal alignment between application data buffer(s) and network interface's network buffer data area(s) is NOT guaranteed and is only possible if the following condition is true :
      1. Network interface's network buffer data area(s) MUST be aligned to a multiple of the CPU's data word size.

      (B) Otherwise, a single and fixed alignment between application data buffer(s) and network interface's buffer data area(s) is NOT possible.
   2. (A) Even when application data buffers and network buffer data areas are aligned in the best case; optimal alignment is NOT guaranteed for every read/write of data to/from application data buffers and network buffer data areas.
      For any single read/write of data to/from application data buffers and network buffer data areas, optimal alignment only occurs if the following conditions are true :
      1. Data read/written to/from application data buffer(s) to network buffer data area(s) MUST start on addresses with the same relative offset from CPU word-aligned addresses.
         In other words, the modulus of the specific read/write address in the application data buffer with the CPU's data word size MUST be equal to the modulus of the specific read/write address in the network buffer data area with the CPU's data word size.
         This condition MIGHT NOT be satisfied whenever :
         1. Data is read/written to/from fragmented packets
         2. Data is NOT maximally read/written to/from stream-type packets (e.g., TCP data segments)
         3. Packets include variable number of header options (e.g., IP options)

      (B) However, even though optimal alignment between application data buffers and network buffer data areas is NOT guaranteed for every read/write; optimal alignment SHOULD occur more frequently leading to improved network data throughput.

   (b) Since the first aligned address in the application data buffer may be 0 to (`CPU_CFG_DATA_SIZE` - 1) octets after the application data buffer's starting address, the application data buffer SHOULD allocate and reserve an additional (`CPU_CFG_DATA_SIZE` - 1) number of octets.
   However, the application data buffer's effective, usable size is still limited to its original declared size (before reserving additional octets) and SHOULD NOT be increased by the additional, reserved octets.

NetIF_GetTxDataAlignPtr()

Description

Gets and places the aligned pointer into a transmit application data buffer.

Files

`net_if.h/net_if.c`

Prototype

```
void  *NetIF_GetTxDataAlignPtr (NET_IF_NBR   if_nbr,
                                void        *p_data,
                                RTOS_ERR    *p_err)
```

Arguments

`if_nbr`

Network interface number to get a transmit application buffer's aligned data pointer.

`p_data`

Pointer to transmit the application data buffer into which to get an aligned pointer.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- Pointer to aligned transmit application data buffer address, if NO error(s).
- Pointer to NULL, otherwise.

Notes / Warnings

1. (a)
    1. (A) Optimal alignment between application data buffer(s) and network interface's network buffer data area(s) is NOT guaranteed and is only possible if the following condition is true :
        1. Network interface's network buffer data area(s) MUST be aligned to a multiple of the CPU's data word size.

       (B) Otherwise, a single and fixed alignment between application data buffer(s) and network interface's buffer data area(s) is NOT possible.
    2. (A) Even when application data buffers and network buffer data areas are aligned in the best case; optimal alignment is NOT guaranteed for every read/write of data to/from application data buffers and network buffer data areas.
       For any single read/write of data to/from application data buffers and network buffer data areas, optimal alignment only occurs if all of the following conditions are true :
        1. Data read/written to/from application data buffer(s) to network buffer data area(s) MUST start on addresses with the same relative offset from CPU word-aligned addresses.

       In other words, the modulus of the specific read/write address in the application data buffer with the CPU's data word size MUST be equal to the modulus of the specific read/write address in the network buffer data area with the CPU's data word size.
       This condition MIGHT NOT be satisfied whenever :
        1. Data is read/written to/from fragmented packets
        2. Data is NOT maximally read/written to/from stream-type packets (e.g. TCP data segments)
        3. Packets include variable number of header options (e.g. IP options)

       (B) However, even though optimal alignment between application data buffers and network buffer data areas is NOT guaranteed for every read/write; optimal alignment SHOULD occur more frequently leading to improved network data throughput.

   (b) Since the first aligned address in the application data buffer may be 0 to (`CPU_CFG_DATA_SIZE` - 1) octets after the application data buffer's starting address, the application data buffer SHOULD allocate and reserve an additional (`CPU_CFG_DATA_SIZE` - 1) number of octets.

NetIF_IO_Ctrl()

Description

1. Handles the network interface and/or device specific (I/O) control(s) :
   (a) Device link :
    1. Get the device link info
    2. Get the device link state
    3. Update the device link state

Files

`net_if.h/net_if.c`

Prototype

```
void  NetIF_IO_Ctrl (NET_IF_NBR   if_nbr,
                    CPU_INT08U   opt,
                    void        *p_data,
                    RTOS_ERR    *p_err)
```

Arguments

`if_nbr`

Network interface number to handle the (I/O) controls.

`opt`

Desired I/O control option code to perform. Additional control options may be defined by the device driver :

- `NET_IF_IO_CTRL_LINK_STATE_GET` Get the device's current physical link state, 'UP' or 'DOWN'.
- `NET_IF_IO_CTRL_LINK_STATE_GET_INFO` Get the device's detailed physical link state information.
- `NET_IF_IO_CTRL_LINK_STATE_UPDATE` Update the device's current physical link state.
- `NET_IF_IO_CTRL_EEE_GET_INFO` Retrieve information if EEE is enabled or not.
- `NET_IF_IO_CTRL_EEE` Enable/Disable EEE support on the interface.

`p_data`

Pointer to variable that will receive possible I/O control data.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

None.

Notes / Warnings

1. '`p_data`' MUST point to a variable or memory buffer that is sufficiently sized AND aligned to receive any return data. The options are as follows :
   (a) `NET_IF_IO_CTRL_LINK_STATE_GET` :
      1. For Ethernet or Wireless interface: `p_data` MUST point to a `CPU_BOOLEAN` variable.
   (b) `NET_IF_IO_CTRL_LINK_STATE_GET_INFO` `NET_IF_IO_CTRL_LINK_STATE_UPDATE`
      1. For an Ethernet interface: `p_data` MUST point to a variable of data type `NET_DEV_LINK_ETHER`.
      2. For a Wireless interface: `p_data` MUST point to a variable of data type `NET_DEV_LINK_WIFI`.

NetIF_ISR_Handler()

Description

Handle network interface's device interrupt service routine (ISR) function(s).

Files

`net_if.h/net_if.c`

Prototype

```
void  NetIF_ISR_Handler (NET_IF_NBR        if_nbr,
                    NET_DEV_ISR_TYPE   type,
                    RTOS_ERR          *p_err)
```

Arguments

`if_nbr`

Network interface number to handle ISR(s).

`type`

Device interrupt type(s) to handle :

- `NET_DEV_ISR_TYPE_UNKNOWN` Handle unknown device ISR(s).
- `NET_DEV_ISR_TYPE_RX` Handle device receive ISR(s).
- `NET_DEV_ISR_TYPE_RX_OVERRUN` Handle device receive overrun ISR(s).
- `NET_DEV_ISR_TYPE_TX_RDY` Handle device transmit ready ISR(s).
- `NET_DEV_ISR_TYPE_TX_COMPLETE` Handle device transmit complete ISR(s).

`p_err`

Pointer to variable that will receive the return error code from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_INVALID_STATE`

Returned Value

none.

Notes / Warnings

Device driver(s)' Board Support Package (BSP) Interrupt Service Routine (ISR) handler(s). This function is a network interface (IF) to network device function & SHOULD be called only by appropriate network device driver ISR handler function(s).

NetIF_IsEn()

Description

Validates the network interface as enabled.

Files

`net_if.h/net_if.c`

Prototype

```
CPU_BOOLEAN  NetIF_IsEn (NET_IF_NBR   if_nbr,
                    RTOS_ERR    *p_err)
```

Arguments

`if_nbr`

Network interface number to validate.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- `DEF_YES` , network interface is valid and enabled.
- `DEF_NO` , network interface is invalid or disabled.

Notes / Warnings

None.

NetIF_IsEnCfgd()

Description

Validates the configured network interface as enabled.

Files

`net_if.h/net_if.c`

Prototype

```
CPU_BOOLEAN  NetIF_IsEnCfgd (NET_IF_NBR   if_nbr,
                             RTOS_ERR    *p_err)
```

Arguments

`if_nbr`

Network interface number to validate.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- `DEF_YES` , network interface is valid and enabled.
- `DEF_NO` , network interface is invalid or disabled.

Notes / Warnings

None.

NetIF_IsValid()

Description

Validates the network interface number.

Files

`net_if.h/net_if.c`

Prototype

```
CPU_BOOLEAN  NetIF_IsValid (NET_IF_NBR   if_nbr,
                            RTOS_ERR    *p_err)
```

Arguments

`if_nbr`

Network interface number to validate.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

- `DEF_YES` , network interface number is valid.
- `DEF_NO` , network interface number is invalid / NOT yet configured.

Notes / Warnings

None.

NetIF_IsValidCfgd()

Description

Validates the configured network interface number.

Files

`net_if.h/net_if.c`

Prototype

```
CPU_BOOLEAN  NetIF_IsValidCfgd (NET_IF_NBR   if_nbr,
                                RTOS_ERR    *p_err)
```

Arguments

`if_nbr`

Network interface number to validate.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

- `DEF_YES` , network interface number is valid.
- `DEF_NO` , network interface number is invalid / NOT yet configured or reserved.

Notes / Warnings

None.

NetIF_LinkStateGet()

Description

Gets the network interface's last known physical link state.

Files

`net_if.h/net_if.c`

Prototype

```
NET_IF_LINK_STATE  NetIF_LinkStateGet (NET_IF_NBR   if_nbr,
                                       RTOS_ERR    *p_err)
```

Arguments

`if_nbr`

Network interface number to get last known physical link state.

`p_err`

Pointer to variable that will receive the return error code from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- `NET_IF_LINK_UP` , if NO error(s) and network interface's last known physical link state was 'UP'.
- `NET_IF_LINK_DOWN` , otherwise.

Notes / Warnings

`NetIF_LinkStateGet()` only returns a network interface's last known physical link state since enabled network interfaces' physical link states are only periodically updated.

NetIF_LinkStateSubscribe()

Description

Subscribes to get notified when an interface link state changes.

Files

`net_if.h/net_if.c`

Prototype

```
void  NetIF_LinkStateSubscribe (NET_IF_NBR                if_nbr,
                                NET_IF_LINK_SUBSCRIBER_FNCT   fcnt,
                                RTOS_ERR                  *p_err)
```

Arguments

`if_nbr`

Network interface number to check the link state.

`fcnt`

Function to call when the link changes.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`

Returned Value

None.

Notes / Warnings

None.

NetIF_LinkStateUnsubscribe()

Description

Removes the subscribe function from the link state changes subscription list.

Files

`net_if.h/net_if.c`

Prototype

```
void  NetIF_LinkStateUnsubscribe (NET_IF_NBR                if_nbr,
                                  NET_IF_LINK_SUBSCRIBER_FNCT   fcnt,
                                  RTOS_ERR                  *p_err)
```

Arguments

`if_nbr`

Specify the Network Interface number to remove the link state check function.

`fcnt`

Function to remove from subscription list.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_POOL_FULL`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

None.

Notes / Warnings

None.

NetIF_LinkStateWaitUntilUp()

Description

Waits for a network interface's link state to be 'UP'.

Files

`net_if.h/net_if.c`

Prototype

```
CPU_BOOLEAN  NetIF_LinkStateWaitUntilUp (NET_IF_NBR    if_nbr,
                                         CPU_INT16U    retry_max,
                                         CPU_INT32U    time_dly_ms,
                                         RTOS_ERR     *p_err)
```

Arguments

`if_nbr`

Network interface number to check link state.

`retry_max`

Maximum number of consecutive wait retries.

`time_dly_ms`

Transitory delay value, in milliseconds.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_NET_IF_LINK_DOWN`

Returned Value

- `NET_IF_LINK_UP` , if NO error(s) and network interface's link state is 'UP'.

- `NET_IF_LINK_DOWN` , otherwise.

Notes / Warnings

If a non-zero number of retries is requested, a non-zero time delay SHOULD also be requested; otherwise, all retries will likely fail immediately since no time will elapse to allow the network interface's link state to successfully be 'UP'.

NetIF_MTU_Get()

Description

Gets the network interface's MTU.

Files

`net_if.h/net_if.c`

Prototype

```
NET_MTU  NetIF_MTU_Get (NET_IF_NBR   if_nbr,
                        RTOS_ERR    *p_err)
```

Arguments

`if_nbr`

Network interface number to get MTU.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- Network interface's MTU, if NO error(s).
- 0, otherwise.

Notes / Warnings

None.

NetIF_MTU_Set()

Description

Sets the network interface's MTU.

Files

`net_if.h/net_if.c`

Prototype

```
void  NetIF_MTU_Set (NET_IF_NBR   if_nbr,
                     NET_MTU      mtu,
                     RTOS_ERR    *p_err)
```

Arguments

`if_nbr`

Network interface number to set MTU.

`mtu`

Desired maximum transmission unit size to configure.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

None.

Notes / Warnings

None.

NetIF_NbrGetFromName()

Description

Get network interface number from it's name.

Files

`net_if.h/net_if.c`

Prototype

```
NET_IF_NBR  NetIF_NbrGetFromName (CPU_CHAR  *p_name)
```

Arguments

`p_name`

Pointer to a string containing interface controller's name.

Returned Value

Network interface number associated with controller name.

Notes / Warnings

None.

NetIF_Start()

This function is DEPRECATED and will be removed in a future version of this product. Instead, use NetIF_Ether_Start() or NetIF_WiFi_Start() .

Description

Start a network interface.

Files

net_if.h/net_if.c

Prototype

```
void  NetIF_Start (NET_IF_NBR   if_nbr,
                   RTOS_ERR    *p_err)
```

Arguments

if_nbr

Network interface number to start.

p_err

Pointer to variable that will receive the return error code from this function :

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_TYPE
- RTOS_ERR_NOT_INIT
- RTOS_ERR_TX
- RTOS_ERR_ALREADY_EXISTS
- RTOS_ERR_NOT_FOUND
- RTOS_ERR_BLK_ALLOC_CALLBACK
- RTOS_ERR_RX
- RTOS_ERR_SEG_OVF
- RTOS_ERR_NOT_SUPPORTED
- RTOS_ERR_NOT_READY
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_NET_IF_LINK_DOWN
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_NET_NEXT_HOP
- RTOS_ERR_INVALID_STATE
- RTOS_ERR_ABORT
- RTOS_ERR_TIMEOUT

Returned Value

None.

Notes / Warnings

None.

NetIF_Stop()

Description

Stop a network interface.

Files

`net_if.h/net_if.c`

Prototype

```
void NetIF_Stop (NET_IF_NBR  if_nbr,
                 RTOS_ERR    *p_err)
```

Arguments

`if_nbr`

Network interface number to stop.

`p_err`

Pointer to variable that will receive the return error code from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_INVALID_STATE`

Returned Value

none.

Notes / Warnings

None.

NetIF_TxSuspendTimeoutGet_ms()

Description

Gets the network interface transmit suspend timeout value.

Files

`net_if.h/net_if.c`

Prototype

```
CPU_INT32U  NetIF_TxSuspendTimeoutGet_ms (NET_IF_NBR  if_nbr,
                                          RTOS_ERR    *p_err)
```

Arguments

`if_nbr`

Interface number to get timeout value.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_HANDLE

Returned Value

- Network transmit suspend timeout value (in milliseconds), if NO error(s).
- 0, otherwise.

Notes / Warnings

None.

NetIF_TxSuspendTimeoutSet()

Description

Sets the network interface transmit suspend timeout value.

Files

net_if.h/net_if.c

Prototype

```
void  NetIF_TxSuspendTimeoutSet (NET_IF_NBR   if_nbr,
                                 CPU_INT32U   timeout_ms,
                                 RTOS_ERR    *p_err)
```

Arguments

if_nbr

Interface number to set timeout value.

timeout_ms

Timeout value (in milliseconds).

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_HANDLE

Returned Value

None.

Notes / Warnings

None.

NetIF_TypeGet()

Description

Gets the network interface's type.

Files

`net_if.h/net_if.c`

Prototype

```
NET_IF_TYPE  NetIF_TypeGet (NET_IF_NBR   if_nbr,
                            RTOS_ERR    *p_err)
```

Arguments

`if_nbr`

Network interface number to get the type.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

Network interface's type :

- `NET_IF_TYPE_ETHER`
- `NET_IF_TYPE_WIFI`
- `NET_IF_TYPE_NONE`, in case of error

Notes / Warnings

None.

NetIF_WaitSetupReady()

Description

Wait for the network interface setup to be complete.

Files

`net_if.h/net_if.c`

Prototype

```
void  NetIF_WaitSetupReady (NET_IF_NBR        if_nbr,
                            NET_IF_APP_INFO  *p_info,
                            CPU_INT32U        timeout_ms,
                            RTOS_ERR         *p_err)
```

Arguments

`if_nbr`

Network interface number on which to wait.

p_info

Pointer to structure in which the setup information will be copied. DEF_NULL, to not receive the setup information.

timeout_ms

Timeout, in milliseconds. A value of 0 will never timeout.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_ABORT

Returned Value

None.

Notes / Warnings

None.

Ethernet Network Interface API

Macro Name	Description
NET_CTRLR_ETHER_REG()	Registers an Ethernet controller to the platform manager.

Function Name	Description
NetIF_Ether_Add()	Add & initialize a specific instance of a network Ethernet interface.
NetIF_Ether_Start()	Start an Ethernet type interface

NET_CTRLR_ETHER_REG()

Description

Registers a Ethernet controller to the platform manager.

Files

net_if_ether.h

Prototype

```
NET_CTRLR_ETHER_REG(name, ctlr_info_ptr)
```

Arguments

name

Unique name for the Ethernet controller. It is recommended to follow the standard "etherX" where X is a digit.

ctlr_info_ptr

Pointer to the Ethernet controller information structure of type NET_IF_ETHER_CTRLR_INFO .

Returned Value

None.

Notes / Warnings

This macro should normally be called from the BSP.

NetIF_Ether_Add()

Description

Add & initialize a specific instance of a network Ethernet interface.

Files

`net_if_ether.h/net_if_ether.c`

Prototype

```
NET_IF_NBR  NetIF_Ether_Add (CPU_CHAR       *p_name,
                             NET_IF_BUF_CFG  *p_buf_cfg,
                             MEM_SEG        *p_mem_seg,
                             RTOS_ERR       *p_err)
```

Arguments

`p_name`

String identifier for the Ethernet interface to add.

`p_buf_cfg`

Pointer to buffer configuration.

`p_mem_seg`

Memory segment from which internal data will be allocated. If `DEF_NULL`, it will be allocated from the global heap.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_NO_MORE_RSRC`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`

Returned Value

Interface number of the added interface, if NO error(s). `NET_IF_NBR_NONE`, otherwise.

Notes / Warnings

When a non-null buffer configuration is passed, a copy of the device configuration that come from the BSP is completed. It is recommended to reduce the memory usage to modify the static configuration specified in the BSP.

NetIF_Ether_Start()

Description

Start an Ethernet type interface :

Files

`net_if_ether.h/net_if_ether.c`

Prototype

```
void  NetIF_Ether_Start (NET_IF_NBR        if_nbr,
                         NET_IF_ETHER_CFG  *p_cfg,
                         RTOS_ERR          *p_err)
```

Arguments

`if_nbr`

Network interface number to start.

`p_cfg`

Pointer to interface configuration. `DEF_NULL` , to just start the interface without Address or DHCP setup.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_TYPE`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_NOT_SUPPORTED`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`
- `RTOS_ERR_FAIL`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_NET_IF_LINK_DOWN`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_NOT_INIT`
- `RTOS_ERR_TX`
- `RTOS_ERR_ALREADY_EXISTS`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_NET_STR_ADDR_INVALID`
- `RTOS_ERR_RX`
- `RTOS_ERR_NOT_READY`
- `RTOS_ERR_POOL_EMPTY`

- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_NET_NEXT_HOP
- RTOS_ERR_IS_OWNER

Returned Value

None.

Notes / Warnings

None.

Wireless Network Interface API

Macro Name	Description
NET_CTRLR_WIFI_SPI_REG()	Registers an external WiFi controller connected via SPI to the platform manager.

Function Name	Description
NetIF_WiFi_Add()	Add & initialize a specific instance of a network WiFi interface.
NetIF_WiFi_CreateAP()	Create a wireless ad-hoc access point.
NetIF_WiFi_GetPeerInfo()	Gets the peer info connected to the access point (when acting as an access point).
NetIF_WiFi_Join()	Join a wireless access point.
NetIF_WiFi_Leave()	Leave the access point previously joined.
NetIF_WiFi_Scan()	Scan available wireless access point.
NetIF_WiFi_Start()	Start a Wi-Fi-type interface.

NET_CTRLR_WIFI_SPI_REG()

Description

Registers a WiFi controller to the platform manager. Those kind of WiFi Controllers are outside the MCU and connected via a SPI interface to the MCU.

Files

`net_if_wifi.h`

Prototype

```
NET_CTRLR_WIFI_SPI_REG(part_name, spi_name, part_info_ptr, bsp_api_ptr, slave_id)
```

Arguments

`part_name`

Unique name for the WiFi controller/part. It is recommended to follow the standard "wifiX" where X is a digit.

`spi_name`

Unique name for the SPI bus to use for the WiFi slave. The SPI bus must already have been registered to the Platform Manager. It is recommended to follow the standard "spiX" where X is a digit.

`part_info_ptr`

Pointer to the WiFi part hardware information structure of type `NET_IF_WIFI_PART_INFO`.

`bsp_api_ptr`

Pointer to the BSP API structure for the WiFi controller.

`slave_id`

Slave ID of the WiFi controller on the SPI bus.

Returned Value

None.

Notes / Warnings

This macro should normally be called from the BSP.

NetIF_WiFi_Add()

Description

Add & initialize a specific instance of a network WiFi interface.

Files

`net_if_wifi.h/net_if_wifi.c`

Prototype

```
NET_IF_NBR  NetIF_WiFi_Add (       CPU_CHAR       *p_name,
                            const  NET_IF_BUF_CFG  *p_cfg,
                                   void            *p_ext_cfg,
                                   MEM_SEG         *p_mem_seg,
                                   RTOS_ERR        *p_err)
```

Arguments

`p_name`

String identifier for the WiFi interface to add.

`p_cfg`

Pointer to buffer configuration.

`p_ext_cfg`

Pointer to extended configuration. DEF_NULL, if not need by WiFi part.

`p_mem_seg`

Memory segment from which internal data will be allocated. If DEF_NULL, it will be allocated from the global heap.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`
- `RTOS_ERR_POOL_EMPTY`

- RTOS_ERR_NOT_FOUND
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_NO_MORE_RSRC
- RTOS_ERR_BLK_ALLOC_CALLBACK
- RTOS_ERR_SEG_OVF

Returned Value

- Interface number of the added interface, if NO error(s).
- NET_IF_NBR_NONE , otherwise.

Notes / Warnings

None.

NetIF_WiFi_CreateAP()

Description

Creates a wireless access point.

Files

net_if_wifi.h/net_if_wifi.c

Prototype

```
void  NetIF_WiFi_CreateAP (NET_IF_NBR                 if_nbr,
                           NET_IF_WIFI_NET_TYPE       net_type,
                           NET_IF_WIFI_DATA_RATE      data_rate,
                           NET_IF_WIFI_SECURITY_TYPE  security_type,
                           NET_IF_WIFI_PWR_LEVEL      pwr_level,
                           NET_IF_WIFI_CH             ch,
                           NET_IF_WIFI_SSID           ssid,
                           NET_IF_WIFI_PSK            psk,
                           RTOS_ERR                  *p_err)
```

Arguments

if_nbr

Wireless network interface number.

net_type

Wireless network type:

- NET_IF_WIFI_NET_TYPE_INFRASTRUCTURE
- NET_IF_WIFI_NET_TYPE_ADHOC

data_rate

Wireless date rate to configure:

- NET_IF_WIFI_DATA_RATE_AUTO
- NET_IF_WIFI_DATA_RATE_1_MBPS
- NET_IF_WIFI_DATA_RATE_2_MBPS
- NET_IF_WIFI_DATA_RATE_5_5_MBPS
- NET_IF_WIFI_DATA_RATE_6_MBPS

- NET_IF_WIFI_DATA_RATE_9_MBPS
- NET_IF_WIFI_DATA_RATE_11_MBPS
- NET_IF_WIFI_DATA_RATE_12_MBPS
- NET_IF_WIFI_DATA_RATE_18_MBPS
- NET_IF_WIFI_DATA_RATE_24_MBPS
- NET_IF_WIFI_DATA_RATE_36_MBPS
- NET_IF_WIFI_DATA_RATE_48_MBPS
- NET_IF_WIFI_DATA_RATE_54_MBPS
- NET_IF_WIFI_DATA_RATE_MCS0
- NET_IF_WIFI_DATA_RATE_MCS1
- NET_IF_WIFI_DATA_RATE_MCS2
- NET_IF_WIFI_DATA_RATE_MCS3
- NET_IF_WIFI_DATA_RATE_MCS4
- NET_IF_WIFI_DATA_RATE_MCS5
- NET_IF_WIFI_DATA_RATE_MCS6
- NET_IF_WIFI_DATA_RATE_MCS7
- NET_IF_WIFI_DATA_RATE_MCS8
- NET_IF_WIFI_DATA_RATE_MCS9
- NET_IF_WIFI_DATA_RATE_MCS10
- NET_IF_WIFI_DATA_RATE_MCS11
- NET_IF_WIFI_DATA_RATE_MCS12
- NET_IF_WIFI_DATA_RATE_MCS13
- NET_IF_WIFI_DATA_RATE_MCS14
- NET_IF_WIFI_DATA_RATE_MCS15

`security_type`

Wireless security type:

- NET_IF_WIFI_SECURITY_OPEN
- NET_IF_WIFI_SECURITY_WEP
- NET_IF_WIFI_SECURITY_WPA
- NET_IF_WIFI_SECURITY_WPA2

`pwr_level`

Wireless radio power to configure:

- NET_IF_WIFI_PWR_LEVEL_LO
- NET_IF_WIFI_PWR_LEVEL_MED
- NET_IF_WIFI_PWR_LEVEL_HI

`ch`

Channel of the wireless network to create:

- NET_IF_WIFI_CH_1
- NET_IF_WIFI_CH_2
- NET_IF_WIFI_CH_3
- NET_IF_WIFI_CH_4
- NET_IF_WIFI_CH_5
- NET_IF_WIFI_CH_6
- NET_IF_WIFI_CH_7
- NET_IF_WIFI_CH_8
- NET_IF_WIFI_CH_9
- NET_IF_WIFI_CH_10
- NET_IF_WIFI_CH_11
- NET_IF_WIFI_CH_12

- NET_IF_WIFI_CH_13
- NET_IF_WIFI_CH_14

ssid

SSID of the access point to create.

psk

Pre-shared key of the access point.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_HANDLE

Returned Value

None.

Notes / Warnings

None.

NetIF_WiFi_GetPeerInfo()

Description

Gets the peer info connected to the access point (when acting as an access point).

Files

net_if_wifi.h/net_if_wifi.c

Prototype

```
CPU_INT16U  NetIF_WiFi_GetPeerInfo (NET_IF_NBR         if_nbr,
                                    NET_IF_WIFI_PEER  *p_buf_peer,
                                    CPU_INT16U         buf_peer_len_max,
                                    RTOS_ERR          *p_err)
```

Arguments

if_nbr

Wireless network interface number.

p_buf_peer

Pointer to the buffer to save the peer information.

buf_peer_len_max

Length in bytes of p_buf_peer .

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_HANDLE

Returned Value

Number of peers on the network and that are set in the buffer.

Notes / Warnings

None.

NetIF_WiFi_Join()

Description

Joins a wireless access point.

Files

net_if_wifi.h/net_if_wifi.c

Prototype

```
void  NetIF_WiFi_Join (NET_IF_NBR                 if_nbr,
                       NET_IF_WIFI_NET_TYPE       net_type,
                       NET_IF_WIFI_DATA_RATE      data_rate,
                       NET_IF_WIFI_SECURITY_TYPE  security_type,
                       NET_IF_WIFI_PWR_LEVEL      pwr_level,
                       NET_IF_WIFI_SSID           ssid,
                       NET_IF_WIFI_PSK            psk,
                       RTOS_ERR                  *p_err)
```

Arguments

if_nbr

Wireless network interface number.

net_type

Wireless network type:

- NET_IF_WIFI_NET_TYPE_INFRASTRUCTURE
- NET_IF_WIFI_NET_TYPE_ADHOC

data_rate

Wireless date rate to configure:

- NET_IF_WIFI_DATA_RATE_AUTO
- NET_IF_WIFI_DATA_RATE_1_MBPS
- NET_IF_WIFI_DATA_RATE_2_MBPS
- NET_IF_WIFI_DATA_RATE_5_5_MBPS
- NET_IF_WIFI_DATA_RATE_6_MBPS
- NET_IF_WIFI_DATA_RATE_9_MBPS
- NET_IF_WIFI_DATA_RATE_11_MBPS
- NET_IF_WIFI_DATA_RATE_12_MBPS
- NET_IF_WIFI_DATA_RATE_18_MBPS
- NET_IF_WIFI_DATA_RATE_24_MBPS

- `NET_IF_WIFI_DATA_RATE_36_MBPS`
- `NET_IF_WIFI_DATA_RATE_48_MBPS`
- `NET_IF_WIFI_DATA_RATE_54_MBPS`
- `NET_IF_WIFI_DATA_RATE_MCS0`
- `NET_IF_WIFI_DATA_RATE_MCS1`
- `NET_IF_WIFI_DATA_RATE_MCS2`
- `NET_IF_WIFI_DATA_RATE_MCS3`
- `NET_IF_WIFI_DATA_RATE_MCS4`
- `NET_IF_WIFI_DATA_RATE_MCS5`
- `NET_IF_WIFI_DATA_RATE_MCS6`
- `NET_IF_WIFI_DATA_RATE_MCS7`
- `NET_IF_WIFI_DATA_RATE_MCS8`
- `NET_IF_WIFI_DATA_RATE_MCS9`
- `NET_IF_WIFI_DATA_RATE_MCS10`
- `NET_IF_WIFI_DATA_RATE_MCS11`
- `NET_IF_WIFI_DATA_RATE_MCS12`
- `NET_IF_WIFI_DATA_RATE_MCS13`
- `NET_IF_WIFI_DATA_RATE_MCS14`
- `NET_IF_WIFI_DATA_RATE_MCS15`

`security_type`

Wireless security type:

- `NET_IF_WIFI_SECURITY_OPEN`
- `NET_IF_WIFI_SECURITY_WEP`
- `NET_IF_WIFI_SECURITY_WPA`
- `NET_IF_WIFI_SECURITY_WPA2`

`pwr_level`

Wireless radio power to configure:

- `NET_IF_WIFI_PWR_LEVEL_LO`
- `NET_IF_WIFI_PWR_LEVEL_MED`
- `NET_IF_WIFI_PWR_LEVEL_HI`

`ssid`

SSID of the access point to join.

`psk`

Pre-shared key of the access point.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

None.

Notes / Warnings

Before join an access point, the access point should have been found during a previous scan.

NetIF_WiFi_Leave()

Description

Leaves the access point previously joined.

Files

`net_if_wifi.h/net_if_wifi.c`

Prototype

```
void  NetIF_WiFi_Leave (NET_IF_NBR   if_nbr,
                        RTOS_ERR    *p_err)
```

Arguments

`if_nbr`

Wireless network interface number.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

None.

Notes / Warnings

None.

NetIF_WiFi_Scan()

Description

Scans the available wireless access point.

Files

`net_if_wifi.h/net_if_wifi.c`

Prototype

```
CPU_INT16U  NetIF_WiFi_Scan (       NET_IF_NBR         if_nbr,
                                    NET_IF_WIFI_AP    *p_buf_scan,
                                    CPU_INT16U         buf_scan_len_max,
                              const  NET_IF_WIFI_SSID  *p_ssid,
                                    NET_IF_WIFI_CH     ch,
                                    RTOS_ERR          *p_err)
```

Arguments

`if_nbr`

Wireless network interface number.

`p_buf_scan`

Pointer to a buffer that will receive available access point.

`buf_scan_len_max`

Maximum number of access point that can be stored

`p_ssid`

Pointer to a strubg that contains the SSID to scan. DEF_NULL to scan for all access point.

`ch`

Channel number:

- `NET_IF_WIFI_CH_ALL`
- `NET_IF_WIFI_CH_1`
- `NET_IF_WIFI_CH_2`
- `NET_IF_WIFI_CH_3`
- `NET_IF_WIFI_CH_4`
- `NET_IF_WIFI_CH_5`
- `NET_IF_WIFI_CH_6`
- `NET_IF_WIFI_CH_7`
- `NET_IF_WIFI_CH_8`
- `NET_IF_WIFI_CH_9`
- `NET_IF_WIFI_CH_10`
- `NET_IF_WIFI_CH_11`
- `NET_IF_WIFI_CH_12`
- `NET_IF_WIFI_CH_13`
- `NET_IF_WIFI_CH_14`

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_NOT_READY`

Returned Value

Number of wireless access points found.

Notes / Warnings

None.

NetIF_WiFi_Start()

Description

Start an WiFi type interface

Files

`net_if_wifi.h/net_if_wifi.c`

Prototype

```
void  NetIF_WiFi_Start (NET_IF_NBR        if_nbr,
                        NET_IF_WIFI_CFG  *p_cfg,
                        RTOS_ERR         *p_err)
```

Arguments

`if_nbr`

Network interface number to start.

`p_cfg`

Pointer to interface configuration. `DEF_NULL`, to just start the interface without Address or DHCP setup.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_TYPE`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_NOT_SUPPORTED`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`
- `RTOS_ERR_FAIL`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_NET_IF_LINK_DOWN`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_NOT_INIT`
- `RTOS_ERR_TX`
- `RTOS_ERR_ALREADY_EXISTS`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_NET_STR_ADDR_INVALID`
- `RTOS_ERR_RX`
- `RTOS_ERR_NOT_READY`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_NET_NEXT_HOP`
- `RTOS_ERR_IS_OWNER`

Returned Value

None.

Notes / Warnings

None.

DHCP Client API

Function Name	Description
DHCPc_IF_Add	Attaches an interface to the DHCP module and starts the DHCP process on this interface.
DHCPc_IF_Reboot	Reboots the DHCP Client process.
DHCPc_IF_Remove	Stops and removes the DHCP operation on the given network interface.

DHCPc_IF_Add()

Description

Attaches an interface to the DHCP module and starts the DHCP process on this interface.

Files

dhcp_client.h/dhcp_client.c

Prototype

```
void  DHCPc_IF_Add (NET_IF_NBR   if_nbr,
                    DHCPc_CFG   *p_cfg,
                    RTOS_ERR    *p_err)
```

Arguments

if_nbr

Interface number on which DHCP must be performed.

p_cfg

Pointer to the DHCP configuration for the given interface.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_ALREADY_EXISTS
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_BLK_ALLOC_CALLBACK
- RTOS_ERR_SEG_OVF
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_OS_ILLEGAL_RUN_TIME
- RTOS_ERR_INVALID_HANDLE

Returned Value

None.

Notes / Warnings

None.

DHCPc_IF_Reboot()

Description

Reboots the DHCP Client process.

Files

`dhcp_client.h/dhcp_client.c`

Prototype

```
void  DHCPc_IF_Reboot (NET_IF_NBR   if_nbr,
                       RTOS_ERR    *p_err)
```

Arguments

`if_nbr`

Interface number on which the DHCP process must be rebooted.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`

Returned Value

None.

Notes / Warnings

The reboot process will be completed automatically by the DHCP client process when an interface link state changes. This API is provided if more flexibility is required by the customer application.

DHCPc_IF_Remove()

Description

Stops and removes the DHCP operation on the given network interface.

Files

`dhcp_client.h/dhcp_client.c`

Prototype

```
void  DHCPc_IF_Remove (NET_IF_NBR   if_nbr,
                       RTOS_ERR    *p_err)
```

Arguments

if_nbr

Interface number on which the DHCP process must be stopped.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_NOT_FOUND
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_OS_ILLEGAL_RUN_TIME
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_BLK_ALLOC_CALLBACK
- RTOS_ERR_SEG_OVF

Returned Value

None.

Notes / Warnings

None.

DNS Client API

Function Name	Description
DNSc_CacheClrAll	Flushes the DNS cache.
DNSc_CacheClrHost	Removes a host from the cache.
DNSc_CfgServerByAddr	Configures the DNS server to use by default with an address structure.
DNSc_CfgServerByStr	Configures the DNS server to use by default using a string.
DNSc_GetServerByAddr	Gets the default DNS server in an address object format.
DNSc_GetServerByStr	Gets the default DNS server in string format.
DNSc_GetHost	Converts a string representation of a host name to its corresponding IP address using DNS service.
DNSc_GetHostAddrs()	Get a list of IP addresses assigned to a host.
DNSc_FreeHostAddrs()	Free Host addresses allocated by DNS_HostAddrsGet().

DNSc_CacheClrAll()

Description

Flushes the DNS cache.

Files

dns_client.h/dns_client.c

Prototype

```
void DNSc_CacheClrAll (RTOS_ERR *p_err)
```

Arguments

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_INIT`

Returned Value

None.

Notes / Warnings

None.

DNSc_CacheClrHost()

Description

Removes a host from the cache.

Files

`dns_client.h/dns_client.c`

Prototype

```
void  DNSc_CacheClrHost (CPU_CHAR  *p_host_name,
                         RTOS_ERR  *p_err)
```

Arguments

`p_host_name`

Pointer to a string that contains the host name to remove from the cache.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_INIT`
- `RTOS_ERR_NET_OP_IN_PROGRESS`
- `RTOS_ERR_NOT_FOUND`

Returned Value

None.

Notes / Warnings

None.

DNSc_CfgServerByAddr()

Description

Configures the DNS server to use by default with an address structure.

Files

`dns_client.h/dns_client.c`

Prototype

```
void  DNSc_CfgServerByAddr (NET_IP_ADDR_OBJ  *p_addr,
                            RTOS_ERR         *p_err)
```

Arguments

`p_addr`

Pointer to the structure that contains the IP address of the DNS server.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

None.

DNSc_CfgServerByStr()

Description

Configures the DNS server to use by default using a string.

Files

`dns_client.h/dns_client.c`

Prototype

```
void  DNSc_CfgServerByStr (CPU_CHAR  *p_server,
                           RTOS_ERR  *p_err)
```

Arguments

`p_server`

Pointer to a string that contains the IP address of the DNS server.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NET_STR_ADDR_INVALID`

Returned Value

None.

Notes / Warnings

None.

DNSc_GetHost()

Description

Converts a string representation of a host name to its corresponding IP address using DNS service.

Files

dns_client.h/dns_client.c

Prototype

```
DNSc_STATUS  DNSc_GetHost (CPU_CHAR        *p_host_name,
                           NET_IP_ADDR_OBJ  *p_addr_tbl,
                           CPU_INT08U       *p_addr_nbr,
                           DNSc_FLAGS        flags,
                           DNSc_REQ_CFG     *p_cfg,
                           RTOS_ERR         *p_err)
```

Arguments

p_host_name

Pointer to a string that contains the host name.

p_addr_tbl

Pointer to the array that will receive the IP address(es).

p_addr_nbr

Pointer to a variable that contains how many addresses can be contained in the array.

flags

DNS client flag:

DNSc_FLAG_NONE	By default, this function is blocking.
DNSc_FLAG_NO_BLOCK	Do not block (only possible if DNSc's task is enabled).
DNSc_FLAG_FORCE_CACHE	Take host from the cache, do not send new DNS request.
DNSc_FLAG_FORCE_RENEW	Force DNS request and remove existing entry in the cache.
DNSc_FLAG_FORCE_RESOLUTION	Force DNS to resolve given host name.
DNSc_FLAG_IPv4_ONLY	Return only the IPv4 address(es).
DNSc_FLAG_IPv6_ONLY	Return only the IPv6 address(es).

p_cfg

Pointer to a request configuration. Should be set to overwrite the default DNS configuration.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_NOT_INIT

- RTOS_ERR_NOT_FOUND
- RTOS_ERR_BLK_ALLOC_CALLBACK
- RTOS_ERR_SEG_OVF
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_OS_ILLEGAL_RUN_TIME
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_ABORT
- RTOS_ERR_TIMEOUT

Returned Value

Resolution status: `DNSc_STATUS_PENDING` Host resolution is pending, call again to see the status. (Processed by DNSc's task) `DNSc_STATUS_RESOLVED` Host is resolved. `DNSc_STATUS_FAILED` Host resolution has failed.

- `DNSc_STATUS_PENDING` : Host resolution is pending, call again to see the status. (Processed by DNSc's task)
- `DNSc_STATUS_RESOLVED` : Host is resolved.
- `DNSc_STATUS_FAILED` : Host resolution has failed.

Notes / Warnings

None.

DNSc_GetHostAddrs()

Description

Get a list of IP addresses assigned to a host.

Files

`dns_client.h/dns_client.c`

Prototype

```
DNSc_STATUS DNSc_GetHostAddrs (CPU_CHAR        *p_host_name,
                               NET_HOST_IP_ADDR **p_addrs,
                               CPU_INT08U       *p_addr_nbr,
                               DNSc_FLAGS        flags,
                               DNSc_REQ_CFG     *p_cfg,
                               RTOS_ERR         *p_err)
```

Arguments

`p_host_name`

Pointer to a string that contains the host name.

`p_addrs`

Pointer to link list that will receive the IP address(es).

`p_addr_nbr`

Pointer to a variable that contains how many addresses can be returned.

`flags`

DNS client flag:

DNSc_FLAG_NONE	By default, this function is blocking.
DNSc_FLAG_NO_BLOCK	Do not block (only possible if DNSc's task is enabled).
DNSc_FLAG_FORCE_CACHE	Take host from the cache, do not send new DNS request.
DNSc_FLAG_FORCE_RENEW	Force DNS request and remove existing entry in the cache.
DNSc_FLAG_FORCE_RESOLUTION	Force DNS to resolve given host name.
DNSc_FLAG_IPv4_ONLY	Return only the IPv4 address(es).
DNSc_FLAG_IPv6_ONLY	Return only the IPv6 address(es).

`p_cfg`

Pointer to a request configuration. Should be set to overwrite the default DNS configuration.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function.

Returned Value

None.

Notes / Warnings

None.

DNSc_FreeHostAddrs()

Description

Free Host addresses allocated by DNSc_GetHostAddrs() .

Files

`dns_client.h/dns_client.c`

Prototype

```
void DNSc_CacheClrAll (RTOS_ERR *p_err)
```

Arguments

`p_addr`

Pointer to the host addresses link-list to free.

Returned Value

None.

Notes / Warnings

None.

DNSc_GetServerByAddr

Description

Gets the default DNS server in a address object format.

Files

`dns_client.h/dns_client.c`

Prototype

```
void  DNSc_GetServerByAddr (NET_IP_ADDR_OBJ  *p_addr,
                            RTOS_ERR     *p_err)
```

Arguments

`p_addr`

Pointer to the structure that will receive the IP address of the DNS server.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_FOUND`

Returned Value

None.

Notes / Warnings

None.

DNSc_GetServerByStr()

Description

Gets the default DNS server in string format.

Files

`dns_client.h/dns_client.c`

Prototype

```
void  DNSc_GetServerByStr (CPU_CHAR    *p_str,
                           CPU_INT08U   str_len_max,
                           RTOS_ERR    *p_err)
```

Arguments

`p_addr`

Pointer to the structure that will receive the IP address of the DNS server.

`str_len_max`

Maximum string length.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_NOT_FOUND

Returned Value

None.

Notes / Warnings

None.

ARP API

Function Name	Description
NetARP_CacheGetAddrHW()	Get the hardware address corresponding to a specific ARP cache's protocol address.
NetARP_CachePoolStatGet()	Get ARP caches' statistics pool.
NetARP_CachePoolStatResetMaxUsed()	Reset ARP caches' statistics pool's maximum number of entries used.
NetARP_CacheProbeAddrOnNet()	Transmit an ARP request to probe the local network for a specific protocol address.
NetARP_CfgAddrFilterEn()	Configures the ARP address filter feature
NetARP_CfgCacheAccessedTh()	Configure ARP cache access promotion threshold.
NetARP_CfgCacheTimeout()	Configure ARP cache timeout for ARP Cache List.
NetARP_CfgCacheTxQ_MaxTh()	Configures the ARP cache maximum number of transmit packet buffers to enqueue.
NetARP_CfgPendReqMaxRetries()	Configure maximum number of ARP request retries for ARP cache in PEND state.
NetARP_CfgPendReqTimeout()	Configure timeout between ARP request timeouts for ARP cache in PEND state.
NetARP_CfgRenewReqMaxRetries()	Configure maximum number of ARP request retries for ARP cache in RENEW state.
NetARP_CfgRenewReqTimeout()	Configure timeout between ARP request timeouts for ARP cache in RENEW state.
NetARP_IsAddrProtocolConflict()	Check interface's protocol address conflict status between this interface's ARP host protocol address(es) and any other host(s) on the local network.
NetARP_TxReqGratuitous()	Prepares and transmits an unrequested ARP Request into the local network.

NetARP_CacheGetAddrHW()

Description

1. Gets the hardware address that corresponds to a specific ARP cache's protocol address by following these steps :
   (a) Acquire the network lock.
   (b) Search the ARP Cache List for ARP cache with the desired protocol address.
   (c) If the corresponding ARP cache is found, get/return the hardware address.
   (d) Release network lock.

Files

`net_arp.h/net_arp.c`

Prototype

```
NET_CACHE_ADDR_LEN  NetARP_CacheGetAddrHW (NET_IF_NBR          if_nbr,
                                           CPU_INT08U          *p_addr_hw,
                                           NET_CACHE_ADDR_LEN   addr_hw_len_buf,
                                           CPU_INT08U          *p_addr_protocol,
                                           NET_CACHE_ADDR_LEN   addr_protocol_len,
                                           RTOS_ERR            *p_err)
```

Arguments

`if_nbr`

Network interface number.

`p_addr_hw`

Pointer to a memory buffer that will receive the hardware address :

`addr_hw_len_buf`

Length of hardware address memory buffer (in octets).

`p_addr_protocol`

Pointer to the specific protocol address to search for corresponding hardware address.

`addr_protocol_len`

Length of protocol addresses (in octets).

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_NET_ADDR_UNRESOLVED`
- `RTOS_ERR_INVALID_STATE`

Returned Value

- Length of returned hardware address, if available.
- 0, otherwise.

Notes / Warnings

1. (a)
   1. The size of the memory buffer that will receive the returned hardware address MUST be greater than or equal to `NET_IF_HW_ADDR_LEN_MAX` .
   2. The length of any returned hardware address is equal to `NET_IF_HW_ADDR_LEN_MAX` .
   3. Address memory buffer array cleared in case of any error(s).
      (A) Address memory buffer array SHOULD be initialized to return a NULL address PRIOR to all validation or function handling in case of any error(s).

   (b) The length of the protocol address MUST be equal to `NET_IPv4_ADDR_SIZE` and is included for correctness and completeness.
2. ARP addresses handled in network-order :
   (a) ' `p_addr_hw` ' returns a hardware address in network-order.

(b) '`p_addr_protocol`' MUST points to a protocol address in network-order.
3. While an ARP cache is in 'PENDING' state, the hardware address is NOT yet resolved, but it MAY be resolved in the near future by an awaited ARP Reply.

NetARP_CachePoolStatGet()

Description

Gets the ARP statistics pool.

Files

`net_arp.h/net_arp.c`

Prototype

```
NET_STAT_POOL  NetARP_CachePoolStatGet (void)
```

Arguments

Returned Value

- ARP statistics pool, if NO error(s).
- NULL statistics pool, otherwise.

Notes / Warnings

None.

NetARP_CachePoolStatResetMaxUsed()

Description

Resets ARP statistics pool's maximum number of entries used.

Files

`net_arp.h/net_arp.c`

Prototype

```
void  NetARP_CachePoolStatResetMaxUsed (void)
```

Arguments

None.

Returned Value

None.

Notes / Warnings

None.

NetARP_CacheProbeAddrOnNet()

Description

1. Transmits an ARP Request to probe the local network for a specific protocol address :
   (a) Acquire network lock.
   (b) Remove the ARP cache with desired protocol address from ARP Cache List, if available.
   (c) Configure the ARP cache :
      1. Get the default-configured ARP cache
      2. ARP cache state
   (d) Transmit the ARP Request to probe local network for desired protocol address.
   (e) Release the network lock.
2. NetARP_CacheProbeAddrOnNet() SHOULD be used in conjunction with NetARP_CacheGetAddrHW() to determine if a specific protocol address is available on the local network :
   (a) After successfully transmitting an ARP Request to probe the local network.
   (b) After some time delay(s) [on the order of ARP Request timeouts & retries].
   (c) Check ARP Cache for the hardware address of a host on the local network that corresponds to the desired protocol address.
   See also NetARP_CacheGetAddrHW() and NetARP_TxReqGratuitous() .

Files

`net_arp.h/net_arp.c`

Prototype

```
void NetARP_CacheProbeAddrOnNet (NET_PROTOCOL_TYPE    protocol_type,
                                 CPU_INT08U          *p_addr_protocol_sender,
                                 CPU_INT08U          *p_addr_protocol_target,
                                 NET_CACHE_ADDR_LEN   addr_protocol_len,
                                 RTOS_ERR            *p_err)
```

Arguments

`protocol_type`

Address protocol type.

`p_addr_protocol_sender`

Pointer to the protocol address to send the probe from (see Note #5a1).

`p_addr_protocol_target`

Pointer to the protocol address to probe the local network (see Note #1.a.2).

`addr_protocol_len`

Length of the protocol address (in octets) [see Note #1.b].

`p_err` Pointer to the variable that will receive one of the following error code(s) from this function :

- `_NONE`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_SEG_OVF`

Returned Value

None.

Notes / Warnings

1. (a)

2. '`p_addr_protocol_target`' MUST point to a valid protocol address in network-order.
(b) The length of the protocol address MUST be equal to `NET_IPv4_ADDR_SIZE` and is included for correctness and completeness.

NetARP_CfgAddrFilterEn()

Description

Configures the ARP address filter feature.

Files

`net_arp.h/net_arp.c`

Prototype

```
void NetARP_CfgAddrFilterEn (CPU_BOOLEAN en)
```

Arguments

`en`

Set Filter to enabled or disabled:

- `DEF_ENABLED` Enable filtering feature.
- `DEF_DISABLED` Disable filtering feature.

Returned Value

None.

Notes / Warnings

None.

NetARP_CfgCacheAccessedTh()

Description

Configures the ARP cache access promotion threshold.

Files

`net_arp.h/net_arp.c`

Prototype

```
CPU_BOOLEAN NetARP_CfgCacheAccessedTh (CPU_INT16U nbr_access)
```

Arguments

`nbr_access`

Desired number of ARP cache accesses before ARP cache is promoted.

Returned Value

- `DEF_OK` , ARP cache access promotion threshold configured.
- `DEF_FAIL` , otherwise.

Notes / Warnings

None.

NetARP_CfgCacheTimeout()

Description

Configures the ARP cache timeout from ARP Cache List.

Files

`net_arp.h/net_arp.c`

Prototype

```
CPU_BOOLEAN  NetARP_CfgCacheTimeout (CPU_INT16U  timeout_sec)
```

Arguments

`timeout_sec`

Desired value for the ARP cache timeout (in seconds).

Returned Value

- `DEF_OK` , ARP cache timeout configured.
- `DEF_FAIL` , otherwise.

Notes / Warnings

1. RFC #1122, Section 2.3.2.1 states that "an implementation of the Address Resolution Protocol (ARP) … MUST provide a mechanism to flush out-of-date cache entries. If this mechanism involves a timeout, it SHOULD be possible to configure the timeout value."
2. The configured timeout does NOT reschedule any current ARP cache timeout in progress, but becomes effective the next time an ARP cache sets its timeout.

NetARP_CfgCacheTxQ_MaxTh()

Description

Configures the ARP cache maximum number of transmit packet buffers to enqueue.

Files

`net_arp.h/net_arp.c`

Prototype

```
CPU_BOOLEAN  NetARP_CfgCacheTxQ_MaxTh (NET_BUF_QTY  nbr_buf_max)
```

Arguments

`nbr_buf_max`

Desired maximum number of transmit packet buffers to enqueue.

Returned Value

- `DEF_OK` , ARP cache transmit packet buffer threshold configured.
- `DEF_FAIL` , otherwise.

Notes / Warnings

None.

NetARP_CfgPendReqMaxRetries()

Description

Configures the ARP Request maximum number of requests for ARP cache in PEND state.

Files

`net_arp.h/net_arp.c`

Prototype

```
CPU_BOOLEAN  NetARP_CfgPendReqMaxRetries (CPU_INT08U  max_nbr_retries)
```

Arguments

`max_nbr_retries`

Desired maximum number of ARP Request attempts.

Returned Value

- `DEF_OK` , ARP Request maximum number of request attempts configured.
- `DEF_FAIL` , otherwise.

Notes / Warnings

1. An ARP cache monitors the number of ARP Requests transmitted before receiving an ARP Reply. In other words, an ARP cache monitors the number of attempted ARP Requests.
   However, the maximum number of ARP Requests that each ARP cache is allowed to transmit is configured in terms of retries. The total number of attempts is equal to the configured number of retries, plus one (1.).

NetARP_CfgPendReqTimeout()

Description

Configures the timeout between ARP Request retries for ARP cache in PEND state.

Files

`net_arp.h/net_arp.c`

Prototype

```
CPU_BOOLEAN  NetARP_CfgPendReqTimeout (CPU_INT08U  timeout_sec)
```

Arguments

`timeout_sec`

Desired value for ARP Request pending ARP Reply timeout (in seconds).

Returned Value

- `DEF_OK` , ARP Request timeout configured.
- `DEF_FAIL` , otherwise.

Notes / Warnings

Configured timeout does NOT reschedule any current ARP Request timeouts in progress but becomes effective the next time an ARP Request is transmitted with timeout.

NetARP_CfgRenewReqMaxRetries()

Description

Configures the ARP Request maximum number of requests for ARP cache in RENEW state.

Files

`net_arp.h/net_arp.c`

Prototype

```
CPU_BOOLEAN  NetARP_CfgRenewReqMaxRetries (CPU_INT08U  max_nbr_retries)
```

Arguments

`max_nbr_retries`

Desired maximum number of ARP Request attempts.

Returned Value

- `DEF_OK` , ARP Request maximum number of request attempts configured.
- `DEF_FAIL` , otherwise.

Notes / Warnings

1. An ARP cache monitors the number of ARP Requests transmitted before receiving an ARP Reply. In other words, an ARP cache monitors the number of attempted ARP Requests.
   However, the maximum number of ARP Requests that each ARP cache is allowed to transmit is configured in terms of retries. The total number of attempts is equal to the configured number of retries, plus one (1.).

NetARP_CfgRenewReqTimeout()

Description

Configures the timeout between ARP Request retries for ARP cache in RENEW state.

Files

`net_arp.h/net_arp.c`

Prototype

```
CPU_BOOLEAN  NetARP_CfgRenewReqTimeout (CPU_INT08U  timeout_sec)
```

Arguments

`timeout_sec`

Desired value for ARP Request pending ARP Reply timeout (in seconds).

Returned Value

- `DEF_OK` , ARP Request timeout configured.
- `DEF_FAIL` , otherwise.

Notes / Warnings

Configured timeout does NOT reschedule any current ARP Request timeouts in progress but becomes effective the next time an ARP Request is transmitted with timeout.

NetARP_IsAddrProtocolConflict()

Description

Gets the interface's protocol address conflict status between this interface's ARP host protocol address(s) and the other host(s) on the local network.

Files

`net_arp.h/net_arp.c`

Prototype

```
CPU_BOOLEAN  NetARP_IsAddrProtocolConflict (NET_IF_NBR   if_nbr,
                                            RTOS_ERR    *p_err)
```

Arguments

`if_nbr`

Interface number to get protocol address conflict status.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- `DEF_YES` , if the address conflict is detected.

- `DEF_NO` , otherwise.

Notes / Warnings

RFC #3927, Section 2.5 states that :

"If a host receives an ARP packet (request *or* reply) on an interface where the 'sender hardware address' does not match the hardware address of that interface, but the 'sender IP address' is a IP address the host has configured for that interface, then this is a conflicting ARP packet, indicating an address conflict."

NetARP_TxReqGratuitous()

Description

1. Prepares and transmits an unrequested ARP Request into the local network by following these steps :
   (a) Acquire the network lock.
   (b) Get the network interface's hardware address.
   (c) Prepare the ARP Request packet :
      1. Configure the sender's hardware address as this interface's hardware address.
      2. Configure the target's hardware address as NULL.
      3. Configure the sender's protocol address as this interface's protocol address.
      4. Configure the target's protocol address as this interface's protocol address.
      5. Configure the ARP operation as an ARP Request.
   (d) Transmit the ARP Request.
   (e) Release the network lock.
2. NetARP_TxReqGratuitous() COULD be used in conjunction with NetARP_IsAddrProtocolConflict() to determine if the host's protocol address is already present on the local network, as follows :
   (a) After successfully transmitting a gratuitous ARP Request onto the local network.
   (b) After a time delay(s) [on the order of ARP Request timeouts & retries].
   (c) After checking this host's ARP protocol address conflict flag to see if any other host(s) are configured with this host's ARP protocol address.

Files

`net_arp.h/net_arp.c`

Prototype

```
void NetARP_TxReqGratuitous (NET_PROTOCOL_TYPE    protocol_type,
                             CPU_INT08U          *p_addr_protocol,
                             NET_CACHE_ADDR_LEN   addr_protocol_len,
                             RTOS_ERR            *p_err)
```

Arguments

`protocol_type`

Address protocol type.

`p_addr_protocol`

Pointer to protocol address used to transmit gratuitous request (see Note #5).

`addr_protocol_len`

Length of protocol address (in octets).

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_TYPE
- RTOS_ERR_TX
- RTOS_ERR_NOT_FOUND
- RTOS_ERR_RX
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_NET_IF_LINK_DOWN
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_TIMEOUT

Returned Value

None.

Notes / Warnings

1. 'p_addr_protocol' MUST point to a valid protocol address in network-order.
2. RFC #3927, Section 2.4 states that one purpose for transmitting a gratuitous ARP Request is for a host to "announce its [claim] ... [on] a unique address ... by broadcasting ... an ARP Request ... to make sure that other hosts on the link do not have stale ARP cache entries left over from some other host that may previously have been using the same address". "The ... ARP Request ... announcement ... sender and target IP addresses are both set to the host's newly selected ... address."

NDP API

Function Name	Description
NetNDP_CfgNeighborCacheTimeout()	Configure NDP neighbor cache timeout for NDP Neighbor Cache List.
NetNDP_CfgReachabilityTimeout()	Configure one of the NDP neighbor timeouts associated with the Neighbors Unreachability Detection.
NetNDP_CfgSolicitMaxNbr()	Configure one of the NDP neighbor maximum solicitations number.

NetNDP_CfgNeighborCacheTimeout()

Description

Configures the NDP Neighbor timeout from NDP Neighbor cache list.

Files

net_ndp.h/net_ndp.c

Prototype

```
CPU_BOOLEAN  NetNDP_CfgNeighborCacheTimeout (CPU_INT16U  timeout_sec)
```

Arguments

timeout_sec

Desired value for NDP neighbor timeout (in seconds).

Returned Value

- `DEF_OK` , NDP neighbor cache timeout configured.
- `DEF_FAIL` , otherwise.

Notes / Warnings

None.

NetNDP_CfgReachabilityTimeout()

Description

Configures possible NDP Neighbor reachability timeouts.

Files

`net_ndp.h/net_ndp.c`

Prototype

```
CPU_BOOLEAN NetNDP_CfgReachabilityTimeout (NET_NDP_TIMEOUT   timeout_type,
                                           CPU_INT16U        timeout_sec)
```

Arguments

`timeout_type`

NDP timeout type :

- `NET_NDP_TIMEOUT_REACHABLE`
- `NET_NDP_TIMEOUT_DELAY`
- `NET_NDP_TIMEOUT_SOLICIT`

`timeout_sec`

Desired value for NDP neighbor reachable timeout (in seconds).

Returned Value

- `DEF_OK` , NDP neighbor cache timeout configured.
- `DEF_FAIL` , otherwise.

Notes / Warnings

multicast.

NetNDP_CfgSolicitMaxNbr()

Description

Configures the NDP maximum number of NDP Solicitation sent for the given type of solicitation.

Files

`net_ndp.h/net_ndp.c`

Prototype

```
CPU_BOOLEAN NetNDP_CfgSolicitMaxNbr (NET_NDP_SOLICIT solicit_type,
                             CPU_INT08U      max_nbr)
```

Arguments

`solicit_type`

NDP Solicitation message type :

- `NET_NDP_SOLICIT_MULTICAST`
- `NET_NDP_SOLICIT_UNICAST`
- `NET_NDP_SOLICIT_DAD`

`max_nbr`

Desired maximum number of NDP solicitation attempts.

Returned Value

- `DEF_OK` , NDP Request maximum number of solicitation attempts configured.
- `DEF_FAIL` , otherwise.

Notes / Warnings

multicast.

IGMP API

Function Name	Description
NetIGMP_HostGrpJoin()	Join a host group.
NetIGMP_HostGrpLeave()	Leave a host group.

NetIGMP_HostGrpJoin()

Description

Joins a host group.

Files

`net_igmp.h/net_igmp.c`

Prototype

```
CPU_BOOLEAN  NetIGMP_HostGrpJoin (NET_IF_NBR      if_nbr,
                         NET_IPv4_ADDR  addr_grp,
                         RTOS_ERR       *p_err)
```

Arguments

`if_nbr`

Interface number to join host group.

`addr_grp`

IP address of host group to join.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- `DEF_OK` , if host group successfully joined.
- `DEF_FAIL` , otherwise.

Notes / Warnings

IP host group address MUST be in host-order.

NetIGMP_HostGrpLeave()

Description

Leaves a host group.

Files

`net_igmp.h/net_igmp.c`

Prototype

```
CPU_BOOLEAN  NetIGMP_HostGrpLeave (NET_IF_NBR      if_nbr,
                                   NET_IPv4_ADDR   addr_grp,
                                   RTOS_ERR       *p_err)
```

Arguments

`if_nbr`

Interface number to leave host group.

`addr_grp`

IP address of host group to leave.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_FOUND`

Returned Value

- `DEF_OK` , if host group successfully left.
- `DEF_FAIL` , otherwise.

Notes / Warnings

IP host group address MUST be in host-order.

MLDP API

Function Name	Description
NetMLDP_HostGrpJoin()	Join a MLDP Multicast host group.
NetMLDP_HostGrpLeave()	Leave a MLDP host group.

NetMLDP_HostGrpJoin()

Description

Joins an IPv6 MLDP group associated with a multicast address.

Files

`net_mldp.h/net_mldp.c`

Prototype

```
CPU_BOOLEAN  NetMLDP_HostGrpJoin (NET_IF_NBR      if_nbr,
                                  NET_IPv6_ADDR  *p_addr,
                                  RTOS_ERR       *p_err)
```

Arguments

`if_nbr`

Interface number associated with the MDLP host group.

`p_addr`

Pointer to the IPv6 address of host group to join.

`p_err`

Pointer to variable that will receive the return error code from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_TYPE`
- `RTOS_ERR_NOT_INIT`
- `RTOS_ERR_TX`
- `RTOS_ERR_ALREADY_EXISTS`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_RX`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_NOT_SUPPORTED`
- `RTOS_ERR_NOT_READY`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_NET_IF_LINK_DOWN`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_NET_NEXT_HOP`

- RTOS_ERR_ABORT
- RTOS_ERR_TIMEOUT

Returned Value

- Pointer to MLDP Host Group object added to the MLDP list.
- DEF_NULL , otherwise.

Notes / Warnings

None.

NetMLDP_HostGrpLeave()

Description

Leaves MDLP group associated with the received IPv6 multicast address.

Files

net_mldp.h/net_mldp.c

Prototype

```
CPU_BOOLEAN  NetMLDP_HostGrpLeave (NET_IF_NBR      if_nbr,
                                   NET_IPv6_ADDR  *p_addr,
                                   RTOS_ERR       *p_err)
```

Arguments

if_nbr

Interface number associated with host group.

p_addr

Pointer to the IPv6 address of host group to leave.

p_err

Pointer to variable that will receive the return error code from this function :

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_TYPE
- RTOS_ERR_NOT_INIT
- RTOS_ERR_TX
- RTOS_ERR_NOT_FOUND
- RTOS_ERR_ALREADY_EXISTS
- RTOS_ERR_RX
- RTOS_ERR_NOT_SUPPORTED
- RTOS_ERR_NOT_READY
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_NET_IF_LINK_DOWN
- RTOS_ERR_OS_OBJ_DEL

- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_NET_NEXT_HOP
- RTOS_ERR_ABORT
- RTOS_ERR_TIMEOUT

Returned Value

- DEF_OK , if host group successfully left.
- DEF_FAIL , otherwise.

Notes / Warnings

None.

ICMP API

Function Name	Description
NetICMP_TxEchoReq()	Send ICMPv4 or ICMPv6 Echo Request message to a network host.

NetICMP_TxEchoReq()

Description

Transmits an ICMPv4 or an ICMPv6 echo request message.

Files

net_icmp.h/net_icmp.c

Prototype

```
CPU_BOOLEAN  NetICMP_TxEchoReq (CPU_INT08U      *p_addr_dest,
                                NET_IP_ADDR_LEN   addr_len,
                                CPU_INT32U        timeout_ms,
                                void             *p_data,
                                CPU_INT16U        data_len,
                                RTOS_ERR         *p_err)
```

Arguments

p_addr_dest

Pointer to the IP destination address to send the ICMP echo request.

addr_len

IP address length :

- NET_IPv4_ADDR_SIZE
- NET_IPv6_ADDR_SIZE

timeout_ms

Timeout value to wait for ICMP echo response.

p_data

Pointer to the data buffer to include in the ICMP echo request.

data_len

Number of data buffer octets to include in the ICMP echo request.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_SEG_OVF
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_NET_INVALID_ADDR_SRC
- RTOS_ERR_NET_NEXT_HOP
- RTOS_ERR_NET_IF_LINK_DOWN
- RTOS_ERR_NET_ICMP_ECHO_REPLY_DATA_CMP
- RTOS_ERR_TIMEOUT

Returned Value

- DEF_OK , if ICMP echo request message successfully sent to remote host.
- DEF_FAIL , otherwise.

Notes / Warnings

None.

IPv4 API

Function Name	Description
NetIPv4_AddrLinkLocalCfg()	Start the IPv4 Link Local process on the given interface.
NetIPv4_AddrLinkLocalCfgRemove()	Stop the IPv4 Link Local process on the given interface and remove the link local address if one has already been configured.
NetIPv4_CfgAddrAdd()	Add a statically-configured IPv4 host address, subnet mask, and default gateway to an interface.
NetIPv4_CfgAddrAddDynamic()	Add a dynamically-configured IPv4 host address, subnet mask, and default gateway to an interface.
NetIPv4_CfgAddrAddDynamicStart()	Start dynamic IPv4 address configuration for an interface.
NetIPv4_CfgAddrAddDynamicStop()	Stop dynamic IPv4 address configuration for an interface.
NetIPv4_CfgAddrRemove()	Remove a configured IPv4 host address from an interface.
NetIPv4_CfgAddrRemoveAll()	Remove all configured IPv4 host address(es) from an interface.
NetIPv4_CfgFragReasmTimeout()	Configure IPv4 fragment reassembly timeout.
NetIPv4_GetAddrDfltGateway()	Get the default gateway IPv4 address for a host's configured IPv4 address.
NetIPv4_GetAddrHost()	Get an interface's configured IPv4 host address(es).
NetIPv4_GetAddrSrc()	Get corresponding configured IPv4 host address for a remote IPv4 address to use as source address.
NetIPv4_GetAddrSubnetMask()	Get the IPv4 address subnet mask for a host's configured IPv4 address.
NetIPv4_IsAddrBroadcast()	Validate an IPv4 address as the limited broadcast IPv4 address.
NetIPv4_IsAddrClassA()	Validate an IPv4 address as a Class-A IPv4 address.
NetIPv4_IsAddrClassB()	Validate an IPv4 address as a Class-B IPv4 address.
NetIPv4_IsAddrClassC()	Validate an IPv4 address as a Class-C IPv4 address.

Function Name	Description
NetIPv4_IsAddrHost()	Validate an IPv4 address as one the host's IPv4 address(es).
NetIPv4_IsAddrHostCfgd()	Validate an IPv4 address as one the host's configured IPv4 address(es).
NetIPv4_IsAddrLocalHost()	Validate an IPv4 address as a Localhost IPv4 address.
NetIPv4_IsAddrLocalLink()	Validate an IPv4 address as a link-local IPv4 address.
NetIPv4_IsAddrMulticast()	Validate an IPv4 address as a multicast IP address.
NetIPv4_IsAddrsCfgdOnIF()	Check if any IPv4 address(es) are configured on an interface.
NetIPv4_IsAddrThisHost()	Validate an IPv4 address as the 'This Host' initialization IPv4 address.
NetIPv4_IsValidAddrHost()	Validate an IPv4 address as a valid IPv4 host address.
NetIPv4_IsValidAddrHostCfgd()	Validate an IPv4 address as a valid, configurable IPv4 host address.
NetIPv4_IsValidAddrSubnetMask()	Validate an IPv4 address subnet mask.

NetIPv4_AddrLinkLocalCfg()

Description

Start the IPv4 Link Local process on the given interface.

Files

net_ipv4.h/net_ipv4.c

Prototype

```
void  NetIPv4_AddrLinkLocalCfg (NET_IF_NBR                       if_nbr,
                                NET_IPv4_LINK_LOCAL_COMPLETE_HOOK   hook,
                                RTOS_ERR                         *p_err)
```

Arguments

if_nbr

Interface number on which the IPv4 Link Local process will be started.

hook

Hook function to be notified when the process is completed.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_ALREADY_EXISTS
- RTOS_ERR_BLK_ALLOC_CALLBACK
- RTOS_ERR_SEG_OVF
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_OS_ILLEGAL_RUN_TIME
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_IS_OWNER
- RTOS_ERR_ABORT

RTOS_ERR_TIMEOUT

Returned Value

None.

Notes / Warnings

An IPv4 link local address will only be configured if no other IPv4 addresses are configured on the interface.

NetIPv4_AddrLinkLocalCfgRemove()

Description

Stop the IPv4 Link Local process on the given interface and remove the link local address if one has already been configured.

Files

net_ipv4.h/net_ipv4.c

Prototype

```
void  NetIPv4_AddrLinkLocalCfgRemove (NET_IF_NBR  if_nbr,
                                      RTOS_ERR   *p_err)
```

Arguments

if_nbr

Interface number.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_ALREADY_EXISTS
- RTOS_ERR_BLK_ALLOC_CALLBACK
- RTOS_ERR_SEG_OVF
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_OS_ILLEGAL_RUN_TIME
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_IS_OWNER
- RTOS_ERR_ABORT
- RTOS_ERR_TIMEOUT

Returned Value

None.

Notes / Warnings

None

NetIPv4_CfgAddrAdd()

Description

Adds a statically-configured IPv4 host address, subnet mask, and default gateway to an interface.

Files

net_ipv4.h/net_ipv4.c

Prototype

```
CPU_BOOLEAN NetIPv4_CfgAddrAdd (NET_IF_NBR      if_nbr,
                                NET_IPv4_ADDR   addr_host,
                                NET_IPv4_ADDR   addr_subnet_mask,
                                NET_IPv4_ADDR   addr_dflt_gateway,
                                RTOS_ERR       *p_err)
```

Arguments

if_nbr

Interface number to configure.

addr_host

Desired IPv4 address to add to this interface.

addr_subnet_mask

Desired IPv4 address subnet mask to configure.

addr_dflt_gateway

Desired IPv4 default gateway address to configure.

DEF_NULL or empty string if no gateway to configure.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_ALREADY_EXISTS
- RTOS_ERR_INVALID_HANDLE

Returned Value

- DEF_OK , if valid IPv4 address, subnet mask, and default gateway configured.
- DEF_FAIL , otherwise.

Notes / Warnings

1. IPv4 addresses MUST be in host-order.
2. (a) A host on an isolated network should be able to correctly operate and communicate with all other hosts on its local network without need of a gateway or configuration of a gateway.

(b) However, a configured gateway MUST be on the same network as the host's IPv4 address (i.e. the network portion of the configured IPv4 address and the configured gateway addresses MUST be identical).

NetIPv4_CfgAddrAddDynamic()

Description

Adds a dynamically-configured IPv4 host address, subnet mask, and default gateway to an interface.

Files

`net_ipv4.h/net_ipv4.c`

Prototype

```
CPU_BOOLEAN  NetIPv4_CfgAddrAddDynamic (NET_IF_NBR      if_nbr,
                                        NET_IPv4_ADDR   addr_host,
                                        NET_IPv4_ADDR   addr_subnet_mask,
                                        NET_IPv4_ADDR   addr_dflt_gateway,
                                        RTOS_ERR       *p_err)
```

Arguments

`if_nbr`

Interface number to configure.

`addr_host`

Desired IPv4 address to add to this interface.

`addr_subnet_mask`

Desired IPv4 address subnet mask to configure.

`addr_dflt_gateway`

Desired IPv4 default gateway address to configure.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_ALREADY_EXISTS`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- `DEF_OK` , if valid IPv4 address, subnet mask, and default gateway configured.
- `DEF_FAIL` , otherwise.

Notes / Warnings

1. Application calls to dynamic address configuration functions MUST be sequenced as follows :
   (a) NetIPv4_CfgAddrAddDynamicStart() MUST precede NetIPv4_CfgAddrAddDynamic().
2. IPv4 addresses MUST be in host-order.

NetIPv4_CfgAddrAddDynamicStart()

Description

Starts the dynamic address configuration for an interface.

Files

`net_ipv4.h/net_ipv4.c`

Prototype

```
CPU_BOOLEAN  NetIPv4_CfgAddrAddDynamicStart (NET_IF_NBR   if_nbr,
                                             RTOS_ERR    *p_err)
```

Arguments

`if_nbr`

Interface number to start dynamic address configuration.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- `DEF_OK` , if dynamic configuration successfully started.
- `DEF_FAIL` , otherwise.

Notes / Warnings

1. Application calls to dynamic address configuration functions MUST be sequenced as follows:
   (a) NetIPv4_CfgAddrAddDynamicStart() MUST precede NetIPv4_CfgAddrAddDynamic().

NetIPv4_CfgAddrAddDynamicStop()

Description

Stops the dynamic address configuration for an interface.

Files

`net_ipv4.h/net_ipv4.c`

Prototype

```
CPU_BOOLEAN  NetIPv4_CfgAddrAddDynamicStop (NET_IF_NBR   if_nbr,
                                            RTOS_ERR    *p_err)
```

Arguments

`if_nbr`

Interface number to stop dynamic address configuration.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_POOL_FULL`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- `DEF_OK` , if dynamic configuration successfully stopped.
- `DEF_FAIL` , otherwise.

Notes / Warnings

1. Application calls to dynamic address configuration functions MUST be sequenced as follows :
   (a) NetIPv4_CfgAddrAddDynamicStop()MUST followNetIPv4_CfgAddrAddDynamicStart() , if and ONLY if the dynamic address initialization fails.

NetIPv4_CfgAddrRemove()

Description

Removes a configured IPv4 host address, subnet mask, and default gateway from an interface.

Files

`net_ipv4.h/net_ipv4.c`

Prototype

```
CPU_BOOLEAN  NetIPv4_CfgAddrRemove (NET_IF_NBR      if_nbr,
                                    NET_IPv4_ADDR   addr_host,
                                    RTOS_ERR       *p_err)
```

Arguments

`if_nbr`

Interface number to remove address configuration.

`addr_host`

IPv4 address to remove.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- `DEF_OK` , if IPv4 address configuration removed.
- `DEF_FAIL` , otherwise.

Notes / Warnings

IPv4 address MUST be in host-order.

NetIPv4_CfgAddrRemoveAll()

Description

Removes all configured IPv4 host address(s) from an interface.

Files

`net_ipv4.h/net_ipv4.c`

Prototype

```
CPU_BOOLEAN  NetIPv4_CfgAddrRemoveAll (NET_IF_NBR   if_nbr,
                                       RTOS_ERR    *p_err)
```

Arguments

`if_nbr`

Interface number to remove address configuration.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- `DEF_OK` , if ALL interface's configured IP host address(s) successfully removed.
- `DEF_FAIL` , otherwise.

Notes / Warnings

None.

NetIPv4_CfgFragReasmTimeout()

Description

Configures the IPv4 fragment reassembly timeout.

Files

`net_ipv4.h/net_ipv4.c`

Prototype

```
CPU_BOOLEAN  NetIPv4_CfgFragReasmTimeout (CPU_INT08U  timeout_sec)
```

Arguments

`timeout_sec`

Desired value for IPv4 fragment reassembly timeout (in seconds).

Returned Value

- `DEF_OK` , IPv4 fragment reassembly timeout configured.
- `DEF_FAIL` , otherwise.

Notes / Warnings

Configured timeout does NOT reschedule any current IP fragment reassembly timeout in progress, but becomes effective the next time IP fragments reassemble with timeout.

NetIPv4_GetAddrDfltGateway()

Description

Gets the default gateway IPv4 address for a configured IPv4 host address.

Files

`net_ipv4.h/net_ipv4.c`

Prototype

```
NET_IPv4_ADDR  NetIPv4_GetAddrDfltGateway (NET_IPv4_ADDR   addr,
                                           RTOS_ERR      *p_err)
```

Arguments

`addr`

Configured IPv4 host address to get the default gateway.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_FOUND`

Returned Value

- Configured IPv4 host address's default gateway in host-order, if NO error(s).
- `NET_IPv4_ADDR_NONE` , otherwise.

Notes / Warnings

IPv4 address returned in host-order.

NetIPv4_GetAddrHost()

Description

Gets an interface's IPv4 host address(s).

Files

`net_ipv4.h/net_ipv4.c`

Prototype

```
CPU_BOOLEAN  NetIPv4_GetAddrHost (NET_IF_NBR        if_nbr,
                                  NET_IPv4_ADDR     *p_addr_tbl,
                                  NET_IP_ADDRS_QTY  *p_addr_tbl_qty,
                                  RTOS_ERR          *p_err)
```

Arguments

`if_nbr`

Interface number to get IPv4 host address(s).

`p_addr_tbl`

Pointer to IPv4 address table that will receive the IPv4 host address(s).

`p_addr_tbl_qty`

Pointer to a variable to pass the table size and will return the number of address added to the table.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- `DEF_OK` , if interface's IPv4 host address(s) successfully returned.
- `DEF_FAIL` , otherwise.

Notes / Warnings

IPv4 address(s) returned in host-order.

NetIPv4_GetAddrSrc()

Description

Gets the corresponding configured IPv4 host address for a remote IPv4 address (to use as source address).

Files

`net_ipv4.h/net_ipv4.c`

Prototype

```
NET_IPv4_ADDR  NetIPv4_GetAddrSrc (NET_IPv4_ADDR   addr_remote,
                                   RTOS_ERR        *p_err)
```

Arguments

`addr_remote`

Remote address to get configured IPv4 host address.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- Configured IPv4 host address, if available.
- `NET_IPv4_ADDR_NONE` , otherwise.

Notes / Warnings

IPv4 addresses MUST be in host-order.

NetIPv4_GetAddrSubnetMask()

Description

Gets the IPv4 address subnet mask for a configured IPv4 host address.

Files

`net_ipv4.h/net_ipv4.c`

Prototype

```
NET_IPv4_ADDR  NetIPv4_GetAddrSubnetMask (NET_IPv4_ADDR   addr,
                                          RTOS_ERR       *p_err)
```

Arguments

`addr`

Configured IPv4 host address to get the subnet mask.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_FOUND`

Returned Value

- Configured IPv4 host address's subnet mask in host-order, if NO error(s).
- `NET_IPv4_ADDR_NONE` , otherwise.

Notes / Warnings

IPv4 address returned in host-order.

NetIPv4_IsAddrBroadcast()

Description

Validates an IPv4 address as a limited broadcast IPv4 address.

Files

`net_ipv4.h/net_ipv4.c`

Prototype

```
CPU_BOOLEAN  NetIPv4_IsAddrBroadcast (NET_IPv4_ADDR  addr)
```

Arguments

`addr`

IPv4 address to validate.

Returned Value

- `DEF_YES` , if IPv4 address is a limited broadcast IP address.
- `DEF_NO` , otherwise.

Notes / Warnings

IPv4 address MUST be in host-order.

NetIPv4_IsAddrClassA()

Description

Validates an IPv4 address as a Class-A IPv4 address.

Files

`net_ipv4.h/net_ipv4.c`

Prototype

```
CPU_BOOLEAN  NetIPv4_IsAddrClassA (NET_IPv4_ADDR  addr)
```

Arguments

`addr`

IPv4 address to validate.

Returned Value

- `DEF_YES` , if IPv4 address is a Class-A IPv4 address.
- `DEF_NO` , otherwise.

Notes / Warnings

1. IPv4 address MUST be in host-order.

NetIPv4_IsAddrClassB()

Description

Validates an IPv4 address as a Class-B IPv4 address.

Files

`net_ipv4.h/net_ipv4.c`

Prototype

```
CPU_BOOLEAN  NetIPv4_IsAddrClassB (NET_IPv4_ADDR  addr)
```

Arguments

`addr`

IPv4 address to validate.

Returned Value

- `DEF_YES` , if IPv4 address is a Class-B IPv4 address.
- `DEF_NO` , otherwise.

Notes / Warnings

IPv4 address MUST be in host-order.

NetIPv4_IsAddrClassC()

Description

Validates an IPv4 address as a Class-C IPv4 address.

Files

`net_ipv4.h/net_ipv4.c`

Prototype

```
CPU_BOOLEAN  NetIPv4_IsAddrClassC (NET_IPv4_ADDR  addr)
```

Arguments

`addr`

IPv4 address to validate.

Returned Value

- `DEF_YES` , if IPv4 address is a Class-C IPv4 address.
- `DEF_NO` , otherwise.

Notes / Warnings

IPv4 address MUST be in host-order.

NetIPv4_IsAddrClassD()

Description

Validates an IPv4 address as a Class-D IPv4 address.

Files

`net_ipv4.h/net_ipv4.c`

Prototype

```
CPU_BOOLEAN  NetIPv4_IsAddrClassD (NET_IPv4_ADDR  addr)
```

Arguments

`addr`

IPv4 address to validate.

Returned Value

- `DEF_YES` , if IPv4 address is a Class-D IPv4 address.
- `DEF_NO` , otherwise.

Notes / Warnings

IPv4 address MUST be in host-order.

NetIPv4_IsAddrHost()

Description

Validates an IPv4 address as an IPv4 host address :

Files

`net_ipv4.h/net_ipv4.c`

Prototype

```
CPU_BOOLEAN  NetIPv4_IsAddrHost (NET_IPv4_ADDR  addr)
```

Arguments

`addr`

IPv4 address to validate.

Returned Value

- `DEF_YES` , if IPv4 address is one of the host's IPv4 addresses.
- `DEF_NO` , otherwise.

Notes / Warnings

IPv4 address MUST be in host-order.

NetIPv4_IsAddrHostCfgd()

Description

Validates an IPv4 address as a configured IPv4 host address on an enabled interface.

Files

`net_ipv4.h/net_ipv4.c`

Prototype

```
CPU_BOOLEAN NetIPv4_IsAddrHostCfgd (NET_IPv4_ADDR addr)
```

Arguments

`addr`

IPv4 address to validate.

Returned Value

- `DEF_YES` , if IPv4 address is one of the host's configured IPv4 addresses.
- `DEF_NO` , otherwise.

Notes / Warnings

IPv4 address MUST be in host-order.

NetIPv4_IsAddrLocalHost()

Description

Validates an IPv4 address as a 'Localhost' IPv4 address.

Files

`net_ipv4.h/net_ipv4.c`

Prototype

```
CPU_BOOLEAN NetIPv4_IsAddrLocalHost (NET_IPv4_ADDR addr)
```

Arguments

`addr`

IPv4 address to validate.

Returned Value

- `DEF_YES` , if IPv4 address is a 'Localhost' IPv4 address.
- `DEF_NO` , otherwise.

Notes / Warnings

IPv4 address MUST be in host-order.

NetIPv4_IsAddrLocalLink()

Description

Validates an IPv4 address as a link-local IPv4 address.

Files

`net_ipv4.h/net_ipv4.c`

Prototype

```
CPU_BOOLEAN  NetIPv4_IsAddrLocalLink (NET_IPv4_ADDR  addr)
```

Arguments

`addr`

IPv4 address to validate.

Returned Value

- `DEF_YES` , if IPv4 address is a link-local IPv4 address.
- `DEF_NO` , otherwise.

Notes / Warnings

1. IPv4 address MUST be in host-order.

NetIPv4_IsAddrMulticast()

Description

Validates an IPv4 address as a multicast IP address.

Files

`net_ipv4.h/net_ipv4.c`

Prototype

```
CPU_BOOLEAN  NetIPv4_IsAddrMulticast (NET_IPv4_ADDR  addr)
```

Arguments

`addr`

IPv4 address to validate.

Returned Value

- `DEF_YES` , if IPv4 address is a multicast IP address.
- `DEF_NO` , otherwise.

Notes / Warnings

1. IPv4 address MUST be in host-order.

NetIPv4_IsAddrThisHost()

Description

Validates an IPv4 address as a 'This Host' initialization IPv4 address.

Files

`net_ipv4.h/net_ipv4.c`

Prototype

```
CPU_BOOLEAN  NetIPv4_IsAddrThisHost (NET_IPv4_ADDR  addr)
```

Arguments

`addr`

IPv4 address to validate.

Returned Value

- `DEF_YES` , if IPv4 address is a 'This Host' initialization IPv4 address.
- `DEF_NO` , otherwise.

Notes / Warnings

IPv4 address MUST be in host-order.

NetIPv4_IsAddrsCfgdOnIF()

Description

Checks if any IPv4 host address(s) configured on an interface.

Files

`net_ipv4.h/net_ipv4.c`

Prototype

```
CPU_BOOLEAN  NetIPv4_IsAddrsCfgdOnIF (NET_IF_NBR   if_nbr,
                                      RTOS_ERR    *p_err)
```

Arguments

`if_nbr`

Interface number to check for configured IPv4 host address(s).

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- DEF_YES , if any IP host address(s) configured on interface.
- DEF_NO , otherwise.

Notes / Warnings

None.

NetIPv4_IsValidAddrHost()

Description

1. Validates an IPv4 host address :
   (a) MUST NOT be one of the following :
      1. This Host RFC #1122, Section 3.2.1.3.(a)
      2. Specified Host RFC #1122, Section 3.2.1.3.(b)
      3. Limited Broadcast RFC #1122, Section 3.2.1.3.(c)
      4. Directed Broadcast RFC #1122, Section 3.2.1.3.(d)
      5. Localhost RFC #1122, Section 3.2.1.3.(g)
      6. Multicast host address RFC #1112, Section 7.2
   (b) 1. RFC #3927, Section 2.1 specifies the "IPv4 Link-Local address" :
      1. "Range ... inclusive" ...
         1. "from 169.254.1.0" ...
         2. "to 169.254.254.255".
2. ONLY validates typical IPv4 host addresses, since 'This Host' and 'Specified Host' IPv4 host addresses are ONLY valid during a host's initialization.
   This function CANNOT be used to validate any 'This Host' or 'Specified Host' host addresses.

Files

net_ipv4.h/net_ipv4.c

Prototype

```
CPU_BOOLEAN  NetIPv4_IsValidAddrHost (NET_IPv4_ADDR  addr_host)
```

Arguments

addr_host

IPv4 host address to validate (see Note #4).

Returned Value

- DEF_YES , if IPv4 host address is valid.
- DEF_NO , otherwise.

Notes / Warnings

IPv4 address MUST be in host-order.

NetIPv4_IsValidAddrHostCfgd()

Description

1. Validates an IPv4 address for a configured IPv4 host address :
   (a) MUST NOT be one of the following :
      1. This Host RFC #1122, Section 3.2.1.3.(a)
      2. Specified Host RFC #1122, Section 3.2.1.3.(b)

    3. Limited Broadcast RFC #1122, Section 3.2.1.3.(c)
    4. Directed Broadcast RFC #1122, Section 3.2.1.3.(d)
    5. Subnet Broadcast RFC #1122, Section 3.2.1.3.(e)
    6. Localhost RFC #1122, Section 3.2.1.3.(g)
    7. Multicast host address RFC #1112, Section 7.2
2. ONLY validates this host's IPv4 address, since 'This Host' and 'Specified Host' IPv4 host addresses are ONLY valid during a host's initialization (see Notes #1a1 and #1a4). This function CANNOT be used to validate any 'This Host' or 'Specified Host' host addresses.

Files

`net_ipv4.h/net_ipv4.c`

Prototype

```
CPU_BOOLEAN  NetIPv4_IsValidAddrHostCfgd (NET_IPv4_ADDR  addr_host,
                                          NET_IPv4_ADDR  addr_subnet_mask)
```

Arguments

`addr_host`

IPv4 host address to validate.

`addr_subnet_mask`

IPv4 address subnet mask.

Returned Value

- `DEF_YES` , if this host's IPv4 address is valid.
- `DEF_NO` , otherwise.

Notes / Warnings

IPv4 addresses MUST be in host-order.

NetIPv4_IsValidAddrSubnetMask()

Description

1. Validates an IPv4 address subnet mask :
   (a) RFC #1122, Section 3.2.1.3 states that :

   > 1. "IP addresses are not permitted to have the value 0 or -1 for any of the ... \<Subnet-number\> fields" ...
   > 2. "This implies that each of these fields will be at least two bits long."

   (b) RFC #950, Section 2.1 'Special Addresses' reiterates that "the values of all zeros and all ones in the subnet field should not be assigned to actual (physical) subnets".
   (c) RFC #950, Section 2.1 also states that "the bits that identify the subnet ... need not be adjacent in the address. However, we recommend that the subnet bits be contiguous and located as the most significant bits of the local address". Therefore, it is assumed that at least the most significant bit of the network portion of the subnet address SHOULD be set.

Files

`net_ipv4.h/net_ipv4.c`

Prototype

```
CPU_BOOLEAN NetIPv4_IsValidAddrSubnetMask (NET_IPv4_ADDR addr_subnet_mask)
```

Arguments

`addr_subnet_mask`

IPv4 address subnet mask to validate.

Returned Value

- `DEF_YES`, if IPv4 address subnet mask is valid.
- `DEF_NO`, otherwise.

Notes / Warnings

IPv4 addresses MUST be in host-order.

IPv6 API

Function Name	Description
NetIPv6_AddrAutoCfgDis()	Disables the IPv6 Stateless Address Auto-Configuration procedure.
NetIPv6_AddrAutoCfgEn()	Enables the IPv6 Stateless Address Auto-Configuation procedure.
NetIPv6_AddrMask()	Applies IPv6 mask on an address.
NetIPv6_AddrMaskByPrefixLen()	Gets the IPv6 address masked with the prefix length.
NetIPv6_AddrSubscribe()	Configures the IPv6 Address Configuration hook function.
NetIPv6_AddrUnsubscribe()	Removes a configured IPv6 host address and multicast solicited mode address from an interface.
NetIPv6_CfgAddrAdd()	Adds a statically-configured IPv6 host address to an interface.
NetIPv6_AddrTypeValidate()	Validates the type of an IPv6 address.
NetIPv6_CfgAddrRemove()	Removes a configured IPv6 host address and multicast solicited mode address from an interface.
NetIPv6_CfgAddrRemoveAll()	Removes all configured IPv6 host address(s) from an interface.
NetIPv6_CfgFragReasmTimeout()	Configures the IPv6 fragment reassembly timeout.
NetIPv6_CreateAddrFromID()	Creates an IPv6 address from a prefix and an identifier.
NetIPv6_CreateIF_ID()	Creates an IPv6 interface identifier.
NetIPv6_GetAddrHost()	Gets an interface's IPv6 host address(es).
NetIPv6_GetAddrSrc()	Finds the best matched source address in the IPv6 configured host addresses for the specified destination address.
NetIPv6_GetAddrMatchingLen()	Computes the number of identical most significant bits of two IPv6 addresses.
NetIPv6_GetAddrScope()	Gets the scope of a specific IPv6 address.
NetIPv6_IsAddrHostCfgd()	Validates an IPv6 address as a configured IPv6 host address on an enabled interface.
NetIPv6_IsAddrsCfgdOnIF()	Checks if any IPv6 host addresses are configured on a specific interface.
NetIPv6_IsValidAddrHost()	Validates an IPv6 host address.
NetIPv6_IsAddrLinkLocal()	Validates an IPv6 address as a link-local IPv6 address.
NetIPv6_IsAddrSiteLocal()	Validates an IPv6 address as a site-local address.

Function Name	Description
NetIPv6_IsAddrMcast()	Validates an IPv6 address as a multicast address.
NetIPv6_IsAddrMcastAllRouters()	Validates an IPv6 address as the all routers multicast address.
NetIPv6_IsAddrMcastAllNodes()	Validates an IPv6 address as the all nodes multicast address.
NetIPv6_IsAddrMcastSolNode()	Validates an IPv6 address as a solicited node multicast address.
NetIPv6_IsAddrMcastRsvd()	Validates an IPv6 address as a reserved multicast IPv6 address.
NetIPv6_IsAddrUnspecified()	Validates an IPv6 address as the unspecified IPv6 address.
NetIPv6_IsAddrLoopback()	Validates an IPv6 address as the IPv6 loopback address.

NetIPv6_AddrAutoCfgDis()

Description

Disables the IPv6 Auto-Configuration.

Files

net_ipv6.h/net_ipv6.c

Prototype

```
CPU_BOOLEAN  NetIPv6_AddrAutoCfgDis (NET_IF_NBR   if_nbr,
                                     RTOS_ERR    *p_err)
```

Arguments

if_nbr

Network interface number on which to disabled address auto-configuration.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_HANDLE

Returned Value

- DEF_OK , if the IPv6 Auto-Configuration was disabled successfully.
- DEF_FAIL , otherwise.

Notes / Warnings

1. After disabling, in a case of a link status change, the auto-configuration will not be called. The Hook function after the Auto-configuration completion will also not occur.
2. The address previously configured with the auto-configuration will NOT be removed after disabling.

NetIPv6_AddrAutoCfgEn()

Description

Enables IPv6 Auto-configuration process. If the link state is UP, the IPv6 Auto-configuration will start. For other states, the Auto-configuration will start when the link becomes UP.

Files

`net_ipv6.h/net_ipv6.c`

Prototype

```
CPU_BOOLEAN  NetIPv6_AddrAutoCfgEn (NET_IF_NBR    if_nbr,
                                    CPU_BOOLEAN   dad_en,
                                    RTOS_ERR     *p_err)
```

Arguments

`if_nbr`

Network interface number to enable the address auto-configuration on.

`dad_en`

- `DEF_YES` , perform the Duplication Address Detection (DAD).
- DEF_NO , otherwise

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_NOT_AVAIL`

Returned Value

- `DEF_OK` , if the IPv6 Auto-Configuration is enabled successfully
- `DEF_FAIL` , otherwise.

Notes / Warnings

None.

NetIPv6_AddrSubscribe()

Description

Configures the IPv6 Address Configuration hook function. This function will be called each time a IPv6 address has finished being configured.

Files

`net_ipv6.h/net_ipv6.c`

Prototype

```
void  NetIPv6_AddrSubscribe (NET_IPv6_ADDR_HOOK_FNCT   fnct,
                             RTOS_ERR                 *p_err)
```

Arguments

`fnct`

Pointer to hook function to call when the IPv6 static address configuration ends.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_POOL_EMPTY`

Returned Value

None.

Notes / Warnings

None.

NetIPv6_AddrUnsubscribe()

Description

Removes the subscribe hook function for IPv6 address configuration.

Files

`net_ipv6.h/net_ipv6.c`

Prototype

```
void NetIPv6_AddrUnsubscribe (NET_IPv6_ADDR_HOOK_FNCT  fnct)
```

Arguments

`fnct`

Pointer to hook function to remove.

Returned Value

None.

Notes / Warnings

None.

NetIPv6_CfgAddrAdd()

Description

Adds a statically-configured IPv6 host address to an interface.

Files

`net_ipv6.h/net_ipv6.c`

Prototype

```
CPU_BOOLEAN  NetIPv6_CfgAddrAdd (NET_IF_NBR      if_nbr,
                                 NET_IPv6_ADDR  *p_addr,
                                 CPU_INT08U      prefix_len,
                                 NET_FLAGS       flags,
                                 RTOS_ERR       *p_err)
```

Arguments

`if_nbr`

Interface number to configure.

`p_addr`

Pointer to desired IPv6 address to add to this interface.

`prefix_len`

Prefix length of the desired IPv6 address to add to this interface.

`flags`

Set of flags to select options for the address configuration:

- `NET_IPv6_FLAG_BLOCK_EN` Enables blocking mode if set.
- `NET_IPv6_FLAG_DAD_EN` Enables DAD if set.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_NOT_AVAIL`

Returned Value

- `DEF_OK` , if the valid IPv6 address is configured.
- `DEF_FAIL` , otherwise.

Notes / Warnings

None.

NetIPv6_CfgAddrRemove()

Description

Removes a configured IPv6 host address and multicast solicited mode address from an interface.

Files

`net_ipv6.h/net_ipv6.c`

Prototype

```
CPU_BOOLEAN  NetIPv6_CfgAddrRemove (NET_IF_NBR      if_nbr,
                                    NET_IPv6_ADDR  *p_addr_host,
                                    RTOS_ERR       *p_err)
```

Arguments

`if_nbr`

Interface number to remove address configuration.

`p_addr_host`

Pointer to the IPv6 address to remove.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- `DEF_OK` , if the IPv6 address is removed.
- `DEF_FAIL` , otherwise.

Notes / Warnings

None.

NetIPv6_CfgAddrRemoveAll()

Description

Removes all configured IPv6 host address(s) from an interface.

Files

`net_ipv6.h/net_ipv6.c`

Prototype

```
CPU_BOOLEAN  NetIPv6_CfgAddrRemoveAll (NET_IF_NBR   if_nbr,
                                       RTOS_ERR    *p_err)
```

Arguments

`if_nbr`

Interface number to remove address configuration.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- `DEF_OK` , if ALL interface's configured IP host address(s) are successfully removed.
- `DEF_FAIL` , otherwise.

Notes / Warnings

None.

NetIPv6_CfgFragReasmTimeout()

Description

Configures the IPv6 fragment reassembly timeout.

Files

`net_ipv6.h/net_ipv6.c`

Prototype

```
CPU_BOOLEAN  NetIPv6_CfgFragReasmTimeout (CPU_INT08U  timeout_sec)
```

Arguments

`timeout_sec`

Desired value for IPv6 fragment reassembly timeout (in seconds).

Returned Value

- `DEF_OK` , IPv6 fragment reassembly timeout configured.
- `DEF_FAIL` , otherwise.

Notes / Warnings

IPv6 fragment reassembly timeout is the maximum time allowed between received IPv6 fragments from the same IPv6 datagram.

NetIPv6_GetAddrHost()

Description

Gets an interface's IPv6 host address(es).

Files

`net_ipv6.h/net_ipv6.c`

Prototype

```
CPU_BOOLEAN  NetIPv6_GetAddrHost (NET_IF_NBR        if_nbr,
                                  NET_IPv6_ADDR     *p_addr_tbl,
                                  NET_IP_ADDRS_QTY  *p_addr_tbl_qty,
                                  RTOS_ERR          *p_err)
```

Arguments

`if_nbr`

Interface number to get IPv6 host address(s).

`p_addr_tbl`

Pointer to the IPv6 address table that will receive the IPv6 host address(s) in host-order for this interface.

`p_addr_tbl_qty`

Pointer to a variable to :

1. Pass the size of the address table, in number of IPv6 addresses, pointed to by ' `p_addr_tbl` '.
2. (a) Return the actual number of IPv6 addresses, if NO error(s).
   (b) Return 0, otherwise.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- `DEF_OK` , if the interface's IPv6 host address(s) are successfully returned.
- `DEF_FAIL` , otherwise.

Notes / Warnings

None.

NetIPv6_GetAddrSrc()

Description

Finds the best matched source address in the IPv6 configured host addresses for the specified destination address.

Files

`net_ipv6.h/net_ipv6.c`

Prototype

```
const  NET_IPv6_ADDR_OBJ  *NetIPv6_GetAddrSrc (      NET_IF_NBR     *p_if_nbr,
                                               const  NET_IPv6_ADDR  *p_addr_src,
                                               const  NET_IPv6_ADDR  *p_addr_dest,
                                                      NET_IPv6_ADDR  *p_addr_nexthop,
                                                      RTOS_ERR       *p_err)
```

Arguments

`p_if_nbr`

Pointer to given interface number if any. Variable that will received the interface number if none is given.

`p_addr_src`

Pointer to the IPv6 suggested source address if any. `DEF_NULL` if no source address is preferred.

`p_addr_dest`

Pointer to the destination address.

`p_addr_nexthop`

Pointer to Next Hop IPv6 address that the function will find.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_NET_NEXT_HOP`

Returned Value

Pointer to the IPv6 addresses structure associated with the best source address for the given destination.

Notes / Warnings

None.

NetIPv6_GetAddrMatchingLen()

Description

Computes the number of identical most significant bits of two IPv6 addresses.

Files

`net_ipv6.h/net_ipv6.c`

Prototype

```
CPU_INT08U  NetIPv6_GetAddrMatchingLen (const  NET_IPv6_ADDR  *p_addr_1,
                                        const  NET_IPv6_ADDR  *p_addr_2,
                                               RTOS_ERR       *p_err)
```

Arguments

`p_addr_1`

First IPv6 address for comparison.

`p_addr_2`

Second IPv6 address for comparison.

Returned Value

- Number of matching bits, if any.
- 0, otherwise.

Notes / Warnings

1. The returned number is based on the number of matching MSB of both IPv6 addresses:
   (a) Calling the function with the following addresses will return 32 matching bits:
      - p_addr_1 : FE80:ABCD:0000:0000:0000:0000:0000:0000
      - p_addr_2 : FE80:ABCD:F000:0000:0000:0000:0000:0000
   (b) Calling the function with the following addresses will return 127 matching bits:
      - p_addr_1 : FE80:ABCD:0000:0000:0000:0000:0000:0000
      - p_addr_2 : FE80:ABCD:0000:0000:0000:0000:0000:0001
   (c) Calling the function with the following addresses will return 0 matching bits:
      - p_addr_1 : FE80:ABCD:0000:0000:0000:0000:0000:0000
      - p_addr_2 : 7E80:ABCD:0000:0000:0000:0000:0000:0000
   (d) Calling the function with identical addresses will return 128 matching bits.

NetIPv6_GetAddrScope()

Description

Gets the scope of the given IPv6 address.

Files

`net_ipv6.h/net_ipv6.c`

Prototype

```
NET_IPv6_SCOPE  NetIPv6_GetAddrScope (const NET_IPv6_ADDR  *p_addr)
```

Arguments

`p_addr`

Pointer to the IPv6 address.

Returned Value

Scope of the given IPv6 address.

Notes / Warnings

1. For an unicast address, the scope is given by the 16 first bits of the address. Three scopes are possible:
   (a) 0xFE80 Link-local
   (b) 0xFEC0 Site-local -> Deprecated
   (c) others Global
2. For a multicast address, the scope is given by a four bits field inside the address. Current possible scopes are:
   (a) 0x0 reserved
   (b) 0x1 interface-local
   (c) 0x2 link-local
   (d) 0x3 reserved
   (e) 0x4 admin-local
   (f) 0x5 site-local
   (g) 0x6 unassigned
   (h) 0x7 unassigned
   (i) 0x8 organization-local
   (j) 0x9 unassigned
   (k) 0xA unassigned
   (l) 0xB unassigned
   (m) 0xC unassigned

(n) 0xD unassigned
(o) 0xE global
(p) 0xF reserved

NetIPv6_IsAddrHostCfgd()

Description

Validates an IPv6 address as a configured IPv6 host address on an enabled interface.

Files

`net_ipv6.h/net_ipv6.c`

Prototype

```
CPU_BOOLEAN  NetIPv6_IsAddrHostCfgd (const  NET_IPv6_ADDR  *p_addr)
```

Arguments

`p_addr`

Pointer to the IPv6 address to validate.

Returned Value

- `DEF_YES` , if IPv6 address is one of the host's configured IPv6 addresses.
- `DEF_NO` , otherwise.

Notes / Warnings

None.

NetIPv6_IsAddrsCfgdOnIF()

Description

Checks if any IPv6 host address(s) configured on an interface.

Files

`net_ipv6.h/net_ipv6.c`

Prototype

```
CPU_BOOLEAN  NetIPv6_IsAddrsCfgdOnIF (NET_IF_NBR   if_nbr,
                                      RTOS_ERR    *p_err)
```

Arguments

`if_nbr`

Interface number to check for configured IPv6 host address(s).

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_HANDLE

Returned Value

- DEF_YES , if any IP host address(s) configured on interface.
- DEF_NO , otherwise.

Notes / Warnings

None.

NetIPv6_IsValidAddrHost()

Description

1. Validates an IPv6 host address :
   (a) MUST NOT be one of the following :
      1. The unspecified address
      2. The loopback address
      3. A Multicast address

Files

net_ipv6.h/net_ipv6.c

Prototype

```
CPU_BOOLEAN  NetIPv6_IsValidAddrHost (const  NET_IPv6_ADDR  *p_addr_host)
```

Arguments

p_addr_host

Pointer to the IPv6 host address to validate.

Returned Value

- DEF_YES , if IPv6 host address valid.
- DEF_NO , otherwise.

Notes / Warnings

None.

NetIPv6_IsAddrLinkLocal()

Description

1. Validates an IPv6 address as a link-local IPv6 address.
   (a) In a text representation, it corresponds to the following format:

```
 FE80:0000:0000:0000:aaaa.bbbb.cccc.dddd       OR:
 FE80::aaaa.bbbb.cccc.dddd                     WHERE:

   aaaa.bbbb.cccc.dddd is the interface ID.
```

Files

`net_ipv6.h/net_ipv6.c`

Prototype

```
CPU_BOOLEAN  NetIPv6_IsAddrLinkLocal (const  NET_IPv6_ADDR  *p_addr)
```

Arguments

`p_addr`

Pointer to the IPv6 address to validate

Returned Value

- `DEF_YES` , if IPv6 address is a link-local IPv6 address.
- `DEF_NO` , otherwise.

Notes / Warnings

None.

NetIPv6_IsAddrSiteLocal()

Description

1. Validates an IPv6 address as a site-local address :
   (a) In a text representation, it corresponds to the following format:

```
FEC0:AAAA:BBBB:CCCC:DDDD:aaaa.bbbb.cccc.dddd WHERE:

AAAA.BBBB.CCCC.DDDD is the subnet ID AND ...
aaaa.bbbb.cccc.dddd is the interface ID.
```

Files

`net_ipv6.h/net_ipv6.c`

Prototype

```
CPU_BOOLEAN  NetIPv6_IsAddrSiteLocal (const  NET_IPv6_ADDR  *p_addr)
```

Arguments

`p_addr`

Pointer to the IPv6 address to validate.

Returned Value

- `DEF_YES` , if IPv6 address is a site-local IPv6 address.
- `DEF_NO` , otherwise.

Notes / Warnings

None.

NetIPv6_IsAddrMcast()

Description

Validates an IPv6 address as a multicast address.

Files

`net_ipv6.h/net_ipv6.c`

Prototype

```
CPU_BOOLEAN  NetIPv6_IsAddrMcast (const  NET_IPv6_ADDR  *p_addr)
```

Arguments

`p_addr`

Pointer to the IPv6 address to validate.

Returned Value

- `DEF_YES` , if IPv6 address is a multicast IPv6 address.
- `DEF_NO` , otherwise.

Notes / Warnings

Refer to RFC #4291, Section 2.6 for the Multicast IPv6 address format.

NetIPv6_IsAddrMcastAllRouters()

Description

1. Validates that an IPv6 address is a multicast to all routers IPv6 address :
   (a) RFC #4291 Section 2.7.1 specifies that the following IPv6 addresses "identify the group of all IPv6 routers" within their respective scope :
    1. FF01:0000:0000:0000:0000:0000:0000:2 Scope 1 -> Interface-local
    2. FF02:0000:0000:0000:0000:0000:0000:2 Scope 2 -> Link-local
    3. FF05:0000:0000:0000:0000:0000:0000:2 Scope 5 -> Site-local

Files

`net_ipv6.h/net_ipv6.c`

Prototype

```
CPU_BOOLEAN  NetIPv6_IsAddrMcastAllRouters (const  NET_IPv6_ADDR  *p_addr)
```

Arguments

`p_addr`

Pointer to the IPv6 address to validate (see Note #2).

Returned Value

- `DEF_YES` , if IPv6 address is a multicast to all routers IPv6 address.
- `DEF_NO` , otherwise.

Notes / Warnings

None.

NetIPv6_IsAddrMcastAllNodes()

Description

1. Validates that an IPv6 address is a multicast to all routers IPv6 nodes :
   (a) RFC #4291 Section 2.7.1 specifies that the following IPv6 addresses "identify the group of all IPv6 nodes" within their respective scope :
      1. FF01:0000:0000:0000:0000:0000:0000:1 Scope 1 -> Interface-local
      2. FF02:0000:0000:0000:0000:0000:0000:1 Scope 2 -> Link-local

Files

`net_ipv6.h/net_ipv6.c`

Prototype

```
CPU_BOOLEAN  NetIPv6_IsAddrMcastAllNodes (const  NET_IPv6_ADDR  *p_addr)
```

Arguments

`p_addr`

Pointer to the IPv6 address to validate.

Returned Value

- `DEF_YES` , if IPv6 address is a multicast to all nodes IPv6 address.
- `DEF_NO` , otherwise.

Notes / Warnings

None.

NetIPv6_IsAddrMcastSolNode()

Description

Validates that an IPv6 address is the solicited node multicast address associated with an IPv6 unicast address.

Files

`net_ipv6.h/net_ipv6.c`

Prototype

```
CPU_BOOLEAN  NetIPv6_IsAddrMcastSolNode (const  NET_IPv6_ADDR  *p_addr,
                                         const  NET_IPv6_ADDR  *p_addr_input)
```

Arguments

`p_addr`

Pointer to the IPv6 address to validate.

`p_addr_input`

Pointer to input IPv6 unicast address.

Returned Value

- `DEF_YES` , if IPv6 address is a solicited node multicast IPv6 address.
- `DEF_NO` , otherwise.

Notes / Warnings

1. RFC #4291 Section 2.7.1 specifies that "Solicited-Node multicast address are computed as a function of a node's unicast and anycast addresses. A Solicited-Node multicast address is formed by taking the low-order 24 bits of an address (unicast or anycast) and appending those bits to the prefix FF02:0:0:0:0:1:FF00::/104 resulting in a multicast address in the range" from :
   (a) `FF02:0000:0000:0000:0000:0001:FF00:0000` to
   (b) `FF02:0000:0000:0000:0000:0001:FFFF:FFFF"`

NetIPv6_IsAddrMcastRsvd()

Description

1. Validates that an IPv6 address is a reserved multicast IPv6 address :
   (a) RFC #4291 Section 2.7.1 specifies that the following addresses "are reserved and shall never be assigned to any multicast group" :
   FF00:0:0:0:0:0:0:0
   FF01:0:0:0:0:0:0:0
   FF02:0:0:0:0:0:0:0
   FF03:0:0:0:0:0:0:0
   FF04:0:0:0:0:0:0:0
   FF05:0:0:0:0:0:0:0
   FF06:0:0:0:0:0:0:0
   FF08:0:0:0:0:0:0:0
   FF09:0:0:0:0:0:0:0
   FF0A:0:0:0:0:0:0:0
   FF0B:0:0:0:0:0:0:0
   FF0C:0:0:0:0:0:0:0
   FF0D:0:0:0:0:0:0:0
   FF0E:0:0:0:0:0:0:0
   FF0F:0:0:0:0:0:0:0

Files

`net_ipv6.h/net_ipv6.c`

Prototype

```
CPU_BOOLEAN NetIPv6_IsAddrMcastRsvd (const NET_IPv6_ADDR *p_addr)
```

Arguments

`p_addr`

Pointer to the IPv6 address to validate (see Note #2).

Returned Value

- `DEF_YES` , if IPv6 address is a reserved multicast IPv6 addresss.
- `DEF_NO` , otherwise.

Notes / Warnings

None.

NetIPv6_IsAddrUnspecified()

Description

1. Validates that an IPv6 address is the unspecified IPv6 address :
   (a) RFC #4291 Section 2.5.2 specifies that the following unicast address "is called the unspecified address" :
   `0000:0000:0000:0000:0000:0000:0000:0000`

Files

`net_ipv6.h/net_ipv6.c`

Prototype

```
CPU_BOOLEAN  NetIPv6_IsAddrUnspecified (const  NET_IPv6_ADDR  *p_addr)
```

Arguments

`p_addr`

Pointer to the IPv6 address to validate.

Returned Value

- `DEF_YES` , if IPv6 address is the unspecified address.
- `DEF_NO` , otherwise.

Notes / Warnings

This address indicates the absence of an address and MUST NOT be assigned to any physical interface. However, it can be used in some cases in the Source Address field of IPv6 packets sent by a host before it learns its own address.

NetIPv6_IsAddrLoopback()

Description

1. Validates that an IPv6 address is the IPv6 loopback address :

(a) RFC #4291 Section 2.5.3 specifies that the following unicast address "is called the loopback address" :

`0000:0000:0000:0000:0000:0000:0000:0001`

Files

`net_ipv6.h/net_ipv6.c`

Prototype

```
CPU_BOOLEAN  NetIPv6_IsAddrLoopback (const  NET_IPv6_ADDR  *p_addr)
```

Arguments

`p_addr`

Pointer to the IPv6 address to validate (see Note #2).

Returned Value

- `DEF_YES` , if IPv6 address is the IPv6 loopback address.
- `DEF_NO` , otherwise.

Notes / Warnings

This address may be used by a node to send an IPv6 packet to itself and MUST NOT be assigned to any physical interface.

NetIPv6_AddrTypeValidate()

Description

Validates the type of an IPv6 address.

Files

`net_ipv6.h/net_ipv6.c`

Prototype

```
NET_IPv6_ADDR_TYPE  NetIPv6_AddrTypeValidate (const  NET_IPv6_ADDR  *p_addr,
                                              NET_IF_NBR      if_nbr)
```

Arguments

`p_addr`

Pointer to the IPv6 address to validate.

`if_nbr`

Interface number associated to the address.

Returned Value

`NET_IPv6_ADDR_TYPE_MCAST` , if IPv6 address is an unknown type muticast address.

- `NET_IPv6_ADDR_TYPE_MCAST_SOL` , if IPv6 address is the multicast solicited node address.
- `NET_IPv6_ADDR_TYPE_MCAST_ROUTERS` , if IPv6 address is the multicast all routers address.
- `NET_IPv6_ADDR_TYPE_MCAST_NODES` , if IPv6 address is the multicast all nodes address.
- `NET_IPv6_ADDR_TYPE_LINK_LOCAL` , if IPv6 address is a link-local address.
- `NET_IPv6_ADDR_TYPE_SITE_LOCAL` , if IPv6 address is a site-local address.
- `NET_IPv6_ADDR_TYPE_UNSPECIFIED` , if IPv6 address is the unspecified address.
- `NET_IPv6_ADDR_TYPE_LOOPBACK` , if IPv6 address is the loopback address.
- `NET_IPv6_ADDR_TYPE_UNICAST` , otherwise.

Notes / Warnings

None.

NetIPv6_CreateIF_ID()

Description

Creates an IPv6 interface identifier.

Files

`net_ipv6.h/net_ipv6.c`

Prototype

```
CPU_INT08U  NetIPv6_CreateIF_ID (NET_IF_NBR            if_nbr,
                                 NET_IPv6_ADDR         *p_addr_id,
                                 NET_IPv6_ADDR_ID_TYPE  id_type,
                                 RTOS_ERR              *p_err)
```

Arguments

`if_nbr`

Network interface number to obtain the link-layer hardware address.

`p_addr_id`

Pointer to the IPv6 address that will receive the IPv6 interface identifier.

`id_type`

IPv6 interface identifier type:

`NET_IPv6_ADDR_AUTO_CFG_ID_IEEE_48` Universal token from IEEE 802 48-bit MAC

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

None.

Notes / Warnings

1. Universal interface identifier generated from IEEE 802 48-bit MAC address is the only interface identifier actually supported.
2. RFC #4291, Appendix A, states that "[EUI-64] actually defines 0xFF and 0xFF as the bits to be inserted to create an IEEE EUI-64 identifier from an IEEE MAC-48 identifier. The 0xFF and 0xFE values are used when starting with an IEEE EUI-48 identifier."

NetIPv6_CreateAddrFromID()

Description

1. Creates an IPv6 address from a prefix and an identifier.
   (a) Validate prefix length.

(b) Append address ID to IPv6 address prefix.

Files

`net_ipv6.h/net_ipv6.c`

Prototype

```
void  NetIPv6_CreateAddrFromID (NET_IPv6_ADDR              *p_addr_id,
                                NET_IPv6_ADDR              *p_addr_prefix,
                                NET_IPv6_ADDR_PREFIX_TYPE   prefix_type,
                                CPU_SIZE_T                  prefix_len,
                                RTOS_ERR                   *p_err)
```

Arguments

`p_addr_id`

Pointer to the IPv6 address ID.

`p_addr_prefix`

Pointer to variable that will receive the created IPv6 address.

`prefix_type`

Prefix type:

- `NET_IPv6_ADDR_PREFIX_CUSTOM` Custom prefix type
- `NET_IPv6_ADDR_PREFIX_LINK_LOCAL` Link-local prefix type

`prefix_len`

Prefix len (in bits).

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

1. If the prefix type is custom, p_addr_prefix SHOULD point on a variable that contain the prefix address. The address ID will be append to this prefix address. If the prefix type is link-local, the content of the variable pointed by p_addr_prefix does not matter and will be overwritten.
2. If the prefix type is link-local, prefix address SHOULD be initialized to the unspecified IPv6 address to make sure resulting address does not have any uninitialized values (i.e., if any of the lower 8 octets of the ID address are NOT initialized).

NetIPv6_MaskGet()

Description

Gets an IPv6 mask based on prefix length.

Files

`net_ipv6.h/net_ipv6.c`

Prototype

```
void NetIPv6_MaskGet (NET_IPv6_ADDR *p_mask_rtn,
                      CPU_INT08U     prefix_len,
                      RTOS_ERR      *p_err)
```

Arguments

`p_mask_rtn`

Pointer to the IPv6 address mask to set.

`prefix_len`

Length of the prefix mask in bits.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

None.

NetIPv6_AddrMaskByPrefixLen()

Description

Gets the IPv6 address masked with the prefix length.

Files

`net_ipv6.h/net_ipv6.c`

Prototype

```
void NetIPv6_AddrMaskByPrefixLen (const NET_IPv6_ADDR *p_addr,
                                  CPU_INT08U     prefix_len,
                                  NET_IPv6_ADDR *p_addr_rtn,
                                  RTOS_ERR      *p_err)
```

Arguments

`p_addr`

Pointer to the IPv6 address.

`prefix_len`

IPv6 address prefix length.

`p_addr_rtn`

Pointer to the IPv6 address that will receive the masked address.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

None.

NetIPv6_AddrMask()

Description

Applies IPv6 mask on an address.

Files

`net_ipv6.h/net_ipv6.c`

Prototype

```
void NetIPv6_AddrMask (const NET_IPv6_ADDR  *p_addr,
                       const NET_IPv6_ADDR  *p_mask,
                             NET_IPv6_ADDR  *p_addr_rtn)
```

Arguments

`p_addr`

Pointer to the IPv6 address to be masked.

`p_mask`

Pointer to the IPv6 address mask.

`p_addr_rtn`

Pointer to the IPv6 address that will received the masked address.

Returned Value

None.

Notes / Warnings

None.

TCP API

Function Name	Description
NetTCP_ConnCfgIdleTimeout()	Configures the TCP connection's idle timeout.
NetTCP_ConnCfgMaxSegSizeLocal()	Configures the TCP connection's local maximum segment size.
NetTCP_ConnCfgMSL_Timeout()	Configures the TCP connection's maximum segment lifetime (MSL) timeout.
NetTCP_ConnCfgReTxMaxTh()	Configures the TCP connection's maximum number of same segment retransmissions.
NetTCP_ConnCfgReTxMaxTimeout()	Configures the TCP connection's maximum retransmission timeout.
NetTCP_ConnCfgRxWinSize()	Configures the TCP connection's receive window size.
NetTCP_ConnCfgTxAckDlyTimeout()	Configures the TCP connection's transmit acknowledgment delay timeout.
NetTCP_ConnCfgTxAckImmedRxdPushEn()	Configures the TCP connection's transmit immediate acknowledgment for received and pushed TCP segments.
NetTCP_ConnCfgTxKeepAliveEn()	Configures the TCP connection's transmit keep-alive enable.
NetTCP_ConnCfgTxKeepAliveRetryTimeout()	Configures the TCP connection's transmit keep-alive retry timeout.
NetTCP_ConnCfgTxKeepAliveTh()	Configures the TCP connection's maximum number of consecutive keep-alives to transmit.
NetTCP_ConnCfgTxNagleEn()	Configures the TCP connection's transmit Nagle algorithm enable.
NetTCP_ConnCfgTxWinSize()	Configures the TCP connection's transmit window size.
NetTCP_ConnPoolStatGet()	Gets the TCP connections' statistics pool.
NetTCP_ConnPoolStatResetMaxUsed()	Resets the TCP connections' statistics pool's maximum number of entries used.
NetTCP_ConnStateGet()	Retrieves the TCP Connection State.

NetTCP_ConnCfgIdleTimeout()

Description

Configures the TCP connection's idle timeout.

Files

`net_tcp.h/net_tcp.c`

Prototype

```
CPU_BOOLEAN  NetTCP_ConnCfgIdleTimeout (NET_TCP_CONN_ID      conn_id_tcp,
                                        NET_TCP_TIMEOUT_SEC  timeout_sec,
                                        RTOS_ERR             *p_err)
```

Arguments

`conn_id_tcp`

Handle identifier of TCP connection to configure connection idle timeout.

`timeout_sec`

Desired value for TCP connection idle timeout (in seconds).

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- `DEF_OK` , TCP connection idle timeout is successfully configured.
- `DEF_FAIL` , otherwise.

Notes / Warnings

Configured timeout does NOT reschedule any current idle timeout in progress but becomes effective the next time a TCP connection sets its idle timeout.

NetTCP_ConnCfgMSL_Timeout()

Description

Configures the TCP connection's maximum segment lifetime (MSL) timeout.

Files

`net_tcp.h/net_tcp.c`

Prototype

```
CPU_BOOLEAN NetTCP_ConnCfgMSL_Timeout (NET_TCP_CONN_ID      conn_id_tcp,
                                       NET_TCP_TIMEOUT_SEC  msl_timeout_sec,
                                       RTOS_ERR            *p_err)
```

Arguments

`conn_id_tcp`

Handle identifier of TCP connection to configure MSL value.

`msl_timeout_sec`

Desired value for TCP connection MSL timeout (in seconds).

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- `DEF_OK` , TCP connection MSL timeout is successfully configured.
- `DEF_FAIL` , otherwise.

Notes / Warnings

None.

NetTCP_ConnCfgMaxSegSizeLocal()

Description

Configures the TCP connection's local maximum segment size.

Files

`net_tcp.h/net_tcp.c`

Prototype

```
CPU_BOOLEAN  NetTCP_ConnCfgMaxSegSizeLocal (NET_TCP_CONN_ID    conn_id_tcp,
                                            NET_TCP_SEG_SIZE   max_seg_size,
                                            RTOS_ERR          *p_err)
```

Arguments

`conn_id_tcp`

Handle identifier of TCP connection to configure local maximum segment size.

`max_seg_size`

Desired maximum segment size.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_INVALID_STATE`

Returned Value

- `DEF_OK` , TCP connection local maximum segment size successfully configured.
- `DEF_FAIL` , otherwise.

Notes / Warnings

1. RFC #793, Section 3.1 'Header Format : Options : Maximum Segment Size' states that a TCP connection advertises its "maximum receive segment size ... only ... in the initial connection request (i.e., in segments with the SYN control bit set)."
   (a) Actively- connected TCP connection
   (b) Passively-connected TCP connection, which is cloned from its previously- configured LISTEN-state TCP connection

NetTCP_ConnCfgReTxMaxTh()

Description

Configures the TCP connection's maximum number of same segment retransmissions.

Files

`net_tcp.h/net_tcp.c`

Prototype

```
CPU_BOOLEAN  NetTCP_ConnCfgReTxMaxTh (NET_TCP_CONN_ID   conn_id_tcp,
                                      NET_PKT_CTR       nbr_max_re_tx,
                                      RTOS_ERR         *p_err)
```

Arguments

conn_id_tcp

Handle identifier of TCP connection to configure maximum number of same segment retransmissions.

nbr_max_re_tx

Desired maximum number of same segment retransmissions.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_HANDLE

Returned Value

- DEF_OK , TCP connection maximum number of same segment retransmissions are successfully configured.
- DEF_FAIL , otherwise.

Notes / Warnings

RFC #1122, Section 4.2.3.5 states that "when the number of transmissions of the same segment reaches a threshold ... close the connection."

NetTCP_ConnCfgReTxMaxTimeout

Description

Configures the TCP connection's maximum retransmission timeout.

Files

net_tcp.h/net_tcp.c

Prototype

```
CPU_BOOLEAN NetTCP_ConnCfgReTxMaxTimeout (NET_TCP_CONN_ID       conn_id_tcp,
                                          NET_TCP_TIMEOUT_SEC   timeout_sec,
                                          RTOS_ERR             *p_err)
```

Arguments

conn_id_tcp

Handle identifier of TCP connection to configure maximum retransmission timeout.

timeout_sec

Desired value for TCP connection maximum retransmission timeout (in seconds).

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_HANDLE

Returned Value

- `DEF_OK` , TCP connection maximum retransmission timeout is successfully configured.
- `DEF_FAIL` , otherwise.

Notes / Warnings

1. (a) RFC #2988, Section 2.4 states that "a maximum value MAY be placed on RTO provided it is at least 60 seconds."
   (b) RFC #1122, Section 4.2.3.1 states that "the recommended ... RTO ... upper bound should be 2*MSL."
   (c) Stevens, TCP/IP Illustrated, Volume 1, 8th Printing, Section 21.2, Page 299 states that "the timeout value ... [has] an upper limit of 64 seconds."
2. Configured timeout does NOT reschedule any current re-transmission timeout in progress, but becomes effective the next time a TCP connection sets its re-transmission timeout.

NetTCP_ConnCfgRxWinSize()

Description

Configures the TCP connection's receive window size.

Files

`net_tcp.h/net_tcp.c`

Prototype

```
CPU_BOOLEAN  NetTCP_ConnCfgRxWinSize (NET_TCP_CONN_ID    conn_id_tcp,
                                      NET_TCP_WIN_SIZE   win_size,
                                      RTOS_ERR          *p_err)
```

Arguments

`conn_id_tcp`

Handle identifier of TCP connection to configure receive window size.

`win_size`

Desired receive window size.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_INVALID_STATE`

Returned Value

- `DEF_OK` , TCP connection receive window size successfully configured.
- `DEF_FAIL` , otherwise.

Notes / Warnings

1. A TCP connection's receive window size SHOULD NOT be updated after the TCP connection is connected. Thus any configuration of the local receive window size MUST be performed by the application layer PRIOR to any TCP connection request/synchronization either from the following :
   (a) Actively- connected TCP connection
   (b) Passively-connected TCP connection, which is cloned from its previously- configured LISTEN-state TCP connection

NetTCP_ConnCfgTxAckDlyTimeout()

Description

Configures the TCP connection's transmit acknowledgment delay timeout.

Files

`net_tcp.h/net_tcp.c`

Prototype

```
CPU_BOOLEAN  NetTCP_ConnCfgTxAckDlyTimeout (NET_TCP_CONN_ID      conn_id_tcp,
                                            NET_TCP_TIMEOUT_MS   timeout_ms,
                                            RTOS_ERR            *p_err)
```

Arguments

`conn_id_tcp`

Handle identifier of TCP connection to configure transmit acknowledgment timeout.

`timeout_ms`

Desired value for TCP connection transmit acknowledgment delay timeout.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- `DEF_OK` , TCP connection transmit acknowledgment delay timeout is successfully configured.
- `DEF_FAIL` , otherwise.

Notes / Warnings

1. (a) RFC #1122, Section 4.2.3.2 states that "an ACK should not be excessively delayed; in particular, the delay MUST be less than 0.5 seconds."
   (b) RFC #2581, Section 4.2 reiterates that "an ACK ... MUST be generated within 500 ms of the arrival of the first unacknowledged packet."
2. Configured timeout does NOT reschedule any current acknowledgment delay timeout in progress but becomes effective the next time a TCP connection sets an acknowledgment delay timeout.

NetTCP_ConnCfgTxAckImmedRxdPushEn

Description

Configures the TCP connection's transmit immediate acknowledgment for received and pushed TCP segments enable.

Files

net_tcp.h/net_tcp.c

Prototype

```
CPU_BOOLEAN NetTCP_ConnCfgTxAckImmedRxdPushEn (NET_TCP_CONN_ID   conn_id_tcp,
                                               CPU_BOOLEAN       tx_immed_ack_en,
                                               RTOS_ERR         *p_err)
```

Arguments

`conn_id_tcp`

Handle identifier of TCP connection to configure transmit immediate acknowledgment.

`tx_immed_ack_en`

Desired value for TCP connection transmit immediate acknowledgment.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- `DEF_OK` , TCP connection transmits immediate acknowledgment for received and pushed TCP segments enable is successfully configured.
- `DEF_FAIL` , otherwise.

Notes / Warnings

RFC #813, Section 5 states that "the receiver of data will refrain from sending an acknowledgment under certain circumstances ... The most obvious event on which to depend is the arrival of another segment. So, if a segment arrives, postpone sending an acknowledgment if ... the push bit is not set in the segment, since it is a reasonable assumption that there is more data coming in a subsequent segment."

NetTCP_ConnCfgTxKeepAliveEn()

Description

Configures the TCP connection's transmit keep-alive algorithm enable.

Files

`net_tcp.h/net_tcp.c`

Prototype

```
CPU_BOOLEAN  NetTCP_ConnCfgTxKeepAliveEn (NET_TCP_CONN_ID   conn_id_tcp,
                                          CPU_BOOLEAN       keep_alive_en,
                                          RTOS_ERR         *p_err)
```

Arguments

`conn_id_tcp`

Handle identifier of TCP connection to configure transmit keep-alive enable.

`keep_alive_en`

Desired value for TCP connection transmit keep-alive enable.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- `DEF_OK` , TCP connection transmit keep-alive enable is successfully configured.
- `DEF_FAIL` , otherwise.

Notes / Warnings

RFC #1122, Section 4.2.3.6 states that "if keep-alives are included, the application MUST be able to turn them on or off for each TCP connection."

NetTCP_ConnCfgTxKeepAliveRetryTimeout

Description

Configures the TCP connection's transmit keep-alive retry timeout.

Files

net_tcp.h/net_tcp.c

Prototype

```
CPU_BOOLEAN NetTCP_ConnCfgTxKeepAliveRetryHandler (NET_TCP_CONN_ID       conn_id_tcp,
                                                   NET_TCP_TIMEOUT_SEC   timeout_sec)
```

Arguments

`conn_id_tcp`

Handle identifier of TCP connection to configure transmit keep-alive retry timeout.

`timeout_sec`

Desired value for TCP connection transmit keep-alive retry timeout (in seconds).

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- `DEF_OK` , TCP connection transmit keep-alive retry timeout is successfully configured.
- `DEF_FAIL` , otherwise.

Notes / Warnings

Configured timeout does NOT reschedule any current keep-alive retry timeout in progress but becomes effective the next time a TCP connection sets its keep-alive retry timeout.

NetTCP_ConnCfgTxKeepAliveTh()

Description

Configures the TCP connection's maximum number of consecutive keep-alives to transmit.

Files

`net_tcp.h/net_tcp.c`

Prototype

```
CPU_BOOLEAN  NetTCP_ConnCfgTxKeepAliveTh (NET_TCP_CONN_ID   conn_id_tcp,
                                          NET_PKT_CTR       nbr_max_keep_alive,
                                          RTOS_ERR         *p_err)
```

Arguments

`conn_id_tcp`

Handle identifier of TCP connection to configure transmit keep-alive threshold.

`nbr_max_keep_alive`

Desired maximum number of consecutive keep-alives to transmit.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- `DEF_OK` , TCP connection transmit keep-alive threshold successfully configured.
- `DEF_FAIL` , otherwise.

Notes / Warnings

Stevens, TCP/IP Illustrated, Volume 1, 8th Printing, Section 23.3 'Other End Crashes', Pages 334-335 states "that the [remote host] ... send[s] ... [N] keepalive probes ... before declaring the connection dead."

NetTCP_ConnCfgTxNagleEn()

Description

Configures the TCP connection's transmit Nagle algorithm enable.

Files

`net_tcp.h/net_tcp.c`

Prototype

```
CPU_BOOLEAN  NetTCP_ConnCfgTxNagleEn (NET_TCP_CONN_ID   conn_id_tcp,
                                      CPU_BOOLEAN      nagle_en,
                                      RTOS_ERR        *p_err)
```

Arguments

`conn_id_tcp`

Handle identifier of TCP connection to configure transmit Nagle enable.

`nagle_en`

Desired value for TCP connection transmit Nagle enable :

- `DEF_ENABLED` TCP connections delay transmitting next data segment(s) until all unacknowledged data is acknowledged OR an MSS-sized segment can be transmitted.
- `DEF_DISABLED` TCP connections transmit all data segment(s) when permitted by local and remote hosts' congestion controls.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- `DEF_OK` , TCP connection transmit Nagle enable is successfully configured.
- `DEF_FAIL` , otherwise.

Notes / Warnings

RFC #1122, Section 4.2.3.4 also states that "a TCP SHOULD implement the Nagle Algorithm ... However, there MUST be a way for an application to disable the Nagle algorithm on an individual connection."

NetTCP_ConnCfgTxWinSize()

Description

Configures the TCP connection's transmit window size.

Files

`net_tcp.h/net_tcp.c`

Prototype

```
CPU_BOOLEAN  NetTCP_ConnCfgTxWinSize (NET_TCP_CONN_ID    conn_id_tcp,
                                      NET_TCP_WIN_SIZE   win_size,
                                      RTOS_ERR          *p_err)
```

Arguments

`conn_id_tcp`

Handle identifier of TCP connection to configure transmit window size.

`win_size`

Desired transmit window size.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_INVALID_STATE`

Returned Value

- `DEF_OK` , TCP connection transmit window size successfully configured.
- `DEF_FAIL` , otherwise.

Notes / Warnings

1. A TCP connection's transmit window size SHOULD NOT be updated after the TCP connection is connected. Thus any configuration of the local transmit window size MUST be performed by the application layer PRIOR to any TCP connection request/synchronization either from the following :
   (a) Actively- connected TCP connection
   (b) Passively-connected TCP connection, which is cloned from its previously- configured LISTEN-state TCP connection.

NetTCP_ConnPoolStatGet()

Description

Gets the TCP connection statistics pool.

Files

`net_tcp.h/net_tcp.c`

Prototype

```
NET_STAT_POOL  NetTCP_ConnPoolStatGet (void)
```

Arguments

Returned Value

TCP connection statistics pool, if NO error(s). NULL statistics pool, otherwise.

Notes / Warnings

None.

NetTCP_ConnPoolStatResetMaxUsed()

Description

Resets the TCP connection's statistics pool's maximum number of entries used.

Files

`net_tcp.h/net_tcp.c`

Prototype

```
void NetTCP_ConnPoolStatResetMaxUsed (void)
```

Arguments

Returned Value

None.

Notes / Warnings

None.

NetTCP_ConnStateGet()

Description

Retrieves the TCP Connection State.

Files

`net_tcp.h/net_tcp.c`

Prototype

```
NET_TCP_CONN_STATE NetTCP_ConnStateGet (NET_TCP_CONN_ID conn_id)
```

Arguments

`conn_id`

TCP Connection ID number.

Returned Value

TCP Connection State.

Notes / Warnings

None.

Socket Functions API

Function Name	Description
NET_SOCK_DESC_CLR()	Remove a socket file descriptor ID as a member of a file descriptor set.
NET_SOCK_DESC_COPY()	Copy a file descriptor set to another file descriptor set.
NET_SOCK_DESC_INIT()	Initialize/zero-clear a file descriptor set.
NET_SOCK_DESC_IS_SET()	Check if a socket file descriptor ID is a member of a file descriptor set.
NET_SOCK_DESC_SET()	Add a socket file descriptor ID as a member of a file descriptor set.
NetSock_Accept()	Wait for new socket connections on a listening server socket.
NetSock_Bind()	Assign network addresses to sockets.

Function Name	Description
NetSock_CfgBlock()	Configure a socket's blocking mode.
NetSock_CfgConnChildQ_SizeGet()	Get socket's connection child queue size value.
NetSock_CfgConnChildQ_SizeSet()	Configure socket's child connection queue size.
NetSock_CfgIF()	Configure the interface that must be used by the socket.
NetSock_CfgRxQ_Size()	Configure socket's receive queue size.
NetSock_CfgSecure()	Configure a socket's secure mode.
NetSock_CfgSecureClientCertKeyInstall()	Install certificate and key that must be used by a client for mutual authentication.
NetSock_CfgSecureClientCommonName()	Configure client socket's common name.
NetSock_CfgSecureClientTrustCallBack()	Configure client socket's trust call back function.
NetSock_CfgSecureServerCertKeyInstall()	Install certificate (CERT) and private key (KEY) from a buffer which must be used by a server.
NetSock_CfgTimeoutConnAcceptDflt()	Set socket's connection accept timeout to configured-default value.
NetSock_CfgTimeoutConnAcceptGet_ms()	Get socket's connection accept timeout value.
NetSock_CfgTimeoutConnAcceptSet()	Set socket's connection accept timeout value.
NetSock_CfgTimeoutConnCloseDflt()	Set socket's connection close timeout to configured-default value.
NetSock_CfgTimeoutConnCloseGet_ms()	Get socket's connection close timeout value.
NetSock_CfgTimeoutConnCloseSet()	Set socket's connection close timeout value.
NetSock_CfgTimeoutConnReqDflt()	Set socket's connection request timeout to configured-default value.
NetSock_CfgTimeoutConnReqGet_ms()	Get socket's connection request timeout value.
NetSock_CfgTimeoutConnReqSet()	Set socket's connection request timeout value.
NetSock_CfgTimeoutRxQ_Dflt()	Set socket's connection receive queue timeout to configured-default value.
NetSock_CfgTimeoutRxQ_Get_ms()	Get socket's receive queue timeout value.
NetSock_CfgTimeoutRxQ_Set()	Set socket's connection receive queue timeout value.
NetSock_CfgTimeoutTxQ_Dflt()	Set socket's connection transmit queue timeout to configured-default value.
NetSock_CfgTimeoutTxQ_Get_ms()	Get socket's transmit queue timeout value.
NetSock_CfgTimeoutTxQ_Set()	Set socket's connection transmit queue timeout value.
NetSock_CfgTxIP_TOS() (IPv4 only)	Configure socket's transmit IPv4 Type of Service (TOS).
NetSock_CfgTxIP_TTL_Multicast() (IPv4 only)	Configure socket's transmit IPv4 multicast Time to live (TTL).
NetSock_CfgTxIP_TTL() (IPv4 only)	Configure socket's transmit IPv4 Time to Live (TTL).
NetSock_CfgTxQ_Size()	Configure socket's transmit queue size.
NetSock_Close()	Terminate communication and free a socket.
NetSock_Conn()	Connect a local socket to a remote socket address.
NetSock_GetConnTransportID()	Gets a socket's transport layer connection handle ID (e.g., TCP connection ID) if available.
NetSock_GetLocalIPAddr()	Gets the local IP address used in the socket connection.
NetSock_IsConn()	Check if a socket is connected to a remote socket.
NetSock_Listen()	Set a socket to accept incoming connections.
NetSock_Open()	Create a datagram (i.e., UDP) or stream (i.e., TCP) type socket.
NetSock_OptGet()	Get the specified socket option from the sock_id socket.
NetSock_OptSet()	Set the specified socket option to the sock_id socket.

Function Name	Description
NetSock_PoolStatGet()	Get Network Sockets' statistics pool.
NetSock_PoolStatResetMaxUsed()	Reset Network Sockets' statistics pool's maximum number of entries used.
NetSock_RxData() / NetSock_RxDataFrom()	Copy up to a specified number of bytes received from a remote socket into an application memory buffer.
NetSock_Sel()	Check if any sockets are ready for available read or write operations or error conditions.
NetSock_SelAbort()	Abort any tasks that are pending on a socket using the select functionality.
NetSock_TxData() / NetSock_TxDataTo()	Copy bytes from an application memory buffer into a socket to send to a remote socket.

NET_SOCK_DESC_CLR()

Remove a socket file descriptor ID as a member of a file descriptor set. See also function NetSock_Sel().

Files

`net_sock.h`

Prototype

```
NET_SOCK_DESC_CLR(desc_nbr, p_desc_set);
```

Arguments

`desc_nbr`

This is the socket file descriptor ID returned by `NetSock_Open()/socket()` when the socket was created *or* by `NetSock_Accept()/accept()` when a connection was accepted.

`p_desc_set`

Pointer to a socket file descriptor set.

Returned Value

None.

Required Configuration

Available only if `NET_SOCK_CFG_SEL_EN` is enabled (see section Socket Layer Configuration.

Notes / Warnings

`NetSock_Sel()/select()` checks or waits for available operations or error conditions on any of the socket file descriptor members of a socket file descriptor set.

No errors are returned even if the socket file descriptor ID or the file descriptor set is invalid, or the socket file descriptor ID is *not* set in the file descriptor set.

NET_SOCK_DESC_COPY()

Copy a file descriptor set to another file descriptor set. See also function NetSock_Sel() .

Files

`net_sock.h`

Prototype

```
NET_SOCK_DESC_COPY(p_desc_set_dest, p_desc_set_src);
```

Arguments

`p_desc_set_dest`

Pointer to the destination socket file descriptor set.

`p_desc_set_src`

Pointer to the source socket file descriptor set to copy.

Returned Value

None.

Required Configuration

Available only if `NET_SOCK_CFG_SEL_EN` is enabled (see section Socket Layer Configuration.

Notes / Warnings

`NetSock_Sel()/select()` checks or waits for available operations or error conditions on any of the socket file descriptor members of a socket file descriptor set.

No errors are returned even if either file descriptor set is invalid.

NET_SOCK_DESC_INIT()

Initialize/zero-clear a file descriptor set. See also function NetSock_Sel() .

Files

`net_sock.h`

Prototype

```
NET_SOCK_DESC_INIT(p_desc_set);
```

Arguments

`p_desc_set`

Pointer to a socket file descriptor set.

Returned Value

None.

Required Configuration

Available only if `NET_SOCK_CFG_SEL_EN` is enabled (see section Socket Layer Configuration.

Notes / Warnings

`NetSock_Sel()/select()` checks or waits for available operations or error conditions on any of the socket file descriptor members of a socket file descriptor set.

No errors are returned even if the file descriptor set is invalid.

NET_SOCK_DESC_IS_SET()

Check if a socket file descriptor ID is a member of a file descriptor set. See also function NetSock_Sel() .

Files

`net_sock.h`

Prototype

```
NET_SOCK_DESC_IS_SET(desc_nbr, p_desc_set);
```

Arguments

`desc_nbr`

This is the socket file descriptor ID returned by `NetSock_Open()/socket()` when the socket was created *or* by `NetSock_Accept()/accept()` when a connection was accepted.

`p_desc_set`

Pointer to a socket file descriptor set.

Returned Value

- 1, if the socket file descriptor ID is a member of the file descriptor set;
- 0, otherwise.

Required Configuration

Available only if `NET_SOCK_CFG_SEL_EN` is enabled (see section Socket Layer Configuration.

Notes / Warnings

`NetSock_Sel()/select()` checks or waits for available operations or error conditions on any of the socket file descriptor members of a socket file descriptor set.

`0` is returned if the socket file descriptor ID or the file descriptor set is invalid.

NET_SOCK_DESC_SET()

Add a socket file descriptor ID as a member of a file descriptor set. See also function NetSock_Sel() .

Files

`net_sock.h`

Prototype

```
NET_SOCK_DESC_SET(desc_nbr, p_desc_set);
```

Arguments

`desc_nbr`

This is the socket file descriptor ID returned by `NetSock_Open()/socket()` when the socket was created *or* by `NetSock_Accept()/accept()` when a connection was accepted.

`p_desc_set`

Pointer to a socket file descriptor set.

Returned Value

None.

Required Configuration

Available only if NET_SOCK_CFG_SEL_EN is enabled (see section Socket Layer Configuration.

Notes / Warnings

`NetSock_Sel()/select()` checks or waits for available operations or error conditions on any of the socket file descriptor members of a socket file descriptor set.

No errors are returned even if the socket file descriptor ID or the file descriptor set is invalid, or the socket file descriptor ID is *not* cleared in the file descriptor set.

NetSock_Accept()

Description

Returns a new socket accepted from a listen socket.

Files

`net_sock.h/net_sock.c`

Prototype

```
NET_SOCK_ID  NetSock_Accept (NET_SOCK_ID        sock_id,
                             NET_SOCK_ADDR      *p_addr_remote,
                             NET_SOCK_ADDR_LEN  *p_addr_len,
                             RTOS_ERR           *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of listen socket.

`p_addr_remote`

Pointer to an address buffer that will receive the socket address structure.

`p_addr_len`

Pointer to a variable to pass the size of the socket address structure and that will received the size of the accepted connection's socket address structure, if no errors, else a 0 size will be returned.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_TYPE
- RTOS_ERR_NOT_FOUND
- RTOS_ERR_NET_INVALID_CONN
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_INVALID_STATE
- RTOS_ERR_NET_CONN_CLOSED_FAULT
- RTOS_ERR_ABORT
- RTOS_ERR_TIMEOUT

Returned Value

- Socket descriptor/handle identifier of new accepted socket, if NO error(s).
- NET_SOCK_BSD_ERR_ACCEPT , otherwise.

Notes / Warnings

1. Since 'p_addr`_len' argument is both an input and output argument:
   (a) Its input value SHOULD be validated prior to use;
   1. In the case that the ' p_addr_len ' argument is passed a null pointer, NO input value is validated or used.

   (b) While its output value MUST be initially configured to return a default value PRIOR to all other validation or function handling in case of any error(s).
2. Socket accept operation valid for stream-type sockets only.

NetSock_Bind()

Description

Binds a network socket to a local address.

Files

net_sock.h/net_sock.c

Prototype

```
NET_SOCK_RTN_CODE  NetSock_Bind (NET_SOCK_ID        sock_id,
                                 NET_SOCK_ADDR      *p_addr_local,
                                 NET_SOCK_ADDR_LEN   addr_len,
                                 RTOS_ERR           *p_err)
```

Arguments

sock_id

Socket descriptor/handle identifier of socket to bind to a local address.

p_addr_local

Pointer to socket address structure.

addr_len

Length of socket address structure (in octets).

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_TYPE`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_ALREADY_EXISTS`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_NET_INVALID_CONN`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_NET_CONN_CLOSED_FAULT`

Returned Value

- `NET_SOCK_BSD_ERR_NONE` , if NO error(s).
- `NET_SOCK_BSD_ERR_BIND` , otherwise.

Notes / Warnings

1. Socket address structure 'AddrFamily' member MUST be configured in host-order and MUST NOT be converted to/from network-order.
2. Socket address structure addresses MUST be configured/converted from host-order to network-order.

NetSock_BlockGet()

Description

Gets the blocking mode configuration of the specified socket.

Files

`net_sock.h/net_sock.c`

Prototype

```
CPU_INT08U  NetSock_BlockGet (NET_SOCK_ID   sock_id,
                              RTOS_ERR     *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket from which to get option.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- `NET_SOCK_BLOCK_SEL_NO_BLOCK` : Socket is in non-blocking mode.

- `NET_SOCK_BLOCK_SEL_BLOCK` : Socket is in blocking mode.
- `NET_SOCK_BLOCK_SEL_NONE` : Error in operation.

Notes / Warnings

None.

NetSock_CfgBlock()

Description

Configures the socket's blocking mode.

Files

`net_sock.h/net_sock.c`

Prototype

```
CPU_BOOLEAN  NetSock_CfgBlock (NET_SOCK_ID   sock_id,
                               CPU_INT08U    block,
                               RTOS_ERR     *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to configure blocking mode.

`block`

Desired value for socket blocking mode :

- `NET_SOCK_BLOCK_SEL_DFLT`
- `NET_SOCK_BLOCK_SEL_BLOCK`
- `NET_SOCK_BLOCK_SEL_NO_BLOCK`

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- `DEF_OK` , socket blocking mode successfully configured.
- `DEF_FAIL` , otherwise.

Notes / Warnings

None.

NetSock_CfgConnChildQ_SizeGet()

Description

Gets the socket's connection child queue size value.

Files

`net_sock.h/net_sock.c`

Prototype

```
NET_SOCK_Q_SIZE  NetSock_CfgConnChildQ_SizeGet (NET_SOCK_ID   sock_id,
                                                RTOS_ERR     *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to configure receive queue size.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

Socket's connection child queue size value :

- `NET_SOCK_Q_SIZE_NONE` , on any error(s).
- `NET_SOCK_Q_SIZE_UNLIMITED` , if unlimited (i.e., NO limit) value is configured.
- Child connection queue size, otherwise.

Notes / Warnings

Despite inconsistency with other 'Get' status functions, `NetSock_CfgConnChildQ_SizeGet()` includes 'Cfg' for consistency with other `NetSock_CfgConn&&&()` functions.

NetSock_CfgConnChildQ_SizeSet()

Description

Configures the socket's child connection queue size.

Files

`net_sock.h/net_sock.c`

Prototype

```
CPU_BOOLEAN  NetSock_CfgConnChildQ_SizeSet (NET_SOCK_ID       sock_id,
                                            NET_SOCK_Q_SIZE   queue_size,
                                            RTOS_ERR         *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to configure receive queue size.

queue_size

Desired child connection queue size :

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_HANDLE

Returned Value

- DEF_OK , socket child connection queue size successfully configured.
- DEF_FAIL , otherwise.

Notes / Warnings

1. NetSock_CfgConnChildQ_SizeSet() allows a listen socket to reject new incoming connection when the number of connection already accepted (currently being processed) plus the listen queue reaches the maximum number of child connection.
   (a) It should be used when resources such as number of received buffers are limited.
   (b) It doesn't remove any connection currently accepted. It becomes effective for later connection request.

NetSock_CfgIF()

Description

Configures the interface that must be used by the socket.

Files

net_sock.h/net_sock.c

Prototype

```
CPU_BOOLEAN  NetSock_CfgIF (NET_SOCK_ID   sock_id,
                            NET_IF_NBR    if_nbr,
                            RTOS_ERR     *p_err)
```

Arguments

sock_id

Socket descriptor/handle identifier of the socket to configure the interface number.

if_nbr

Interface number to bind to the socket.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_TYPE
- RTOS_ERR_INVALID_HANDLE

Returned Value

- `DEF_OK` , If the interface number was successfully set.
- `DEF_FAIL` , Otherwise.

Notes / Warnings

None.

NetSock_CfgRxQ_Size()

Description

Configures the socket's receive queue size.

Files

`net_sock.h/net_sock.c`

Prototype

```
CPU_BOOLEAN  NetSock_CfgRxQ_Size (NET_SOCK_ID         sock_id,
                                  NET_SOCK_DATA_SIZE   size,
                                  RTOS_ERR            *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to configure receive queue size.

`size`

Desired receive queue size.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_TYPE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_NET_INVALID_CONN`
- `RTOS_ERR_INVALID_STATE`

Returned Value

- `DEF_OK` , socket receive queue size successfully configured.
- `DEF_FAIL` , otherwise.

Notes / Warnings

1. For datagram sockets, configured size does NOT :
   (a) Limit or remove any received data currently queued but becomes effective for later received data.
   (b) Partially truncate any received data but instead allows data from exactly one received packet buffer to overflow the configured size since each datagram MUST be received atomically.
2. For stream sockets, size MAY be required to be configured prior to connecting.

NetSock_CfgSecure()

Description

Configures the socket's secure mode.

Files

`net_sock.h/net_sock.c`

Prototype

```
CPU_BOOLEAN  NetSock_CfgSecure (NET_SOCK_ID   sock_id,
                                CPU_BOOLEAN   secure,
                                RTOS_ERR     *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to configure secure mode.

`secure`

Desired value for socket secure mode :

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_TYPE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_INVALID_STATE`

Returned Value

- `DEF_OK` , socket `secure` mode successfully configured.
- `DEF_FAIL` , otherwise.

Notes / Warnings

A socket's `secure` session `('p_sock-\>SecureSession')` will be initialized if a `secure` port session buffer/object is available.

NetSock_CfgSecureClientCertKeyInstall()

Description

Installs the certificate and key that must be used by a client for mutual authentication.

Files

`net_sock.h/net_sock.c`

Prototype

```
CPU_BOOLEAN  NetSock_CfgSecureClientCertKeyInstall (      NET_SOCK_ID                 sock_id,
                                    const  void                  *p_cert,
                                         CPU_INT32U                 cert_len,
                                    const  void                  *p_key,
                                         CPU_INT32U                 key_len,
                                         NET_SOCK_SECURE_CERT_KEY_FMT   fmt,
                                         CPU_BOOLEAN                cert_chain,
                                         RTOS_ERR                  *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of server socket to configure secure certificate and key.

`p_cert`

Pointer to buffer that contains the certificate.

`cert_len`

Certificate length.

`p_key`

Pointer to buffer that contains the key.

`key_len`

Key length.

`fmt`

Certificate and key format:

- `NET_SOCK_SECURE_CERT_KEY_FMT_PEM`
- `NET_SOCK_SECURE_CERT_KEY_FMT_DER`

`cert_chain`

Certificate point to a chain of certificate.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- `DEF_OK` , certificate and key successfully installed.
- `DEF_FAIL` , otherwise.

Notes / Warnings

MUST BE CALLED ONLY AFTER `NetSock_CfgSecure()` has been called.

NetSock_CfgSecureClientCommonName()

Description

Configures the client socket's common name.

Files

`net_sock.h/net_sock.c`

Prototype

```
CPU_BOOLEAN  NetSock_CfgSecureClientCommonName (NET_SOCK_ID   sock_id,
                                                CPU_CHAR     *p_common_name,
                                                RTOS_ERR     *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of client socket to configure the common name.

`p_common_name`

Pointer to string that contain the common name.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- `DEF_OK`, common name successfully installed.
- `DEF_FAIL`, otherwise.

Notes / Warnings

MUST BE CALLED ONLY AFTER `NetSock_CfgSecure()` has been called.

NetSock_CfgSecureClientTrustCallBack()

Description

Configure client socket's trust call back function.

Files

`net_sock.h/net_sock.c`

Prototype

```
CPU_BOOLEAN  NetSock_CfgSecureClientTrustCallBack (NET_SOCK_ID                 sock_id,
                                                   NET_SOCK_SECURE_TRUST_FNCT   call_back_fnct,
                                                   RTOS_ERR                    *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of client socket to configure trust call back function.

`call_back_fnct`

Pointer to the trust call back function

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- `DEF_OK` , trust call back function successfully configured.
- `DEF_FAIL` , otherwise.

Notes / Warnings

MUST BE CALLED ONLY AFTER `NetSock_CfgSecure()` has been called.

NetSock_CfgSecureServerCertKeyInstall()

Description

Installs the certificate and key that must be used by a server socket.

Files

`net_sock.h/net_sock.c`

Prototype

```
CPU_BOOLEAN  NetSock_CfgSecureServerCertKeyInstall (      NET_SOCK_ID                  sock_id,
                                            const  void                    *p_cert,
                                                   CPU_INT32U                cert_len,
                                            const  void                    *p_key,
                                                   CPU_INT32U                key_len,
                                                   NET_SOCK_SECURE_CERT_KEY_FMT   fmt,
                                                   CPU_BOOLEAN                cert_chain,
                                                   RTOS_ERR                  *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of server socket to configure secure certificate and key.

`p_cert`

Pointer to buffer that contains the certificate.

`cert_len`

Certificate length.

`p_key`

Pointer to buffer that contains the key.

`key_len`

Key length.

`fmt`

Certificate and key format:

- `NET_SOCK_SECURE_CERT_KEY_FMT_PEM`
- `NET_SOCK_SECURE_CERT_KEY_FMT_DER`

`cert_chain`

Certificate point to a chain of certificate.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- `DEF_OK` , certificate and key successfully installed.
- `DEF_FAIL` , otherwise.

Notes / Warnings

MUST BE CALLED ONLY AFTER `NetSock_CfgSecure()` has been called.

NetSock_CfgTimeoutConnAcceptDflt()

Description

Sets the socket's connection accept configured-default timeout value.

Files

`net_sock.h/net_sock.c`

Prototype

```
CPU_BOOLEAN  NetSock_CfgTimeoutConnAcceptDflt (NET_SOCK_ID   sock_id,
                                               RTOS_ERR     *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to set connection accept timeout.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- `DEF_OK` , socket connection accept configured-default timeout successfully set.
- `DEF_FAIL` , otherwise.

Notes / Warnings

Configured timeout does NOT reschedule any current socket connection accept timeout in progress, but becomes effective the next time a socket pends on a connection accept with timeout.

NetSock_CfgTimeoutConnAcceptGet_ms()

Description

Gets the socket's connection accept timeout value.

Files

`net_sock.h/net_sock.c`

Prototype

```
CPU_INT32U  NetSock_CfgTimeoutConnAcceptGet_ms (NET_SOCK_ID   sock_id,
                                                RTOS_ERR     *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to get connection accept timeout.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

Socket's connection accept network timeout value :

- 0, on any error(s).
- `NET_TMR_TIME_INFINITE` , if an infinite (i.e., NO timeout) value is configured.
- In number of milliseconds, otherwise.

Notes / Warnings

Despite inconsistency with other 'Get' status functions, `NetSock_CfgTimeoutConnAcceptGet_ms()` includes 'Cfg' for consistency with other `NetSock_CfgTimeout&&&()` functions.

NetSock_CfgTimeoutConnAcceptSet()

Description

Sets the socket's connection accept timeout value.

Files

`net_sock.h/net_sock.c`

Prototype

```
CPU_BOOLEAN  NetSock_CfgTimeoutConnAcceptSet (NET_SOCK_ID   sock_id,
                                              CPU_INT32U    timeout_ms,
                                              RTOS_ERR     *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to set connection accept timeout.

`timeout_ms`

Desired timeout value :

- `NET_TMR_TIME_INFINITE` , if an infinite (i.e., NO timeout) value is desired.
- In number of milliseconds, otherwise.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- `DEF_OK` , socket connection accept timeout successfully set.
- `DEF_FAIL` , otherwise.

Notes / Warnings

Configured timeout does NOT reschedule any current socket connection accept timeout in progress, but becomes effective the next time a socket pends on a connection accept with timeout.

NetSock_CfgTimeoutConnCloseDflt()

Description

Sets the socket's connection close configured-default timeout value.

Files

`net_sock.h/net_sock.c`

Prototype

```
CPU_BOOLEAN  NetSock_CfgTimeoutConnCloseDflt (NET_SOCK_ID   sock_id,
                                              RTOS_ERR     *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to set connection close timeout.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_HANDLE

Returned Value

- DEF_OK , socket connection close configured-default timeout successfully set.
- DEF_FAIL , otherwise.

Notes / Warnings

Configured timeout does NOT reschedule any current socket connection close timeout in progress, but becomes effective the next time a socket pends on a connection close with timeout.

NetSock_CfgTimeoutConnCloseGet_ms()

Description

Gets the socket's connection close timeout value.

Files

net_sock.h/net_sock.c

Prototype

```
CPU_INT32U  NetSock_CfgTimeoutConnCloseGet_ms (NET_SOCK_ID   sock_id,
                                              RTOS_ERR     *p_err)
```

Arguments

sock_id

Socket descriptor/handle identifier of socket to get connection close timeout.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_HANDLE

Returned Value

Socket's connection close network timeout value :

- 0, on any error(s).
- NET_TMR_TIME_INFINITE , if an infinite (i.e., NO timeout) value is configured.
- In number of milliseconds, otherwise.

Notes / Warnings

Despite inconsistency with other 'Get' status functions, NetSock_CfgTimeoutConnCloseGet_ms() includes 'Cfg' for consistency with other NetSock_CfgTimeout&&&() functions.

NetSock_CfgTimeoutConnCloseSet()

Description

Sets the socket's connection close timeout value.

Files

`net_sock.h/net_sock.c`

Prototype

```
CPU_BOOLEAN  NetSock_CfgTimeoutConnCloseSet (NET_SOCK_ID   sock_id,
                                             CPU_INT32U    timeout_ms,
                                             RTOS_ERR     *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to set connection close timeout.

`timeout_ms`

Desired timeout value :

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- `DEF_OK` , socket connection close timeout successfully set.
- `DEF_FAIL` , otherwise.

Notes / Warnings

Configured timeout does NOT reschedule any current socket connection close timeout in progress, but becomes effective the next time a socket pends on a connection close with timeout.

NetSock_CfgTimeoutConnReqDflt()

Description

Sets the socket's connection request configured-default timeout value.

Files

`net_sock.h/net_sock.c`

Prototype

```
CPU_BOOLEAN  NetSock_CfgTimeoutConnReqDflt (NET_SOCK_ID   sock_id,
                                            RTOS_ERR     *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to set connection request timeout.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- `DEF_OK` , socket connection request configured-default timeout successfully set.
- `DEF_FAIL` , otherwise.

Notes / Warnings

Configured timeout does NOT reschedule any current socket connection request timeout in progress, but becomes effective the next time a socket pends on a connection request with timeout.

NetSock_CfgTimeoutConnReqGet_ms()

Description

Gets the socket's connection request timeout value.

Files

`net_sock.h/net_sock.c`

Prototype

```
CPU_INT32U  NetSock_CfgTimeoutConnReqGet_ms (NET_SOCK_ID   sock_id,
                                             RTOS_ERR     *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to get connection request timeout.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

Socket's connection request network timeout value :

- 0, on any error(s).
- `NET_TMR_TIME_INFINITE` , if an infinite (i.e., NO timeout) value is configured.
- In number of milliseconds, otherwise.

Notes / Warnings

1. Despite inconsistency with other 'Get' status functions, `NetSock_CfgTimeoutConnReqGet_ms()` includes 'Cfg' for consistency with other `NetSock_CfgTimeout&&&()` functions.

NetSock_CfgTimeoutConnReqSet()

Description

Sets the socket's connection request timeout value.

Files

`net_sock.h/net_sock.c`

Prototype

```
CPU_BOOLEAN  NetSock_CfgTimeoutConnReqSet (NET_SOCK_ID   sock_id,
                                           CPU_INT32U    timeout_ms,
                                           RTOS_ERR     *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to set connection request timeout.

`timeout_ms`

Desired timeout value :

- `NET_TMR_TIME_INFINITE` , if an infinite (i.e., NO timeout) value is desired.
- In number of milliseconds, otherwise.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- `DEF_OK` , socket connection request timeout successfully set.
- `DEF_FAIL` , otherwise.

Notes / Warnings

Configured timeout does NOT reschedule any current socket connection request timeout in progress, but becomes effective the next time a socket pends on a connection request with timeout.

NetSock_CfgTimeoutRxQ_Dflt()

Description

Sets the socket's receive queue configured-default timeout value.

Files

`net_sock.h/net_sock.c`

Prototype

```
CPU_BOOLEAN  NetSock_CfgTimeoutRxQ_Dflt (NET_SOCK_ID   sock_id,
                                         RTOS_ERR     *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to set the configured receive queue.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_NET_INVALID_CONN`

Returned Value

- `DEF_OK` , socket receive queue configured-default timeout successfully set.
- `DEF_FAIL` , otherwise.

Notes / Warnings

Configured timeout does NOT reschedule any current socket receive queue timeout in progress, but becomes effective the next time a socket pends on its receive queue with timeout.

NetSock_CfgTimeoutRxQ_Get_ms()

Description

Gets the socket's receive queue timeout value.

Files

`net_sock.h/net_sock.c`

Prototype

```
CPU_INT32U  NetSock_CfgTimeoutRxQ_Get_ms (NET_SOCK_ID   sock_id,
                                          RTOS_ERR     *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to get receive queue timeout.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_NET_INVALID_CONN`

Returned Value

Socket's receive queue network timeout value :

- 0, on any error(s).
- `NET_TMR_TIME_INFINITE` , if an infinite (i.e., NO timeout) value is configured.
- In number of milliseconds, otherwise.

Notes / Warnings

1. Despite inconsistency with other 'Get' status functions, `NetSock_CfgTimeoutRxQ_Get_ms()` includes 'Cfg' for consistency with other `NetSock_CfgTimeout&&&()` functions.

NetSock_CfgTimeoutRxQ_Set()

Description

Sets the socket's receive queue timeout value.

Files

`net_sock.h/net_sock.c`

Prototype

```
CPU_BOOLEAN  NetSock_CfgTimeoutRxQ_Set (NET_SOCK_ID   sock_id,
                                        CPU_INT32U    timeout_ms,
                                        RTOS_ERR     *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to set receive queue timeout.

`timeout_ms`

Desired timeout value :

- `NET_TMR_TIME_INFINITE` , if an infinite (i.e., NO timeout) value is desired.
- In number of milliseconds, otherwise.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_NET_INVALID_CONN`

Returned Value

- `DEF_OK` , socket receive queue timeout successfully set.
- `DEF_FAIL` , otherwise.

Notes / Warnings

Configured timeout does NOT reschedule any current socket receive queue timeout in progress, but becomes effective the next time a socket pends on its receive queue with timeout.

NetSock_CfgTimeoutTxQ_Dflt()

Description

1. Sets the socket's transmit queue configured-default timeout value.

Files

`net_sock.h/net_sock.c`

Prototype

```
CPU_BOOLEAN  NetSock_CfgTimeoutTxQ_Dflt (NET_SOCK_ID   sock_id,
                                         RTOS_ERR     *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to set transmit queue configured-default timeout.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_TYPE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_NET_INVALID_CONN`

Returned Value

- `DEF_OK`, socket transmit queue configured-default timeout successfully set.
- `DEF_FAIL`, otherwise.

Notes / Warnings

Configured timeout does NOT reschedule any current socket transmit queue timeout in progress, but becomes effective the next time a socket pends on its transmit queue with timeout.

NetSock_CfgTimeoutTxQ_Get_ms()

Description

1. Gets the socket's transmit queue timeout value.

Files

`net_sock.h/net_sock.c`

Prototype

```
CPU_INT32U  NetSock_CfgTimeoutTxQ_Get_ms (NET_SOCK_ID   sock_id,
                                          RTOS_ERR     *p_err)
```

Arguments

sock_id

Socket descriptor/handle identifier of socket to get transmit queue timeout.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_TYPE
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_NET_INVALID_CONN

Returned Value

Socket's transmit queue network timeout value :

- 0, on any error(s).
- NET_TMR_TIME_INFINITE , if an infinite (i.e., NO timeout) value is configured.
- In number of milliseconds, otherwise.

Notes / Warnings

Despite inconsistency with other 'Get' status functions, NetSock_CfgTimeoutTxQ_Get_ms() includes 'Cfg' for consistency with other NetSock_CfgTimeout&&&() functions.

NetSock_CfgTimeoutTxQ_Set()

Description

1. Sets the socket's transmit queue timeout value.

Files

net_sock.h/net_sock.c

Prototype

```
CPU_BOOLEAN  NetSock_CfgTimeoutTxQ_Set (NET_SOCK_ID   sock_id,
                                        CPU_INT32U    timeout_ms,
                                        RTOS_ERR     *p_err)
```

Arguments

sock_id

Socket descriptor/handle identifier of socket to set transmit queue timeout.

timeout_ms

Desired timeout value :

- NET_TMR_TIME_INFINITE , if an infinite (i.e., NO timeout) value is desired.
- In number of milliseconds, otherwise.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE

- RTOS_ERR_INVALID_TYPE
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_NET_INVALID_CONN

Returned Value

- DEF_OK , socket transmit queue timeout successfully set.
- DEF_FAIL , otherwise.

Notes / Warnings

Configured timeout does NOT reschedule any current socket transmit queue timeout in progress, but becomes effective the next time a socket pends on its transmit queue with timeout.

NetSock_CfgTxIP_TOS() (IPv4 only)

Description

Configures the socket's transmit IP TOS.

Files

net_sock.h/net_sock.c

Prototype

```
#ifdef  NET_IPv4_MODULE_EN

CPU_BOOLEAN  NetSock_CfgTxIP_TOS (NET_SOCK_ID    sock_id,
                                  NET_IPv4_TOS   ip_tos,
                                  RTOS_ERR      *p_err)
```

Arguments

sock_id

Socket descriptor/handle identifier of socket to configure transmit IP TOS.

ip_tos

Desired transmit IP TOS.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_INVALID_STATE

Returned Value

- DEF_OK , socket transmit IP TOS successfully configured.
- DEF_FAIL , otherwise.

Notes / Warnings

None.

NetSock_CfgTxIP_TTL() (IPv4 only)

Description

Configures the socket's transmit IP TTL.

Files

`net_sock.h/net_sock.c`

Prototype

```
#ifdef  NET_IPv4_MODULE_EN

CPU_BOOLEAN  NetSock_CfgTxIP_TTL (NET_SOCK_ID    sock_id,
                                  NET_IPv4_TTL   ip_ttl,
                                  RTOS_ERR      *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to configure transmit IP TTL.

`ip_ttl`

Desired transmit IP TTL :

- `NET_IPv4_TTL_MIN` Minimum TTL transmit value (1)
- `NET_IPv4_TTL_MAX` Maximum TTL transmit value (255)
- `NET_IPv4_TTL_DFLT` Default TTL transmit value (128)
- `NET_IPv4_TTL_NONE` Replace with default TTL

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_INVALID_STATE`

Returned Value

- `DEF_OK` , socket transmit IP TTL successfully configured.
- `DEF_FAIL` , otherwise.

Notes / Warnings

None.

NetSock_CfgTxIP_TTL_Multicast() (IPv4 only)

Description

Configures the socket's transmit IP multicast TTL.

Files

`net_sock.h/net_sock.c`

Prototype

```
CPU_BOOLEAN NetSock_CfgTxIP_TTL_Multicast (NET_SOCK_ID    sock_id,
                                            NET_IPv4_TTL   ip_ttl,
                                            RTOS_ERR      *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to configure transmit IP multicast TTL.

`ip_ttl`

Desired transmit IP multicast TTL:

- `NET_IPv4_TTL_MIN` Minimum TTL transmit value (1)
- `NET_IPv4_TTL_MAX` Maximum TTL transmit value (255)
- `NET_IPv4_TTL_DFLT` Default TTL transmit value (1)
- `NET_IPv4_TTL_NONE` Replace with default TTL

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_INVALID_STATE`

Returned Value

- `DEF_OK` , socket transmit IP multicast TTL successfully configured.
- `DEF_FAIL` , otherwise.

Notes / Warnings

None.

NetSock_CfgTxQ_Size()

Description

Configures the socket's transmit queue size.

Files

`net_sock.h/net_sock.c`

Prototype

```
CPU_BOOLEAN  NetSock_CfgTxQ_Size (NET_SOCK_ID          sock_id,
                                  NET_SOCK_DATA_SIZE   size,
                                  RTOS_ERR            *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to configure transmit queue size.

`size`

Desired transmit queue size.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_NET_INVALID_CONN`
- `RTOS_ERR_INVALID_STATE`

Returned Value

- `DEF_OK` , socket transmit queue `size` successfully configured.
- `DEF_FAIL` , otherwise.

Notes / Warnings

1. For datagram sockets, configured size does NOT :
   (a) Partially truncate any received data. Instead, it allows data from exactly one received packet buffer to overflow the configured `size` since each datagram MUST be received atomically.
2. For stream sockets, `size` MAY be required to be configured prior to connecting.

NetSock_Close()

Description

Closes a network socket.

Files

`net_sock.h/net_sock.c`

Prototype

```
NET_SOCK_RTN_CODE  NetSock_Close (NET_SOCK_ID   sock_id,
                                  RTOS_ERR     *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to close.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_TYPE`
- `RTOS_ERR_NET_RETRY_MAX`
- `RTOS_ERR_NET_SOCK_CLOSED`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`

- RTOS_ERR_NOT_SUPPORTED
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_NET_INVALID_ADDR_SRC
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_NET_IF_LINK_DOWN
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_INVALID_STATE
- RTOS_ERR_ABORT
- RTOS_ERR_TIMEOUT
- RTOS_ERR_NET_OP_IN_PROGRESS
- RTOS_ERR_TX
- RTOS_ERR_NOT_FOUND
- RTOS_ERR_NET_INVALID_CONN
- RTOS_ERR_RX
- RTOS_ERR_NOT_READY
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_NET_NEXT_HOP

Returned Value

- NET_SOCK_BSD_ERR_NONE , if NO error(s).
- NET_SOCK_BSD_ERR_CLOSE , otherwise.

Notes / Warnings

1. Once an application closes its socket, NO further operations on the socket are allowed and the application MUST NOT continue to access the socket.
2. NO BSD socket error is returned for any internal error while closing the socket.

NetSock_Conn()

Description

Connects a network socket to a remote host.

Files

net_sock.h/net_sock.c

Prototype

```
NET_SOCK_RTN_CODE  NetSock_Conn (NET_SOCK_ID        sock_id,
                                 NET_SOCK_ADDR      *p_addr_remote,
                                 NET_SOCK_ADDR_LEN  addr_len,
                                 RTOS_ERR           *p_err)
```

Arguments

sock_id

Socket descriptor/handle identifier of socket to connect.

p_addr_remote

Pointer to socket address structure.

`addr_len`

Length of socket address structure (in octets).

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_TYPE`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_NOT_SUPPORTED`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_FAIL`
- `RTOS_ERR_NET_INVALID_ADDR_SRC`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_NET_IF_LINK_DOWN`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_NET_OP_IN_PROGRESS`
- `RTOS_ERR_TX`
- `RTOS_ERR_ALREADY_EXISTS`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_NET_INVALID_CONN`
- `RTOS_ERR_RX`
- `RTOS_ERR_NOT_READY`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_NET_NEXT_HOP`
- `RTOS_ERR_NET_CONN_CLOSED_FAULT`

Returned Value

- `NET_SOCK_BSD_ERR_NONE` , if NO error(s).
- `NET_SOCK_BSD_ERR_CONN` , otherwise.

Notes / Warnings

1. Socket address structure 'AddrFamily' member MUST be configured in host-order and MUST NOT be converted to/from network-order.
2. Socket address structure addresses MUST be configured/converted from host-order to network-order.

NetSock_GetConnTransportID()

Description

Get a socket's transport layer handle identifier.

Files

`net_sock.h/net_sock.c`

Prototype

```
NET_CONN_ID  NetSock_GetConnTransportID (NET_SOCK_ID   sock_id,
                                         RTOS_ERR     *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to get transport layer handle identifier.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_NET_INVALID_CONN`

Returned Value

- Socket's transport layer handle identifier, if NO error(s).
- `NET_CONN_ID_NONE`, otherwise.

Notes / Warnings

None.

NetSock_GetLocalIPAddr()

Description

Gets the local IP address used in the socket connection.

Files

`net_sock.h/net_sock.c`

Prototype

```
void  NetSock_GetLocalIPAddr (NET_SOCK_ID       sock_id,
                              CPU_INT08U       *p_buf_addr,
                              NET_SOCK_FAMILY  *p_family,
                              RTOS_ERR         *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier.

`p_buf_addr`

Pointer to a buffer to return the local IP address.

`p_family`

Pointer to the variable that will receive the connection family type of the local IP address.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_TYPE
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_NET_INVALID_CONN

Returned Value

None.

Notes / Warnings

' p_buf_addr ' must be a buffer at least of NET_CONN_ADDR_LEN_MAX bytes large.

NetSock_IsConn()

Description

Validates the socket currently in use and connected.

Files

net_sock.h/net_sock.c

Prototype

```
CPU_BOOLEAN  NetSock_IsConn (NET_SOCK_ID   sock_id,
                             RTOS_ERR     *p_err)
```

Arguments

sock_id

Socket descriptor/handle identifier of socket to validate.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_HANDLE

Returned Value

- DEF_YES , socket valid and connected.
- DEF_NO , socket invalid or NOT connected.

Notes / Warnings

None.

NetSock_Listen()

Description

Sets socket to listen for connection requests.

Files

`net_sock.h/net_sock.c`

Prototype

```
NET_SOCK_RTN_CODE  NetSock_Listen (NET_SOCK_ID      sock_id,
                                   NET_SOCK_Q_SIZE   sock_q_size,
                                   RTOS_ERR         *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to listen.

`sock_q_size`

Maximum number of connection requests to accept and queue on listen socket.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_TYPE`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_NET_INVALID_CONN`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_NET_CONN_CLOSED_FAULT`

Returned Value

- `NET_SOCK_BSD_ERR_NONE` , if NO error(s).
- `NET_SOCK_BSD_ERR_LISTEN` , otherwise.

Notes / Warnings

Socket listen operation valid for stream-type sockets only.

NetSock_Open()

Description

Opens a network socket.

Files

`net_sock.h/net_sock.c`

Prototype

```
NET_SOCK_ID  NetSock_Open (NET_SOCK_PROTOCOL_FAMILY   protocol_family,
                           NET_SOCK_TYPE              sock_type,
                           NET_SOCK_PROTOCOL          protocol,
                           RTOS_ERR                  *p_err)
```

Arguments

`protocol_family`

Socket `protocol` family :

- `NET_SOCK_PROTOCOL_FAMILY_IP_V4`
- `NET_SOCK_PROTOCOL_FAMILY_IP_V6`

`sock_type`

Socket type :

- `NET_SOCK_TYPE_DATAGRAM`
- `NET_SOCK_TYPE_STREAM`

`protocol`

Socket `protocol` :

- `NET_SOCK_PROTOCOL_DFLT`
- `NET_SOCK_PROTOCOL_TCP`
- `NET_SOCK_PROTOCOL_UDP`

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_POOL_EMPTY`

Returned Value

- Socket descriptor/handle identifier, if NO error(s).
- `NET_SOCK_BSD_ERR_OPEN` , otherwise.

Notes / Warnings

None.

NetSock_OptGet()

Description

Gets the specified socket option from the sock_id socket.

Files

`net_sock.h/net_sock.c`

Prototype

```
NET_SOCK_RTN_CODE  NetSock_OptGet (NET_SOCK_ID        sock_id,
                                  NET_SOCK_PROTOCOL   level,
                                  NET_SOCK_OPT_NAME   opt_name,
                                  void               *p_opt_val,
                                  NET_SOCK_OPT_LEN   *p_opt_len,
                                  RTOS_ERR           *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket from which to get the option.

`level`

Protocol level from which to retrieve the socket option.

`opt_name`

Socket option to get the value.

`p_opt_val`

Pointer to a socket option value buffer.

`p_opt_len`

Pointer to a socket option value buffer length.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_NET_INVALID_CONN`

Returned Value

- `NET_SOCK_BSD_ERR_NONE` , if NO error(s).
- `NET_SOCK_BSD_ERR_OPT_GET` , otherwise.

Notes / Warnings

1. The supported options are:
   (a) Level `NET_SOCK_PROTOCOL_SOCK` :
   - NET_SOCK_OPT_SOCK_TYPE
   - NET_SOCK_OPT_SOCK_KEEP_ALIVE
   - NET_SOCK_OPT_SOCK_ACCEPT_CONN
   - NET_SOCK_OPT_SOCK_TX_BUF_SIZE
   - NET_SOCK_OPT_SOCK_RX_BUF_SIZE
   - NET_SOCK_OPT_SOCK_TX_TIMEOUT
   - NET_SOCK_OPT_SOCK_RX_TIMEOUT

   (b) Level `NET_SOCK_PROTOCOL_IP` :
   - NET_SOCK_OPT_IP_TOS
   - NET_SOCK_OPT_IP_TTL
   - NET_SOCK_OPT_IP_RX_IF

   (c) Level `NET_SOCK_PROTOCOL_TCP` :
   - NET_SOCK_OPT_TCP_NO_DELAY
   - NET_SOCK_OPT_TCP_KEEP_CNT
   - NET_SOCK_OPT_TCP_KEEP_IDLE

- NET_SOCK_OPT_TCP_KEEP_INTVL

NetSock_OptSet()

Description

Sets the specified socket option of the socket to a specified value.

Files

`net_sock.h/net_sock.c`

Prototype

```
NET_SOCK_RTN_CODE  NetSock_OptSet (NET_SOCK_ID        sock_id,
                                   NET_SOCK_PROTOCOL  level,
                                   NET_SOCK_OPT_NAME  opt_name,
                                   void              *p_opt_val,
                                   NET_SOCK_OPT_LEN   opt_len,
                                   RTOS_ERR          *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket from which to get the option.

`level`

Protocol level at which the option resides.

`opt_name`

Name of the single option to set.

`p_opt_val`

Pointer to the value to set to the socket option.

`opt_len`

Option length.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_TYPE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_NET_INVALID_CONN`
- `RTOS_ERR_INVALID_STATE`

Returned Value

- `NET_SOCK_BSD_ERR_NONE` , if NO error(s).
- `NET_SOCK_BSD_ERR_OPT_SET` , otherwise.

Notes / Warnings

1. The size of the `p_opt_val` and the value of `opt_len` must be equal to the size of the socket option requested to be set.
2. The supported options are:
   (a) Level `NET_SOCK_PROTOCOL_SOCK` :
   - NET_SOCK_OPT_SOCK_KEEP_ALIVE
   - NET_SOCK_OPT_SOCK_TX_BUF_SIZE
   - NET_SOCK_OPT_SOCK_RX_BUF_SIZE
   - NET_SOCK_OPT_SOCK_TX_TIMEOUT
   - NET_SOCK_OPT_SOCK_RX_TIMEOUT

   (b) Level `NET_SOCK_PROTOCOL_IP` :
   - NET_SOCK_OPT_IP_TOS
   - NET_SOCK_OPT_IP_TTL
   - NET_SOCK_OPT_IP_ADD_MEMBERSHIP
   - NET_SOCK_OPT_IP_DROP_MEMBERSHIP

   (c) Level `NET_SOCK_PROTOCOL_TCP` :
   - NET_SOCK_OPT_TCP_NO_DELAY
   - NET_SOCK_OPT_TCP_KEEP_CNT
   - NET_SOCK_OPT_TCP_KEEP_IDLE
   - NET_SOCK_OPT_TCP_KEEP_INTVL

NetSock_PoolStatGet()

Description

Gets the socket statistics pool.

Files

`net_sock.h/net_sock.c`

Prototype

```
NET_STAT_POOL  NetSock_PoolStatGet (void)
```

Arguments

Returned Value

- Socket statistics pool, if NO error(s).
- NULL statistics pool, otherwise.

Notes / Warnings

None.

NetSock_PoolStatResetMaxUsed()

Description

Resets the socket statistics pool's maximum number of entries used.

Files

`net_sock.h/net_sock.c`

Prototype

```
void NetSock_PoolStatResetMaxUsed (void)
```

Arguments

Returned Value

None.

Notes / Warnings

None.

NetSock_RxData()

Description

Receive data from a network socket.

Files

`net_sock.h/net_sock.c`

Prototype

```
NET_SOCK_RTN_CODE  NetSock_RxData (NET_SOCK_ID         sock_id,
                                   void               *p_data_buf,
                                   CPU_INT16U          data_buf_len,
                                   NET_SOCK_API_FLAGS  flags,
                                   RTOS_ERR           *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to receive data.

`p_data_buf`

Pointer to an application data buffer that will receive the socket's received data.

`data_buf_len`

Size of the application data buffer (in octets).

`flags`

Flags to select receive options; bit-field flags logically OR'd :

- `NET_SOCK_FLAG_NONE` No socket flags selected.
- `NET_SOCK_FLAG_RX_DATA_PEEK` Receive socket data without consuming the socket data; i.e., socket data NOT removed from application receive queue(s).
- `NET_SOCK_FLAG_RX_NO_BLOCK` Receive socket data without blocking.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_FOUND`

- RTOS_ERR_NET_INVALID_CONN
- RTOS_ERR_NET_CONN_CLOSE_RX
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_INVALID_STATE
- RTOS_ERR_NET_CONN_CLOSED_FAULT
- RTOS_ERR_ABORT
- RTOS_ERR_TIMEOUT
- RTOS_ERR_WOULD_OVF

Returned Value

- Number of positive data octets received, if NO error(s).
- `NET_SOCK_BSD_RTN_CODE_CONN_CLOSED` , if socket connection closed.
- `NET_SOCK_BSD_ERR_RX` , otherwise.

Notes / Warnings

1. (a)
    1. Datagram-type sockets transmit and receive all data atomically -- i.e., every single, complete datagram transmitted MUST be received as a single, complete datagram.
    2. Thus if the socket's type is datagram and the receive data buffer size is NOT large enough for the received data, the receive data buffer is maximally filled with receive data but the remaining data octets are discarded & `RTOS_ERR_WOULD_OVF` error is returned.

   (b)
    1. (A) Stream-type sockets transmit and receive all data octets in one or more non-distinct packets. In other words, the application data is NOT bounded by any specific packet(s); rather, it is contiguous and sequenced from one packet to the next.
       (B) Thus if the socket's type is stream and the receive data buffer size is NOT large enough for the received data, the receive data buffer is maximally filled with receive data and the remaining data octets remain queued for later application-socket receives.
    2. Thus it is typical, but NOT absolutely required, that a single application task ONLY receive or request to receive data from a stream-type socket.
2. Only some socket receive flag options are implemented. If other flag options are requested, `NetSock_RxData()` handler function(s) abort and return appropriate error codes so that requested flag options are NOT silently ignored.

NetSock_RxDataFrom()

Description

Receive data from a network socket.

This must be used with Datagram socket to know which remote host sent the data when there is no connection established for the Datagram type socket. This can be used for Stream type Socket, but it is not mandatory.

Files

`net_sock.h/net_sock.c`

Prototype

```
NET_SOCK_RTN_CODE NetSock_RxDataFrom (NET_SOCK_ID        sock_id,
                                      void              *p_data_buf,
                                      CPU_INT16U         data_buf_len,
                                      NET_SOCK_API_FLAGS  flags,
                                      NET_SOCK_ADDR      *p_addr_remote,
                                      NET_SOCK_ADDR_LEN  *p_addr_len,
                                      void              *p_ip_opts_buf,
                                      CPU_INT08U         ip_opts_buf_len,
                                      CPU_INT08U        *p_ip_opts_len,
                                      RTOS_ERR          *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to receive data.

`p_data_buf`

Pointer to an application data buffer that will receive the socket's received data.

`data_buf_len`

Size of the application data buffer (in octets).

`flags`

Flags to select receive options; bit-field flags logically OR'd :

- `NET_SOCK_FLAG_NONE` No socket flags selected.
- `NET_SOCK_FLAG_RX_DATA_PEEK` Receive socket data without consuming the socket data; i.e., socket data NOT removed from application receive queue(s).
- `NET_SOCK_FLAG_RX_NO_BLOCK` Receive socket data without blocking.

`p_addr_remote`

Pointer to an address buffer that will receive the socket address structure.

`p_addr_len`

Pointer to a variable to pass the size of the socket address structure and that will received the size of the accepted connection’s socket address structure, if no errors. Otherwise, a 0 size is returned.

`p_ip_opts_buf`

Pointer to buffer to receive possible IP options, if NO error(s).

`ip_opts_buf_len`

Size of IP options receive buffer (in octets).

`p_ip_opts_len`

Pointer to variable that will receive the return size of any received IP options, if NO error(s).

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_NET_INVALID_CONN`
- `RTOS_ERR_NET_CONN_CLOSE_RX`

- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_INVALID_STATE
- RTOS_ERR_NET_CONN_CLOSED_FAULT
- RTOS_ERR_ABORT
- RTOS_ERR_TIMEOUT
- RTOS_ERR_WOULD_OVF

Returned Value

- Number of positive data octets received, if NO error(s).
- `NET_SOCK_BSD_RTN_CODE_CONN_CLOSED` , if socket connection closed.
- `NET_SOCK_BSD_ERR_RX` , otherwise.

Notes / Warnings

1. (a)
   1. Datagram-type sockets transmit and receive all data atomically -- i.e., every single, complete datagram transmitted MUST be received as a single, complete datagram.
   2. Thus if the socket's type is datagram and the receive data buffer size is NOT large enough for the received data, the receive data buffer is maximally filled with receive data but the remaining data octets are discarded and RTOS_ERR_WOULD_OVF error is returned.

   (b)
   1. A) Stream-type sockets transmit and receive all data octets in one or more non-distinct packets. In other words, the application data is NOT bounded by any specific packet(s); rather, it is contiguous and sequenced from one packet to the next.
      (B) Thus if the socket's type is stream and the receive data buffer size is NOT large enough for the received data, the receive data buffer is maximally filled with receive data and the remaining data octets remain queued for later application-socket receives.
   2. Thus it is typical, but NOT absolutely required, that a single application task ONLY receive or request to receive data from a stream-type socket.
2. Only some socket receive flag options are implemented. If other flag options are requested, NetSock_RxData() handler function(s) abort and return appropriate error codes so that requested flag options are NOT silently ignored.
3. (a) If :
   1. NO IP options were received OR
   2. NO IP options receive buffer is provided OR
   3. IP options receive buffer NOT large enough for the received IP options then NO IP options are returned and any received IP options are silently discarded.

   (b) The IP options receive buffer size SHOULD be large enough to receive the maximum IP options size, `NET_IPv4_HDR_OPT_SIZE_MAX` .

   (c) Received IP options should be provided/decoded via appropriate IP layer API.
4. IP options arguments may NOT be necessary (remove if unnecessary).

NetSock_Sel()

Description

Checks multiple sockets for available resources &/or operations.

Files

`net_sock.h/net_sock.c`

Prototype

```
#if (NET_SOCK_CFG_SEL_EN == DEF_ENABLED)

NET_SOCK_RTN_CODE  NetSock_Sel (NET_SOCK_QTY       sock_nbr_max,
                                NET_SOCK_DESC     *p_sock_desc_rd,
                                NET_SOCK_DESC     *p_sock_desc_wr,
                                NET_SOCK_DESC     *p_sock_desc_err,
                                NET_SOCK_TIMEOUT  *p_timeout,
                                RTOS_ERR          *p_err)
```

Arguments

`sock_nbr_max`

Maximum number of socket descriptors/handle identifiers in the socket descriptor sets.

`p_sock_desc_rd`

Pointer to a set of socket descriptors/handle identifiers to :

1. Check for available read operation(s).
2. (a) Return the actual socket descriptors/handle identifiers ready for available read operation(s), if NO error(s).
   (b) Return the initial, non-modified set of socket descriptors/handle identifiers, on any error(s).
   (c) Return a null-valued (i.e., zero-cleared) descriptor set, if any timeout expires.

`p_sock_desc_wr`

Pointer to a set of socket descriptors/handle identifiers to :

1. Check for available write operation(s).
2. (a) Return the actual socket descriptors/handle identifiers ready for available write operation(s), if NO error(s).
   (b) Return the initial, non-modified set of socket descriptors/handle identifiers, on any error(s).
   (c) Return a null-valued (i.e., zero-cleared) descriptor set, if any timeout expires.

`p_sock_desc_err`

Pointer to a set of socket descriptors/handle identifiers to :

1. Check for any error(s) and/or exception(s).
2. (a) Return the actual socket descriptors/handle identifiers flagged with any error(s) &/or exception(s), if NO non-descriptor-related error(s).
   (b) Return the initial, non-modified set of socket descriptors/handle identifiers, on any error(s).
   (c) Return a null-valued (i.e., zero-cleared) descriptor set, if any timeout expires.

`p_timeout`

Pointer to a timeout.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_TYPE`
- `RTOS_ERR_NET_INVALID_CONN`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`

- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_INVALID_STATE
- RTOS_ERR_NET_CONN_CLOSED_FAULT
- RTOS_ERR_TIMEOUT
- RTOS_ERR_ABORT

Returned Value

- Number of socket descriptors with available resources &/or operations, if any.
- NET_SOCK_BSD_RTN_CODE_TIMEOUT , on timeout.
- NET_SOCK_BSD_ERR_SEL , otherwise.

Notes / Warnings

1. (a) IEEE Std 1003.1, 2004 Edition, Section ' select() : DESCRIPTION' states that :
    1. (A) " select() ... shall support" the following file descriptor types :
        1. "regular files," ...
        2. "terminal and pseudo-terminal devices," ...
        3. "STREAMS-based files," ...
        4. "FIFOs," ...
        5. "pipes," ...
        6. "sockets."

       (B) "The behavior of ... select() on ... other types of ... file descriptors ... is unspecified."
    2. Network Socket Layer supports BSD 4.x select() functionality with the following restrictions/constraints :

       (A) ONLY supports the following file descriptor types :
        1. Sockets

   (b) IEEE Std 1003.1, 2004 Edition, Section ' select() : DESCRIPTION' states that :
    1. (A) "The ' nfds ' argument (' sock_nbr_max ') specifies the range of descriptors to be tested. The first 'nfds' descriptors shall be checked in each set. that is, the [following] descriptors ... in the descriptor sets shall be examined" :
        1. "from zero" ...
        2. "through nfds-1."

       (B) Stevens/Fenner/Rudoff, UNIX Network Programming, Volume 1, 3rd Edition, 6th Printing, Section 6.3, Page 163 states that :
        1. ... since "descriptors start at 0" ...
        2. "the ['nfds'] argument" specifies :

           (a) "the number of descriptors," ...

           (b) "not the largest value."
    2. "The select() function shall ... examine the file descriptor sets whose addresses are passed in the ' readfds ' (' p_sock_desc_rd '), ' writefds ' (' p_sock_desc_wr '), and ' errorfds ' (' p_sock_desc_err ') parameters to see whether some of their descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively."

       (A)
        1. (a) "If the ' readfds ' argument (' p_sock_desc_rd ') is not a null pointer, it points to an object of type ' fd_set ' (' NET_SOCK_DESC ') that on input specifies the file descriptors to be checked for being ready to read, and on output indicates which file descriptors are ready to read."

           (b) "A descriptor shall be considered ready for reading when a call to an input function ... would not block, whether or not the function would transfer data successfully. (The function might return data, an end-of-file indication, or an error other than one indicating that it is blocked, and in each of these cases the descriptor shall be considered ready for reading.)" :
            1. "If the socket is currently listening, then it shall be marked as readable if an incoming connection request has been received, and a call to the accept() function shall complete without blocking."

           (c) Stevens/Fenner/Rudoff, UNIX Network Programming, Volume 1, 3rd Edition, 6th Printing, Section 6.3, Pages 164-165 states that "a socket is ready for reading if any of the following ... conditions is true" :
            1. "A read operation on the socket will not block and will return a value greater than 0 (i.e., the data that is ready to be read)."

2. "The read half of the connection is closed (i.e., a TCP connection that has received a FIN). A read operation ... will not block and will return 0 (i.e., EOF)."
3. "The socket is a listening socket and the number of completed connections is nonzero. An `accept()` on the listening socket will ... not block."
4. "A socket error is pending. A read operation on the socket will not block and will return an error (-1) with 'errno' set to the specific error condition."
(A) Stevens/Fenner/Rudoff, UNIX Network Programming, Volume 1, 3rd Edition, 6th Printing, Section 6.3, Page 165 states "that when an error occurs on a socket, it is [also] marked as both readable and writeable by `select()` ".

2. (a) "If the '`writefds`' argument ('`p_sock_desc_wr`') is not a null pointer, it points to an object of type '`fd_set`' ('`NET_SOCK_DESC`') that on input specifies the file descriptors to be checked for being ready to write, and on output indicates which file descriptors are ready to write."
(b) "A descriptor shall be considered ready for writing when a call to an output function ... would not block, whether or not the function would transfer data successfully" :
   1. "If a non-blocking call to the connect() function has been made for a socket, and the connection attempt has either succeeded or failed leaving a pending error, the socket shall be marked as writable."

(c) Stevens/Fenner/Rudoff, UNIX Network Programming, Volume 1, 3rd Edition, 6th Printing, Section 6.3, Page 165 states that "a socket is ready for writing if any of the following ... conditions is true" :
   1. "A write operation will not block and will return a positive value (e.g., the number of bytes accepted by the transport layer)."
   2. "The write half of the connection is closed."
   3. "A socket using a non-blocking `connect()` has completed the connection, or the `connect()` has failed."
   4. "A socket error is pending. A write operation on the socket will not block and will return an error (-1) with 'errno' set to the specific error condition."
   (A) Stevens/Fenner/Rudoff, UNIX Network Programming, Volume 1, 3rd Edition, 6th Printing, Section 6.3, Page 165 states "that when an error occurs on a socket, it is [also] marked as both readable and writeable by `select()` ".

3. (a) "If the '`errorfds`' argument ('`p_sock_desc_err`') is not a null pointer, it points to an object of type 'fd_set' ('`NET_SOCK_DESC`') that on input specifies the file descriptors to be checked for error conditions pending, and on output indicates which file descriptors have error conditions pending."
(b) "A file descriptor ... shall be considered to have an exceptional condition pending ... as noted below" :
   1. (A) "A socket ... receive operation ... [that] would return out-of-band data without blocking."
   (B) "A socket ... [with] out-of-band data ... present in the receive queue."
   2. "If a socket has a pending error."
   3. "Other circumstances under which a socket may be considered to have an exceptional condition pending are protocol-specific and implementation-defined."

(c) Stevens/Fenner/Rudoff, UNIX Network Programming, Volume 1, 3rd Edition, 6th Printing, Section 6.3, Page 165 states that "a socket has an exception condition pending if ... any of the following ... conditions is true" :
   1. (A) "Out-of-band data for the socket" is currently available; ...
   (B) "Or the socket is still at the out-of-band mark."

(d) Stevens/Fenner/Rudoff, UNIX Network Programming, Volume 1, 3rd Edition, 6th Printing, Section 6.3, Page 165 states "that when an error occurs on a socket, it is [also] marked as both readable and writeable by `select()` ".

(B)

1. (a) "Upon successful completion, ... `select()` ... shall" :
   1. "modify the objects pointed to by the 'readfds' ('`p_sock_desc_rd`'), 'writefds' ('`p_sock_desc_wr`'), and 'errorfds' ('`p_sock_desc_err`') arguments to indicate which file descriptors are ready for reading, ready for writing, or have an error condition pending, respectively," ...
   2. "and shall return the total number of ready descriptors in all the output sets."

(b)
   1. "For each file descriptor less than nfds ('`sock_nbr_max`'), the corresponding bit shall be set on successful completion" :
   (A) "if it was set on input" ...
   (B) "and the associated condition is true for that file descriptor."
   2. Conversely, "for each file descriptor ... the corresponding bit shall be ... [clear] ... on ... completion" :
   (A) If it was clear on input. ...
   (B) If the associated condition is NOT true for that file descriptor.

(C) Or it was set on input, but any timeout occurs (see Note #3c2B).

2. `select()` can NOT absolutely guarantee that descriptors returned as ready will still be ready during subsequent operations since any higher priority tasks or processes may asynchronously consume the descriptors' operations and/or resources. This can occur since `select()` functionality and subsequent operations are NOT atomic operations protected by network, file system, or operating system mechanisms.
3. (A) "The '`timeout`' parameter ('`p_timeout`') controls how long ... `select()` ... shall take before timing out."
   1. (a) "If the '`timeout`' parameter ('`p_timeout`') is not a null pointer, it specifies a maximum interval to wait for the selection to complete."
      1. "If none of the selected descriptors are ready for the requested operation, ... `select()` ... shall block until at least one of the requested operations becomes ready ... or ... until the timeout occurs."
      2. "If the specified time interval expires without any requested operation becoming ready, the function shall return."
      3. "To effect a poll, the '`timeout`' parameter ('`p_timeout`') should not be a null pointer, and should point to a zero-valued timespec structure ('`NET_SOCK_TIMEOUT`')."

      (b)
      1. (A) "If the '`readfds`' ('`p_sock_desc_rd`'), '`writefds`' ('`p_sock_desc_wr`'), and '`errorfds`' ('`p_sock_desc_err`') arguments are" ...
         1. "all null pointers" ...
         2. [or all null-valued (i.e., no file descriptors set)]...

         (B) "and the '`timeout`' argument ('`p_timeout`') is not a null pointer," ...
      2. ... then "select() ... shall block for the time specified".
   2. "If the '`timeout`' parameter ('`p_timeout`') is a null pointer, then the call to ... `select()` shall block indefinitely until at least one descriptor meets the specified criteria."

      (a)
      1. Although IEEE Std 1003.1, 2004 Edition, Section '`select()` : DESCRIPTION' states that `select()` may "block indefinitely until ... one of the requested operations becomes ready ... or until interrupted by a signal", it does NOT explicitly specify how to handle the case where the descriptor arguments and the timeout parameter argument are all NULL pointers.
      2. However, since inter-process signals are NOT currently supported, it does NOT seem reasonable to block a task or process indefinitely (i.e., forever) if the descriptor arguments and the timeout parameter argument are all NULL pointers. Instead, an 'invalid timeout interval' error should be returned.

   (B)
   1. "For the `select()` function, the timeout period is given ... in an argument ('`p_timeout`') of type struct '`timeval`' ('`NET_SOCK_TIMEOUT`')" ... :

      (a) "in seconds" ...

      (b) "and microseconds."
   2. (a)
      1. "Implementations may place limitations on the maximum timeout interval supported" :

         (A) "All implementations shall support a maximum timeout interval of at least 31 days."
         1. However, since maximum timeout interval values are dependent on the specific OS implementation; a maximum timeout interval can NOT be guaranteed.

         (B) "If the '`timeout`' argument ('`p_timeout`') specifies a timeout interval greater than the implementation-defined maximum value, the maximum value shall be used as the actual timeout value."
      2. "Implementations may also place limitations on the granularity of timeout intervals" :

         (A) "If the requested 'timeout' interval requires a finer granularity than the implementation supports, the actual timeout interval shall be rounded up to the next supported value."

      (b) Operating systems may have different minimum/maximum ranges and resolutions for timeouts while pending or waiting on an operating system resource to become available (see Note #3b3A1a) versus time delays for suspending a task or process that is NOT pending or waiting for an operating system resource (see Note #3b3A1b).

      (c)
      1. (A) IEEE Std 1003.1, 2004 Edition, Section '`select()` : RETURN VALUE' states that :
         1. "Upon successful completion, ... `select()` ... shall return the total number of bits set in the bit masks."
         2. (a) "Otherwise, -1 shall be returned," ...

            (b) "and 'errno' shall be set to indicate the error." 'errno' NOT currently supported (see '`net_bsd.h` Note #1b').

         (B) Stevens/Fenner/Rudoff, UNIX Network Programming, Volume 1, 3rd Edition, 6th Printing, Section 6.3, Page 161 states that BSD select() function "returns ... 0 on timeout".

IEEE Std 1003.1, 2004 Edition, Section 'select() : DESCRIPTION' states that :
(A) "On failure, the objects pointed to by the 'readfds' ('p_sock_desc_rd'), 'writefds' ('p_sock_desc_wr'), and 'errorfds' ('p_sock_desc_err') arguments shall not be modified."

1. Since 'p_sock_desc_rd', 'p_sock_desc_wr', and 'p_sock_desc_err' arguments are both input and output arguments; their input values, prior to use, MUST be copied to return their initial input values PRIOR to all other validation or function handling in case of any error(s).

(B) "If the 'timeout' interval expires without the specified condition being true for any of the specified file descriptors, the objects pointed to by the 'readfds' ('p_sock_desc_rd'), 'writefds' ('p_sock_desc_wr'), and 'errorfds' ('p_sock_desc_err') arguments shall have all bits set to 0."

(d) IEEE Std 1003.1, 2004 Edition, Section 'select() : ERRORS' states that "under the following conditions, ... select() shall fail and set 'errno' to" :

1. "[EBADF] - One or more of the file descriptor sets specified a file descriptor that is not a valid open file descriptor."
2. "[EINVAL]" -
   (A) "The 'nfds' argument ('sock_nbr_max') is" :
   1. "less than 0 or" ...
   2. "greater than FD_SETSIZE."

   (B) "An invalid timeout interval was specified."
3. A socket events table lists requested socket or connection ID numbers/handle identifiers and their respective socket event operations.
   (a) Socket event tables are terminated with NULL table entry values.
   (b)
   1. NET_SOCK_CFG_SEL_NBR_EVENTS_MAX configures socket event tables' maximum number of socket events/operations.
   2. This value is used to declare the size of the socket events tables. Note that one additional table entry is added for a terminating NULL table entry at a maximum table index 'NET_SOCK_CFG_SEL_NBR_EVENTS_MAX'.
4. Since datagram-type sockets typically never wait on transmit operations, no socket event should wait on datagram-type socket transmit operations or transmit errors.

NetSock_SelAbort()

Description

Aborts (unblocks) all tasks that are pending on a particular socket using the select.

Files

net_sock.h/net_sock.c

Prototype

```
void  NetSock_SelAbort (NET_SOCK_ID   sock_id,
                        RTOS_ERR     *p_err)
```

Arguments

sock_id

Socket descriptor/handle identifier of socket to abort the select.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_HANDLE

Returned Value

None.

Notes / Warnings

None.

NetSock_TxData()

Description

Transmits data through a socket.

Files

net_sock.h/net_sock.c

Prototype

```
NET_SOCK_RTN_CODE  NetSock_TxData (NET_SOCK_ID         sock_id,
                                   void               *p_data,
                                   CPU_INT16U          data_len,
                                   NET_SOCK_API_FLAGS  flags,
                                   RTOS_ERR           *p_err)
```

Arguments

sock_id

Socket descriptor/handle identifier of socket to transmit data.

p_data

Pointer to application data to transmit.

data_len

Length of application data to transmit (in octets).

flags

Flags to select transmit options; bit-field flags logically OR'd :

- NET_SOCK_FLAG_NONE No socket flags selected.
- NET_SOCK_FLAG_TX_NO_BLOCK Transmit socket data without blocking.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_TYPE
- RTOS_ERR_NET_RETRY_MAX
- RTOS_ERR_BLK_ALLOC_CALLBACK
- RTOS_ERR_NOT_SUPPORTED
- RTOS_ERR_SEG_OVF
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_FAIL

- RTOS_ERR_NET_INVALID_ADDR_SRC
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_NET_IF_LINK_DOWN
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_INVALID_STATE
- RTOS_ERR_ABORT
- RTOS_ERR_TIMEOUT
- RTOS_ERR_TX
- RTOS_ERR_NOT_FOUND
- RTOS_ERR_ALREADY_EXISTS
- RTOS_ERR_NET_INVALID_CONN
- RTOS_ERR_RX
- RTOS_ERR_NOT_READY
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_NET_NEXT_HOP
- FAULT
- RTOS_ERR_NET_ADDR_UNRESOLVED

Returned Value

- Number of positive data octets transmitted, if NO error(s).
- `NET_SOCK_BSD_RTN_CODE_CONN_CLOSED` , if socket connection closed.
- `NET_SOCK_BSD_ERR_TX` , otherwise.

Notes / Warnings

1. (a)
   1. (A) Datagram-type sockets transmit and receive all data atomically -- i.e., every single, complete datagram transmitted MUST be received as a single, complete datagram. Thus each call to transmit data MUST be transmitted in a single, complete datagram.
      (B) Since IP transmit fragmentation is NOT currently supported (see 'net_ip.h Note #1d'), if the socket's type is datagram and the requested transmit data length is greater than the socket/transport layer MTU, then NO data is transmitted and `RTOS_ERR_WOULD_OVF` error is returned.
   2. (A)
      1. Stream-type sockets transmit and receive all data octets in one or more non-distinct packets. In other words, the application data is NOT bounded by any specific packet(s); rather, it is contiguous and sequenced from one packet to the next.
      2. Thus if the socket's type is stream and the socket's transmit data queue(s) are NOT large enough for the transmitted data, the transmit data queue(s) are maximally filled with transmit data and the remaining data octets are discarded but may be re-transmitted by later application-socket transmits.
      3. Therefore, NO stream-type socket transmit data length should be "too long to pass through the underlying protocol" and cause the socket transmit to "fail ... [with] no data ... transmitted" (see Note #3a1B1).

      (B) Thus it is typical -- but NOT absolutely required -- that a single application task ONLY transmit or request to transmit data to a stream-type socket.

   (b) ' `data_len` ' of 0 octets NOT allowed.
2. Only some socket transmit flag options are implemented. If other flag options are requested, `NetSock_TxData()` handler function(s) abort and return appropriate error codes so that requested flag options are NOT silently ignored.
3. Although NO socket send() specification states to return '0' when the socket's connection is closed, it seems reasonable to return '0' since it is possible for the socket connection to be close()'d or shutdown() by the remote host.

NetSock_TxDataTo()

Description

Transmit data through a network socket. This must be used with the Datagram type socket to specify the remote destination address, since no connection has been established for Datagram socket type.

Files

`net_sock.h/net_sock.c`

Prototype

```
NET_SOCK_RTN_CODE  NetSock_TxDataTo (NET_SOCK_ID         sock_id,
                                     void                *p_data,
                                     CPU_INT16U          data_len,
                                     NET_SOCK_API_FLAGS  flags,
                                     NET_SOCK_ADDR       *p_addr_remote,
                                     NET_SOCK_ADDR_LEN   addr_len,
                                     RTOS_ERR            *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to transmit data.

`p_data`

Pointer to application data to transmit.

`data_len`

Length of application data to transmit (in octets).

`flags`

Flags to select transmit options; bit-field flags logically OR'd :

- `NET_SOCK_FLAG_NONE` No socket flags selected.
- `NET_SOCK_FLAG_TX_NO_BLOCK` Transmit socket data without blocking.

`p_addr_remote`

Pointer to destination address buffer.

`addr_len`

Length of destination address buffer (in octets).

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_TYPE`
- `RTOS_ERR_NET_RETRY_MAX`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_NOT_SUPPORTED`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_FAIL`
- `RTOS_ERR_NET_INVALID_ADDR_SRC`
- `RTOS_ERR_WOULD_OVF`

- RTOS_ERR_NET_IF_LINK_DOWN
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_INVALID_STATE
- RTOS_ERR_ABORT
- RTOS_ERR_TIMEOUT
- RTOS_ERR_TX
- RTOS_ERR_NOT_FOUND
- RTOS_ERR_ALREADY_EXISTS
- RTOS_ERR_NET_INVALID_CONN
- RTOS_ERR_RX
- RTOS_ERR_NOT_READY
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_NET_NEXT_HOP
- RTOS_ERR_NET_CONN_CLOSED_FAULT
- RTOS_ERR_NET_ADDR_UNRESOLVED

Returned Value

- Number of positive data octets transmitted, if NO error(s).
- `NET_SOCK_BSD_RTN_CODE_CONN_CLOSED` , if socket connection closed.
- `NET_SOCK_BSD_ERR_TX` , otherwise.

Notes / Warnings

1. (a)
   1. (A) Datagram-type sockets transmit and receive all data atomically -- i.e., every single, complete datagram transmitted MUST be received as a single, complete datagram. Thus each call to transmit data MUST be transmitted in a single, complete datagram.
      (B) Since IP transmit fragmentation is NOT currently supported, if the socket's type is datagram and the requested transmit data length is greater than the socket/transport layer MTU, then NO data is transmitted and `RTOS_ERR_WOULD_OVF` error is returned.
   2. (A)
      1. Stream-type sockets transmit and receive all data octets in one or more non-distinct packets. In other words, the application data is NOT bounded by any specific packet(s); rather, it is contiguous and sequenced from one packet to the next.
      2. Thus if the socket's type is stream and the socket's transmit data queue(s) are NOT large enough for the transmitted data, the transmit data queue(s) are maximally filled with transmit data and the remaining data octets are discarded but may be re-transmitted by later application-socket transmits.
      3. Therefore, NO stream-type socket transmit data length should be "too long to pass through the underlying protocol" and cause the socket transmit to "fail ... [with] no data ... transmitted".

      (B) Thus it is typical -- but NOT absolutely required -- that a single application task ONLY transmit or request to transmit data to a stream-type socket.

   (b) ' `data_len` ' of 0 octets NOT allowed.
2. Only some socket transmit flag options are implemented. If other flag options are requested, `NetSock_TxData()` handler function(s) abort and return appropriate error codes so that requested flag options are NOT silently ignored.
3. Although NO socket send() specification states to return '0' when the socket's connection is closed, it seems reasonable to return '0' since it is possible for the socket connection to be close()'d or shutdown() by the remote host.

BSD Sockets Functions API

Function Name	Description
accept()	Wait for new socket connections on a listening server socket.
bind()	Assign network addresses to sockets.

Function Name	Description
close()	Terminate communication and free a socket.
connect()	Connect a local socket to a remote socket address.
FD_CLR()	Remove a socket file descriptor ID as a member of a file descriptor set.
FD_ISSET()	Check if a socket file descriptor ID is a member of a file descriptor set.
FD_SET()	Add a socket file descriptor ID as a member of a file descriptor set.
FD_ZERO()	Initialize/zero-clear a file descriptor set.
getsockopt()	Get a specific option value on a specific TCP socket.
htonl()	Convert 32-bit integer values from CPU host-order to network-order.
htons()	Convert 16-bit integer values from CPU host-order to network-order.
inet_addr()	Convert a string of an IPv4 address in dotted-decimal notation to an IPv4 address in host-order.
inet_aton()	Convert an IPv4 address in ASCII dotted-decimal notation to a network protocol IPv4 address in network-order.
inet_ntoa()	Convert an IPv4 address in host-order into an IPv4 dotted-decimal notation ASCII strin
inet_ntop()	Converts an IPv4 or IPv6 Internet network address into a string in Internet standard format.
inet_pton()	Converts an IPv4 or IPv6 Internet network address in its standard text presentation form into its numeric binary form.
getaddrinfo()	Converts human-readable text strings representing hostnames or IP addresses into a dynamically allocated linked list of struct addrinfo structures.
freeaddrinfo()	Frees addrinfo structures information that getaddrinfo() has allocated.
listen()	Set a socket to accept incoming connections.
ntohl()	Convert 32-bit integer values from network-order to CPU host-order.
ntohs()	Convert 16-bit integer values from network-order to CPU host-order.
recv() / recvfrom()	Copy up to a specified number of bytes received from a remote socket into an application memory buffer.
select()	Check if any sockets are ready for available read or write operations or error conditions.
send() / sentto()	Copy bytes from an application memory buffer into a socket to send to a remote socket.
setsockopt()	Set a specific option on a specific TCP socket.
socket()	Create a datagram (i.e., UDP) or stream (i.e., TCP) type socket.

accept() (TCP only)

Description

Gets a new accepted socket from a socket set to listen for connection requests.

Files

`net_bsd.h/net_bsd.c`

Prototype

```
#ifdef  NET_SOCK_TYPE_STREAM_MODULE_EN

int  accept (       int         sock_id,
         struct  sockaddr   *p_addr_remote,
               socklen_t  *p_addr_len)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of listen socket.

`p_addr_remote`

Pointer to an address buffer that will receive the socket address structure of the accepted socket's remote address, if NO error(s).

`p_addr_len`

Pointer to a variable to pass the size of the address buffer pointed to by '`p_addr_remote`' and returns the actual size of socket address structure with the accepted socket's remote address, if NO error(s). Return 0, otherwise.

Returned Value

- Socket descriptor/handle identifier of new accepted socket, if NO error(s).
- -1, otherwise.

Notes / Warnings

1. Socket address structure '`sa_family`' member returned in host-order and SHOULD NOT be converted to network-order.
2. Socket address structure addresses returned in network-order and SHOULD be converted from network-order to host-order.

bind()

Description

Binds a socket to a local address.

Files

`net_bsd.h/net_bsd.c`

Prototype

```
int bind (      int         sock_id,
         struct sockaddr   *p_addr_local,
               socklen_t   addr_len)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to bind to a local address.

`p_addr_local`

Pointer to socket address structure.

`addr_len`

Length of socket address structure (in octets).

Returned Value

- 0, if NO error(s).

- -1, otherwise.

Notes / Warnings

1. Socket address structure '`sa_family`' member MUST be configured in host-order and MUST NOT be converted to/from network-order.
2. Socket address structure addresses MUST be configured/converted from host-order to network-order.

close()

Description

Closes a socket.

Files

`net_bsd.h/net_bsd.c`

Prototype

```
int close (int sock_id)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to close.

Returned Value

- 0, if NO error(s).
- -1, otherwise.

Notes / Warnings

Once an application closes its socket, NO further operations on the socket are allowed and the application MUST NOT continue to access the socket.

connect()

Description

Connects a socket to a remote server.

Files

`net_bsd.h/net_bsd.c`

Prototype

```
int connect (      int        sock_id,
          struct  sockaddr  *p_addr_remote,
                socklen_t  addr_len)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to connect.

`p_addr_remote`

Pointer to socket address structure (see Note #1).

`addr_len`

Length of socket address structure (in octets).

Returned Value

- 0, if NO error(s). -1, otherwise.

Notes / Warnings

1. Socket address structure 'sa_family' member MUST be configured in host-order and MUST NOT be converted to/from network-order.
2. Socket address structure addresses MUST be configured/converted from host-order to network-order.

FD_CLR()

Remove a socket file descriptor ID as a member of a file descriptor set. See function NET_SOCK_DESC_CLR() for more information.

Files

`net_bsd.h`

Prototype

```
FD_CLR(fd, fdsetp);
```

Required Configuration

None.

FD_ISSET()

Check if a socket file descriptor ID is a member of a file descriptor set. See function NET_SOCK_DESC_IS_SET() for more information.

Files

`net_bsd.h`

Prototype

```
    FD_ISSET(fd, fdsetp);
```

Required Configuration

None.

FD_SET()

Add a socket file descriptor ID as a member of a file descriptor set. See function NET_SOCK_DESC_SET() for more information.

Files

`net_bsd.h`

Prototype

```
FD_SET(fd, fdsetp);
```

Required Configuration

None.

FD_ZERO()

Initialize/zero-clear a file descriptor set. See function NET_SOCK_DESC_INIT() for more information.

Files

`net_bsd.h`

Prototype

```
FD_ZERO(fdsetp);
```

Required Configuration

None.

getsockopt()

Description

Gets the socket option.

Files

`net_bsd.h/net_bsd.c`

Prototype

```
int getsockopt(int        sock_id,
               int        protocol,
               int        opt_name,
               void       *p_opt_val,
               socklen_t  *p_opt_len)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to get the option.

`protocol`

Protocol level at which the option resides.

`opt_name`

Name of the single socket option to get.

`p_opt_val`

Pointer to the socket option value to get.

`opt_len`

Option length.

Returned Value

- 0, if NO error(s).
- -1, otherwise.

Notes / Warnings

None.

htonl()

Convert 32-bit integer values from CPU host-order to network-order. See function NET_UTIL_HOST_TO_NET_32 for more information.

Files

`net_bsd.h`

Prototype

```
htonl(val);
```

Required Configuration

None.

htons()

Convert 16-bit integer values from CPU host-order to network-order. See function NET_UTIL_HOST_TO_NET_16 for more information.

Files

`net_bsd.h`

Prototype

```
htons(val);
```

inet_addr() (IPv4 only)

Description

Converts an IPv4 address in ASCII dotted-decimal notation to a network protocol IPv4 address in network-order.

Files

`net_bsd.h/net_bsd.c`

Prototype

```
in_addr_t  inet_addr (char  *p_addr)
```

Arguments

`p_addr`

Pointer to an ASCII string that contains a dotted-decimal IPv4 address (see Note #2).

Returned Value

- Network-order IPv4 address represented by ASCII string, if NO error(s).
- -1, otherwise.

Notes / Warnings

1. RFC #1983 states that "dotted decimal notation ... refers [to] IP addresses of the form A.B.C.D; where each letter represents, in decimal, one byte of a four byte IP address." In other words, the dotted-decimal notation separates four decimal octet values by the dot (or period) character ('.'). Each decimal value represents one octet of the IP address starting with the most significant octet in network-order.
   IP Address Examples : `192.168.1.64 = 0xC0A80140`
2. The dotted-decimal ASCII string MUST :
   (a) Include ONLY decimal values and the dot (or period) character ('.'). ALL other characters are trapped as invalid, including any leading or trailing characters.
   (b) Include UP TO four decimal values separated by UP TO three dot characters and MUST be terminated with the NULL character.
   (c) Ensure that each decimal value does NOT exceed the maximum octet value (i.e., 255).
   (d) Ensure that each decimal value does NOT include leading zeros.

inet_aton() (IPv4 only)

Description

Converts an IPv4 address in ASCII dotted-decimal notation to a network protocol IPv4 address in network-order.

Files

`net_bsd.h/net_bsd.c`

Prototype

```
#ifdef  NET_IPv4_MODULE_EN

int  inet_aton(      char    *p_addr_in,
          struct in_addr  *p_addr)
```

Arguments

`p_addr_in`

Pointer to an ASCII string that contains a dotted-decimal IPv4 address.

`p_addr`

Pointer to an IPv4 address.

Returned Value

- 1 if the supplied address is valid.
- 0, otherwise.

Notes / Warnings

1. RFC #1983 states that "dotted decimal notation ... refers [to] IP addresses of the form A.B.C.D; where each letter represents, in decimal, one byte of a four byte IP address."
2. IEEE Std 1003.1, 2004 Edition - `inet_addr`, `inet_ntoa` - IPv4 address manipulation:
   (a) Values specified using IPv4 dotted decimal notation take one of the following forms:
   1. a.b.c.d - With a four-part address, each part shall be interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address.
   2. a.b.c - With a three-part address, the last part shall be interpreted as a 16-bit quantity and placed in the right-most two bytes of the network address. This makes the three-part address format convenient for specifying Class B network addresses as "128.net.host".
   3. a.b - With a two-part address, the last part shall be interpreted as a 24-bit quantity and placed in the right-most three bytes of the network address. This makes the two-part address format convenient for specifying Class A network addresses as "net.host".
   4. a - With a one-part address, the value shall be stored directly in the network address without any byte rearrangement.
3. The dotted-decimal ASCII string MUST :
   (a) Include ONLY decimal values and the dot (or period) character ('.'). ALL other characters trapped are invalid, including any leading or trailing characters.
   (b) Include UP TO four decimal values separated by UP TO three dot characters and MUST be terminated with the NULL character.
   (c) Ensure that each decimal value does NOT exceed the maximum value for its form:
   1. a.b.c.d - 255.255.255.255
   2. a.b.c - 255.255.65535
   3. a.b - 255.16777215
   4. a - 4294967295
   (d) Ensure that each decimal value does NOT include leading zeros.

inet_ntoa() (IPv4 only)

Description

Converts a network protocol IPv4 address into a dotted-decimal notation ASCII string.

Files

`net_bsd.h/net_bsd.c`

Prototype

```
#ifdef  NET_IPv4_MODULE_EN

char  *inet_ntoa (struct  in_addr  addr)
```

Arguments

`addr`

IPv4 address.

Returned Value

Pointer to ASCII string of converted IPv4 address, if NO error(s). Pointer to NULL, otherwise.

Notes / Warnings

1. RFC #1983 states that "dotted decimal notation ... refers [to] IP addresses of the form A.B.C.D; where each letter represents, in decimal, one byte of a four byte IP address."
   In other words, the dotted-decimal notation separates four decimal octet values by the dot (or period) character ('.'). Each decimal value represents one octet of the IP address starting with the most significant octet in network-order.
   IP Address Examples : 192.168.1.64 = 0xC0A80140
2. IEEE Std 1003.1, 2004 Edition, Section ' `inet_ntoa()` : DESCRIPTION' states that " `inet_ntoa()` ... need not be reentrant ... [and] is not required to be thread-safe."
   Since the character string is returned in a single, global character string array, this conversion function is NOT re-entrant.

inet_ntop()

Description

Converts an IPv4 or IPv6 Internet network address into a string in Internet standard format.

Files

`net_bsd.h/net_bsd.c`

Prototype

```
const char *inet_ntop (    int        af,
                     const void     *src,
                         char      *dst,
                         socklen_t  size)
```

Arguments

`af`

Address family:

- `AF_INET` IPv4 Address Family
- `AF_INET6` IPv6 Address Family

`src`

A pointer to the IP address in network byte to convert to a string.

`dst`

A pointer to a buffer in which to store the NULL-terminated string representation of the IP address.

`size`

Length, in characters, of the buffer pointed to by dst.

Returned Value

- Pointer to a buffer containing the string representation of IP address in the standard format, If no error occurs.
- `DEF_NULL` , otherwise.

Notes / Warnings

None.

inet_pton()

Description

Converts an IPv4 or IPv6 Internet network address in its standard text presentation form into its numeric binary form.

Files

`net_bsd.h/net_bsd.c`

Prototype

```
int inet_pton (      int   af,
              const char *src,
                   void *dst);
```

Arguments

`af`

Address family:

- `AF_INET` IPv4 Address Family
- `AF_INET6` IPv6 Address Family

`src`

A pointer to the IP address in network byte to convert to a string.

`dst`

A pointer to a buffer in which to store the NULL-terminated string representation of the IP address.

Returned Value

- 1, if no error.
- 0, if src does not contain a character string representing a valid network address in the specified address family.
- -1, if af does not contain a valid address family.

Notes / Warnings

None.

getaddrinfo()

Description

Converts human-readable text strings representing hostnames or IP addresses into a dynamically allocated linked list of struct addrinfo structures.

Files

`net_bsd.h/net_bsd.c`

Prototype

```
int getaddrinfo (const       char     *p_node_name,
                 const       char     *p_service_name,
                 const struct addrinfo  *p_hints,
                       struct addrinfo **pp_res)
```

Arguments

`p_node_name`

A pointer to a string that contains a host (node) name or a numeric host address string. For the Internet protocol, the numeric host address string is a dotted-decimal IPv4 address or an IPv6 hex address.

`p_service_name`

A pointer to a string that contains either a service name or port number represented as a string.

`p_hints`

A pointer to an addrinfo structure that provides hints about the type of socket the caller supports.

`pp_res`

A pointer to a linked list of one or more addrinfo structures that contains response information about the host.

Returned Value

- 0, if no error,

Most nonzero error codes returned map to the set of errors outlined by Internet Engineering Task Force (IETF) recommendations:

- EAI_ADDRFAMILY
- EAI_AGAIN
- EAI_BADFLAGS
- EAI_FAIL
- EAI_FAMILY
- EAI_MEMORY
- EAI_NONAME
- EAI_OVERFLOW
- EAI_SERVICE
- EAI_SOCKTYPE
- EAI_SYSTEM

Notes / Warnings

None.

freeaddrinfo()

Description

Frees addrinfo structures information that getaddrinfo() has allocated.

Files

`net_bsd.h/net_bsd.c`

Prototype

```
void freeaddrinfo(struct addrinfo *res)
```

Arguments

`res`

A pointer to the addrinfo structure or linked list of addrinfo structures to be freed.

Returned Value

None.

Notes / Warnings

None.

listen() (TCP only)

Description

Sets a socket to listen for connection requests.

Files

`net_bsd.h/net_bsd.c`

Prototype

```
#ifdef  NET_SOCK_TYPE_STREAM_MODULE_EN

int  listen (int  sock_id,
          int  sock_q_size)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to listen.

`sock_q_size`

Number of connection requests to queue on listen socket.

Returned Value

- 0, if NO error(s).
- -1, otherwise.

Notes / Warnings

None.

ntohl()

Convert 32-bit integer values from network-order to CPU host-order. See function NET_UTIL_NET_TO_HOST_32 for more information.

Files

`net_bsd.h`

Prototype

```
ntohl(val);
```

Required Configuration

None.

ntohs()

Convert 16-bit integer values from network-order to CPU host-order. See function NET_UTIL_NET_TO_HOST_16 for more information.

Files

`net_bsd.h`

Prototype

```
ntohs(val);
```

Required Configuration

None.

recv()

Description

Receives data from a socket.

Files

`net_bsd.h/net_bsd.c`

Prototype

```
ssize_t recv (int     sock_id,
          void   *p_data_buf,
         _size_t  data_buf_len,
          int     flags)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to receive data.

`p_data_buf`

Pointer to an application data buffer that will receive the socket's received data.

`data_buf_len`

Size of the application data buffer (in octets).

`flags`

Flags to select receive options; bit-field flags logically OR'd :

- 0 No socket flags selected.
- `MSG_PEEK` Receive socket data without consuming the socket data.
- `MSG_DONTWAIT` Receive socket data without blocking.

Returned Value

- Number of positive data octets received, if NO error(s). - 0, if socket connection closed.
- -1, otherwise.

Notes / Warnings

1. (a)
    1. Datagram-type sockets transmit and receive all data atomically -- i.e., every single, complete datagram transmitted MUST be received as a single, complete datagram.
    2. Thus if the socket's type is datagram and the receive data buffer size is NOT large enough for the received data, the receive data buffer is maximally filled with receive data but the remaining data octets are discarded and RTOS_ERR_WOULD_OVF error is returned.

    (b)
    1. (A) Stream-type sockets transmit and receive all data octets in one or more non-distinct packets. In other words, the application data is NOT bounded by any specific packet(s); rather, it is contiguous and sequenced from one packet to the next.
       (B) Thus if the socket's type is stream and the receive data buffer size is NOT large enough for the received data, the receive data buffer is maximally filled with receive data and the remaining data octets remain queued for later application-socket receives.
    2. It is typical, but NOT absolutely required, that a single application task ONLY receive or request to receive data from a stream-type socket.
2. Only some socket receive flag options are implemented. If other flag options are requested, socket receive handler function(s) abort and return appropriate error codes so that requested flag options are NOT silently ignored.
3. IEEE Std 1003.1, 2004 Edition, Section 'recv() : RETURN VALUE' states that :
   (a) "Upon successful completion, recv() shall return the length of the message in bytes."
   (b) "If no messages are available to be received and the peer has performed an orderly shutdown, recv() shall return 0."
   (c)
    1. "Otherwise, -1 shall be returned"
    2. "and 'errno' set to indicate the error." 'errno' is NOT currently supported.

recvfrom()

Description

Receives data from a socket.

Files

`net_bsd.h/net_bsd.c`

Prototype

```
ssize_t recvfrom (      int          sock_id,
                        void        *p_data_buf,
                        _size_t      data_buf_len,
                        int          flags,
                 struct sockaddr    *p_addr_remote,
                        socklen_t   *p_addr_len)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to receive data.

`p_data_buf`

Pointer to an application data buffer that will receive the socket's received data.

`data_buf_len`

Size of the application data buffer (in octets).

`flags`

Flags to select receive options; bit-field flags logically OR'd :

- 0 No socket flags selected.
- `MSG_PEEK` Receive socket data without consuming the socket data.
- `MSG_DONTWAIT` Receive socket data without blocking.

`p_addr_remote`

Pointer to an address buffer that will receive the socket address structure with the received data's remote address, if NO error(s).

`p_addr_len`

Pointer to a variable to pass the size of the address buffer pointed to by 'p_addr_remote' and returns the actual size of socket address structure with the received data's remote address, if NO error(s). Return 0, otherwise.

Returned Value

- Number of positive data octets received, if NO error(s).
- 0, if socket connection closed.
- -1, otherwise.

Notes / Warnings

1. (a)
    1. Datagram-type sockets transmit and receive all data atomically (i.e., every single, complete datagram transmitted MUST be received as a single, complete datagram).
    2. If the socket's type is datagram and the receive data buffer size is NOT large enough for the received data, the receive data buffer is maximally filled with receive data but the remaining data octets are discarded and NET_SOCK_ERR_INVALID_DATA_SIZE error is returned.

    (b)
    1. (A) Stream-type sockets transmit and receive all data octets in one or more non-distinct packets. In other words, the application data is NOT bounded by any specific packet(s); rather, it is contiguous and sequenced from one packet to the next.
       (B) Thus if the socket's type is stream and the receive data buffer size is NOT large enough for the received data, the receive data buffer is maximally filled with receive data and the remaining data octets remain queued for later application-socket receives.
    2. It is typical, but NOT absolutely required, that a single application task ONLY receive or request to receive data from a stream-type socket.
2. Only some socket receive flag options are implemented. If other flag options are requested, socket receive handler function(s) abort and return appropriate error codes so that requested flag options are NOT silently ignored.
3. (a) Socket address structure '`sa_family`' member returned in host-order and SHOULD NOT be converted to network-order.
   (b) Socket address structure addresses returned in network-order and SHOULD be converted from network-order to host-order.
4. IEEE Std 1003.1, 2004 Edition, Section 'recvfrom() : RETURN VALUE' states that :
   (a) "Upon successful completion, recvfrom() shall return the length of the message in bytes."
   (b) "If no messages are available to be received and the peer has performed an orderly shutdown, recvfrom() shall return 0."
   (c)
    1. "Otherwise, [-1 shall be returned]"
    2. "and 'errno' set to indicate the error." 'errno' is NOT currently supported.

select()

Description

Checks multiple file descriptors for available resources and/or operations.

Files

`net_bsd.h/net_bsd.c`

Prototype

```
#if    (NET_SOCK_CFG_SEL_EN == DEF_ENABLED)

int  select (      int       desc_nbr_max,
          struct  fd_set   *p_desc_rd,
          struct  fd_set   *p_desc_wr,
          struct  fd_set   *p_desc_err,
          struct  timeval  *p_timeout)
```

Arguments

`desc_nbr_max`

Maximum number of file descriptors in the file descriptor sets.

`p_desc_rd`

Pointer to a set of file descriptors to :

1. Check for available read operation(s).
2. (a) Return the actual file descriptors ready for available read operation(s), if NO error(s).
   (b) Return the initial, non-modified set of file descriptors, on any error(s).
   (c) Return a null-valued (i.e., zero-cleared) descriptor set, if any timeout expires.

`p_desc_wr`

Pointer to a set of file descriptors to :

1. Check for available write operation(s).
2. (a) Return the actual file descriptors ready for available write operation(s), if NO error(s).
   (b) Return the initial, non-modified set of file descriptors, on any error(s).
   (c) Return a null-valued (i.e., zero-cleared) descriptor set, if any timeout expires.

`p_desc_err`

Pointer to a set of file descriptors to :

1. Check for any error(s) and/or exception(s).
2. (a) Return the actual file descriptors flagged with any error(s) and/or exception(s), if NO non-descriptor-related error(s);
   (b) Return the initial, non-modified set of file descriptors, on any error(s);
   (c) Return a null-valued (i.e., zero-cleared) descriptor set, if any timeout expires.

`p_timeout`

Pointer to a timeout.

Returned Value

- Number of file descriptors with available resources and/or operations, if any.
- 0, on timeout.
- -1, otherwise.

Notes / Warnings

1. (a) IEEE Std 1003.1, 2004 Edition, Section 'select() : DESCRIPTION' states that :
   1. (A) "select() ... shall support" the following file descriptor types :
      1. "regular files"
      2. "terminal and pseudo-terminal devices"
      3. "STREAMS-based files"
      4. "FIFOs"
      5. "pipes"
      6. "sockets"

      (B) "The behavior of ... select() on ... other types of ... file descriptors ... is unspecified."
   2. Network Socket Layer supports BSD 4.x select() functionality with the following restrictions/constraints :

      (A) ONLY supports the following file descriptor types :
      1. Sockets

   (b) IEEE Std 1003.1, 2004 Edition, Section 'select() : DESCRIPTION' states that :
   1. (A) "The 'nfds' argument ('desc_nbr_max') specifies the range of descriptors to be tested. The first 'nfds' descriptors shall be checked in each set; the descriptors from zero through nfds-1 in the descriptor sets shall be examined."

      (B) Stevens/Fenner/Rudoff, UNIX Network Programming, Volume 1, 3rd Edition, 6th Printing, Section 6.3, Page 163 states that "the ['nfds'] argument" specifies :
      1. "the number of descriptors" ...
      2. "not the largest value"
   2. "The select() function shall ... examine the file descriptor sets whose addresses are passed in the 'readfds' ('p_desc_rd'), 'writefds' ('p_desc_wr'), and 'errorfds' ('p_desc_err') parameters to see whether some of their descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively."

      (A)
      1. (a) "If the 'readfds' argument ('p_desc_rd') is not a null pointer, it points to an object of type 'fd_set' that on input specifies the file descriptors to be checked for being ready to read, and on output indicates which file descriptors are ready to read."

         (b) "A descriptor shall be considered ready for reading when a call to an input function ... would not block, whether or not the function would transfer data successfully. (The function might return data, an end-of-file indication, or an error other than one indicating that it is blocked, and in each of these cases the descriptor shall be considered ready for reading.)" :
         1. "If the socket is currently listening, then it shall be marked as readable if an incoming connection request has been received, and a call to the accept() function shall complete without blocking."

         (c) Stevens/Fenner/Rudoff, UNIX Network Programming, Volume 1, 3rd Edition, 6th Printing, Section 6.3, Pages 164-165 states that "a socket is ready for reading if any of the following ... conditions is true" :
         1. "A read operation on the socket will not block and will return a value greater than 0 (i.e., the data that is ready to be read)."
         2. "The read half of the connection is closed (i.e., a TCP connection that has received a FIN). A read operation ... will not block and will return 0 (i.e., EOF)."
         3. "The socket is a listening socket and the number of completed connections is nonzero. An accept() on the listening socket will ... not block."
         4. "A socket error is pending. A read operation on the socket will not block and will return an error (-1) with 'errno' set to the specific error condition."
      2. (a) "If the 'writefds' argument ('`p_desc_wr`') is not a null pointer, it points to an object of type 'fd_set' that on input specifies the file descriptors to be checked for being ready to write, and on output indicates which file descriptors are ready to write."

         (b) "A descriptor shall be considered ready for writing when a call to an output function ... would not block, whether or not the function would transfer data successfully" :
         1. "If a non-blocking call to the connect() function has been made for a socket, and the connection attempt has either succeeded or failed leaving a pending error, the socket shall be marked as writable."

         (c) Stevens/Fenner/Rudoff, UNIX Network Programming, Volume 1, 3rd Edition, 6th Printing, Section 6.3, Page 165 states that "a socket is ready for writing if any of the following ... conditions is true" :
         1. "A write operation will not block and will return a positive value (e.g., the number of bytes accepted by the transport layer)."
         2. "The write half of the connection is closed."
         3. "A socket using a non-blocking connect() has completed the connection, or the connect() has failed."

4. "A socket error is pending. A write operation on the socket will not block and will return an error (-1) with 'errno' set to the specific error condition."
3. (a) "If the 'errorfds' argument ('p_desc_err') is not a null pointer, it points to an object of type 'fd_set' that on input specifies the file descriptors to be checked for error conditions pending, and on output indicates which file descriptors have error conditions pending."
   (b) "A file descriptor ... shall be considered to have an exceptional condition pending ... as noted below" :
   1. "If a socket has a pending error."
   2. "Other circumstances under which a socket may be considered to have an exceptional condition pending are protocol-specific and implementation-defined."

   (c) Stevens/Fenner/Rudoff, UNIX Network Programming, Volume 1, 3rd Edition, 6th Printing, Section 6.3, Page 165 states "that when an error occurs on a socket, it is [also] marked as both readable and writeable by select()".

(B)

1. (a) "Upon successful completion, ... select() ... shall" :
   1. "modify the objects pointed to by the 'readfds' ('`p_desc_rd`'), 'writefds' ('`p_desc_wr`'), and 'errorfds' ('`p_desc_err`') arguments to indicate which file descriptors are ready for reading, ready for writing, or have an error condition pending, respectively," ...
   2. "and shall return the total number of ready descriptors in all the output sets."

   (b)
   1. "For each file descriptor less than nfds ('`desc_nbr_max`'), the corresponding bit shall be set on successful completion" :
      (A) "if it was set on input" ...
      (B) "and the associated condition is true for that file descriptor."
2. select() can NOT absolutely guarantee that descriptors returned as ready will still be ready during subsequent operations since any higher priority tasks or processes may asynchronously consume the descriptors' operations and/or resources. This can occur since select() functionality and subsequent operations are NOT atomic operations protected by network, file system, or operating system mechanisms.
   However, as long as no higher priority tasks or processes access any of the same descriptors, then a single task or process can assume that all descriptors returned as ready by select() will still be ready during subsequent operations.
3. (A) "The 'timeout' parameter ('p_timeout') controls how long ... select() ... shall take before timing out."
   1. (a) "If the 'timeout' parameter ('p_timeout') is not a null pointer, it specifies a maximum interval to wait for the selection to complete."
      1. "If none of the selected descriptors are ready for the requested operation, ... select() ... shall block until at least one of the requested operations becomes ready ... or ... until the timeout occurs."
      2. "If the specified time interval expires without any requested operation becoming ready, the function shall return."
      3. "To effect a poll, the 'timeout' parameter ('`p_timeout`') should not be a null pointer, and should point to a zero-valued timespec structure ('timeval')."

      (b)
      1. (A) "If the 'readfds' ('`p_desc_rd`'), 'writefds' ('`p_desc_wr`'), and 'errorfds' ('p_desc_err') `arguments` are"
         1. "all null pointers"
         2. [or all null-valued (i.e., no file descriptors set)]

         (B) "and the 'timeout' argument ('p_timeout') is not a null pointer,"
      2. ... then "select() ... shall block for the time specified".
   2. "If the 'timeout' parameter ('p_timeout') is a null pointer, then the call to ... select() shall block indefinitely until at least one descriptor meets the specified criteria."

   (B)
   1. "For the select() function, the timeout period is given ... in an argument ('p_timeout') of type struct 'timeval'" ... :
      (a) "in seconds"
      (b) "and microseconds"
   2. (a)
      1. "Implementations may place limitations on the maximum timeout interval supported" :
         (A) "All implementations shall support a maximum timeout interval of at least 31 days."
         1. However, since maximum timeout interval values are dependent on the specific OS implementation; a maximum timeout interval CANNOT be guaranteed.

         (B) "If the 'timeout' argument ('p_timeout') specifies a timeout interval greater than the implementation-defined maximum value, the maximum value shall be used as the actual timeout value."
      2. "Implementations may also place limitations on the granularity of timeout intervals" :

(A) "If the requested 'timeout' interval requires a finer granularity than the implementation supports, the actual timeout interval shall be rounded up to the next supported value."

(c)

1. (A) IEEE Std 1003.1, 2004 Edition, Section 'select() : RETURN VALUE' states that :
   1. "Upon successful completion, ... select() ... shall return the total number of bits set in the bit masks."
   2. (a) "Otherwise, -1 shall be returned," ...
      (b) "and 'errno' shall be set to indicate the error." 'errno' NOT currently supported (see 'net_bsd.c Note #1b').
      (c) Stevens/Fenner/Rudoff, UNIX Network Programming, Volume 1, 3rd Edition, 6th Printing, Section 6.3, Page 161 states that BSD select() function "returns ... 0 on timeout".
   3. IEEE Std 1003.1, 2004 Edition, Section 'select() : DESCRIPTION' states that :
      (A) "On failure, the objects pointed to by the 'readfds' (`p_desc_rd`), 'writefds' (`p_desc_wr`), and 'errorfds' (`p_desc_err`) arguments shall not be modified."
      (B) "If the 'timeout' interval expires without the specified condition being true for any of the specified file descriptors, the objects pointed to by the 'readfds' (`p_desc_rd`), 'writefds' (`p_desc_wr`), and 'errorfds' (`p_desc_err`) arguments shall have all bits set to 0."

(d) IEEE Std 1003.1, 2004 Edition, Section 'select() : ERRORS' states that "under the following conditions, ... select() shall fail and set 'errno' to" :

1. "[EBADF] - One or more of the file descriptor sets specified a file descriptor that is not a valid open file descriptor."
2. "[EINVAL]" -
   (A) "The 'nfds' argument ('desc_nbr_max') is" :
   1. "less than 0 or" ...
   2. "greater than FD_SETSIZE."

   (B) "An invalid timeout interval was specified."
   'errno' is NOT currently supported.

send()

Description

Sends data through a socket.

Files

`net_bsd.h/net_bsd.c`

Prototype

```
ssize_t send (int      sock_id,
              void    *p_data,
             _size_t   data_len,
              int      flags)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to send data.

`p_data`

Pointer to application data to send.

`data_len`

Length of application data to send (in octets).

`flags`

Flags to select send options; bit-field flags logically OR'd :

- 0 No socket flags selected.
- `MSG_DONTWAIT` Send socket data without blocking.

Returned Value

- Number of positive data octets sent, if NO error(s).
- 0, if socket connection closed.
- -1, otherwise.

Notes / Warnings

1. (a)
   1. (A) Datagram-type sockets send and receive all data atomically -- i.e., every single, complete datagram sent MUST be received as a single, complete datagram. Thus each call to send data MUST be transmitted in a single, complete datagram.
      (B)
      1. IEEE Std 1003.1, 2004 Edition, Section 'send() : DESCRIPTION' states that "if the message is too long to pass through the underlying protocol, send() shall fail and no data shall be transmitted".
      2. Since IP transmit fragmentation is NOT currently supported (see 'net_ip.h Note #1d'), if the socket's type is datagram and the requested send data length is greater than the socket/transport layer MTU, then NO data is sent and NET_SOCK_ERR_INVALID_DATA_SIZE error is returned.
   2. (A)
      1. Stream-type sockets send and receive all data octets in one or more non-distinct packets. In other words, the application data is NOT bounded by any specific packet(s); rather, it is contiguous and sequenced from one packet to the next.
      2. Thus if the socket's type is stream and the socket's send data queue(s) are NOT large enough for the send data, the send data queue(s) are maximally filled with send data and the remaining data octets are discarded, but may be re-sent by later application-socket sends.
      3. Therefore, NO stream-type socket send data length should be "too long to pass through the underlying protocol" and cause the socket send to "fail ... [with] no data ... transmitted".
      (B) Thus it is typical -- but NOT absolutely required -- that a single application task ONLY send or request to send data to a stream-type socket.
   (b) '`data_len`' of 0 octets NOT allowed.
2. Only some socket send flag options are implemented. If other flag options are requested, socket send handler function(s) abort and return appropriate error codes so that requested flag options are NOT silently ignored.
3. (a) IEEE Std 1003.1, 2004 Edition, Section 'send() : RETURN VALUE' states that :
   1. "Upon successful completion, send() shall return the number of bytes sent."
      (A) Section 'send() : DESCRIPTION' elaborates that "successful completion of a call to sendto() does not guarantee delivery of the message".
      (B)
      1. Thus applications SHOULD verify the actual returned number of data octets transmitted and/or prepared for transmission.
      2. In addition, applications MAY desire verification of receipt and/or acknowledgement of transmitted data to the remote host, either inherently by the transport layer or explicitly by the application.
   2. (A) "Otherwise, -1 shall be returned."
      1. Section 'send() : DESCRIPTION' elaborates that "a return value of -1 indicates only locally-detected errors".
      (B) "and 'errno' set to indicate the error." 'errno' is NOT currently supported.
   (b) Although NO socket send() specification states to return '0' when the socket's connection is closed, it seems reasonable to return '0' since it is possible for the socket connection to be close()'d or shutdown() by the remote host.

sendto()

Description

Sends data through a socket.

Files

net_bsd.h/net_bsd.c

Prototype

```
ssize_t sendto (        int         sock_id,
                        void        *p_data,
                        _size_t     data_len,
                        int         flags,
                struct  sockaddr    *p_addr_remote,
                        socklen_t   addr_len)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to send data.

`p_data`

Pointer to application data to send.

`data_len`

Length of application data to send (in octets).

`flags`

Flags to select send options; bit-field flags logically OR'd :

- 0 No socket flags selected.
- `MSG_DONTWAIT` Send socket data without blocking.

`p_addr_remote`

Pointer to destination address buffer; required for datagram sockets, optional for stream sockets.

`addr_len`

Length of destination address buffer (in octets).

Returned Value

- Number of positive data octets sent, if NO error(s).
- 0, if socket connection closed.
- -1, otherwise.

Notes / Warnings

1. (a)
   1. (A) Datagram-type sockets send and receive all data atomically (i.e., every single, complete datagram sent MUST be received as a single, complete datagram). Each call to send data MUST be transmitted in a single, complete datagram. (B) Since IP transmit fragmentation is NOT currently supported, if the socket's type is datagram and the requested send data length is greater than the socket/transport layer MTU, then NO data is sent and RTOS_ERR_WOULD_OVF error is returned.
   2. (A)
      1. Stream-type sockets send and receive all data octets in one or more non-distinct packets. In other words, the application data is NOT bounded by any specific packet(s); rather, it is contiguous and sequenced from one packet to the next.
      2. Thus if the socket's type is stream and the socket's send data queue(s) are NOT large enough for the send data, the send data queue(s) are maximally filled with send data and the remaining data octets are discarded but may be re-sent by later application-socket sends.

3. Therefore, NO stream-type socket send data length should be "too long to pass through the underlying protocol" and cause the socket send to "fail ... [with] no data ... transmitted".

(B) It is typical, but NOT absolutely required, that a single application task ONLY send or request to send data to a stream-type socket.

(b) 'data_len' of 0 octets NOT allowed.

2. Only some socket send flag options are implemented. If other flag options are requested, the socket send handler function(s) abort, then return appropriate error codes so that the requested flag options are NOT silently ignored.
3. (a) Socket address structure 'sa_family' member MUST be configured in host-order and MUST NOT be converted to/from network-order.

   (b) Socket address structure addresses MUST be configured/converted from host-order to network-order.
4. (a) IEEE Std 1003.1, 2004 Edition, Section 'sendto() : RETURN VALUE' states that :
   1. "Upon successful completion, sendto() shall return the number of bytes sent."

      (A) Section 'sendto() : DESCRIPTION' elaborates that "successful completion of a call to sendto() does not guarantee delivery of the message."

      (B)
      1. Thus applications SHOULD verify the actual returned number of data octets transmitted and/or prepared for transmission.
      2. In addition, applications MAY desire verification of receipt and/or acknowledgement of transmitted data to the remote host, either inherently by the transport layer or explicitly by the application.
   2. (A) "Otherwise, -1 shall be returned" ...
      1. Section 'sendto() : DESCRIPTION' elaborates that "a return value of -1 indicates only locally-detected errors".

      (B) "and 'errno' set to indicate the error." 'errno' is NOT currently supported.

   (b) Although NO socket send() specification states to return '0' when the socket's connection is closed, it seems reasonable to return '0' since it is possible for the socket connection to be close()'d or shutdown() by the remote host.

setsockopt()

Description

Sets the socket option.

Files

`net_bsd.h/net_bsd.c`

Prototype

```
int  setsockopt (int        sock_id,
                 int        protocol,
                 int        opt_name,
                 void       *p_opt_val,
                 socklen_t   opt_len)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to set the option.

`protocol`

Protocol level at which the option resides.

`opt_name`

Name of the single socket option to set.

`p_opt_val`

Pointer to the socket option value to set.

`opt_len`

Option length.

Returned Value

- 0, if NO error(s).
- -1, otherwise.

Notes / Warnings

None.

socket()

Description

Creates a socket.

Files

`net_bsd.h/net_bsd.c`

Prototype

```
int socket (int protocol_family,
        int sock_type,
        int protocol)
```

Arguments

`protocol_family`

Socket `protocol` family :

- `PF_INET` Internet Protocol version 4 (IPv4).
- `PF_INET6` Internet Protocol version 6 (IPv6).

`sock_type`

Socket type :

- `SOCK_DGRAM` Datagram-type socket.
- `SOCK_STREAM` Stream -type socket.

`protocol`

Socket `protocol` :

- 0 Default protocol for socket type.
- `IPPROTO_UDP` User Datagram Protocol (UDP).
- `IPPROTO_TCP` Transmission Control Protocol (TCP).

Returned Value

- Socket descriptor/handle identifier, if NO error(s).
- -1, otherwise.

Notes / Warnings

None.

Network Application Interface API

Function Name	Description
NetApp_ClientDatagramOpen()	Open a UDP datagram using IPv4 or IPv6 address.
NetApp_ClientDatagramOpenByHostname()	Open a UDP datagram to a server using the server's hostname (select remote address using DNS)
NetApp_ClientStreamOpen()	Open and connect a TCP Stream to a server
NetApp_ClientStreamOpenByHostname()	Open and connect a TCP Stream to a server using the server's hostname (select remote address using DNS)
NetApp_SockOpen()	Open an application socket.
NetApp_SockClose()	Close an application socket.
NetApp_SockBind()	Bind an application socket to a local address.
NetApp_SockConn()	Connect an application socket to a remote address.
NetApp_SockListen()	Set an application socket to listen for connection requests.
NetApp_SockAccept()	Return a new application socket accepted from a listen application socket.
NetApp_SockRx()	Receive application data via socket.
NetApp_SockTx()	Transmit application data via socket.
NetApp_SetSockAddr()	Setup a socket address from an IPv4 or an IPv6 address.
NetApp_TimeDly_ms()	Delay for specified time, in milliseconds.

NetApp_ClientDatagramOpen()

Description

1. Connects a client to a server using an IP address (IPv4 or IPv6) with a datagram socket by following these steps :
   (a) Open a datagram socket.
   (b) Set connection timeout.

Files

`net_app.h/net_app.c`

Prototype

```
NET_SOCK_ID  NetApp_ClientDatagramOpen (CPU_INT08U         *p_addr,
                                        NET_IP_ADDR_FAMILY   addr_family,
                                        NET_PORT_NBR        remote_port_nbr,
                                        NET_SOCK_ADDR       *p_sock_addr,
                                        RTOS_ERR           *p_err)
```

Arguments

`p_addr`

Pointer to IP address.

`addr_family`

IP family of the address.

`remote_port_nbr`

Port of the remote host.

`p_sock_addr`

Pointer to a variable that receives the socket address of the remote host.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_POOL_EMPTY`

Returned Value

- Socket ID, if no error(s).
- `NET_SOCK_ID_NONE` , otherwise.

Notes / Warnings

None.

NetApp_ClientDatagramOpenByHostname()

Description

1. Opens a datagram type (UDP) socket to the server using its host name. (select IP address automatically, see Note #2): (a) Get the IP address of the remote host from a string that contains either the IP address or the host name that will be resolved using DNS (see Note #2). (b) Open a datagram socket.

Files

`net_app.h/net_app.c`

Prototype

```
NET_IP_ADDR_FAMILY  NetApp_ClientDatagramOpenByHostname (NET_SOCK_ID        *p_sock_id,
                                                         CPU_CHAR           *p_remote_host_name,
                                                         NET_PORT_NBR        remote_port_nbr,
                                                         NET_IP_ADDR_FAMILY  ip_family,
                                                         NET_SOCK_ADDR      *p_sock_addr,
                                                         CPU_BOOLEAN        *p_is_hostname,
                                                         RTOS_ERR           *p_err)
```

Arguments

`p_sock_id`

Pointer to a variable that receives the socket ID opened from this function.

`p_remote_host_name`

Pointer to a string that contains the remote host name to resolve.

`remote_port_nbr`

Port of the remote host.

`ip_family`

Select IP family of addresses returned by DNS resolution :

- `NET_IP_ADDR_FAMILY_IPv4`
- `NET_IP_ADDR_FAMILY_IPv6`

`p_sock_addr`

Pointer to a variable that receives the socket address of the remote host.

`p_is_hostname`

Pointer to variable that receives the boolean to indicate if the string passed in `p_remote_host_name` was a hostname or a IP address.

- `DEF_YES` , hostname was received.
- `DEF_NO` , otherwise.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_NET_ADDR_UNRESOLVED`
- `RTOS_ERR_NET_STR_ADDR_INVALID`

Returned Value

- `NET_IP_ADDR_FAMILY_IPv4` , if the opening was successful using an IPv4 address.
- `NET_IP_ADDR_FAMILY_IPv6` , if the opening was successful using an IPv6 address.
- `NET_IP_ADDR_FAMILY_UNKNOWN` , otherwise.

Notes / Warnings

1. When a host name is passed into the remote host name parameter, this function tries to resolve the address of the remote host using DNS.
   (a) The DNS must be present and enabled in the project for the resolve to be possible.
   (b) The `ip_family` argument let the application choose the IP family of the addresses returned by the DNS resolution.
   (c) This function always blocks and the fail timeout depends on the DNS resolution timeout, the number of remote addresses found, and the connection timeout parameter.

NetApp_ClientStreamOpen()

Description

1. Connects a client to a server using an IP address (IPv4 or IPv6) with a stream socket by following these steps :
   (a) Open a stream socket.
   (b) Set Security parameter (TLS/SSL), if required.
   (c) Set connection timeout.
   (d) Connect to the remote host.

Files

`net_app.h/net_app.c`

Prototype

```
NET_SOCK_ID  NetApp_ClientStreamOpen (CPU_INT08U             *p_addr,
                          NET_IP_ADDR_FAMILY       addr_family,
                          NET_PORT_NBR            remote_port_nbr,
                          NET_SOCK_ADDR          *p_sock_addr,
                          NET_APP_SOCK_SECURE_CFG  *p_secure_cfg,
                          CPU_INT32U             req_timeout_ms,
                          RTOS_ERR              *p_err)
```

Arguments

`p_addr`

Pointer to IP address.

`addr_family`

IP family of the address.

`remote_port_nbr`

Port of the remote host.

`p_sock_addr`

Pointer to a variable that receives the socket address of the remote host.

`p_secure_cfg`

Pointer to the secure configuration (TLS/SSL), if needed.

`req_timeout_ms`

Connection timeout in milliseconds.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_TYPE`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_NOT_SUPPORTED`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_FAIL`
- `RTOS_ERR_NET_INVALID_ADDR_SRC`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_NET_IF_LINK_DOWN`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_NET_OP_IN_PROGRESS`
- `RTOS_ERR_TX`
- `RTOS_ERR_ALREADY_EXISTS`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_NET_INVALID_CONN`
- `RTOS_ERR_RX`

- RTOS_ERR_NOT_READY
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_NET_NEXT_HOP
- RTOS_ERR_NET_CONN_CLOSED_FAULT

Returned Value

- Socket ID, if no error(s).
- NET_SOCK_ID_NONE , otherwise.

Notes / Warnings

This function is in blocking mode, which means that at the end of the function, the socket will have succeeded or failed to connect to the remote host.

NetApp_ClientStreamOpenByHostname()

Description

1. Connects a client to a server using its host name with a stream (TCP) socket (select IP address automatically, see Note #2) by following these steps :
   (a) Get the IP address of the remote host from a string that contains either the IP address or the host name that will be resolved using DNS (see Note #2).
   (b) Open a stream socket.
   (c) Connect a stream socket.

Files

net_app.h/net_app.c

Prototype

```
NET_IP_ADDR_FAMILY  NetApp_ClientStreamOpenByHostname (NET_SOCK_ID            *p_sock_id,
                                                       CPU_CHAR               *p_remote_host_name,
                                                       NET_PORT_NBR            remote_port_nbr,
                                                       NET_SOCK_ADDR          *p_sock_addr,
                                                       NET_APP_SOCK_SECURE_CFG *p_secure_cfg,
                                                       CPU_INT32U              req_timeout_ms,
                                                       RTOS_ERR               *p_err)
```

Arguments

p_sock_id

Pointer to a variable that receives the socket ID opened from this function.

p_remote_host_name

Pointer to a string that contains the remote host name to resolve.

remote_port_nbr

Port of the remote host.

p_sock_addr

Pointer to a variable that receives the socket address of the remote host.

`p_secure_cfg`

Pointer to the secure configuration (TLS/SSL), if needed.

`req_timeout_ms`

Connection timeout in ms.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_TYPE`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_NOT_SUPPORTED`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_FAIL`
- `RTOS_ERR_NET_INVALID_ADDR_SRC`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_NET_IF_LINK_DOWN`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_NET_OP_IN_PROGRESS`
- `RTOS_ERR_TX`
- `RTOS_ERR_ALREADY_EXISTS`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_NET_INVALID_CONN`
- `RTOS_ERR_RX`
- `RTOS_ERR_NOT_READY`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_NET_ADDR_UNRESOLVED`
- `RTOS_ERR_NET_NEXT_HOP`
- `RTOS_ERR_NET_CONN_CLOSED_FAULT`

Returned Value

- `NET_IP_ADDR_FAMILY_IPv4` , if connected successfully using an IPv4 address.
- `NET_IP_ADDR_FAMILY_IPv6` , if connected successfully using an IPv6 address.
- `NET_IP_ADDR_FAMILY_UNKNOWN` , otherwise.

Notes / Warnings

1. When a host name is passed to the remote host name parameter, this function tries to resolve the address of the remote host using DNS.
   (a) The DNS must be present and enabled in the project for the resolve to be possible.
   (b) If an IPv6 and an IPv4 address are found for the remote host. This function will first try to connect to the remote host using the IPv6 address. If the connection fails using IPv6, a connection retry will occur using the IPv4 address.
   (c) This function always blocks and the fail timeout depends on the DNS resolution timeout, the number of remote addresses found, and the connection timeout parameter.

2. This function is in blocking mode, which means that at the end of the function, the socket will have succeeded or failed to connect to the remote host.

NetApp_SetSockAddr()

Description

Sets up a socket address from an IPv4 or an IPv6 address.

Files

`net_app.h/net_app.c`

Prototype

```
void  NetApp_SetSockAddr (NET_SOCK_ADDR        *p_sock_addr,
                          NET_SOCK_ADDR_FAMILY  addr_family,
                          NET_PORT_NBR          port_nbr,
                          CPU_INT08U           *p_addr,
                          NET_IP_ADDR_LEN       addr_len,
                          RTOS_ERR             *p_err)
```

Arguments

`p_sock_addr`

Pointer to the socket address that will be configure by this function.

`addr_family`

IP address family to configure, possible values:

- `NET_SOCK_ADDR_FAMILY_IP_V4`
- `NET_SOCK_ADDR_FAMILY_IP_V6`

`port_nbr`

Port number.

`p_addr`

Pointer to IP address to use.

`addr_len`

Length of the IP address to use.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

None.

NetApp_SockAccept()

Description

1. Returns a new application socket accepted from a listen application socket with error handling :
   (a) Configure the accept timeout, if any
   (b) Wait for the accept socket
   (c) Restore the accept timeout, if necessary

Files

`net_app.h/net_app.c`

Prototype

```
NET_SOCK_ID  NetApp_SockAccept (NET_SOCK_ID        sock_id,
                                NET_SOCK_ADDR     *p_addr_remote,
                                NET_SOCK_ADDR_LEN *p_addr_len,
                                CPU_INT16U         retry_max,
                                CPU_INT32U         timeout_ms,
                                CPU_INT32U         time_dly_ms,
                                RTOS_ERR          *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of listen socket.

`p_addr_remote`

Pointer to an address buffer that receives the socket address structure.

`p_addr_len`

Pointer to a variable to pass the size of the socket address structure and that will receive the size of the accepted connection’s socket address structure, if no errors. Otherwise, a 0 size will be returned.

`retry_max`

Maximum number of consecutive socket accept retries.

`timeout_ms`

Socket accept timeout value per attempt/retry.

`time_dly_ms`

Transitory socket accept delay value (in milliseconds).

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function::

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_TYPE`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_NET_INVALID_CONN`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_POOL_EMPTY`

- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_INVALID_STATE
- RTOS_ERR_NET_CONN_CLOSED_FAULT
- RTOS_ERR_ABORT
- RTOS_ERR_TIMEOUT

Returned Value

- Socket descriptor/handle identifier of new accepted socket, if NO error(s).
- NET_SOCK_BSD_ERR_ACCEPT , otherwise.

Notes / Warnings

1. Socket arguments and/or operations are validated in network socket handler functions. Some arguments validated only if the validation code is enabled (i.e., RTOS_ARG_CHK_EXT_EN is DEF_ENABLED in ' rtos_cfg .h').
2. Socket accept operation valid for stream-type sockets only.
3. (a) Socket address structure 'AddrFamily' member returned in host-order and SHOULD NOT be converted to network-order.
   (b) Socket address structure addresses returned in network-order and SHOULD be converted from network-order to host-order.
4. If a non-zero number of retries is requested and the global socket blocking (' NET_SOCK_CFG_BLOCK_SEL ') is configured for non-blocking operation (' NET_SOCK_BLOCK_SEL_NO_BLOCK '), then one or more of the following SHOULD also be requested. Otherwise, all retries will likely fail immediately since no time will elapse to wait for and allow socket operation(s) to successfully complete :
   (a) One or more of the following SHOULD also be requested. Otherwise, all retries will likely fail immediately since no time will have elapsed to allow socket operation(s) to successfully complete :
      1. A non-zero timeout
      2. A non-zero time delay(s).

NetApp_SockBind()

Description

Binds an application socket to a local address with error handling.

Files

net_app.h/net_app.c

Prototype

```
CPU_BOOLEAN  NetApp_SockBind (NET_SOCK_ID        sock_id,
                              NET_SOCK_ADDR      *p_addr_local,
                              NET_SOCK_ADDR_LEN   addr_len,
                              CPU_INT16U          retry_max,
                              CPU_INT32U          time_dly_ms,
                              RTOS_ERR           *p_err)
```

Arguments

sock_id

Socket descriptor/handle identifier of the application to which to bind the socket.

p_addr_local

Pointer to socket address structure.

`addr_len`

Length of socket address structure (in octets).

`retry_max`

Maximum number of consecutive socket bind retries).

`time_dly_ms`

Transitory socket bind delay value, in milliseconds.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_TYPE`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_ALREADY_EXISTS`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_NET_INVALID_CONN`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_NET_CONN_CLOSED_FAULT`

Returned Value

- `DEF_OK` , application socket is successfully bound to a local address.
- `DEF_FAIL` , otherwise.

Notes / Warnings

1. Socket arguments and/or operations are validated in network socket handler functions. Some arguments are validated only if the validation code is enabled (i.e., `RTOS_ARG_CHK_EXT_EN` is `DEF_ENABLED` in ' `rtos_cfg` .h').
2. (a) Socket address structure 'AddrFamily' member MUST be configured in host-order and MUST NOT be converted to/from network-order.
   (b) Socket address structure addresses MUST be configured/converted from host-order to network-order.
3. If a non-zero number of retries is requested, a non-zero time delay SHOULD also be requested. Otherwise, all retries will most likely fail immediately since no time will have elapsed to allow socket operation(s) to successfully complete.

NetApp_SockClose()

Description

1. Closes an application socket, with error handling :
   (a) Configure close timeout, if any
   (b) Close application socket

Files

`net_app.h/net_app.c`

Prototype

```
CPU_BOOLEAN NetApp_SockClose (NET_SOCK_ID   sock_id,
                              CPU_INT32U    timeout_ms,
                              RTOS_ERR     *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of application socket to close.

`timeout_ms`

Socket close timeout value.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_TYPE`
- `RTOS_ERR_NET_RETRY_MAX`
- `RTOS_ERR_NET_SOCK_CLOSED`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_NOT_SUPPORTED`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_NET_INVALID_ADDR_SRC`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_NET_IF_LINK_DOWN`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_NET_OP_IN_PROGRESS`
- `RTOS_ERR_TX`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_NET_INVALID_CONN`
- `RTOS_ERR_RX`
- `RTOS_ERR_NOT_READY`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_NET_NEXT_HOP`

Returned Value

- `DEF_OK` , application socket successfully closed.
- `DEF_FAIL` , otherwise.

Notes / Warnings

1. Socket arguments and/or operations are validated in network socket handler functions. Some arguments are validated only if the validation code is enabled (i.e., `RTOS_ARG_CHK_EXT_EN` is `DEF_ENABLED` in ' `net_cfg` .h').
2. (a) Once an application closes its socket, NO further operations on the socket are allowed and the application MUST NOT continue to access the socket.
   (b) NO error is returned for any internal error while closing the socket.

NetApp_SockConn()

Description

1. Connects an application socket to a remote address with error handling :
   (a) Configure connect timeout, if any
   (b) Connect application socket to remote address
   (c) Restore connect timeout, if necessary

Files

`net_app.h/net_app.c`

Prototype

```
CPU_BOOLEAN  NetApp_SockConn (NET_SOCK_ID        sock_id,
                              NET_SOCK_ADDR      *p_addr_remote,
                              NET_SOCK_ADDR_LEN   addr_len,
                              CPU_INT16U          retry_max,
                              CPU_INT32U          timeout_ms,
                              CPU_INT32U          time_dly_ms,
                              RTOS_ERR           *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of application socket to which to connect.

`p_addr_remote`

Pointer to socket address structure.

`addr_len`

Length of socket address structure (in octets).

`retry_max`

Maximum number of consecutive socket connect retries.

`timeout_ms`

Socket connect timeout value per attempt/retry.

`time_dly_ms`

Transitory socket connect delay value (in milliseconds).

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_TYPE`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_NOT_SUPPORTED`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_FAIL`

- RTOS_ERR_NET_INVALID_ADDR_SRC
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_NET_IF_LINK_DOWN
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_INVALID_STATE
- RTOS_ERR_ABORT
- RTOS_ERR_TIMEOUT
- RTOS_ERR_NET_OP_IN_PROGRESS
- RTOS_ERR_TX
- RTOS_ERR_ALREADY_EXISTS
- RTOS_ERR_NOT_FOUND
- RTOS_ERR_NET_INVALID_CONN
- RTOS_ERR_RX
- RTOS_ERR_NOT_READY
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_NET_NEXT_HOP
- RTOS_ERR_NET_CONN_CLOSED_FAULT

Returned Value

- DEF_OK , application socket has successfully connected to a remote address.
- DEF_FAIL , otherwise.

Notes / Warnings

1. Socket arguments and/or operations are validated in network socket handler functions. Some arguments are validated only if the validation code is enabled (i.e., RTOS_ARG_CHK_EXT_EN is DEF_ENABLED in ' rtos_cfg .h').
2. (a) Socket address structure 'AddrFamily' member MUST be configured in host-order and MUST NOT be converted to/from network-order.
   (b) Socket address structure addresses MUST be configured/converted from host-order to network-order.
3. (a)
   1. If a non-zero number of retries is requested AND ...
   2. global socket blocking (' NET_SOCK_CFG_BLOCK_SEL ') is configured ... for non-blocking operation (' NET_SOCK_BLOCK_SEL_NO_BLOCK '); ...

   (b) ... one or more of the following SHOULD also be requested. Otherwise, all retries will most likely fail immediately since no time will have elapsed to allow socket operation(s) to successfully complete :
   1. A non-zero timeout
   2. A non-zero time delay

NetApp_SockListen()

Description

Sets an application socket to listen for connection requests with error handling.

Files

net_app.h/net_app.c

Prototype

```
CPU_BOOLEAN  NetApp_SockListen (NET_SOCK_ID       sock_id,
                                NET_SOCK_Q_SIZE   sock_q_size,
                                RTOS_ERR         *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to listen.

`sock_q_size`

Maximum number of connection requests to accept and queue on listen socket.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_TYPE`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_NET_INVALID_CONN`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_NET_CONN_CLOSED_FAULT`

Returned Value

- `DEF_OK` , application socket is successfully set to listen.
- `DEF_FAIL` , otherwise.

Notes / Warnings

1. Socket arguments and/or operations are validated in network socket handler functions. Some arguments are validated only if the validation code is enabled (i.e., `RTOS_ARG_CHK_EXT_EN` is `DEF_ENABLED` in '`net_cfg` .h').
2. Socket listen operation is only valid for stream-type sockets.

NetApp_SockOpen()

Description

Opens an application socket, with error handling.

Files

`net_app.h/net_app.c`

Prototype

```
NET_SOCK_ID  NetApp_SockOpen (NET_SOCK_PROTOCOL_FAMILY   protocol_family,
                              NET_SOCK_TYPE              sock_type,
                              NET_SOCK_PROTOCOL          protocol,
                              CPU_INT16U                 retry_max,
                              CPU_INT32U                 time_dly_ms,
                              RTOS_ERR                  *p_err)
```

Arguments

`protocol_family`

Socket protocol family :

- `NET_SOCK_PROTOCOL_FAMILY_IP_V4`

- `NET_SOCK_PROTOCOL_FAMILY_IP_V6`

`sock_type`

Socket type :

- `NET_SOCK_TYPE_DATAGRAM`
- `NET_SOCK_TYPE_STREAM`

`protocol`

Socket protocol :

- `NET_SOCK_PROTOCOL_TCP`
- `NET_SOCK_PROTOCOL_UDP`

`retry_max`

Maximum number of consecutive socket open retries.

`time_dly_ms`

Transitory socket open delay value (in milliseconds).

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_POOL_EMPTY`

Returned Value

- Socket descriptor/handle identifier, if NO error(s).
- `NET_SOCK_BSD_ERR_OPEN` , otherwise.

Notes / Warnings

1. Socket arguments and/or operations are validated in network socket handler functions. Some arguments validated only if the validation code is enabled (i.e., `RTOS_ARG_CHK_EXT_EN` is `DEF_ENABLED` in '`rtos_cfg` .h').
2. If a non-zero number of retries is requested, a non-zero time delay SHOULD also be requested. Otherwise, all retries will most likely fail immediately since no time will have elapsed to allow socket operation(s) to successfully complete.

NetApp_SockRx()

Description

1. Receive application data via socket, with error handling :
   (a) Validate receive arguments
   (b) Configure receive timeout, if any
   (c) Receive application data via socket
   (d) Restore receive timeout, if necessary

Files

`net_app.h/net_app.c`

Prototype

```
CPU_INT16U  NetApp_SockRx (NET_SOCK_ID          sock_id,
                           void                *p_data_buf,
                           CPU_INT16U           data_buf_len,
                           CPU_INT16U           data_rx_th,
                           NET_SOCK_API_FLAGS   flags,
                           NET_SOCK_ADDR       *p_addr_remote,
                           NET_SOCK_ADDR_LEN   *p_addr_len,
                           CPU_INT16U           retry_max,
                           CPU_INT32U           timeout_ms,
                           CPU_INT32U           time_dly_ms,
                           RTOS_ERR            *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to receive application data.

`p_data_buf`

Pointer to an application data buffer that will receive application data.

`data_buf_len`

Size of the application data buffer (in octets).

`data_rx_th`

Application data receive threshold :

- 0, NO minimum receive threshold; i.e., receive ANY amount of data. Recommended for datagram sockets.
- Minimum amount of application data to receive (in octets) within maximum number of retries, otherwise.

`flags`

Flags to select receive options; bit-field flags logically OR'd :

- `NET_SOCK_FLAG_NONE`
- `NET_SOCK_FLAG_RX_DATA_PEEK`
- `NET_SOCK_FLAG_RX_NO_BLOCK`

`p_addr_remote`

Pointer to an address buffer that receives the socket address structure.

`p_addr_len`

Pointer to a variable to pass the size of the socket address structure and that will receive the size of the accepted connection's socket address structure, if no errors. Otherwise, a 0 size will be returned.

`retry_max`

Maximum number of consecutive socket receive retries.

`timeout_ms`

Socket receive timeout value per attempt/retry.

`time_dly_ms`

Transitory socket receive delay value, in milliseconds.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_NET_INVALID_CONN`
- `RTOS_ERR_NET_CONN_CLOSE_RX`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_NET_CONN_CLOSED_FAULT`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

- Number of positive data octets received, if NO error(s).
- 0, otherwise.

Notes / Warnings

1. Socket arguments and/or operations are validated in network socket handler functions. Some arguments are validated only if the validation code is enabled (i.e., RTOS_ARG_CHK_EXT_EN is DEF_ENABLED in 'rtos_cfg.h').
2. (a) Socket address structure 'AddrFamily' member returned in host-order and SHOULD NOT be converted to network-order.
   (b) Socket address structure addresses returned in network-order and SHOULD be converted from network-order to host-order.
3. (a)
    1. (A) Datagram-type sockets transmit and receive all data atomically -- i.e., every single, complete datagram transmitted MUST be received as a single, complete datagram.
       (B) IEEE Std 1003.1, 2004 Edition, Section 'recvfrom() : DESCRIPTION' summarizes that "for message-based sockets, such as ... SOCK_DGRAM ... the entire message shall be read in a single operation. If a message is too long to fit in the supplied buffer, and MSG_PEEK is not set in the flags argument, the excess bytes shall be discarded."
    2. If the socket's type is datagram and the receive data buffer size is NOT large enough for the received data, the receive data buffer is maximally filled with receive data but the remaining data octets are discarded and RTOS_ERR_WOULD_OVF error is returned.

   (b)
    1. (A) Stream-type sockets transmit and receive all data octets in one or more non-distinct packets. In other words, the application data is NOT bounded by any specific packet
       (s); rather, it is contiguous and sequenced from one packet to the next.
       (B) IEEE Std 1003.1, 2004 Edition, Section 'recv() : DESCRIPTION' summarizes that "for stream-based sockets, such as SOCK_STREAM, message boundaries shall be ignored. In this case, data shall be returned to the user as soon as it becomes available, and no data shall be discarded."
    2. If the socket's type is stream and the receive data buffer size is NOT large enough for the received data, the receive data buffer is filled with received data and the remaining data octets remain queued for subsequent application reception.
4. (a)
    1. If a non-zero number of retries is requested and one of the following conditions:
    2. (A) global socket blocking ('NET_SOCK_CFG_BLOCK_SEL') is configured for non-blocking operation ('NET_SOCK_BLOCK_SEL_NO_BLOCK'), OR
       (B) socket 'flags' argument set to 'NET_SOCK_FLAG_RX_BLOCK'.

   (b) One or more of the following SHOULD also be requested. Otherwise, all retries will most likely fail immediately since no time will have elapsed to allow the socket operation(s) to successfully complete :
    1. A non-zero timeout
    2. A non-zero time delay(s).

NetApp_SockTx()

Description

1. Transmits application data via socket with error handling :
   (a) Configure transmit timeout, if any
   (b) Transmit application data via socket
   (c) Restore transmit timeout, if necessary

Files

`net_app.h/net_app.c`

Prototype

```
CPU_INT16U  NetApp_SockTx (NET_SOCK_ID         sock_id,
                           void               *p_data,
                           CPU_INT16U          data_len,
                           NET_SOCK_API_FLAGS  flags,
                           NET_SOCK_ADDR      *p_addr_remote,
                           NET_SOCK_ADDR_LEN   addr_len,
                           CPU_INT16U          retry_max,
                           CPU_INT32U          timeout_ms,
                           CPU_INT32U          time_dly_ms,
                           RTOS_ERR           *p_err)
```

Arguments

`sock_id`

Socket descriptor/handle identifier of socket to transmit application data.

`p_data`

Pointer to application data to transmit.

`data_len`

Length of application data to transmit (in octets).

`flags`

Flags to select transmit options; bit-field flags logically OR'd :

- `NET_SOCK_FLAG_NONE`
- `NET_SOCK_FLAG_TX_NO_BLOCK`

`p_addr_remote`

Pointer to destination address buffer.

`addr_len`

Length of destination address buffer (in octets).

`retry_max`

Maximum number of consecutive socket transmit retries.

`timeout_ms`

Socket transmit timeout value per attempt/retry.

time_dly_ms

Transitory socket transmit delay value (in milliseconds).

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_TYPE
- RTOS_ERR_NET_RETRY_MAX
- RTOS_ERR_BLK_ALLOC_CALLBACK
- RTOS_ERR_NOT_SUPPORTED
- RTOS_ERR_SEG_OVF
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_FAIL
- RTOS_ERR_NET_INVALID_ADDR_SRC
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_NET_IF_LINK_DOWN
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_INVALID_STATE
- RTOS_ERR_ABORT
- RTOS_ERR_TIMEOUT
- RTOS_ERR_TX
- RTOS_ERR_NOT_FOUND
- RTOS_ERR_ALREADY_EXISTS
- RTOS_ERR_NET_INVALID_CONN
- RTOS_ERR_RX
- RTOS_ERR_NOT_READY
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_NET_NEXT_HOP
- RTOS_ERR_NET_CONN_CLOSED_FAULT

Returned Value

- Number of positive data octets transmitted, if NO error(s).
- 0, otherwise.

Notes / Warnings

1. Socket arguments and/or operations are validated in network socket handler functions. Some arguments are validated only if the validation code is enabled (i.e., RTOS_ARG_CHK_EXT_EN is DEF_ENABLED in 'rtos_cfg.h').
2. (a) Socket address structure 'AddrFamily' member MUST be configured in host-order and MUST NOT be converted to/from network-order.
   (b) Socket address structure addresses MUST be configured/converted from host-order to network-order.
3. If a non-zero number of retries is requested and the global socket blocking ('NET_SOCK_CFG_BLOCK_SEL') is configured for non-blocking operation ('NET_SOCK_BLOCK_SEL_NO_BLOCK'), then one or more of the following SHOULD also be requested. Otherwise, all retries will most likely fail immediately since no time will elapse to wait for & allow socket operation(s) to successfully complete :
   (a) A non-zero timeout
   (b) A non-zero time delay
4. Datagram sockets NOT currently blocked during transmit and therefore require NO transmit timeout.

NetApp_TimeDly_ms()

Description

Delays for a specified time (in milliseconds).

Files

`net_app.h/net_app.c`

Prototype

```
void  NetApp_TimeDly_ms (CPU_INT32U   time_dly_ms,
                         RTOS_ERR    *p_err)
```

Arguments

`time_dly_ms`

Time delay value (in milliseconds).

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

1. (a) Time delay of 0 milliseconds allowed.
   (b) Time delay limited to the maximum possible OS time delay if it is greater than the maximum possible OS time delay.

HTTP Client API

HTTP Client API

- HTTPc_ConfigureConnParam()
- HTTPc_ConfigureMemSeg()
- HTTPc_ConfigureTaskStk()
- HTTPc_ConfigureQty()
- HTTPc_Init()
- HTTPc_TaskPrioSet()
- HTTPc_TaskDlySet()
- HTTPc_ConnClose()
- HTTPc_ConnClr()
- HTTPc_ConnSetParam()
- HTTPc_ConnOpen()
- HTTPc_ReqClr()
- HTTPc_ReqSetParam()
- HTTPc_ReqSend()
- HTTPc_FormAppFmt()
- HTTPc_FormMultipartFmt()
- HTTPc_FormAddKeyVal()
- HTTPc_FormAddKeyValExt()
- HTTPc_FormAddFile()
- HTTPc_WebSockSetParam()
- HTTPc_WebSockMsgSetParam()
- HTTPc_WebSockUpgrade()
- HTTPc_WebSockSend()
- HTTPc_WebSockClr()
- HTTPc_WebSockMsgClr()
- HTTPc_WebSockFmtCloseMsg()
- HTTP Client Hook Functions

HTTPc_ConfigureConnParam()

Description

Overwrite the Connection configuration object for HTTP client.

Files

`http_client.h/http_client.c`

Prototype

```
void  HTTPc_ConfigureConnParam (HTTPc_CONN_CFG  *p_conn_cfg)
```

Arguments

`p_conn_cfg`

Pointer to structure containing the connection parameters.

Returned Value

None.

Notes / Warnings

None.

HTTPc_ConfigureMemSeg()

Description

Configure the memory segment that will be used to allocate internal data needed by HTTP client module instead of the default memory segment.

Files

`http_client.h/http_client.c`

Prototype

```
void  HTTPc_ConfigureMemSeg (MEM_SEG  *p_mem_seg)
```

Arguments

`p_mem_seg`

Pointer to the memory segment from which the internal data will be allocated. If `DEF_NULL`, the internal data will be allocated from the global Heap.

Returned Value

None.

Notes / Warnings

1. This function is optional. If it is called, it must be called before `HTTPc_Init()`. If it is not called, default values will be used to initialize the module.

HTTPc_ConfigureTaskStk()

Description

Configure the HTTP client task stack properties to use the parameters contained in the passed structure instead of the default parameters.

Files

`http_client.h/http_client.c`

Prototype

```
void  HTTPc_ConfigureTaskStk (CPU_STK_SIZE   stk_size_elements,
                              void          *p_stk_base)
```

Arguments

`stk_size_elements`

Size of the stack, in `CPU_STK` elements.

`p_stk_base`

Pointer to base of the stack.

Returned Value

None.

Notes / Warnings

1. This function is optional. If it is called, it must be called before `HTTPc_Init()`. If it is not called, default values will be used to initialize the module.

HTTPc_ConfigureQty()

Description

Overwrite the Quantity configuration object for HTTP client.

Files

`http_client.h/http_client.c`

Prototype

```
void HTTPc_ConfigureQty(HTTPc_QTY_CFG *p_qty_cfg)
```

Arguments

`p_qty_cfg`

Pointer to a structure containing the quantity parameters.

Returned Value

None.

Notes / Warnings

None.

HTTPc_Init()

Description

1. Initializes the HTTP Client Suite by following these steps :
   (a) Validate Configuration.
   (b) Create HTTP Client Task, if necessary.

Files

`http_client.h/http_client.c`

Prototype

```
void HTTPc_Init(RTOS_ERR *p_err)
```

Arguments

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_ALREADY_INIT`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`

Returned Value

None.

Notes / Warnings

None.

HTTPc_TaskPrioSet()

Description

Sets priority of the HTTP client task.

Files

`http_client.h/http_client.c`

Prototype

```
void  HTTPc_TaskPrioSet (RTOS_TASK_PRIO   prio,
                         RTOS_ERR        *p_err)
```

Arguments

`prio`

New priority for the HTTP client task.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_ARG`

Returned Value

None.

Notes / Warnings

None.

HTTPc_TaskDlySet()

Description

Sets delay of the HTTP client task.

Files

`http_client.h/http_client.c`

Prototype

```
void HTTPc_TaskDlySet (CPU_INT08U   dly_ms,
                       RTOS_ERR    *p_err)
```

Arguments

`dly_ms`

New delay in millisecond for the HTTP client task.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function.

Returned Value

None.

Notes / Warnings

None.

HTTPc_ConnClose()

Description

Closes a persistent HTTP client connection.

Files

`http_client.h/http_client.c`

Prototype

```
void  HTTPc_ConnClose (HTTPc_CONN_OBJ  *p_conn_obj,
                       HTTPc_FLAGS      flags,
                       RTOS_ERR        *p_err)
```

Arguments

`p_conn_obj`

Pointer to the HTTPc Connection to close.

`flags`

Configuration flags :

- `HTTPc_FLAG_CONN_NO_BLOCK`

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_INIT`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_NO_MORE_RSRC`
- `RTOS_ERR_INVALID_TYPE`
- `RTOS_ERR_NET_RETRY_MAX`
- `RTOS_ERR_NET_SOCK_CLOSED`
- `RTOS_ERR_NOT_SUPPORTED`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NET_INVALID_ADDR_SRC`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_NET_IF_LINK_DOWN`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

- RTOS_ERR_TX
- RTOS_ERR_NOT_FOUND
- RTOS_ERR_NET_INVALID_CONN
- RTOS_ERR_RX
- RTOS_ERR_NOT_READY
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_NET_NEXT_HOP

Returned Value

None.

Notes / Warnings

None.

HTTPc_ConnClr()

Description

Clears an HTTP client connection object before the first usage.

Files

http_client.h/http_client.c

Prototype

```
void  HTTPc_ConnClr (HTTPc_CONN_OBJ   *p_conn_obj,
                     RTOS_ERR         *p_err)
```

Arguments

p_conn_obj

Pointer to the current HTTPc Connection object.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_NOT_INIT

Returned Value

None.

Notes / Warnings

1. This function MUST be called before the HTTPc_CONN object is used for the first time.

HTTPc_ConnSetParam()

Description

Sets the parameters related to the HTTP Client Connection.

Files

http_client.h/http_client.c

Prototype

```
void  HTTPc_ConnSetParam (HTTPc_CONN_OBJ    *p_conn_obj,
                          HTTPc_PARAM_TYPE   type,
                          void              *p_param,
                          RTOS_ERR          *p_err)
```

Arguments

`p_conn`

Pointer to the current HTTPc Connection object.

`type`

Parameter type :

- `HTTPc_PARAM_TYPE_SERVER_PORT`
- `HTTPc_PARAM_TYPE_PERSISTENT`
- `HTTPc_PARAM_TYPE_CONNECT_TIMEOUT`
- `HTTPc_PARAM_TYPE_INACTIVITY_TIMEOUT`
- `HTTPc_PARAM_TYPE_SECURE_COMMON_NAME`
- `HTTPc_PARAM_TYPE_SECURE_TRUST_CALLBACK`
- `HTTPc_PARAM_TYPE_CONN_CONNECT_CALLBACK`
- `HTTPc_PARAM_TYPE_CONN_CLOSE_CALLBACK`

`p_param`

Pointer to the parameter.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_INIT`
- `RTOS_ERR_OWNERSHIP`

Returned Value

None

Notes / Warnings

None.

HTTPc_ConnOpen()

Description

Opens a new HTTP connection.

Files

`http_client.h/http_client.c`

Prototype

```
CPU_BOOLEAN  HTTPc_ConnOpen (HTTPc_CONN_OBJ    *p_conn_obj,
                             CPU_CHAR          *p_buf,
                             CPU_INT16U         buf_len,
                             CPU_CHAR          *p_hostname_str,
                             CPU_INT16U         hostname_str_len,
                             HTTPc_FLAGS        flags,
                             RTOS_ERR          *p_err)
```

Arguments

`p_conn_obj`

Pointer to the HTTPc Connection object to open.

`p_buf`

Pointer to the HTTP buffer that sends (TX) and receives (RX) data.

`buf_len`

Length of the HTTP buffer.

`p_hostname_str`

Pointer to the hostname string.

`hostname_str_len`

Length of the hostname string.

`flags`

Configuration flags :

- `HTTPc_FLAG_CONN_NO_BLOCK`

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_INIT`
- `RTOS_ERR_OWNERSHIP`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`
- `RTOS_ERR_NO_MORE_RSRC`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_INVALID_TYPE`
- `RTOS_ERR_NOT_SUPPORTED`
- `RTOS_ERR_FAIL`
- `RTOS_ERR_NET_INVALID_ADDR_SRC`
- `RTOS_ERR_NET_IF_LINK_DOWN`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_NET_OP_IN_PROGRESS`
- `RTOS_ERR_TX`
- `RTOS_ERR_ALREADY_EXISTS`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_NET_INVALID_CONN`
- `RTOS_ERR_RX`
- `RTOS_ERR_NOT_READY`
- `RTOS_ERR_NET_ADDR_UNRESOLVED`

- RTOS_ERR_NET_NEXT_HOP
- RTOS_ERR_NET_CONN_CLOSED_FAULT

Returned Value

- DEF_OK , if the connection opening has successfully completed.
- DEF_FAIL , otherwise.

Notes / Warnings

None.

HTTPc_ReqClr()

Description

Clears the Request object members.

Files

http_client.h/http_client.c

Prototype

```
void  HTTPc_ReqClr (HTTPc_REQ_OBJ  *p_req_obj,
                    RTOS_ERR       *p_err)
```

Arguments

p_req_obj

Pointer to the request object to clear.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function :

- RTOS_ERR_NONE
- RTOS_ERR_NOT_INIT

Returned Value

None.

Notes / Warnings

None.

HTTPc_ReqSetParam()

Description

Sets a parameter related to a given HTTP Request.

Files

http_client.h/http_client.c

Prototype

```
void  HTTPc_ReqSetParam (HTTPc_REQ_OBJ     *p_req_obj,
                         HTTPc_PARAM_TYPE   type,
                         void              *p_param,
                         RTOS_ERR          *p_err)
```

Arguments

`p_req_obj`

Pointer to the request object.

`type`

Parameter type :

- `HTTPc_PARAM_TYPE_REQ_QUERY_STR_TBL`
- `HTTPc_PARAM_TYPE_REQ_QUERY_STR_HOOK`
- `HTTPc_PARAM_TYPE_REQ_HDR_TBL`
- `HTTPc_PARAM_TYPE_REQ_HDR_HOOK`
- `HTTPc_PARAM_TYPE_REQ_FORM_TBL`
- `HTTPc_PARAM_TYPE_REQ_BODY_CONTENT_TYPE`
- `HTTPc_PARAM_TYPE_REQ_BODY_CONTENT_LEN`
- `HTTPc_PARAM_TYPE_REQ_BODY_CHUNK`
- `HTTPc_PARAM_TYPE_REQ_BODY_HOOK`
- `HTTPc_PARAM_TYPE_RESP_HDR_HOOK`
- `HTTPc_PARAM_TYPE_RESP_BODY_HOOK`
- `HTTPc_PARAM_TYPE_TRANS_COMPLETE_CALLBACK`
- `HTTPc_PARAM_TYPE_TRANS_ERR_CALLBACK`

`p_param`

Pointer to the parameter.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_INIT`
- `RTOS_ERR_OWNERSHIP`

Returned Value

None.

Notes / Warnings

None.

HTTPc_ReqSend()

Description

Sends an HTTP request.

Files

`http_client.h/http_client.c`

Prototype

```
CPU_BOOLEAN  HTTPc_ReqSend (HTTPc_CONN_OBJ    *p_conn_obj,
                            HTTPc_REQ_OBJ     *p_req_obj,
                            HTTPc_RESP_OBJ    *p_resp_obj,
                            HTTP_METHOD        method,
                            CPU_CHAR          *p_resource_path,
                            CPU_INT16U         resource_path_len,
                            HTTPc_FLAGS        flags,
                            RTOS_ERR          *p_err)
```

Arguments

`p_conn_obj`

Pointer to the valid HTTPc Connection on which request will be sent.

`p_req_obj`

Pointer to the request to send.

`p_resp_obj`

Pointer to the response object that will be filled with the received response.

`method`

HTTP method of the request.

`p_resource_path`

Pointer to the complete URI (or only resource path) of the request.

`resource_path_len`

Resource path length.

`flags`

Configuration flags :

HTTPc_FLAG_REQ_NO_BLOCK

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_INIT`
- `RTOS_ERR_OWNERSHIP`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_NO_MORE_RSRC`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

- `DEF_YES` , if the HTTP Response received successfully.
- `DEF_NO` , otherwise.

Notes / Warnings

None.

HTTPc_FormAppFmt()

Description

Formats an application type form based on an array containing the form fields.

Files

`http_client.h/http_client.c`

Prototype

```
CPU_INT32U  HTTPc_FormAppFmt (CPU_CHAR             *p_buf,
                              CPU_INT16U            buf_len,
                              HTTPc_FORM_TBL_FIELD *p_form_tbl,
                              CPU_INT16U            form_tbl_size,
                              RTOS_ERR             *p_err)
```

Arguments

`p_buf`

Pointer to the buffer where the form will be written.

`buf_len`

Buffer length.

`p_form_tbl`

Pointer to the form field's table.

`form_tbl_size`

Table size.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_TYPE`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_FAIL`

Returned Value

Length of the formatted form, if no errors. 0 , otherwise.

Notes / Warnings

1. To format the form, only standard Key-Value Pair objects are supported in the table.

HTTPc_FormMultipartFmt()

Description

Formats a multipart type form based on an array containing the form fields.

Files

`http_client.h/http_client.c`

Prototype

```
CPU_INT32U  HTTPc_FormMultipartFmt (CPU_CHAR             *p_buf,
                                    CPU_INT16U            buf_len,
                                    HTTPc_FORM_TBL_FIELD  *p_form_tbl,
                                    CPU_INT16U            form_tbl_size,
                                    RTOS_ERR             *p_err)
```

Arguments

`p_buf`

Pointer to the buffer where the form will be written.

`buf_len`

Buffer length.

`p_form_tbl`

Pointer to the form field's table.

`form_tbl_size`

Table size.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_TYPE`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_FAIL`

Returned Value

Length of the formatted form, if no errors. 0 , otherwise.

Notes / Warnings

1. To format the form, only standard Key-Value Pair object are supported in the table.

HTTPc_FormAddKeyVal()

Description

Adds a Key-Value Pair object to the form table.

Files

`http_client.h/http_client.c`

Prototype

```
void  HTTPc_FormAddKeyVal (HTTPc_FORM_TBL_FIELD  *p_form_tbl,
                           HTTPc_KEY_VAL         *p_key_val,
                           RTOS_ERR              *p_err)
```

Arguments

`p_form_tbl`

Pointer to the the form table.

`p_kvp`

Pointer to the Key-Value Pair object to put in table.

`p_err`

Pointer to the the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

None.

HTTPc_FormAddKeyValExt()

Description

Adds an Extended Key-Value Pair object to the form table.

Files

`http_client.h/http_client.c`

Prototype

```
void  HTTPc_FormAddKeyValExt (HTTPc_FORM_TBL_FIELD  *p_form_tbl,
                              HTTPc_KEY_VAL_EXT     *p_key_val_ext,
                              RTOS_ERR              *p_err)
```

Arguments

`p_form_tbl`

Pointer to the form table.

`p_key_val_ext`

Pointer to the Extended Key-Value Pair object to put in table.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

1. `HTTPc_KVP_BIG` type object allows you to setup a Hook function that will be called when the form is sent. This lets the application write the value directly into the buffer.

HTTPc_FormAddFile()

Description

Adds a multipart file object to the form table.

Files

`http_client.h/http_client.c`

Prototype

```
void  HTTPc_FormAddFile (HTTPc_FORM_TBL_FIELD  *p_form_tbl,
                         HTTPc_MULTIPART_FILE  *p_file_obj,
                         RTOS_ERR              *p_err)
```

Arguments

`p_form_tbl`

Pointer to the form table.

`p_file_obj`

Pointer to the multipart file object to put in table.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

None.

HTTPc_WebSockSetParam()

Description

Sets a parameter related to a given WebSocket object.

Files

`http_client.h/http_client.c`

Prototype

```
void  HTTPc_WebSockSetParam (HTTPc_WEBSOCK_OBJ  *p_ws_obj,
                             HTTPc_PARAM_TYPE    type,
                             void               *p_param,
                             RTOS_ERR           *p_err)
```

Arguments

`p_ws_obj`

Pointer to the WebSocket object.

`type`

Parameter type :

- HTTPc_PARAM_TYPE_WEBSOCK_ON_OPEN
- HTTPc_PARAM_TYPE_WEBSOCK_ON_CLOSE
- HTTPc_PARAM_TYPE_WEBSOCK_ON_MSG_RX_INIT
- HTTPc_PARAM_TYPE_WEBSOCK_ON_MSG_RX_DATA
- HTTPc_PARAM_TYPE_WEBSOCK_ON_MSG_RX_COMPLETE
- HTTPc_PARAM_TYPE_WEBSOCK_ON_ERR
- HTTPc_PARAM_TYPE_WEBSOCK_ON_PONG

p_param

Pointer to the parameter.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function :

- RTOS_ERR_NONE
- RTOS_ERR_NOT_INIT
- RTOS_ERR_OWNERSHIP

Returned Value

None.

Notes / Warnings

None.

HTTPc_WebSockMsgSetParam()

Description

Sets a parameter related to a given WebSocket message object.

Files

http_client.h/http_client.c

Prototype

```
void  HTTPc_WebSockMsgSetParam (HTTPc_WEBSOCK_MSG_OBJ  *p_msg_obj,
                                HTTPc_PARAM_TYPE        type,
                                void                   *p_param,
                                RTOS_ERR               *p_err)
```

Arguments

p_msg_obj

Pointer to the WebSocket message object.

type

Parameter type :

- HTTPc_PARAM_TYPE_WEBSOCK_MSG_USER_DATA
- HTTPc_PARAM_TYPE_WEBSOCK_MSG_ON_TX_INIT
- HTTPc_PARAM_TYPE_WEBSOCK_MSG_ON_TX_DATA
- HTTPc_PARAM_TYPE_WEBSOCK_MSG_ON_TX_COMPLETE

p_param

Pointer to the parameter.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_INIT`
- `RTOS_ERR_OWNERSHIP`

Returned Value

None.

Notes / Warnings

None.

HTTPc_WebSockUpgrade()

Description

Upgrades an HTTP client connection to a WebSocket.

Files

`http_client.h/http_client.c`

Prototype

```
CPU_BOOLEAN  HTTPc_WebSockUpgrade (HTTPc_CONN_OBJ       *p_conn_obj,
                                   HTTPc_REQ_OBJ        *p_req_obj,
                                   HTTPc_RESP_OBJ       *p_resp_obj,
                                   HTTPc_WEBSOCK_OBJ    *p_ws_obj,
                                   CPU_CHAR             *p_resource_path,
                                   CPU_INT16U            resource_path_len,
                                   HTTPc_FLAGS           flags,
                                   RTOS_ERR             *p_err)
```

Arguments

`p_conn_obj`

Pointer to a valid HTTPc Connection on which request will occurred.

`p_req_obj`

Pointer to a request to send.

`p_resp_obj`

Pointer to a response object that will be filled with the received response.

`p_ws_obj`

Pointer to a WebSocket object.

`p_resource_path`

Pointer to a complete URI (or only resource path) of the request.

`resource_path_len`

Resource path length.

`flags`

Configuration flags :

- `HTTPc_FLAG_WEBSOCK_NO_BLOCK`

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_INIT`
- `RTOS_ERR_OWNERSHIP`
- `RTOS_ERR_ALREADY_EXISTS`
- `RTOS_ERR_NET_INVALID_CONN`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_NO_MORE_RSRC`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

- `DEF_OK` , if the operation is successful.
- `DEF_FAIL` , if the operation has failed.

Notes / Warnings

1. Connection Object:
   (a) Since the WebSocket Upgrade handshake is based on a HTTP request, the connection object `p_conn_obj` MUST have already established a connection to the desired host server.
2. Request and Response Object.
   (a) Any HTTP request/response features available can be used during the WebSocket Upgrade Request.
3. WebSocket Upgrade related Header field
   (a) During a WebSocket Upgrade handshake, the following mandatory header fields are managed by the WebSocket module and MUST NOT be set by the Application:
   - Connection
   - Upgrade
   - Sec-WebSocket-Key
   - Sec-WebSocket-Accept
   - Sec-WebSocket-Version

   (b) During a WebSocket Upgrade Handshake, the following optional header fields MAY be managed by the Application using the standard HTTPc Request/Response API:
   - Sec-WebSocket-Protocol
   - Sec-WebSocket-Extensions
4. If the WebSocket Upgrade Handshake is successful, the Status Code in the response object should be '101'. This means that the Host server has switched to the WebSocket protocol. Otherwise, the response object will TYPICALLY be described the reason of the failure.
5. When the WebSocket Upgrade is completed, any HTTP request is no more allowed and it is not possible to switch the connection protocol to HTTP. However, if an HTTP request is required to be sent by the Application, it SHOULD do one of the following procedure:
   (a) Close the current WebSocket Connection by sending a Close Frame and open again the HTTP connection.

(b) Open a different HTTP connection with the Host Server.

HTTPc_WebSockSend()

Description

Sends a WebSocket message.

Files

`http_client.h/http_client.c`

Prototype

```
CPU_BOOLEAN  HTTPc_WebSockSend (HTTPc_CONN_OBJ         *p_conn_obj,
                                HTTPc_WEBSOCK_MSG_OBJ   *p_msg_obj,
                                HTTPc_WEBSOCK_MSG_TYPE   msg_type,
                                CPU_CHAR               *p_data,
                                CPU_INT32U              payload_len,
                                HTTPc_FLAGS               flags,
                                RTOS_ERR               *p_err)
```

Arguments

`p_conn_obj`

Pointer to the valid HTTPc Connection upgraded to WebSocket.

`p_msg_obj`

Pointer to the valid WebSocket message object.

`msg_type`

Type of message to send.

`p_data`

Pointer to the payload to send.

`payload_len`

Length of the payload to send.

`flags`

Configuration flags :

- `HTTPc_FLAG_WEBSOCK_NO_BLOCK`

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_INIT`
- `RTOS_ERR_OWNERSHIP`
- `RTOS_ERR_NET_INVALID_CONN`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`

- RTOS_ERR_FAIL
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_NO_MORE_RSRC
- RTOS_ERR_ABORT
- RTOS_ERR_TIMEOUT

Returned Value

- DEF_OK , the operation is successful.
- DEF_FAIL , the operation has failed.

Notes / Warnings

1. The connection object p_conn MUST has been previously upgraded to WebSocket to use this function. Refer to the HTTPc_WebSockUpgrade() function.
2. The available Message Types are as follows :
   - HTTPc_WEBSOCK_MSG_TYPE_TXT_FRAME
   - HTTPc_WEBSOCK_MSG_TYPE_BIN_FRAME
   - HTTPc_WEBSOCK_MSG_TYPE_CLOSE
   - HTTPc_WEBSOCK_MSG_TYPE_PING
   - HTTPc_WEBSOCK_MSG_TYPE_PONG
3. Payload Content:
   (a) The "Payload data" (argument p_data) is defined as "Extension data" concatenated with "Application data".
   (b) Extension data length is 0 Bytes unless an extension has been negotiated during the handshake.
   (c) "Extension data" content and length are defines by the extension negotiated.
   (d) If negotiated, the "Extension data" must be handled by the application.
4. Data message Restrictions:
   (a) Even if the RFC6455 allows you to sent a message payload of up to 2^64 bytes. Only 2^32 bytes is allowed in the current implementation.
5. Control message Restrictions:
   (a) According to the RFC 6455 section 5.5:
   "All control frames MUST have a payload length of 125 bytes or less and MUST NOT be fragmented."
   (b) Closed frames have a specific payload format. For more information, refer to the HTTPc_WebSockFmtCloseMsg() function.
6. Client-to-Server Masking:
   (a) According to the RFC 6455 section 5.2:
   "All frames sent from the client to the server are masked by a 32-bit value that is contained within the frame."
   (b) The application DO NOT need to mask the payload since it's handled by the WebSocket module.

HTTPc_WebSockClr()

Description

Clears an HTTPc WebSock object before its first usage.

Files

http_client.h/http_client.c

Prototype

```
void  HTTPc_WebSockClr (HTTPc_WEBSOCK_OBJ  *p_ws_obj,
                        RTOS_ERR           *p_err)
```

Arguments

p_ws_obj

Pointer to the current HTTPc Websock object.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function :

- RTOS_ERR_NONE
- RTOS_ERR_NOT_INIT

Returned Value

None.

Notes / Warnings

None.

HTTPc_WebSockMsgClr()

Description

Clears an HTTPc WebSock Message object before its first usage.

Files

http_client.h/http_client.c

Prototype

```
void  HTTPc_WebSockMsgClr (HTTPc_WEBSOCK_MSG_OBJ *p_msg_obj,
                           RTOS_ERR              *p_err)
```

Arguments

p_msg_obj

Pointer to the WebSock Message object to clear.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function :

- RTOS_ERR_NONE
- RTOS_ERR_NOT_INIT

Returned Value

None.

Notes / Warnings

None.

HTTPc_WebSockFmtCloseMsg()

Description

Formats a Close Frame.

Files

http_client.h/http_client.c

Prototype

```
CPU_INT16U  HTTPc_WebSockFmtCloseMsg (HTTPc_WEBSOCK_CLOSE_CODE   close_code,
                                      CPU_CHAR                  *p_reason,
                                      CPU_CHAR                  *p_buf,
                                      CPU_INT16U                 buf_len,
                                      RTOS_ERR                  *p_err)
```

Arguments

`close_code`

Value that defines the origin of connection closure.

`p_reason`

Pointer to the string that contains a reason of the connection closure.

`p_buf`

Pointer to the destination buffer.

`buf_len`

Length of the destination buffer.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_INIT`
- `RTOS_ERR_WOULD_OVF`

Returned Value

- Payload length, if the operation is successful.
- 0, if the operation has failed.

Notes / Warnings

1. Close Frames have a specific format. According to the RFC 6455 section :
   "If there is a body, the first two bytes of the body MUST be a 2-byte unsigned integer (in network byte order) representing a status code with value /code/ defined in Section 7.4. Following the 2-byte integer,the body MAY contain UTF-8-encoded data with value /reason/, the interpretation of which is not defined by this specification."
2. The following Close Codes are available :

- `HTTPc_WEBSOCK_CLOSE_CODE_NORMAL`
- `HTTPc_WEBSOCK_CLOSE_CODE_GOING_AWAY`
- `HTTPc_WEBSOCK_CLOSE_CODE_PROTOCOL_ERR`
- `HTTPc_WEBSOCK_CLOSE_CODE_DATA_TYPE_NOT_ALLOWED`
- `HTTPc_WEBSOCK_CLOSE_CODE_DATA_TYPE_ERR`
- `HTTPc_WEBSOCK_CLOSE_CODE_POLICY_VIOLATION`
- `HTTPc_WEBSOCK_CLOSE_CODE_MSG_TOO_BIG`
- `HTTPc_WEBSOCK_CLOSE_CODE_INVALID_EXT`
- `HTTPc_WEBSOCK_CLOSE_CODE_UNEXPECTED_CONDITION`

1. Close Reason:
   (a) Except for its length, the reason string has no restriction and is user-definable. This should be used for debugging purposes.
   (b) This field can be empty.

HTTP Client Hook Functions

The HTTP Client module provides a set of hook functions to notify your application and to customize the stack behavior for your application's needs. Those are only available when the HTTP Client internal task is active (HTTPc_CFG_MODE_ASYNC_TASK_EN). Here's a list of description for the different hooks type:

Hook Type	Description	Structure name	SetParam function	Must be set before calling
Connection Hooks	Notify event about connection status	HTTPc_CONN_OBJ	HTTPc_ConnSetParam()	HTTPc_ConnOpen()
Request Hooks	Notify event that happen during a HTTP request and allows the application to customize it.	HTTPc_REQ_OBJ	HTTPc_ReqSetParam()	HTTPc_ReqSend()
Hook Functions WebSocket	Notify event that happen after a WebSocket Upgrade is successful and gives access to the application to the messages when received.	HTTPc_WEBSOCK_OBJ	HTTPc_WebSockSetParam()	HTTPc_WebSockUpgrade()
WebSocket Message Hooks	Notify event that happen during a WebSocket Message transmission allows the application to customize it.	HTTPc_WEBSOCK_MSG_OBJ	HTTPc_WebSockMsgSetParam()	HTTPc_WebSockSend()

HTTP Client Connection Hooks

The HTTP Client module provides a set of hook to notify the connection status to the application. Like for every hook functions, those are only available when the HTTP Client internal task is active (`HTTPc_CFG_MODE_ASYNC_TASK_EN`). Those hook functions can be set with the API function `HTTPc_ConnSetParam()` and are mandatory when the API function `HTTPc_ConnOpen()` is called in non-blocking mode.

Table - Connection Hooks

Hook Function	Description
Connection Hooks	
On Connection Connect	Notify that a connection has finish the Connect process.
On Connection Close	Notify that a connection has been close.

HTTP Client On Connection Connect

Called after a connection connect tryout with the remote HTTP server.

Like for every hook functions, it's only available when the HTTP Client internal task is active (`HTTPc_CFG_MODE_ASYNC_TASK_EN`). This hook function can be set up with the API function `HTTPc_ConnSetParam()` with parameter type `HTTPc_PARAM_TYPE_CONN_CONNECT_CALLBACK` . This hook is mandatory when the API function `HTTPc_ConnOpen()` is called in no-blocking mode.

Prototype

```
void  HTTPc_ConnConnectHook (HTTPc_CONN_OBJ  *p_conn_obj,
                             CPU_BOOLEAN      open_status);
```

Arguments

`p_conn`

Pointer to the current *HTTPc Connection Object*.

`open_status`

Status of the connection:

- `DEF_OK` , if the connection with the server was successful.
- `DEF_FAIL` , otherwise.

Return Values

None.

Required Configuration

The HTTPc Internal Task must be active. Refer to HTTP Client Compile-Time Configurations for more infmicrium-ormation.

Notes / Warnings

None.

Example Template

Listing - Connection Connect Hook Function Example Code

```
static void HTTPc_ConnConnectHook (HTTPc_CONN_OBJ *p_conn,
                                  CPU_BOOLEAN      open_status)
{
    if (open_status == DEF_OK) {
        printf("Connection to server succeeded.\n\r");
`    } else {`
        printf("Connection to server failed.\n\r");
    }
}
```

HTTP Client On Connection Close

Called after a connection close tryout.

Like for every hook functions, it's only available when the HTTP Client internal task is active (`HTTPc_CFG_MODE_ASYNC_TASK_EN`). This hook is mandatory when the internal task is enabled and can be set up with the API function `HTTPc_ConnSetParam()` with the parameter type `HTTPc_PARAM_TYPE_CONN_CLOSE_CALLBACK` .

Prototype

```
void HTTPc_ConnCloseHook(HTTPc_CONN_OBJ *p_conn_obj,
                         CPU_BOOLEAN     close_status,
                         RTOS_ERR        err);
```

Arguments

`p_conn_obj`

Pointer to the current *HTTPc Connection Object*.

`close_status`

Status of the connection close:

- `DEF_OK` , if the connection with the server was successful closed.
- `DEF_FAIL` , otherwise.

`err`

Internal error code when the connection was closed.

Return Values

None.

Required Configuration

The HTTPc Internal Task must be active. Refer to HTTP Client Compile-Time Configurations for more information.

Notes / Warnings

None.

Example Template

Listing - Connection Close Hook Function Example Code

```
static void  HTTPc_ConnCloseHook (HTTPc_CONN_OBJ  *p_conn,
                                  CPU_BOOLEAN      close_status,
                                  RTOS_ERR         err)
{
  printf("Connection closed with error code %i.\n\r", err.Code);
}
```

HTTP Client Request Hooks

The HTTP Client module provides a set of hook functions to customize the stack behavior during an HTTP Request. Hook functions are called during each HTTP transaction processing and allows to personalize the HTTP requests to send and to recover the data included in HTTP responses received. Those are only available when the HTTP Client internal task is active (`HTTPc_CFG_MODE_ASYNC_TASK_EN`) and can be set up with the API function `HTTPc_ReqSetParam()` .

Table - Request Hooks

Hook Function	Description
Transaction Hooks	
On Transaction Complete	Notify that an HTTP Transaction (Request + Response) is done.
On Transaction Error	Notify that an error occurred during an specific HTTP Transaction.
Request Hooks	
On Request Query String	Set a Query String field (Key-Value Pair) to add to the Request.
On Request Header	Set a Header field to add to the Request.
On Request Body	Set the data to put in the Request when using the Chunked Transfer Encoding.
Response Hooks	
On Response Header	Recover a header field received in the Response.
On Response Body	Recover each piece of data received in the Response.
Form Field Hooks	
On Form Key-Value Extended	Copy the data of the value into the transmission buffer.
On Form Multipart File	Copy the file data into the transmission buffer.

HTTP Client On Transaction Complete

Called after an HTTP transaction was completed.

Like for every hook functions, it's only available when the HTTP Client internal task is active (`HTTPc_CFG_MODE_ASYNC_TASK_EN`). This hook is not mandatory when the API function `HTTPc_ReqSend()` is called in blocking mode. It can be set up with the API function `HTTPc_ReqSetParam()` and the parameter type `HTTPc_PARAM_TYPE_TRANS_COMPLETE_CALLBACK` .

Prototype

```
void HTTPc_TransCompleteHook (HTTPc_CONN_OBJ *p_conn_obj,
                              HTTPc_REQ_OBJ  *p_req_obj,
                              HTTPc_RESP_OBJ *p_resp_obj,
                              CPU_BOOLEAN     status);
```

Arguments

`p_conn_obj`

Pointer to the current *HTTPc Connection Object*.

`p_req_obj`

Pointer to the current *HTTPc Request Object*.

`p_resp_obj`

Pointer to the *HTTPc Response Object* fill with the data received in the HTTP response.

`status`

Status of the Transaction:

- `DEF_OK` , Transaction was completed successfully.
- `DEF_FAIL` , otherwise.

Return Values

None.

Required Configuration

The HTTP Client Internal Task must be active. See section Run-time Configuration of the Compile-Time Configuration page.

Notes / Warnings

None.

Example Template

Listing - Transaction Complete Hook Function Example Code

```
static void HTTPc_TransCompleteHook (HTTPc_CONN_OBJ *p_conn,
                                     HTTPc_REQ_OBJ  *p_req,
                                     HTTPc_RESP_OBJ *p_resp,
                                     CPU_BOOLEAN     status)
{
    if (status == DEF_OK) {
        printf("%s\n\r", p_resp-\>ReasonPhrasePtr);
    } else {
        printf("Transaction failed\n\r");
    }
}
```

HTTP Client On Transaction Error

Called after an error occurred during an HTTP transaction process.

Like for every hook functions, it's only available when the HTTP Client internal task is active (`HTTPc_CFG_MODE_ASYNC_TASK_EN`). This hook is not mandatory when the API function `HTTPc_ReqSend()` is called in blocking

mode. This hook can be set up with the API function `HTTPc_ReqSetParam()` and the parameter type HTTPc_PARAM_TYPE_TRANS_ERR_CALLBACK.

Prototype

```
void HTTPc_TransErrHook (HTTPc_CONN_OBJ  *p_conn_obj,
                         HTTPc_REQ_OBJ   *p_req_obj,
                         RTOS_ERR         err);
```

Arguments

`p_conn_obj`

Pointer to the current *HTTPc Connection Object*.

`p_req_obj`

Pointer to the current *HTTPc Request Object*.

`err`

Error code associated with the occurred error.

Return Values

None.

Required Configuration

The HTTP Client Internal Task must be active. See the section Compile-Time Configuration page.

Notes / Warnings

None.

Example Template

Listing - Transaction Error Callback Function Example Code

```
static void HTTPc_TransErrHook (HTTPc_CONN_OBJ  *p_conn,
                                HTTPc_REQ_OBJ   *p_req,
                                RTOS_ERR         err)
{
 printf("Transaction error: %i\n\r", err);
}
```

HTTP Client On Request Query String

Called to include a Key-Value Pair to the Query String of the ongoing Request.

The parameter `HTTPc_PARAM_TYPE_REQ_QUERY_STR_HOOK` must be set up using the function `HTTPc_ReqSetParam()` for the hook function to be called.

Prototype

```
CPU_BOOLEAN HTTPc_ReqQueryStrHook (HTTPc_CONN_OBJ  *p_conn_obj,
                                   HTTPc_REQ_OBJ   *p_req_obj,
                                   HTTPc_KEY_VAL  **p_key_val);
```

Arguments

`p_conn_obj`

Pointer to the current *HTTPc Connection Object*.

`p_req_obj`

Pointer to the current *HTTPc Request Object*.

`p_key_val`

Variable that will received the pointer to the Key-Value Pair to include in the Query String.

Return Values

- `DEF_YES`, if all the Key-Value Pairs have been passed by the application.
- `DEF_NO`, if Key-Value Pairs remain to be passed by the application (Hook function will be called again).

Required Configuration

- The `HTTPc_CFG_QUERY_STR_EN` configuration macro **MUST** be enabled.

Notes / Warnings

- The Key-Value Pair object **MUST** stay valid until the HTTP transaction is completed.

Example Template

Listing - Connection Connect Callback Function Example Code

```
#define  HTTPc_QUERY_STR_KEY_LEN_MAX   10
#define  HTTPc_QUERY_STR_VAL_LEN_MAX   20

HTTPc_KEY_VAL  HTTPc_KeyVal;
CPU_CHAR       HTTPc_KeyStr[HTTPc_QUERY_STR_KEY_LEN_MAX];
CPU_CHAR       HTTPc_ValStr[HTTPc_QUERY_STR_VAL_LEN_MAX];

static  CPU_BOOLEAN  HTTPc_ReqQueryStrHook (HTTPc_CONN_OBJ   *p_conn,
                                            HTTPc_REQ_OBJ    *p_req,
                                            HTTPc_KEY_VAL   **p_key_val)
{
    HTTPc_KEY_VAL  *p_kvp;

    p_kvp         = &HTTPc_KeyVal;
    p_kvp-\>KeyPtr = &HTTPc_KeyStr[0];
    p_kvp-\>ValPtr = &HTTPc_ValStr[0];

    Str_Copy_N(p_kvp-\>KeyPtr, "Name", HTTPc_QUERY_STR_KEY_LEN_MAX);
    Str_Copy_N(p_kvp-\>ValPtr, "John", HTTPc_QUERY_STR_VAL_LEN_MAX);

    p_kvp-\>KeyLen = Str_Len_N(p_kvp-\>KeyPtr, HTTPc_QUERY_STR_KEY_LEN_MAX);
    p_kvp-\>ValLen = Str_Len_N(p_kvp-\>ValPtr, HTTPc_QUERY_STR_VAL_LEN_MAX);

   *p_key_val = p_kvp;

    return (DEF_YES);
}
```

HTTP Client On Request Header

Called by the HTTP Client core to allow the application to include a Header Field to the ongoing HTTP Request.

The parameter `HTTPc_PARAM_TYPE_REQ_HDR_HOOK` must be set up using the function `HTTPc_ReqSetParam()` for the hook function to be called.

Prototype

```
CPU_BOOLEAN  HTTPc_ReqHdrHook (HTTPc_CONN_OBJ   *p_conn_obj,
                               HTTPc_REQ_OBJ    *p_req_obj,
                               HTTP_HDR        **p_hdr);
```

Arguments

`p_conn_obj`

Pointer to the current *HTTPc Connection Object*.

`p_req_obj`

Pointer to the current *HTTPc Request Object*.

`p_hdr`

Variable that will received the pointer to the header field to include in the HTTP request.

Return Values

- `DEF_YES` , if all the header fields have been passed by the application.
- `DEF_NO` , if header fields remain to be passed by the application (Hook function will be called again).

Required Configuration

- The HTTPc_CFG_HDR_TX_EN configuration macro **MUST** be enabled.

Notes / Warnings

- The header field object **MUST** stay valid until the HTTP transaction is completed.

Example Template

Listing - Connection Connect Callback Function Example Code

```
#define  HTTPc_HDR_VAL_LEN_MAX         10

HTTPc_HDR  HTTPc_ReqHdr;
CPU_CHAR   HTTPc_ReqHdrValStr[HTTPc_HDR_VAL_LEN_MAX];

static  CPU_BOOLEAN  HTTPc_ReqHdrHook (HTTPc_CONN_OBJ   *p_conn,
                                       HTTPc_REQ_OBJ    *p_req,
                                       HTTPc_HDR       **p_hdr)
{
 HTTPc_HDR  *p_hdr_tmp;

  p_hdr_tmp         = &HTTPc_ReqHdr;
  p_hdr_tmp-\>ValPtr = &HTTPc_ReqHdrValStr[0];

  p_hdr_tmp-\>HdrField = HTTP_HDR_FIELD_COOKIE;
  Str_Copy_N(p_hdr_tmp-\>ValPtr, "ID=234668", HTTPc_HDR_VAL_LEN_MAX);
  p_hdr_tmp-\>ValLen   = Str_Len_N(p_hdr_tmp-\>ValPtr, HTTPc_HDR_VAL_LEN_MAX);

 *p_hdr = p_hdr_tmp;

  return (DEF_YES);
}
```

HTTP Client On Request Body

Called by the HTTP Client core to retrieve from the application the body data to put in the ongoing HTTP Request.

The parameter `HTTPc_PARAM_TYPE_REQ_BODY_HOOK` must be set up using the function `HTTPc_ReqSetParam()` for the hook function to be called. This hook function is used for both the Standard Transfer (content length of body is specified) and for the Chunked Transfer Encoding .

This hook function gives the choice to set the pointer to the application data (with p_data argument) that the HTTP Client stack will take care of transferring; or to directly copy the data in the HTTP transmit buffer (with `p_buf` argument).

Prototype

```
CPU_BOOLEAN  HTTPc_ReqBodyHook (HTTPc_CONN_OBJ   *p_conn_obj,
                                HTTPc_REQ_OBJ    *p_req_obj,
`                                void          **p_data,`
                                CPU_CHAR        *p_buf,
                                CPU_INT16U      *p_buf_len
                                CPU_INT16U      *p_data_len);
```

Arguments

`p_conn_obj`

Pointer to the current *HTTPc Connection Object*.

`p_req_obj`

Pointer to the current *HTTPc Request Object*.

`p_data`

Variable that will received the pointer to the data chunk to include in the HTTP request.

`p_buf`

Pointer to the HTTP transmit buffer.

`buf_len`

Length remaining in the HTTP transmit buffer.

`p_data_len`

Length of the data chunk.

Return Values

- `DEF_YES` , if all the data to transmit have been passed by the application.
- `DEF_NO` , if data to transmit remains to be passed by the application (Hook function will be called again).

Required Configuration

None.

Notes / Warnings

- The data to transmit **MUST** stay valid until the HTTP transaction is completed.

Example Template

Listing - Connection Connect Callback Function Example Code

```
CPU_CHAR  AppHTTPc_ReqBodyBuf[1024];

static  CPU_BOOLEAN  HTTPc_ReqBodyHook (HTTPc_CONN_OBJ   *p_conn,
                                        HTTPc_REQ_OBJ    *p_req,
                                        void            **p_data,
                                        CPU_CHAR         *p_buf,
                                        CPU_INT16U        buf_len,
                                        CPU_INT16U       *p_data_len)
{
    CPU_SIZE_T  data_len;

    data_len  = App_ReadDataStream(&AppHTTPc_ReqBodyBuf);  /* Theoretical application's function to read a stream of data and  ... */
                                                            /* ... copy it in the buffer and return the length of data copied.      */

   *p_data     = &AppHTTPc_ReqBodyBuf;
   *p_data_len = data_len;

    if (data_len == 0) {
        return (DEF_YES);
    } else {
        return (DEF_NO);
    }
}
```

HTTP Client On Response Header

Called by the HTTP Client core for each header field received in the HTTP response and recognize by the HTTP Client stack.

The application can decide to copy the header received or to ignore the header in the hook function. The parameter `HTTPc_PARAM_TYPE_RESP_HDR_HOOK` must be set up using the function `HTTPc_ReqSetParam()` for the hook function to be called.

Prototype

```
void  HTTPc_RespHdrHook (HTTPc_CONN_OBJ  *p_conn_obj,
                         HTTPc_REQ_OBJ   *p_req_obj,
                         HTTP_HDR_FIELD   hdr_field,
                         CPU_CHAR        *p_hdr_val,
                         CPU_INT16U       val_len);
```

Arguments

`p_conn_obj`

Pointer to the current *HTTPc Connection Object*.

`p_req_obj`

Pointer to the current *HTTPc Request Object*.

`hdr_field`

HTTP header type of the header field received in the HTTP response.

`p_hdr_val`

Pointer to the value string received in the Response header field.

`val_len`

Length of the value string.

Return Values

None.

Required Configuration

- The `HTTPc_CFG_HDR_RX_EN` configuration macro **MUST** be enabled.

Notes / Warnings

None.

Example Template

Listing - Connection Connect Callback Function Example Code

```
HTTPc_HDR  AppHTTPc_RespHdr;
CPU_CHAR   AppHTTPc_RespHdrValBuf[100];

static  void  HTTPc_RespHdrHook (HTTPc_CONN_OBJ  *p_conn,
                                 HTTPc_REQ_OBJ   *p_req,
                                 HTTP_HDR_FIELD   hdr_field,
                                 CPU_CHAR        *p_hdr_val,
                                 CPU_INT16U       val_len)
{
    HTTPc_HDR  *p_hdr;

    p_hdr         = &AppHTTPc_RespHdr;
    p_hdr-\>ValPtr = &AppHTTPc_RespHdrValBuf;

    switch (hdr_field) {
        case HTTP_HDR_FIELD_COOKIE:
             p_hdr-\>HdrField = hdr_field;
             Str_Copy_N(p_hdr-\>ValPtr, p_hdr_val, val_len);
             p_hdr-\>ValLen   = val_len;
             break;

        default:`
             break;
    }
}
```

HTTP Client On Response Body

Called by the HTTP Client core when body's data of the HTTP response is received.

The parameter `HTTPc_PARAM_TYPE_RESP_BODY_HOOK` must be set up using the function `HTTPc_ReqSetParam()` for the hook function to be called.

Prototype

```
CPU_INT32U  HTTPc_RespBodyHook (HTTPc_CONN_OBJ     *p_conn_obj,
                                HTTPc_REQ_OBJ      *p_req_obj,
                                HTTP_CONTENT_TYPE   content_type,
                                void               *p_data,
                                CPU_INT32U          data_len,
                                CPU_BOOLEAN         last_chunk);
```

Arguments

`p_conn_obj`

Pointer to the current *HTTPc Connection Object*.

`p_req_obj`

Pointer to the current *HTTPc Request Object*.

`content_type`

HTTP Content Type of the HTTP Response body's data.

`p_data`

Pointer to a data piece of the HTTP Response body.

`p_data_len`

Length of the data piece received.

`last_chunk`

- `DEF_YES` , if this is the last piece of data.
- `DEF_NO` , if more data is up coming.

Return Values

The number of bytes read during the callback.

Required Configuration

None.

Notes / Warnings

None.

Example Template

Listing - Connection Connect Callback Function Example Code

```
static void HTTPc_RespBodyHook (HTTPc_CONN_OBJ      *p_conn,
                                HTTPc_REQ_OBJ       *p_req,
                                HTTP_CONTENT_TYPE   content_type,
                                void                *p_data,
                                CPU_INT32U          data_len,
                                CPU_BOOLEAN         last_chunk)
{
    FS_FILE_HANDLE     file_handle;
    FS_FLAGS           fs_flags;
    CPU_SIZE_T         size_wr;
    CPU_SIZE_T         size_wr_tot;
    FS_ENTRY_INFO      entry_info;
    RTOS_ERR           err;

    file_handle = (FS_FILE_HANDLE)p_req-\>UserDataPtr;

    FSFile_Query(file_handle,
                 &entry_info,
                 &err);
    if (err.Code = RTOS_ERR_ENTRY_CLOSED) {

        fs_flags = 0;
        DEF_BIT_SET(fs_flags, FS_FILE_ACCESS_MODE_WR);
        DEF_BIT_SET(fs_flags, FS_FILE_ACCESS_MODE_CREATE);
        DEF_BIT_SET(fs_flags, FS_FILE_ACCESS_MODE_TRUNCATE);
        file_handle = FSFile_Open(FS_WRK_DIR_NULL,
                                  "index.html",
                                  fs_flags,
                                  &err);
        if (err.Code != RTOS_ERR_NONE) {
            return;
        }

        p_req-\>UserDataPtr = (void *)file_handle;
 }

    switch (content_type) {
        case HTTP_CONTENT_TYPE_HTML:
             if (p_data != DEF_NULL) {
           size_wr     = 0;
                 size_wr_tot = 0;
                 while (size_wr \< data_len) {
                     size_wr = FSFile_Wr(file_handle,
                                         p_data,
                                         data_len,
                                        &err);
                     if (err.Code != RTOS_ERR_NONE) {
                         return;
                     }
                     size_wr_tot += size_wr;
                 }
             }
             break;

        case HTTP_CONTENT_TYPE_OCTET_STREAM:
        case HTTP_CONTENT_TYPE_PDF:
        case HTTP_CONTENT_TYPE_ZIP:
        case HTTP_CONTENT_TYPE_GIF:
        case HTTP_CONTENT_TYPE_JPEG:
        case HTTP_CONTENT_TYPE_PNG:
        case HTTP_CONTENT_TYPE_JS:
        case HTTP_CONTENT_TYPE_PLAIN:
        case HTTP_CONTENT_TYPE_CSS:
        case HTTP_CONTENT_TYPE_JSON:
        default:
             break;
    }
```

```
if(last_chunk == DEF_YES){FSFile_Close(file_handle,&err);}}
```

HTTP Client On Form Key-Value Extended

Called by the HTTP Client core when the form table is formatted to be sent.

Each `HTTPc_KEY_VAL_EXT` objects contained in a table form must have set up a hook function pointer with the parameter OnValTx. The HTTPc transmit buffer pointer and the length available inside are passed as argument to allow the hook function to copy the data value directly inside the HTTPc buffer.

Prototype

```
CPU_BOOLEAN  HTTPc_FormKeyValExtHook (HTTPc_CONN_OBJ     *p_conn_obj,
                                      HTTPc_REQ_OBJ      *p_req_obj,
                                      HTTPc_KEY_VAL_EXT  *p_key_val_obj,
                                      CPU_CHAR           *p_buf,
                                      CPU_INT16U          buf_len,
                                      CPU_INT16U         *p_len_wr);
```

Arguments

`p_conn_obj`

Pointer to the current *HTTPc Connection Object*.

`p_req_obj`

Pointer to the current *HTTPc Request Object*.

`p_key_val_obj`

Pointer to the Key-Value Extended object.

`p_buf`

Pointer to transmit buffer.

`buf_len`

Length available inside the transmit buffer.

`p_len_wr`

Data length actually written inside the buffer by the hook function.

Return Values

- `DEF_YES` , if all the data have been written to the buffer.
- `DEF_NO` , otherwise

Required Configuration

None.

Notes / Warnings

None.

Example Template

Listing - Connection Connect Callback Function Example Code

```
CPU_CHAR  HTTPc_AppBuf[512];

static  CPU_BOOLEAN  HTTPc_FormKeyValExtHook (HTTPc_CONN_OBJ     *p_conn_obj,
                                              HTTPc_REQ_OBJ      *p_req_obj,
                                              HTTPc_KEY_VAL_EXT  *p_key_val_obj,
                                              CPU_CHAR           *p_buf,
                                              CPU_INT16U          buf_len,
                                              CPU_INT16U         *p_len_wr);
{
         HTTPc_KEY_VAL_EXT  *p_field;
         CPU_CHAR           *p_data;
         CPU_INT16U          min_len;
  static  CPU_INT16U          data_ix;

  p_data = &HTTPc_AppBuf[data_ix];

  min_len = DEF_MIN(buf_len, (p_key_val_obj-\>ValLen - data_ix));
  if (min_len \<= 0) {
     data_ix = 0;
    *p_len_wr = min_len;
     return (DEF_YES);
  }

  Str_Copy_N(p_buf, p_data, min_len);
 *p_len_wr  = min_len;
  data_ix  += min_len;

  return (DEF_NO);
}
```

This code uses a variable declared as static for example purpose. It cannot work if multiple connections call the same hook function. In that case, the UserDataPtr parameter inside the `HTTPc_REQ_OBJ` or `HTTPc_CONN_OBJ` can be used to store data for the application usage.

HTTP Client On Form Multipart File

Called by the HTTP Client core when the form table is formatted to be sent.

Each `HTTPc_MULTIPART_FILE` objects contained in a table form must have set up a hook function pointer with the parameter OnFileTx. The HTTPc transmit buffer pointer and the length available inside are passed as argument to allow the hook function to copy the data file directly inside the HTTPc buffer.

Prototype

```
CPU_BOOLEAN  HTTPc_FormMultipartFileHook (HTTPc_CONN_OBJ        *p_conn_obj,
                                          HTTPc_REQ_OBJ         *p_req_obj,
                                          HTTPc_MULTIPART_FILE  *p_file_obj,
                                          CPU_CHAR              *p_buf,
                                          CPU_INT16U             buf_len,
                                          CPU_INT16U            *p_len_wr);
```

Arguments

`p_conn_obj`

Pointer to the current *HTTPc Connection Object*.

`p_req_obj`

Pointer to the current *HTTPc Request Object*.

`p_file_obj`

Pointer to the Multipart File object.

`p_buf`

Pointer to transmit buffer.

`buf_len`

Length available inside the transmit buffer.

`p_len_wr`

Data length actually written inside the buffer by the hook function.

Return Values

- `DEF_YES` , if all the data have been written to the buffer.
- `DEF_NO` , otherwise

Required Configuration

None.

Notes / Warnings

None.

Example Template

Listing - Connection Connect Callback Function Example Code

```
static  CPU_BOOLEAN  HTTPc_FormMultipartFileHook (HTTPc_CONN_OBJ       *p_conn_obj,
                                                  HTTPc_REQ_OBJ        *p_req_obj,
                                                  HTTPc_MULTIPART_FILE  *p_file_obj,
                                                  CPU_CHAR             *p_buf,
                                                  CPU_INT16U            buf_len,
                                                  CPU_INT16U           *p_len_wr);
{
   FS_FILE_HANDLE   file_handle;
   FS_ENTRY_INFO    entry_info;
   FS_FLAGS         fs_flags;
   CPU_SIZE_T       file_rem;
   CPU_SIZE_T       size;
   CPU_SIZE_T       size_rd;
   CPU_BOOLEAN      finish;
   RTOS_ERR         err;

   finish      = DEF_YES;
   file_handle = (FS_FILE_HANDLE) p_req_obj-\>UserDataPtr;

   FSFile_Query(file_handle,
               &entry_info,
               &err);
   if (err.Code == RTOS_ERR_ENTRY_CLOSED) {

       fs_flags = 0;
       DEF_BIT_SET(fs_flags, FS_FILE_ACCESS_MODE_RD);
       DEF_BIT_SET(fs_flags, FS_FILE_ACCESS_MODE_CREATE);
       file_handle = FSFile_Open(FS_WRK_DIR_NULL,
                                 "index.html",
                                 fs_flags,
                                &err);
       if (err.Code != RTOS_ERR_NONE) {
           return (finish);
       }
       p_req-\>UserDataPtr = (void *) file_handle;

   }

   file_rem = p_file-\>Size - p_file-\>Pos;
   if (file_rem \<= 0) {
      *p_len_wr = 0;
       finish   = DEF_YES;
       FSFile_Close(file_handle,
                   &err);
       goto exit;
   }

   size = DEF_MIN(file_rem,
                  buf_len);

   size_rd = FSFile_Rd(file_handle,
                       p_buf,
                       size,
                      &err);
   if (err.Code != RTOS_ERR_NONE) {
       return (finish);
   }

  *p_len_wr = size_rd;
   finish   = DEF_NO;

exit:
   return (finish);
}
```

This code uses the Micrium OS File System for example purposes.

HTTP Client Hook Functions WebSocket

The HTTP Client provides a set of hook functions for WebSocket to customize the stack behavior for your application needs. Those are only available when the HTTP Client internal task is active (`HTTPc_CFG_MODE_ASYNC_TASK_EN`) and can be set up with the API function `HTTPc_WebSockMsgSetParam()`

Table - WebSocket Hooks

Hook Function	Description
Control Hooks	
On Open	Called when a WebSocket Upgrade is successful.
On Close	Called when an error occurred.
On Error	Called when a connection that has been successfully upgraded close.
Message RX Hooks	
On Message Reception Initialization	Called at the beginning of the WebSocket data message reception.
On Message RX Data	Called each time a chunk of the data message is received is available.
On Message RX Complete	Called when the data message is completely received.
On Pong	Called when a Pong message is received.

HTTP Client On Open

Called when a WebSocket upgrade is successful.

It notify that the application can start to receive and transmit WebSocket messages.

Prototype

```
void  HTTPc_WebSockOnOpen (HTTPc_CONN_OBJ  *p_conn)
```

Arguments

`p_conn`

Pointer to the current *HTTPc Connection Object*.

Return Values

None.

Required Configuration

1. The `HTTPc_CFG_WEBSOCKET_EN` configuration macro **MUST** be enabled.

Notes / Warnings

- This hook is set in a `HTTPc_WEBSOCK_OBJ` using `HTTPc_WebSockSetParam()` with the parameter `HTTPc_PARAM_TYPE_WEBSOCK_ON_OPEN` .
- Must be set to the `HTTPc_WEBSOCK_OBJ` before upgrading the connection with `HTTPc_WebSockUpgrade()` .

HTTP Client On Close

Called when a WebSocket Connection has closed.

Note that it doesn't guarantee that the closing handshake is completed, but only notify that the connection will not send or receive WebSocket Message anymore.

Prototype

```
void  HTTPc_WebSockOnClose (HTTPc_CONN_OBJ              *p_conn,
                            HTTPc_WEBSOCK_CLOSE_CODE     close_code,
                            HTTPc_WEBSOCK_CLOSE_REASON  *p_reason);
```

Arguments

`p_conn`

Pointer to the current *HTTPc Connection Object*.

`close_code`

Code defined by the RFC to explain the reason of the connection closing.

`p_reason`

Pointer that contains the data following the close code and the length.

Return Values

None.

Required Configuration

- The `HTTPc_CFG_WEBSOCKET_EN` configuration macro **MUST** be enabled.

Notes / Warnings

- This hook is set in a `HTTPc_WEBSOCK_OBJ` using `HTTPc_WebSockSetParam()` with the parameter `HTTPc_PARAM_TYPE_WEBSOCK_ON_CLOSE`.
- Must be set to the `HTTPc_WEBSOCK_OBJ` before upgrading the connection with `HTTPc_WebSockUpgrade()`.

HTTP Client On Error

Called when an error occurs in a WebSocket connection.

Prototype

```
void  HTTPc_WebSockOnError (HTTPc_CONN_OBJ  *p_conn,
                            RTOS_ERR         err);
```

Arguments

`p_conn`

Pointer to the current *HTTPc Connection Object*.

`err`

Error Code causing the connection to close.

Return Values

None.

Required Configuration

- The `HTTPc_CFG_WEBSOCKET_EN` configuration macro **MUST** be enabled.

Notes / Warnings

- This hook is set in a `HTTPc_WEBSOCK_OBJ` using `HTTPc_WebSockSetParam()` with the parameter `HTTPc_PARAM_TYPE_WEBSOCK_ON_ERR`.
- Must be set to the `HTTPc_WEBSOCK_OBJ` before upgrading the connection with `HTTPc_WebSockUpgrade()`.

HTTP Client On Pong

Called every time a Pong message is received.

Prototype

```
void  HTTPc_WebSockOnPong (HTTPc_CONN_OBJ *p_conn,
                           CPU_CHAR       *p_data,
                           CPU_INT16U      data_len);
```

Arguments

`p_conn`

Pointer to the current *HTTPc Connection Object*.

`p_data`

Pointer to the data of the Pong message

`data_len`

Data length of the Pong message.

Return Values

None.

Required Configuration

- The `HTTPc_CFG_WEBSOCKET_EN` configuration macro **MUST** be enabled.

Notes / Warnings

- This hook is set in a `HTTPc_WEBSOCK_OBJ` using `HTTPc_WebSockSetParam()` with the parameter `HTTPc_PARAM_TYPE_WEBSOCK_ON_PONG`.
- Must be set to the `HTTPc_WEBSOCK_OBJ` before upgrading the connection with `HTTPc_WebSockUpgrade()`.

HTTP Client On Message Reception Initialization

Called at every beginning of a WebSocket Data Message.

It first describes the type of data message and the total length of the message to receive. Then, it is possible to set an optional pointer to specify where the data must be copied automatically.

Prototype

```
        void  HTTPc_WebSockOnMsgRxInit (HTTPc_CONN_OBJ          *p_conn,
                                        HTTPc_WEBSOCK_MSG_TYPE   msg_type,
                                        CPU_INT32U               msg_len,
`                                        void                   **p_data);
```

Arguments

`p_conn`

Pointer to the current *HTTPc Connection Object*.

msg_type

Type of WebSocket message to receive.

msg_len

The total length if the message to receive.

p_data

Pointer to a pointer that can be set to specify where the data must be copied. Specific to the Auto Mode.

Return Values

None.

Required Configuration

- The HTTPc_CFG_WEBSOCKET_EN configuration macro **MUST** be enabled.

Notes / Warnings

- When a message is fragmented, thus the total message length is unknown, the value will be HTTPc_WEBSOCK_TX_MSG_LEN_NOT_DEFINED.
- This hook is set in a HTTPc_WEBSOCK_OBJ using HTTPc_WebSockSetParam() with the parameter HTTPc_PARAM_TYPE_WEBSOCK_ON_MSG_RX_INIT.
- Must be set to the HTTPc_WEBSOCK_OBJ before upgrading the connection with HTTPc_WebSockUpgrade().

HTTP Client On Message RX Data

Called every time a chunk of the data message is available to be read.

Prototype

```
CPU_INT32U  HTTPc_WebSockOnMsgRxData  (HTTPc_CONN_OBJ         *p_conn,
                                       HTTPc_WEBSOCK_MSG_TYPE   msg_type,
`                                       void                   *p_data,`
                                       CPU_INT16U              data_len);
```

Arguments

p_conn

Pointer to the current *HTTPc Connection Object*.

msg_type

Type of WebSocket message to receive.

p_data

Pointer to the message data chunk.

data_len

Length of the data chunk available.

Return Values

The total number of bytes read from the data chunk.

Required Configuration

- The `HTTPc_CFG_WEBSOCKET_EN` configuration macro **MUST** be enabled.

Notes / Warnings

- When the number of bytes read is less than the available data, the remaining bytes will be available for the next time this hook is called with new received data concatenated, if applicable.
- This hook is set in a `HTTPc_WEBSOCK_OBJ` using `HTTPc_WebSockSetParam()` with the parameter `HTTPc_PARAM_TYPE_WEBSOCK_ON_MSG_RX_DATA`.
- Must be set to the `HTTPc_WEBSOCK_OBJ` before upgrading the connection with `HTTPc_WebSockUpgrade()`.

HTTP Client On Message RX Complete

Called when a message is received completely.

Prototype

```
void HTTPc_WebSockOnMsgRxComplete (HTTPc_CONN_OBJ *p_conn);
```

Arguments

`p_conn`

Pointer to the current *HTTPc Connection Object*.

Return Values

None.

Required Configuration

- The `HTTPc_CFG_WEBSOCKET_EN` configuration macro **MUST** be enabled.

Notes / Warnings

- This hook is set in a `HTTPc_WEBSOCK_OBJ` using `HTTPc_WebSockSetParam()` with the parameter `HTTPc_PARAM_TYPE_WEBSOCK_ON_MSG_RX_COMPLETE`.
- Must be set to the `HTTPc_WEBSOCK_OBJ` before upgrading the connection with `HTTPc_WebSockUpgrade()`.

HTTP Client WebSocket Message Hooks

The HTTP Client provides a set of hook functions for WebSocket Message to customize the stack behavior for your application needs. Those are only available when the HTTP Client internal task is active (`HTTPc_CFG_MODE_ASYNC_TASK_EN`) and can be set up with the API function `HTTPc_WebSockSetParam()`.

Table - WebSocket Message Hooks

Hook Function	Description
Message TX Hooks	
On Message TX Initialization	Called when the message is ready to be sent.
On Message TX Data	Called to set the internal connection buffer of a data chunk of a message to send.
On Message TX Complete	Called when the message is completely transmitted

HTTP Client On Message TX Initialization

Called when the message is ready to be transmit.

Prototype

```
CPU_INT32U HTTPc_WebSockOnMsgTxInit (HTTPc_CONN_OBJ         *p_conn,
                                     HTTPc_WEBSOCK_MSG_OBJ   *p_msg);
```

Arguments

`p_conn`

Pointer to the current *HTTPc Connection Object*.

`p_msg`

Pointer to the WebSocket Message to transmit

Return Values

The length of the message to transmit.

Required Configuration

- The `HTTPc_CFG_WEBSOCKET_EN` configuration macro **MUST** be enabled.

Notes / Warnings

- When the dynamic mode is used, thus the total message length is unknown, the value returned will be `HTTPc_WEBSOCK_TX_MSG_LEN_NOT_DEFINED`.
- This hook is set in a `HTTPc_WEBSOCK_MSG_OBJ` using `HTTPc_WebSockMsgSetParam()` with the parameter `HTTPc_PARAM_TYPE_WEBSOCK_MSG_ON_MSG_TX_INIT`.
- Must be set to the `HTTPc_WEBSOCK_MSG_OBJ` before upgrading the connection with `HTTPc_WebSockSend()`.

HTTP Client On Message TX Data

Called to set a chunk of the WebSocket message payload to the connection buffer.

Prototype

```
CPU_INT32U  HTTPc_WebSockOnMsgTxData (HTTPc_CONN_OBJ        *p_conn,
                                      HTTPc_WEBSOCK_MSG_OBJ *p_msg,
                                      CPU_CHAR              *p_buf,
                                      CPU_INT32U             buf_len);
```

Arguments

`p_conn`

Pointer to the current *HTTPc Connection Object*.

`p_msg`

Pointer to the WebSocket message object.

`p_buf`

Pointer to the start of the transmit buffer.

`buf_len`

The available size of the transmit buffer.

Return Values

Number of bytes copied in the buffer and ready to be transmitted.

Required Configuration

- The `HTTPc_CFG_WEBSOCKET_EN` configuration macro **MUST** be enabled.

Notes / Warnings

- In dynamic mode, thus the message length has been previously set to `HTTPc_WEBSOCK_TX_MSG_LEN_NOT_DEFINED`, this hook will be called until the value 0 is returned.
- This hook is set in a `HTTPc_WEBSOCK_MSG_OBJ` using `HTTPc_WebSockMsgSetParam()` with the parameter `HTTPc_PARAM_TYPE_WEBSOCK_MSG_ON_MSG_TX_INIT`.
- Must be set to the `HTTPc_WEBSOCK_MSG_OBJ` before upgrading the connection with `HTTPc_WebSockSend()`.

HTTP Client On Message TX Complete

Called when the message is completely transmitted. Can be also called for every pending message when a connection is close.

Prototype

```
void HTTPc_WebSockOnMsgTxComplete (HTTPc_CONN_OBJ        *p_conn,
                                   HTTPc_WEBSOCK_MSG_OBJ *p_msg
                                  CPU_BOOLEAN            status);
```

Arguments

`p_conn`

Pointer to the current *HTTPc Connection Object*.

`p_msg`

Pointer to the WebSocket Message transmitted.

`status`

Bool that indicate if the message has been transmitted successfully.

Return Values

None.

Required Configuration

- The `HTTPc_CFG_WEBSOCKET_EN` configuration macro **MUST** be enabled.

Notes / Warnings

- This hook is set in a `HTTPc_WEBSOCK_MSG_OBJ` using `HTTPc_WebSockMsgSetParam()` with the parameter `HTTPc_PARAM_TYPE_WEBSOCK_MSG_ON_MSG_TX_COMPLETE`.
- Must be set to the `HTTPc_WEBSOCK_MSG_OBJ` before upgrading the connection with `HTTPc_WebSockSend()`.

HTTP Server API

HTTP Server API

- HTTPs_ConfigureMemSeg()
- HTTPs_Init()
- HTTPs_InstanceInit()
- HTTPs_InstanceTaskPrioSet()
- HTTPs_InstanceStart()
- HTTPs_InstanceStop()
- HTTPs_RespBodySetParamFile()
- HTTPs_RespBodySetParamNoBody()
- HTTPs_RespBodySetParamStaticData()
- HTTPs_RespHdrGet()
- HTTPsREST_Publish()
- HTTPs_FS_Init()
- HTTPs_FS_AddFile()
- HTTPs_FS_SetTime()
- HTTP Server Hook Functions API
- HTTP Server Add-on API

HTTPs_ConfigureMemSeg()

Description

Configure the memory segment that will be used to allocate internal data needed by the HTTP server module instead of the default memory segment.

Files

`http_server.h/http_server.c`

Prototype

```
void HTTPs_ConfigureMemSeg (MEM_SEG *p_mem_seg)
```

Arguments

`p_mem_seg`

Pointer to the memory segment from which the internal data will be allocated. If `DEF_NULL`, the internal data will be allocated from the global Heap.

Returned Value

None.

Notes / Warnings

1. This function is optional. If it is called, it must be called before `HTTPs_Init()`. If it is not called, default values will be used to initialize the module.

HTTPs_Init()

Description

Initializes the HTTP server suite.

Files

`http_server.h/http_server.c`

Prototype

```
void HTTPs_Init (RTOS_ERR *p_err)
```

Arguments

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ALREADY_INIT`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`

Returned Value

None.

Notes / Warnings

None.

HTTPs_InstanceInit()

Description

Initializes an HTTP server instance.

Files

`http_server.h/http_server.c`

Prototype

```
HTTPs_INSTANCE *HTTPs_InstanceInit (const HTTPs_CFG      *p_cfg,
                                    const RTOS_TASK_CFG *p_task_cfg,
                                          RTOS_ERR      *p_err)
```

Arguments

`p_cfg`

Pointer to the instance configuration object.

`p_task_cfg`

Pointer to the instance task configuration object.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_READY`
- `RTOS_ERR_POOL_EMPTY`

- RTOS_ERR_BLK_ALLOC_CALLBACK
- RTOS_ERR_SEG_OVF
- RTOS_ERR_ALLOC
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_OS_ILLEGAL_RUN_TIME
- RTOS_ERR_INIT

Returned Value

- Pointer to the instance handler, if NO error(s).
- NULL pointer, otherwise.

Notes / Warnings

None.

HTTPs_InstanceTaskPrioSet()

Description

Sets the priority of the given HTTP server instance's task.

Files

http_server.h/http_server.c

Prototype

```
void  HTTPs_InstanceTaskPrioSet (HTTPs_INSTANCE  *p_instance,
                                 RTOS_TASK_PRIO   prio,
                                 RTOS_ERR        *p_err)
```

Arguments

p_instance

Pointer to specific HTTP server instance handler.

prio

Priority of the HTTP instance's task.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function :

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_ARG

Returned Value

None.

Notes / Warnings

None.

HTTPs_InstanceStart()

Description

Starts a specific HTTPs server instance which had been previously initialized.

Files

http_server.h/http_server.c

Prototype

```
void  HTTPs_InstanceStart (HTTPs_INSTANCE  *p_instance,
                           RTOS_ERR        *p_err)
```

Arguments

`p_instance`

Pointer to specific HTTP server instance handler.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- RTOS_ERR_NONE
- RTOS_ERR_OS_ILLEGAL_RUN_TIME
- RTOS_ERR_INVALID_TYPE
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_NOT_FOUND
- RTOS_ERR_ALREADY_EXISTS
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_NET_INVALID_CONN
- RTOS_ERR_INVALID_STATE
- RTOS_ERR_NET_CONN_CLOSED_FAULT

Returned Value

None.

Notes / Warnings

None.

HTTPs_InstanceStop()

Description

Stops a specific HTTPs server instance.

Files

http_server.h/http_server.c

Prototype

```
void  HTTPs_InstanceStop (HTTPs_INSTANCE  *p_instance,
                          RTOS_ERR        *p_err)
```

Arguments

`p_instance`

Pointer to Instance handler.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- RTOS_ERR_NONE
- RTOS_ERR_OS_OBJ_DEL

- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_ABORT
- RTOS_ERR_TIMEOUT

Returned Value

None.

Notes / Warnings

None.

HTTPs_RespBodySetParamFile()

Description

Sets the parameters for the response body when the body's data is a file inside a File System infrastructure.

Files

http_server.h/http_server.c

Prototype

```
void  HTTPs_RespBodySetParamFile (const  HTTPs_INSTANCE        *p_instance,
                                  HTTPs_CONN          *p_conn,
                                  CPU_CHAR            *p_path,
                                  HTTP_CONTENT_TYPE      content_type,
                                  CPU_BOOLEAN           token_en,
                                  RTOS_ERR            *p_err)
```

Arguments

p_instance

Pointer to the instance.

p_conn

Pointer to the connection.

p_path

Pointer to the string file path.

content_type

Content type of the file.

If unknown, this can be set to HTTP_CONTENT_TYPE_UNKNOWN . The server core will find it with the file extension. See HTTP_CONTENT_TYPE enum in http.h for possible content types.

token_en

- DEF_YES , if the file contents tokens the server needs to replace.
- DEF_NO , otherwise.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function :

- RTOS_ERR_NONE

Returned Value

None.

Notes / Warnings

1. Must be called from a callback function only.

(a) The instance lock must be acquired before calling this function, which is why this function must be called from a callback function.

HTTPs_RespBodySetParamNoBody()

Description

Sets the parameters to let the server know that no body is needed in the response.

Files

`http_server.h/http_server.c`

Prototype

```
void  HTTPs_RespBodySetParamNoBody (const HTTPs_INSTANCE  *p_instance,
                                    HTTPs_CONN      *p_conn,
                                    RTOS_ERR        *p_err)
```

Arguments

`p_instance`

Pointer to the instance.

`p_conn`

Pointer to the connection.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

1. Must be called from a callback function only.
   (a) The instance lock must be acquired before calling this function, which is why this function must be called from a callback function

HTTPs_RespBodySetParamStaticData()

Description

Sets the parameters for the response body when the body's data is a static data contained in a memory space.

Files

`http_server.h/http_server.c`

Prototype

```
void HTTPs_RespBodySetParamStaticData (const HTTPs_INSTANCE    *p_instance,
                                      HTTPs_CONN         *p_conn,
                                      HTTP_CONTENT_TYPE   content_type,
                                      void               *p_data,
                                      CPU_INT32U          data_len,
                                      CPU_BOOLEAN         token_en,
                                      RTOS_ERR           *p_err)
```

Arguments

`p_instance`

Pointer to the instance.

`p_conn`

Pointer to the connection.

`content_type`

Content type of the file. See `HTTP_CONTENT_TYPE` enum in http.h for possible content types.

`p_data`

- Pointer to memory section containing data.
- `DEF_NULL`, if data will added to the response with the '`OnRespChunkHook`' Hook.

`data_len`

Data length.

`token_en`

- `DEF_YES`, if the data contents tokens the server needs to replace.
- `DEF_NO`, otherwise.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

1. Must be called from a callback function only.
   (a) The instance lock must be acquired before calling this function, which is why this function must be called from a callback function.
2. This function can be used when the data to put in the response body is not in a file in a File System.
3. If all the data to send is inside a memory space, the '`p_data`' parameter can be set to point to the memory space and the '`data_len`' must be set, since the data length is known.
4. When the data to send is a stream of unknown size, the Chunked Transfer Encoding must be used. In this case, the function can work with the parameter '`p_data`' when set to `DEF_NULL`. This tells the server to use the hook function '`p_cfg->p_hooks->OnRespChunkHook`' to retrieve the data to put in the HTTP response.

HTTPs_RespHdrGet()

Description

Acquires a new response header block.

Files

`http_server.h/http_server.c`

Prototype

```
#if (HTTPs_CFG_HDR_TX_EN == DEF_ENABLED)

HTTPs_HDR_BLK *HTTPs_RespHdrGet (const HTTPs_INSTANCE     *p_instance,
                                 HTTPs_CONN         *p_conn,
                                 HTTP_HDR_FIELD      hdr_field,
                                 HTTPs_HDR_VAL_TYPE  val_type,
                                 RTOS_ERR           *p_err)
```

Arguments

`p_instance`

Pointer to the instance.

`p_conn`

Pointer to the connection.

`hdr_field`

Type of the response header value :

See enumeration `HTTPs_HDR_FIELD` .

`val_type`

Data type of the response header field value :

- `HTTP_HDR_VAL_TYPE_NONE` Header field does not require a value.
- `HTTP_HDR_VAL_TYPE_STR_CONST` Header value type is a constant string.
- `HTTP_HDR_VAL_TYPE_STR` Header value type is a variable string.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`

Returned Value

Pointer to the response header block that can be filled with data, if no error(s). Null pointer, otherwise

Notes / Warnings

1. Must be called from a callback function only. Should be called by the function pointed by RespHdrTxFnctPtr in the instance configuration.
   (a) The instance lock must be acquired before calling this function, which is is why this function must be called from a callback function.
2. The header block is automatically added to the header blocks list, so the caller does not need to add the block to the list. Only filling the value and value length should be required by the caller.

HTTPsREST_Publish()

Description

Adds a REST resource to a list of resources.

Files

`http_server_rest.h/http_server_rest.c`

Prototype

```
void  HTTPsREST_Publish (const HTTPs_REST_RESOURCE  *p_resource,
                         CPU_INT32U          list_ID,
                         RTOS_ERR           *p_err)
```

Arguments

`p_resource`

Pointer to the REST resource to publish.

`list_ID`

Identification of the list on which to publish.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

1. Must be called after the HTTP server initialization and before HTTP server start.

HTTPs_FS_Init()

Description

Initializes the static file system.

Files

`http_server_fs_port_static.h/http_server_fs_port_static.c`

Prototype

```
CPU_BOOLEAN  HTTPs_FS_Init (void)
```

Arguments

None.

Returned Value

- `DEF_OK` , if the file system was initialized.
- `DEF_FAIL` , otherwise.

Notes / Warnings

None.

HTTPs_FS_AddFile()

Description

Adds a file to the static file system.

Files

`http_server_fs_port_static.h/http_server_fs_port_static.c`

Prototype

```
CPU_BOOLEAN  HTTPs_FS_AddFile (CPU_CHAR    *p_name,
                               void        *p_data,
                               CPU_INT32U   size)
```

Arguments

`p_name`

Name of the file.

`p_data`

Pointer to buffer holding file data.

`size`

Size of file, in octets.

Returned Value

- `DEF_OK` , if the file was added.
- `DEF_FAIL` , otherwise.

Notes / Warnings

1. The file name must meet the following criteria:
   (a) begin with a path separator character.
   (b) be no longer than `HTTPs_FS_MAX_PATH_NAME_LEN` .
   (c) not end with a path separator character.
   (d) not duplicate the parent directory of a file already added.
   (e) not duplicate a file already added.

HTTPs_FS_SetTime()

Description

Sets the date/time of files and directories.

Files

`http_server_fs_port_static.h/http_server_fs_port_static.c`

Prototype

```
CPU_BOOLEAN  HTTPs_FS_SetTime (NET_FS_DATE_TIME  *p_time)
```

Arguments

`p_time`

Pointer to date/time to set.

Returned Value

- DEF_OK , if the time is set.
- DEF_FAIL , otherwise.

Notes / Warnings

1. This time will be returned in the directory entry for ALL files and directories.

HTTP Server Hook Functions API

This section provides a reference to the HTTP Server Hook Functions API.

- Connection Objects Initialization
- Receive Request Header Field
- Connection Request
- On Request Body Received Hook
- Request Ready Signal
- Request Ready Poll
- Add Response Header Field
- Get Token Value
- On Response Chunk Hook
- Get Error Document
- Connection Error
- On Transaction Complete Hook
- Connection Close

Connection Objects Initialization

This hook function, if defined, is called by the HTTP Server every time an instance is initialized as shown in the figure HTTP Server Hook Functions . This function should be used to initialize application' objects for a web server instance. For example, this function can be used to Initialize the memory pool and chained list for a session or initialize a periodic timer that checks for expired session and release them if it is the case.

Prototype

```
void  HTTPs_InstanceInitHook (const  HTTPs_INSTANCE  *p_instance,
                              const  void            *p_hook_cfg);
```

Arguments

p_instance

Pointer to the instance structure (read only).

p_hook_cfg

Pointer to hook application data.

Return Values

None.

Required Configuration

See section Hook Configuration .

Notes / Warnings

None.

Example Template

Listing - HTTPs Instance Initialization Hook Example Code

```
typedef struct HTTPs_Session HTTPs_SESSION;
struct HTTPs_Session{
 CPU_INT16U     SessionID;
  CLK_TS_SEC     ExpireTS;
  HTTPs_SESSION *NextPtr;
  HTTPs_SESSION *PrevPtr;
};

static MEM_DYN_POOL   SessionPool;
static HTTPs_SESSION *SessionFirstPtr;
static OS_TMR     SesssionIDReleaseTMR;

void HTTPs_InstanceInitHook (const HTTPs_INSTANCE *p_instance,
                             const void          *p_hook_cfg)
{
 CPU_SIZE_T octets_reqd;
 RTOS_ERR   err;

  Mem_DynPoolCreate("HTTPs Instance Session Pool",
                   &SessionPool,                          /* Create the session memory pool.                */
                    DEF_NULL,
                    sizeof(HTTPs_SESSION),
                    sizeof(CPU_ALIGN),
                    HTTPs_USER_LOGGED_MAX_NBR,
                    HTTPs_USER_LOGGED_MAX_NBR,
                   &err);
                          /* Set the first pointer to NULL which indicate ... */
  SessionFirstPtr = DEF_NULL;                            /* there is no active session.            */

                                          /* Create and Start the timer which check for ...       */
                                          /* releasing expired session.                          */
  OSTmrCreate(&SesssionIDReleaseTMR,
              HTTPs_SESSION_RELEASE_TMR_NAME_STR,
              0,
              OSCfg_TmrTaskRate_Hz,
               OS_OPT_TMR_PERIODIC,
             &HTTPs_LoginReleaseSessionIDTmr,
               DEF_NULL,
             &err);

 OSTmrStart(&SesssionIDReleaseTMR, &err);
}
```

Receive Request Header Field

This hook function, if defined, is called by the HTTP Server during the parsing of a request message as shown in the figure HTTP Server Hook Functions . It allows the upper application to choose which header field must be stored to be read and used later in other hook functions.

Prototype

```
void HTTPs_ReqHdrRxHook (const HTTPs_INSTANCE  *p_instance,
                    const HTTPs_CONN      *p_conn,
                    const void            *p_hook_cfg,
                         HTTP_HDR_FIELD    hdr_field);
```

Arguments

`p_instance`

Pointer to the instance structure (read only).

`p_conn`

Pointer to the connection structure (read only).

`p_hook_cfg`

Pointer to hook application data.

`hdr_field`

Header field type received.

Return Values

- `DEF_YES`

The header field must be processed and the value stored.

- `DEF_NO`

The header field must be discarded.

Required Configuration

`HTTPs_CFG_HDR_EN` configuration must be enabled in `http-s_cfg.h` . See section Compile-time Configurations .

The header feature must also be enabled in the Instance configuration (see section Instance Configuration).

See section Hook Configuration .

Notes / Warnings

- The instance structure is for read-only. It *must not* be modified.
- Connection structure should not be modified by this function it should be only read to determine if the header type must be stored.

Example Template

The listing below is shown to demonstrate the HTTP Server module capabilities. This code simply ensures that cookies header are stored.

Listing - Request Header Received Hook Example Code

```
static  CPU_BOOLEAN  HTTPs_ReqHdrRxHook (const  HTTPs_INSTANCE   *p_instance,
                                        const  HTTPs_CONN       *p_conn,
                                        const  void             *p_hook_cfg,
                                               HTTP_HDR_FIELD    hdr_field)
{
  switch (hdr_field) {
    case HTTPs_HDR_FIELD_COOKIE:
        case HTTPs_HDR_FIELD_COOKIE2:
             return(DEF_YES);

        default:
             return(DEF_NO);
  }

  return (DEF_NO);
}
```

Connection Request

This hook function, if defined, is called by the HTTP Server module every time a new request has been received and processed as shown in the figure HTTP Server Hook Functions . This function can restrict the access to some resource by analyzing the connection parameter such as remote IP address, the method, resource requested, HTTP header fields, etc. It is also possible in this hook function to redirect to another file. If the upper application requires memory or any specific resource to process the request, it should be allocated within this hook function.

Prototype

```
CPU_BOOLEAN  HTTPs_ReqHook(const  HTTPs_INSTANCE  *p_instance,
                                  HTTPs_CONN      *p_conn,
                           const  void            *p_hook_cfg);
```

Arguments

`p_instance`

Pointer to the instance structure (read only).

`p_conn`

Pointer to the connection structure.

`p_hook_cfg`

Pointer to hook application data.

Return Values

- `DEF_YES,`

If the client is authorized to access the requested resource

- `DEF_NO,`

otherwise, Status code returned will be set automatically to *unauthorized*

Required Configuration

See section Hook Configuration .

Notes / Warnings

- The instance structure is for read-only. It must not be modified at any point in this hook function.
- The following connection attributes can be accessed to analyze the connection (see section Control Structures for further details on each parameters):
  - ClientAddr
  - Method
  - PathPtr
  - ReqHdrCtr
  - ReqHdrFirstPtr
- In this hook function, only the under-mentioned connection parameters are allowed to be modified (see section Control Structures for further details on each parameters):
  - StatusCode
  - PathPtr
  - FilePtr
  - FileLen
  - BodyDataType
  - ConnDataPtr

Example Template

Listing below is shown to demonstrate the HTTP Server module capabilities. That code simply read all header field and print the cookie header field value.

Listing - Request Hook Example Code

```
static  CPU_BOOLEAN  HTTPs_ReqHook(const  HTTPs_INSTANCE  *p_instance,
                                          HTTPs_CONN      *p_conn,
                                   const  void            *p_hook_cfg)
{
#if (HTTPs_CFG_HDR_EN == DEF_ENABLED)
  HTTPs_HDR_BLK  *p_req_hdr_blk;
#endif

#if (HTTPs_CFG_HDR_EN == DEF_ENABLED)
  p_req_hdr_blk = p_conn->ReqHdrFirstPtr;

  while (p_req_hdr_blk != DEF_NULL) {
    switch (p_req_hdr_blk->HdrField) {
      case HTTP_HDR_FIELD_COOKIE:
        printf("cookie received: %s\n", (CPU_CHAR *)p_req_hdr_blk->ValPtr);
        break;

      default:
        break;
    }
    p_req_hdr_blk = p_req_hdr_blk->HdrBlkNextPtr;
  }
#endif

  return (DEF_OK);
}
```

On Request Body Received Hook

This hook function, if defined, is called by the HTTP Server stack when data is received in the HTTP request body. It allows the upper application to parse the data received to save it or take action on it. The hook function will NOT be called when the HTTP server receives a POST request containing a form. In that case, the server core will take care of parsing the body and saving the data into CGI field blocks. See section HTTP Server Hook Functions for the diagram showing at what moment the hook is called in the HTTP transaction processing.

Prototype

```
CPU_BOOLEAN HTTPs_ReqBodyRxHook (const HTTPs_INSTANCE *p_instance,
                                       HTTPs_CONN      *p_conn,
                                 const void            *p_hook_cfg,
                                       void            *p_buf,
                                 const CPU_SIZE_T       buf_size,
                                       CPU_SIZE_T      *p_buf_size_used);
```

Arguments

`p_instance`

Pointer to the instance structure (read only).

`p_conn`

Pointer to the connection structure (read only).

`p_hook_cfg`

Pointer to hook application data.

`p_buf`

Pointer to the data buffer.

`buf_size`

Size of the data received available inside the buffer.

`p_buf_size_used`

Pointer to the variable that will received the length of the data consumed by the app.

Return Value

- `DEF_YES`

If the data have been consumed by the application.

- `DEF_NO`

Otherwise.

Required Configuration

See section Hook Configuration .

Notes / Warnings

- The instance structure is read-only. It *must not* be modified.
- Connection structure *must not* be modified by this function since the response is mostly ready to be transmitted.

Example Template

The listing below is shown to demonstrate the HTTP Server module capabilities. That code simply copies the data received in an application buffer.

Listing - Add Header field hook function example code

```
#define  APP_BUF_LEN   4096

CPU_CHAR  AppBuf[APP_BUF_LEN];

static  CPU_BOOLEAN  HTTPs_ReqBodyRxHook (const  HTTPs_INSTANCE  *p_instance,
                                          HTTPs_CONN      *p_conn,
                                    const  void           *p_hook_cfg,
                      void           *p_buf,
                                    const  CPU_SIZE_T      buf_size,
                                          CPU_SIZE_T      *p_buf_size_used)
{
 static  CPU_INT16U  buf_ix = 0;
         CPU_INT16U  len;

 len = DEF_MIN(buf_size, (APP_BUF_LEN - buf_ix));
 if (len == 0) {
  return (DEF_NO);
 }

 Mem_Copy(&AppBuf[buf_ix], p_buf, len);

  *p_buf_size_used = len;
  buf_ix          += len;

 return (DEF_YES);
}
```

Request Ready Signal

This hook function, if defined, is called by the HTTP Server after an HTTP request has been completely received as shown in the figure HTTP Server Hook Functions . It allows the upper application to do whatever it needs with data received in the request body. The hook function SHOULD NOT be blocking and SHOULD return quickly. A time consuming function will block the processing of the other connections and reduce the HTTP server performance. In case the request processing is time consuming, the Poll hook function SHOULD be enabled to allow the server to periodically verify if the upper application has finished the request processing. If the hook is not required by the upper application, it can be set as `DEF_NULL` and no function will be called.

Prototype

```
CPU_BOOLEAN  HTTPs_ReqRdySignalHook (const  HTTPs_INSTANCE  *p_instance,
                                            HTTPs_CONN      *p_conn,
                                     const  void            *p_hook_cfg,
                                     const  HTTPs_KEY_VAL   *p_data);
```

Arguments

`p_instance`

Pointer to the instance structure (read only).

`p_conn`

Pointer to the connection structure.

`p_hook_cfg`

Pointer to hook application data.

`p_data`

- Pointer to the first control key value pair, if any form was received.

- `DEF_NULL`, otherwise.

Return Values

- `DEF_YES`

  If the response can be sent.

- `DEF_NO`

  if the response cannot be sent after this call and Poll function MUST be called before sending the response

Required Configuration

See section Hook Configuration .

Notes / Warnings

- The instance structure is read-only. It must not be modified.
- The following connection attributes can be accessed to analyze the connection (see `HTTPs_CONN` in Control Structures for further details on each parameters):
  - ClientAddr
  - Method
  - PathPtr
  - ReqHdrCtr
  - ReqHdrFirstPtr
  - ConnDataPtr
- In this hook function, only the under-mentioned connection parameters are allowed to be modified (see `HTTPs_CONN` in Control Structures for further details on each parameters):
  - StatusCode
  - PathPtr
  - FilePtr
  - FileLen
  - BodyDataType
- If the request data take a while to be processed:
  - The processing should be done in a separate task and not in this callback function to avoid blocking other connections.
  - The poll callback function should be used to allow the connection to poll periodically the upper application and verify if the request processing has been completed.
    Note that The ConnDataPtr attribute inside de `HTTPs_CONN` structure can be used to store a semaphore pointer related or anything else that can help to determine the completion of the request processing.

Example Template

The listing below is shown to demonstrate the HTTP Server module capabilities. That code analyses each key value pair in the list and try to find control name which is linked to led blinking functionality and change the document returned to display the led state using a data in memory as well.

Listing - CGI post hook function example code

```
static CPU_BOOLEAN HTTPs_ReqRdySignalHook (const HTTPs_INSTANCE *p_instance,
                                          HTTPs_CONN *p_conn,
                                          const void *p_hook_cfg,
                                          const HTTPs_KEY_VAL *p_data)
{
        HTTPs_KEY_VAL  *p_key_val;
 static CPU_CHAR       buf[4];
        CPU_INT16S     str_cmp;
        CPU_SIZE_T     str_len;
 static CPU_BOOLEAN    led1_val;
 static CPU_BOOLEAN    led2_val;
        CPU_BOOLEAN    led_val;
        RTOS_ERR       err;

   p_key_val = (HTTPs_KEY_VAL *)p_data;

   while (p_key_val != DEF_NULL) {                                      (1)

   printf("Ctrl Name = %s, Value = %s\n\r",
           p_key_val->KeyPtr,
           p_key_val->ValPtr);

   if (p_key_val->DataType == HTTPs_KEY_VAL_TYPE_PAIR) {                (2)

    str_cmp = Str_Cmp_N(p_key_val->KeyPtr, "LED", p_key_val->KeyLen);     (3)

    if (str_cmp == 0) {

     str_cmp = Str_Cmp_N(p_key_val->ValPtr, "LED1", p_key_val->ValLen);   (4)

     if (str_cmp == 0) {
          led1_val = !led1_val;
          led_val  = led1_val;
        } else {
          str_cmp = Str_Cmp_N(p_key_val->ValPtr, "LED2", p_key_val->ValLen);        if (str_cmp == 0) {
            led2_val = !led2_val;
            led_val  = led2_val;
          }
     }

     if (led_val == DEF_ON) {                                          (6)
          str_len       = Str_Len("ON");
          Str_Copy(&buf[0], "ON");
        } else {
          str_len       = Str_Len("OFF");
          Str_Copy(&buf[0], "OFF");
        }

         HTTPs_RespBodySetParamStaticData(p_instance,                  (7)
                                          p_conn,
                                          HTTP_CONTENT_TYPE_PLAIN,
                                          &buf[0],
                                          str_len,
                                          DEF_NO,
                                          &err);
   }

   } else if (p_key_val->DataType == HTTPs_KEY_VAL_TYPE_FILE) {         (8)

       HTTPs_RespBodySetParamFile(p_instance,                           (9)
                                  p_conn,
                                  p_key_val->ValPtr,
                                  HTTP_CONTENT_TYPE_UNKNOWN,
                                  DEF_YES,
                                  &err);
    }
```

```
    p_key_val = p_key_val->DataNextPtr;}return(DEF_NO);}
```

1. Analyze each key value pair into the list
2. Make sure the key value pair item is a control key and not a file.
3. If the control name starts with the key word 'LED' than a led should be toggled.
4. If the value contains 'LED1' than the first led should be toggled.
5. Else if the value contains 'LED2' than the second led should be toggled.
6. Update memory data to transmit based on the led state.
7. Set the data to send in the response body.
8. If the key value pair is for a file uploaded.
9. Change the file to be transmitted equal to the file just received.

Request Ready Poll

The Poll hook function SHOULD be enable in case the request processing require lots of time. It allows the HTTP server to periodically poll the upper application and verify if the request processing has finished, as shown in the figure HTTP Server Hook Functions . If the Poll feature is not required, this field SHOULD be set as `DEF_NULL` .

Prototype

```
CPU_BOOLEAN  HTTPs_ReqRdyPollHook (const  HTTPs_INSTANCE  *p_instance,
                                   HTTPs_CONN      *p_conn,
                              const  void          *p_hook_cfg);
```

Arguments

`p_instance`

Pointer to the instance structure (read only).

`p_conn`

Pointer to the connection structure.

`p_hook_cfg`

Pointer to hook application data.

Return Values

- `DEF_YES`

If the request processing is completed and the response can be sent.

- `DEF_NO`

If the request processing is not completed and the Poll function must be called again to know the end of processing.

Required Configuration

See section Hook Configuration .

Notes / Warnings

- The instance structure is for read-only. It must not be modified.
- The following connection attributes can be accessed to analyze the connection (see `HTTPs_CONN` in Control Structures for further details on each parameters):
  - ClientAddr
  - Method
  - PathPtr

  - ReqHdrCtr
  - ReqHdrFirstPtr
  - ConnDataPtr
- In this hook function, only the under-mentioned connection parameters are allowed to be modified (see `HTTPs_CONN` in Control Structures for further details on each parameters):
  - StatusCode
  - PathPtr
  - FilePtr
  - FileLen
  - BodyDataType
- If request data take a while to be processed:

The poll callback function should be used to allow the HTTP server core to poll periodically the upper application and verify if the request data processing has been completed. Note that The ConnDataPtr attribute inside the `HTTPs_CONN` structure can be used to store a semaphore pointer related to the completion of the request processing.

Example Template

The listing below demonstrates the HTTP Server module capabilities. That code just returns the state of the request processing state, assuming it's possible to post only one processing at a time.

Listing - CGI poll hook function example code

```
static  CPU_BOOLEAN  req_processing_state = DEF_YES;

static  CPU_BOOLEAN  HTTPs_ReqRdyPollHook (const  HTTPs_INSTANCE  *p_instance,
                                          HTTPs_CONN      *p_conn,
                                    const  void          *p_hook_cfg)
{
  return (req_processing_state)
}
```

Add Response Header Field

This hook function, if defined, is called by the HTTP Server module before starting to transmit the response message as shown on HTTP Server Hook Functions . It allows the upper application to add header field to the response by calling a specific API function with header field type and value.

Prototype

```
CPU_BOOLEAN  HTTPs_RespHdrTxHook (const  HTTPs_INSTANCE  *p_instance,
                                         HTTPs_CONN      *p_conn,
                                  const  void          *p_hook_cfg);
```

Arguments

`p_instance`

Pointer to the instance structure (read only).

`p_conn`

Pointer to the connection structure (read only).

`p_hook_cfg`

Pointer to hook application data.

Return Value

- `DEF_YES`

If all header fields are added without error.

- `DEF_NO`

otherwise.

Required Configuration

`HTTPs_CFG_HDR_EN` must be enabled in the `http_server_cfg.h` file (see section Module Configuration).

The header feature must also be enabled in the Instance configuration (see section Instance Configuration).

See section Hook Configuration .

Notes / Warnings

- The instance structure is for read-only. It *must not* be modified.
- Connection structure *must not* be modified by this function since the response is mostly ready to be transmitted.

Example Template

The listing below is shown to demonstrate the HTTP Server module capabilities. That code simply add two header fields to a response message for an HTML document.

Listing - Add Header field hook function example code

```
static  CPU_BOOLEAN  HTTPs_RespHdrTxHook (const  HTTPs_INSTANCE  *p_instance,
                                         HTTPs_CONN      *p_conn,
                                   const  void           *p_hook_cfg)
{
#if (HTTPs_CFG_HDR_EN == DEF_ENABLED)
         HTTPs_HDR_BLK  *p_resp_hdr_blk;
  const  HTTPs_CFG      *p_cfg;
         CPU_CHAR       *str_data;
         CPU_SIZE_T      str_len;
         RTOS_ERR        http_err;

  p_cfg = p_instance->CfgPtr;

    switch (p_conn->StatusCode) {
      case HTTPs_STATUS_OK:                                        (1)
          if (p_conn->FileContentType == HTTPs_CONTENT_TYPE_HTML) {            (2)
              p_resp_hdr_blk = HTTPs_RespHdrGet(p_instance,                (3)
                                                p_conn,
                                                HTTPs_HDR_FIELD_SET_COOKIE,
                                                HTTPs_HDR_VAL_TYPE_STR_DYN,
                                               &http_err);
              if (p_resp_hdr_blk == DEF_NULL) {
                  return(DEF_FAIL);
              }

        str_data = "user=micrium";
              str_len = Str_Len_N(str_data, p_cfg->RespHdrStrLenMax);
              Str_Copy_N(p_resp_hdr_blk->ValPtr,                          (4)
                         str_data,
                         str_len);

              p_resp_hdr_blk->ValLen = str_len;

              p_resp_hdr_blk = HTTPs_RespHdrGet(p_instance,                (5)
                                                p_conn,
                                                HTTPs_HDR_FIELD_SERVER,
                                                HTTPs_HDR_VAL_TYPE_STR_DYN,
                                               &http_err);
              if (p_resp_hdr_blk == DEF_NULL) {
                  return(DEF_FAIL);
              }

              str_data = "uC-HTTPs V2.00.00";
              str_len = Str_Len_N(str_data, p_cfg->RespHdrStrLenMax);          (6)
              Str_Copy_N(p_resp_hdr_blk->ValPtr,
                         str_data,
                         str_len);

              p_resp_hdr_blk->ValLen = str_len;
          }
          break;

        default:
          break;
  }
#endif

  return (DEF_YES);
}
```

1. Ensure that we don't add header field to a response with an error
2. Make sure to add header field only on HTML document.
3. Acquire and add a first header field block of type 'Cookie' and with a string value type to the connection

4. Set the value string value (cookie content) and the string length.
5. Acquire and add a second header field block of type 'Server' and with a string value type to the connection
6. Set the value string value (server name) and the string length.

Get Token Value

This hook function, if defined, is called by the HTTP Server when a token is found into a plain document such as HTML during the response construct process as shown in the figure HTTP Server Hook Functions . It allows the upper application to populate some part of a document dynamically.

Prototype

```
CPU_BOOLEAN HTTPs_RespTokenValGetHook(const HTTPs_INSTANCE *p_instance,
                                      HTTPs_CONN     *p_conn,
`                   const void         *p_hook_cfg,`
                 const CPU_CHAR      *p_token,
                                      CPU_INT16U     token_len,
                                      CPU_CHAR       *p_val,
                                      CPU_INT16U     val_len_max);
```

Arguments

`p_instance`

Pointer to the instance structure (read only).

`p_conn`

Pointer to the connection structure.

`p_hook_cfg`

Pointer to hook application data.

`p_token`

Pointer to the string that contains the token name.

`token_len`

Length of the token.

`p_val`

Pointer to string where to copy the token substitution value.

`val_len_max`

Maximum string value length.

Return Values

- `DEF_OK`

If token value copied successfully.

- `DEF_FAIL`

Otherwise (see Notes).

Required Configuration

See Token Replacement Compile-time Configuration and Token Replacement Run-time Configuration .

Notes / Warnings

- The instance structure is for read-only. It *must not* be modified.
- Connection structure *must not* be modified by this function since the response transmission is started.
- If the token replacement failed, the token will be replaced by a line of tilde (~) of length equal to `val_len_max`.

Example Template

The listing below demonstrates the HTTP Server module capabilities. That code just look if the token is equal to " `TEXT_STRING` " and replace it by " `TEXT` " else it returns an error and let the server replace the token by the default value.

Listing - CGI poll hook function example code

```
static CPU_BOOLEAN HTTPs_RespTokenValGetHook (const HTTPs_INSTANCE *p_instance,
                                             HTTPs_CONN     *p_conn,
                  const void         *p_hook_cfg,
                  const CPU_CHAR      *p_token,
                                             CPU_INT16U      token_len,
                                             CPU_CHAR       *p_val,
                                             CPU_INT16U      val_len_max)
{
 if (Str_Cmp_N(p_token, "TEXT_STRING", 11) == 0) {
     Str_Copy_N(p_val, "Text", val_len_max);
     return (DEF_OK);
 }

 return (DEF_FAIL);
}
```

On Response Chunk Hook

To allow the upper application to transmit data with the Chunked Transfer Encoding, this hook function is available. If defined, it will be called at the moment of the Response body transfer (as shown in section HTTP Server Hook Functions), and it will be called until the application has transfer all its data. If the hook is not required by the upper application, it can be set as `DEF_NULL` and no function will be called.

Furthermore, for the hook to be called, the RespBodyDataType parameter of `HTTPs_CONN` must be set to `HTTPs_BODY_DATA_TYPE_STATIC_DATA` and the DataPtr parameter must be null. See section Response Body Data for more details on how to send application data in chunk.

Prototype

```
CPU_BOOLEAN HTTPs_RespChunkDataGetHook(const HTTPs_INSTANCE *p_instance,
                                      HTTPs_CONN     *p_conn,
                                const void          *p_hook_cfg,
                                      void          *p_buf,
                                      CPU_SIZE_T      buf_len_max,
                                      CPU_SIZE_T     *p_tx_len);
```

Arguments

`p_instance`

Pointer to the instance structure (read only).

`p_conn`

Pointer to the connection structure.

`p_hook_cfg`

Pointer to hook application data.

`p_buf`

Pointer to the buffer to fill.

`buf_len_max`

Maximum length the buffer can contain.

`p_tx_len`

Pointer to variable that will received the length written in the buffer.

Return Values

- `DEF_YES`

if there is no more data to send.

- `DEF_NO`

Otherwise.

Required Configuration

See section Hook Configuration .

Notes / Warnings

- The instance structure is for read-only. It *must not* be modified.
- Connection structure *must not* be modified by this function. All the parameters to change are already passed as arguments.

Example Template

The listing below demonstrates the HTTP Server module capabilities.

Listing - CGI poll hook function example code

```
static CPU_BOOLEAN HTTPs_RespChunkDataGetHook (const HTTPs_INSTANCE *p_instance,
                                               HTTPs_CONN      *p_conn,
                     const void          *p_hook_cfg,
                           void          *p_buf,
                                               CPU_SIZE_T       buf_len_max,
                                               CPU_SIZE_T      *p_tx_len)
{
#if (HTTPs_CFG_FORM_EN == DEF_ENABLED)
   const HTTPs_CFG      *p_cfg = p_instance->CfgPtr;
        HTTPs_KEY_VAL  *p_key_val;
        CPU_INT16S      str_cmp;

   str_cmp = Str_Cmp_N(p_conn->PathPtr, FORM_SUBMIT_URL, p_conn->PathLenMax);
   if (str_cmp == 0) {
                                                 /* Construct JSON for user                          */
      Str_Copy(p_buf, "{\"user\": {\"first name\": \"");     /* Add First Name field.                         */
      p_key_val = p_conn->FormDataListPtr;
      while (p_key_val != DEF_NULL) {
         str_cmp = Str_Cmp_N(p_key_val->KeyPtr, "firstname", p_cfg->FormCfgPtr->KeyLenMax);
         if (str_cmp == 0) {
            Str_Cat_N(p_buf, p_key_val->ValPtr, p_key_val->ValLen);
            break;
         }
         p_key_val = p_key_val->NextPtr;
      }
      Str_Cat(p_buf, "\", \"last name\":\"");              /* Add Last Name field.                          */
      p_key_val = p_conn->FormDataListPtr;
      while (p_key_val != DEF_NULL) {
         str_cmp = Str_Cmp_N(p_key_val->KeyPtr, "lastname", p_cfg->FormCfgPtr->KeyLenMax);
         if (str_cmp == 0) {
            Str_Cat_N(p_buf, p_key_val->ValPtr, p_key_val->ValLen);
            break;
         }
         p_key_val = p_key_val->NextPtr;
      }
      Str_Cat(p_buf, "\"}}");
   }
  *p_tx_len = Str_Len_N(p_buf, p_cfg->BufLen);
#else
    CPU_SW_EXCEPTION(;);
#endif
   return (DEF_YES);
}
```

Get Error Document

This hook function, if defined, is called by the HTTP Server instance every time the response status code has been changed to a value that is associated with an error to allow the upper application to send the appropriate error page as shown in the figure HTTP Server Hook Functions . The change could be the result of the request processing in the Connection request hook function or the result of an internal error in the HTTP Server core. This function is intended to set the name of the file which will be sent with the response message. If no file is set, a default status page will be sent including the status code number and the reason phrase.

Prototype

```
void HTTPs_ErrFileGetHook (const void            *p_hook_cfg,
                           HTTP_STATUS_CODE      status_code,
                           CPU_CHAR              *p_file_str,
                           CPU_INT32U            file_len_max,
                           HTTPs_BODY_DATA_TYPE  *p_file_type,
                           HTTP_CONTENT_TYPE     *p_content_type,
                           void                  **p_data,
                           CPU_INT32U            *p_data_len);
```

Arguments

`p_hook_cfg`

Pointer to hook application data.

`status_code`

Status code number of the response message.

`p_file_str`

Pointer to the string buffer where the filename string must be copied.

`file_len_max`

Maximum string length of the filename.

`p_file_type`

Pointer to the variable holding the file type, can be modified by this function when data must be transmitted instead of a file:

- `HTTPs_BODY_DATA_TYPE_FILE`
- `HTTPs_BODY_DATA_TYPE_STATIC_DATA`
- `HTTPs_BODY_DATA_TYPE_NONE`

`p_content_type`

Content type of the body. If the data is a File, the content type doesn't need to be set. It will be set according to the file extension. If the data is Static Data, the parameter **MUST** be set.

`p_data`

- Pointer to the data memory block,
- Must set if file type is configured as `HTTPs_BODY_DATA_TYPE_STATIC_DATA`.
- `DEF_NULL`, otherwise.

`p_data_len`

- Pointer to variable holding the length of the data,
- Must set if file type is configured as `HTTPs_FILE_TYPE_STATIC_DATA`.
- `DEF_NULL`, otherwise

Return Values

None.

Required Configuration

See section Hook Configuration .

Notes / Warnings

If the configured file doesn't exist the instance will transmit the default web page instead, defined by `“HTTPs_CFG_HTML_DFLT_ERR_PAGE”` in `http_server_cfg.h` .

Example Template

The listing below demonstrates the HTTP Server module capabilities. That code simply use a file for 404 error and the default document for any other error.

Listing - Connection get error document hook function example code

```
#define  HTTPs_CFG_INSTANCE_STR_FILE_ERR_404           "404.html"

static  void  HTTPs_ErrFileGetHookt (const  void               *p_hook_cfg,
                                     HTTPs_STATUS_CODE      status_code,
                                     CPU_CHAR             *p_file_str,
                                     CPU_INT32U           file_len_max,
                                     HTTPs_BODY_DATA_TYPE  *p_file_type,
                                     HTTP_CONTENT_TYPE     *p_content_type,
                                     void               **p_data,
                                     CPU_INT32U           *p_data_len)
{
  switch (status_code) {
    case HTTP_STATUS_NOT_FOUND:
         Str_Copy_N(p_file_str, HTTPs_CFG_INSTANCE_STR_FILE_ERR_404, file_len_max);
        *p_file_type = HTTPs_BODY_DATA_TYPE_FILE;
         return;

      default:
        *p_data        = HTTPs_CFG_HTML_DFLT_ERR_PAGE;
        *p_data_len     = HTTPs_HTML_DLFT_ERR_LEN;
        *p_file_type    = HTTPs_BODY_DATA_TYPE_STATIC_DATA;
        *p_content_type = HTTP_CONTENT_TYPE_HTML;
         return;
  }

}
```

Connection Error

This hook function, if defined, is called every time an error occurred when processing the request or the method to notify the upper application and change the default behavior as shown shown in the figure HTTP Server Hook Functions . This function can analyses the error and change the status code and the file to return.

Prototype

```
void  HTTPs_ErrHook (const  HTTPs_INSTANCE  *p_instance,
                            HTTPs_CONN     *p_conn,
                     const  void          *p_hook_cfg,
                            HTTPs_ERR       err);
```

Arguments

`p_instance`

Pointer to the instance structure (read only).

`p_conn`

Pointer to the connection structure.

`p_hook_cfg`

Pointer to hook application data.

`err`

Error code .

Return Values

None.

Required Configuration

See section Hook Configuration .

Notes / Warnings

- The instance structure is for read-only. It must not be modified at any point in this hook function.
- The following connection attributes can be accessed to analyze the connection (see HTTPs_CONN in Control Structures for further details on each parameters):
  - ClientAddr
  - Method
  - PathPtr
  - HdrCtr
  - HdrListPtr
- In this hook function, only the under-mentioned connection parameters are allowed to be modified (see HTTPs_CONN in Control Structures for further details on each parameters):
  - StatusCode
  - PathPtr
  - DataPtr
  - DataLen
  - BodyDataType
  - ConnDataPtr

Example Template

The listing below demonstrates the HTTP Server module capabilities. That code simply read error and print file not found errors.

Listing - Error Hook Example Code

```
static void HTTPs_ErrHook (const HTTPs_INSTANCE *p_instance,
                          HTTPs_CONN      *p_conn,
                       const void         *p_hook_cfg,
                    HTTPs_ERR       err)
{
 switch (err) {
  case HTTPs_ERR_FILE_404_NOT_FOUND:
    printf("Error 404 File not found occurred.\n");
        return;

      default:
        break;
 }
}
```

On Transaction Complete Hook

This hook function, if defined, is called every time a HTTP transaction is completed to notify the upper application to release resource allocated related to a transaction, as shown in the figure HTTP Server Hook Functions .

Prototype

```
void HTTPs_TransCompleteHook (const HTTPs_INSTANCE *p_instance,
                              HTTPs_CONN     *p_conn,
                        const void           *p_hook_cfg);
```

Arguments

`p_instance`

Pointer to the instance structure (read only).

`p_conn`

Pointer to the connection structure.

`p_hook_cfg`

Pointer to hook application data.

Return Values

None.

Required Configuration

See section Hook Configuration .

Notes / Warnings

- The instance structure is for read-only. It must not be modified.
- The connection structure is read-only since the connection will be freed after this call.

ConnDataPtr parameter should be used to store the location of the data allocated.

Example Template

The listing below is shown to demonstrate the HTTP Server module capabilities. That code simply print that the HTTP transaction is completed. The hook will be usually used to release objects allocated for the duration of an HTTP transaction.

Listing - Transaction Complete Hook Example Code

```
static void HTTPs_TransCompleteHook (const HTTPs_INSTANCE *p_instance,
                                     HTTPs_CONN     *p_conn,
                  const void           *p_hook_cfg)
{
  printf("HTTP transaction is completed.\n\r");
}
```

Connection Close

This hook function, if defined, is called every time a connection is closed to notify the upper application to release resource allocated related to an HTTP connection, as shown in the figure HTTP Server Hook Functions .

Prototype

```
void  HTTPs_ConnCloseHook (const  HTTPs_INSTANCE  *p_instance,
                                HTTPs_CONN      *p_conn,
                          const  void           *p_hook_cfg);
```

Arguments

`p_instance`

Pointer to the instance structure (read only).

`p_conn`

Pointer to the connection structure.

`p_hook_cfg`

Pointer to hook application data.

Return Values

None.

Required Configuration

See Hook Configuration .

Notes / Warnings

- The instance structure is for read-only. It must not be modified.
- The connection structure is read-only since the connection will be freed after this call.

ConnDataPtr parameter should be used to store the location of the data allocated.

Example Template

The listing below is shown to demonstrate the HTTP Server module capabilities. That code simply print that the connection is closing.

Listing - Connection Close Hook Example Code

```
static  void  HTTPs_ConnCloseHook (const  HTTPs_INSTANCE  *p_instance,
                                        HTTPs_CONN      *p_conn,
                                  const  void           *p_hook_cf)
{
 printf("Connection is closing.\n\r");
}
```

HTTP Server Add-on API

This section provides a reference to the HTTP Server add-on modules API.

- Auth Module API
- REST Module API

Auth Module API

The Authentication add-on requires you to set three hook functions to work properly:

- Get Required Rights Hook
- Login Hook
- Logout Hook

You will also need to use some functions from the Authentication module of Micrium OS Common to create users and right levels.

Get Required Rights Hook

This hook will be called by the Authentication Layer of the Control Layer when a request is received to check with the upper application what rights are associated with this request.

Prototype

```
AUTH_RIGHT  HTTPsAuth_GetRequiredRightsHook (const  HTTPs_INSTANCE  *p_instance,
                                             const  HTTPs_CONN      *p_conn);
```

Arguments

`p_instance`

Pointer to the instance structure (read only).

`p_conn`

Pointer to the connection structure (read only).

Return Values

The authorization right level for this HTTP transaction.

Required Configuration

None.

Notes / Warnings

None.

Example Template

Listing - Get Required Rights Hook Example

```
#define  HTTP_USER_ACCESS  AUTH_RIGHT_2

/*
*********************************************************************************************************
*                              AppGlobal_Auth_GetRequiredRightsHook()
*
* Description : Returns the rights required to fulfill the needs of a given request.
*
* Argument(s) : p_instance  Pointer to the HTTP server instance object.
*
*               p_conn      Pointer to the HTTP connection object.
*
* Return(s)   : returns the authorization right level for this example application.
*
* Note(s)     : none.
*********************************************************************************************************
*/

AUTH_RIGHT  AppGlobal_Auth_GetRequiredRightsHook (const  HTTPs_INSTANCE  *p_instance,
                                                  const  HTTPs_CONN      *p_conn)
{
    return (HTTP_USER_ACCESS);
}
```

Login Hook

This hook will be called by the Application Layer of the Control Layer to found requests related to the log in process, e.g., the GET requests for the resources of the *log in* page (html, css, images) and the POST request with the *log in* form.

This hook functions are called on two occasions:

- When the URL and headers of the request were received and parse by the server (state `HTTPs_AUTH_STATE_REQ_URL`)
- When the request body was received and parse (state `HTTPs_AUTH_STATE_REQ_COMPLETE`)

Prototype

```
CPU_BOOLEAN  AppGlobal_Auth_ParseLoginHook (const  HTTPs_INSTANCE    *p_instance,
                                            const  HTTPs_CONN        *p_conn,
                                                   HTTPs_AUTH_STATE   state,
                                                   HTTPs_AUTH_RESULT  *p_result);
```

Arguments

`p_instance`

Pointer to the instance structure (read only).

`p_conn`

Pointer to the connection structure (read only).

`state`

State of the Authentication module:

- `HTTPs_AUTH_STATE_REQ_URL`
- `HTTPs_AUTH_STATE_REQ_COMPLETE`

`p_result`

Pointer to the authentication result structure that the hook needs to fill.

Return Values

- `DEF_YES`, if the request is the POST with the log in information in a form.
- `DEF_NO`, otherwise.

Required Configuration

None.

Notes / Warnings

None.

Example Template

Listing - Login Hook Example

```
/*
*********************************************************************************************************
*                               AppGlobal_Auth_ParseLoginHook()
*
* Description : (1) Check all the HTTP requests received to see if they are related to resources of the login page.
*                   (a) Check if the POST login is received.
*                   (b) For each request set the redirect paths on no, invalid & valid credentials.
*                   (c) Parse the form fields received in the body for the user and password.
*
*
* Argument(s) : p_instance  Pointer to the HTTP server instance object.
*
*               p_conn      Pointer to the HTTP connection object.
*
*               state       State of the Authentication module:
*                               HTTPs_AUTH_STATE_REQ_URL:      The url and the headers were received and parse.
*                               HTTPs_AUTH_STATE_REQ_COMPLETE: All the request (url + headers + body) was received and parse.
*
*               p_result    Pointer to the authentication result structure to fill.
*
* Return(s)   : DEF_YES, if the request is the POST login.
*               DEF_NO,  otherwise.
*
* Note(s)     : (2) This hook will be called twice for a request processed by the Authentication module:
*                   (a) When the Start line of the request (with the url) and the headers have been
*                       received and parse -> HTTPs_AUTH_STATE_REQ_URL state.
*                   (b) When all the request has been completely received and parse including the body
*                       -> HTTPs_AUTH_STATE_REQ_COMPLETE state.
*
*               (3) for each request received the redirect paths is set as follow:
*                   (a) RedirectPath_OnValidCred    INDEX_PAGE_URL
*                   (b) RedirectPath_OnInvalidCred  LOGIN_PAGE_URL
*                   (c) RedirectPath_OnNoCred
*                       (1) if the path is an unprotected path, let it go. (DEF_NULL) (i.e., the logo)
*                       (2) otherwise               LOGIN_PAGE_URL
*********************************************************************************************************
*/
CPU_BOOLEAN  AppGlobal_Auth_ParseLoginHook (const  HTTPs_INSTANCE     *p_instance,
                                            const  HTTPs_CONN         *p_conn,
                                                   HTTPs_AUTH_STATE    state,
                                                   HTTPs_AUTH_RESULT  *p_result)
{
    CPU_INT16S      cmp_val;
    CPU_BOOLEAN     is_login = DEF_NO;
#if (HTTPs_CFG_FORM_EN == DEF_ENABLED)
    HTTPs_KEY_VAL  *p_current;
    CPU_INT16S      cmp_val_username;
    CPU_INT16S      cmp_val_password;
#endif
    p_result->UsernamePtr = DEF_NULL;
    p_result->PasswordPtr = DEF_NULL;
                                                                /* Set redirect paths for each requests.              */
    p_result->RedirectPathOnInvalidCredPtr = LOGIN_PAGE_URL;
    p_result->RedirectPathOnValidCredPtr   = INDEX_PAGE_URL;
    switch (state) {
                                                                /* --------------- REQUEST URL RECEIVED --------------- */
        case HTTPs_AUTH_STATE_REQ_URL:
                                                                /* Set redirect paths for each requests.              */
             cmp_val = Str_Cmp(p_conn->PathPtr, LOGIN_PAGE_URL);
             if (cmp_val != 0) {
                 cmp_val = Str_Cmp(p_conn->PathPtr, MICRIUM_LOGO_URL);
                 if (cmp_val != 0) {
                     cmp_val = Str_Cmp(p_conn->PathPtr, MICRIUM_CSS_URL);
                 }
             }
             p_result->RedirectPathOnNoCredPtr = (cmp_val == 0) ? DEF_NULL : LOGIN_PAGE_URL;
                                                                /* Check if POST login received.                      */
```

```
        cmp_val = Str_Cmp(p_conn->PathPtr, LOGIN_PAGE_URL);if(cmp_val != 0){
           cmp_val = Str_Cmp(p_conn->PathPtr, MICRIUM_LOGO_URL);if(cmp_val != 0){
              cmp_val = Str_Cmp(p_conn->PathPtr, MICRIUM_CSS_URL);}}
        p_result->RedirectPathOnNoCredPtr = (cmp_val == 0) ? DEF_NULL : LOGIN_PAGE_URL;/* Check if POST login received.            */
        cmp_val = Str_Cmp(p_conn->PathPtr, LOGIN_PAGE_CMD);if(cmp_val == 0){
           is_login = DEF_YES;}break;/* -------------- REQUEST BODY RECEIVED --------------- */
    case HTTPs_AUTH_STATE_REQ_COMPLETE:
#if(HTTPs_CFG_FORM_EN == DEF_ENABLED)/* Parse form fields received for user/password.       */
        p_current = p_conn->FormDataListPtr;while((p_current          != DEF_NULL)&&((p_result->UsernamePtr == DEF_NULL)||(p_result-
>PasswordPtr == DEF_NULL))){if(p_current->DataType == HTTPs_KEY_VAL_TYPE_PAIR){
              cmp_val_username = Str_CmpIgnoreCase_N(p_current->KeyPtr,
                                              FORM_USERNAME_FIELD_NAME,
                                             p_current->KeyLen);
              cmp_val_password = Str_CmpIgnoreCase_N(p_current->KeyPtr,
                                              FORM_PASSWORD_FIELD_NAME,
                                             p_current->KeyLen);if(cmp_val_username == 0){
                 p_result->UsernamePtr = p_current->ValPtr;
                 is_login = DEF_YES;}elseif(cmp_val_password == 0){
                 p_result->PasswordPtr = p_current->ValPtr;}}
           p_current = p_current->NextPtr;}
#endif
        break;
    default:break;}return(is_login);}
```

Logout Hook

This hook will be called by the Application Layer of the Control Layer to find out if the request received is a *log out* request. Depending on the implementation of the this hook, the log out accepted can be a GET or a POST request.

This hook functions will be called on two occasions:

- When the URL and headers of the request were received and parse by the server (state `HTTPs_AUTH_STATE_REQ_URL`)
- When the request body was received and parse (state `HTTPs_AUTH_STATE_REQ_COMPLETE`)

Prototype

```
CPU_BOOLEAN  AppGlobal_Auth_ParseLogoutHook (const  HTTPs_INSTANCE    *p_instance,
                                             const  HTTPs_CONN        *p_conn,
                                                    HTTPs_AUTH_STATE   state);
```

Arguments

`p_instance`

Pointer to the instance structure (read only).

`p_conn`

Pointer to the connection structure (read only).

`state`

State of the Authentication module:

- `HTTPs_AUTH_STATE_REQ_URL`
- `HTTPs_AUTH_STATE_REQ_COMPLETE`

Return Values

- `DEF_YES` , if the request is the POST with the log out information in a form.
- `DEF_NO` , otherwise.

Required Configuration

None.

Notes / Warnings

None.

Example Template

Listing - Logout Hook Example

```
/*
*********************************************************************************************************
*                             AppGlobal_Auth_ParseLogoutHook()
*
* Description : Parse requests received for logout URL and form data logout info.
*
* Argument(s) : p_instance  Pointer to HTTPs instance object.
*
*               p_conn      Pointer to HTTPs connection object.
*
*               state       State of the Authentication module:
*                               HTTPs_AUTH_STATE_REQ_URL:      The URL and the headers were received and parse.
*                               HTTPs_AUTH_STATE_REQ_COMPLETE: All the request (URL + headers + body) was received and parse.
*
* Return(s)   : DEF_YES, if Logout received.
*               DEF_NO,  otherwise.
*
* Note(s)     : (1) This hook will be called twice for a request processed by the Authentication module:
*                   (a) When the Start line of the request (with the URL) and the headers have been
*                       received and parse -> HTTPs_AUTH_STATE_REQ_URL state.
*                   (b) When all the request has been completely received and parse including the body
*                       -> HTTPs_AUTH_STATE_REQ_COMPLETE state.
*********************************************************************************************************
*/
CPU_BOOLEAN  AppGlobal_Auth_ParseLogoutHook (const  HTTPs_INSTANCE   *p_instance,
                                             const  HTTPs_CONN       *p_conn,
                                                    HTTPs_AUTH_STATE   state)
{
    CPU_BOOLEAN     is_logout = DEF_NO;
#if (HTTPs_CFG_FORM_EN == DEF_ENABLED)
    HTTPs_KEY_VAL  *p_current;
    CPU_INT16S      cmp_val;
#endif
    switch (state) {
                                                                /* --------------- REQUEST URL RECEIVED --------------- */
        case HTTPs_AUTH_STATE_REQ_URL:
                                                                /* Check if POST logout received.                      */
             cmp_val = Str_Cmp(p_conn->PathPtr, LOGOUT_PAGE_CMD);
             if (cmp_val == 0) {
                 is_logout = DEF_YES;
             }
             break;
                                                                /* -------------- REQUEST BODY RECEIVED --------------- */
        case HTTPs_AUTH_STATE_REQ_COMPLETE:
#if (HTTPs_CFG_FORM_EN == DEF_ENABLED)
                                                                /* Parse form fields received for logout.              */
             p_current = p_conn->FormDataListPtr;
             while (p_current != DEF_NULL) {
                 if (p_current->DataType == HTTPs_KEY_VAL_TYPE_PAIR) {
                     cmp_val = Str_CmpIgnoreCase_N(p_current->KeyPtr,
                                                   FORM_LOGOUT_FIELD_NAME,
                                                  p_current->KeyLen);
                     if (cmp_val == 0) {
                         is_logout = DEF_YES;
                         break;
                     }
                 }
                 p_current = p_current->NextPtr;
             }
#endif
             break;
        default:
             break;
    }
    return (is_logout);
}
```

REST Module API

The REST add-on module has only one API function to publish, in the global resources list, REST resources created by the application.

- `HTTPsREST_Publish`

> This function can only be called after the HTTP server initialization and before the HTTP server start. Also, the `HTTPsREST_Init()` hook function must be bound to the HTTP server or to the Control layer prior the initialization. Otherwise, the REST memory pools won't be initialized and the resource lists cannot be created.

For each REST resources, a set of hook functions will also need to be defined. For more information refer to section REST module .

HTTPsREST_Publish

Prototype

```
void  HTTPsREST_Publish (const HTTPs_REST_RESOURCE  *p_resource,
                         CPU_INT32U           list_ID,
                         HTTPs_REST_ERR       *p_err);
```

Arguments

`p_resource`

Pointer to the rest resource to publish (read only).

`list_ID`

Identification of the list to publish on.

`p_err`

Pointer to variable that will receive the return error code from this function

Return Values

None.

Required Configuration

None.

Notes / Warnings

None.

Example Template

Listing - REST Publish Resources Example

```
/*
*********************************************************************************************************
*                                   AppREST_ResourcesInit()
*
* Description : Initialize the REST application resources.
*
* Argument(s) : none.
*
* Return(s)   : DEF_OK,   if initialization was successful.
*               DEF_FAIL, otherwise.
*
* Note(s)     : none.
```

```
*****************************************************************************************************************
*/
CPU_BOOLEAN  AppREST_ResourcesInit (void){
   RTOS_ERR  err;HTTPsREST_Publish(&AppREST_List_Resource,0,&err);if(err.Code != RTOS_ERR_NONE)
{return(DEF_FAIL);}HTTPsREST_Publish(&AppREST_User_Resource,0,&err);if(err.Code != RTOS_ERR_NONE)
{return(DEF_FAIL);}HTTPsREST_Publish(&AppREST_File_Resource,0,&err);if(err.Code != RTOS_ERR_NONE){return(DEF_FAIL);}return(DEF_OK);}
```

MQTT Client API

MQTT Client API

- MQTTc_ConfigureTaskStk()
- MQTTc_ConfigureMemSeg()
- MQTTc_ConfigureQty()
- MQTTc_Init()
- MQTTc_TaskPrioSet()
- MQTTc_TaskDlySet()
- MQTTc_InactivityTimeoutDfltSet()
- MQTTc_ConnSetParam()
- MQTTc_ConnClose()
- MQTTc_ConnClr()
- MQTTc_Connect()
- MQTTc_ConnOpen()
- MQTTc_Disconnect()
- MQTTc_MsgClr()
- MQTTc_MsgSetParam()
- MQTTc_PingReq()
- MQTTc_Publish()
- MQTTc_Subscribe()
- MQTTc_SubscribeMult()
- MQTTc_Unsubscribe()
- MQTTc_UnsubscribeMult()

MQTTc_ConfigureTaskStk()

Description

Configure the MQTT client task stack properties to use the parameters contained in the passed structure instead of the default parameters.

Files

`mqtt_client.h/mqtt_client.c`

Prototype

```
void  HTTPc_ConfigureTaskStk (CPU_STK_SIZE   stk_size_elements,
                              void          *p_stk_base)
```

Arguments

`stk_size_elements`

Size of the stack, in `CPU_STK` elements.

`p_stk_base`

Pointer to base of the stack.

Returned Value

None.

Notes / Warnings

1. This function is optional. If it is called, it must be called before `MQTTc_Init()`. If it is not called, default values will be used to initialize the module.

MQTTc_ConfigureMemSeg()

Description

Configure the memory segment that will be used to allocate internal data needed by MQTT client module instead of the default memory segment.

Files

`mqtt_client.h/mqtt_client.c`

Prototype

```
void MQTTc_ConfigureMemSeg (MEM_SEG *p_mem_seg)
```

Arguments

`p_mem_seg`

Pointer to the memory segment from which the internal data will be allocated. If `DEF_NULL`, the internal data will be allocated from the global Heap.

Returned Value

None.

Notes / Warnings

1. This function is optional. If it is called, it must be called before `MQTTc_Init()`. If it is not called, default values will be used to initialize the module.

MQTTc_ConfigureQty()

Description

Overwrite the quantity configuration object for MQTT client.

Files

`mqtt_client.h/mqtt_client.c`

Prototype

```
void MQTTc_ConfigureQty (MQTTc_QTY_CFG *p_qty_cfg)
```

Arguments

`p_qty_cfg`

Pointer to structure containing the quantity parameters.

Returned Value

None.

Notes / Warnings

1. This function is optional. If it is called, it must be called before `MQTTc_Init()`. If it is not called, default values will be used to initialize the module.

MQTTc_Init()

Description

Initializes the MQTT client module.

Files

`mqtt_client.h/mqtt_client.c`

Prototype

```
void  MQTTc_Init (const  MQTTc_CFG      *p_cfg,
            const  RTOS_TASK_CFG  *p_task_cfg,
                 MEM_SEG        *p_mem_seg,
                 RTOS_ERR       *p_err)
```

Arguments

`p_cfg`

Pointer to MQTT Client Configuration Object.

`p_task_cfg`

Pointer to task configuration structure.

`p_mem_seg`

Memory segment from which internal data will be allocated. If `DEF_NULL`, it will be allocated from the global heap.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`

Returned Value

None.

Notes / Warnings

None.

MQTTc_TaskPrioSet()

Description

Sets priority of the MQTT client task.

Files

`mqtt_client.h/mqtt_client.c`

Prototype

```
void MQTTc_TaskPrioSet (RTOS_TASK_PRIO   prio,
                  RTOS_ERR       *p_err)
```

Arguments

`prio`

New priority for the MQTT client task.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function.

Returned Value

None.

Notes / Warnings

None.

MQTTc_TaskDlySet()

Description

Sets task delay.

Files

`mqtt_client.h/mqtt_client.c`

Prototype

```
void MQTTc_TaskDlySet (CPU_INT08U   dly_ms,
                       RTOS_ERR    *p_err)
```

Arguments

`dly_ms`

New delay in millisecond for the MQTT client task.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function.

Returned Value

None.

Notes / Warnings

None.

MQTTc_InactivityTimeoutDfltSet()

Description

Sets default inactivity timeout in second of the MQTT client connection.

Files

`mqtt_client.h/mqtt_client.c`

Prototype

```
void MQTTc_InactivityTimeoutDfltSet (CPU_INT16U   inactivity_timeout_s,
                                     RTOS_ERR    *p_err)
```

Arguments

`inactivity_timeout_s`

New default timeout in second to set on new MQTT connection.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function.

Returned Value

None.

Notes / Warnings

1. It is possible to set each connection with a different inactivity timeout by calling `MQTTc_ConnSetParam()` before opening the connection.

MQTTc_ConnSetParam()

Description

Sets parameters related to the TCP and MQTT Client Connection.

Files

`mqtt_client.h/mqtt_client.c`

Prototype

```
void  MQTTc_ConnSetParam (MQTTc_CONN       *p_conn,
                          MQTTc_PARAM_TYPE   type,
                          void              *p_param,
                          RTOS_ERR          *p_err)
```

Arguments

`p_conn`

Pointer to the current MQTTc Connection.

`type`

Parameter type :

- `MQTTc_PARAM_TYPE_BROKER_IP_ADDR` Broker's IP address.
- `MQTTc_PARAM_TYPE_BROKER_NAME` Broker's name.
- `MQTTc_PARAM_TYPE_BROKER_PORT_NBR` Broker's port number.
- `MQTTc_PARAM_TYPE_INACTIVITY_TIMEOUT_S` Inactivity timeout (in seconds).
- `MQTTc_PARAM_TYPE_CLIENT_ID_STR` Client ID string.
- `MQTTc_PARAM_TYPE_USERNAME_STR` Client username string.
- `MQTTc_PARAM_TYPE_PASSWORD_STR` Client password string.
- `MQTTc_PARAM_TYPE_KEEP_ALIVE_TMR_SEC` Keep alive timer (in seconds).
- `MQTTc_PARAM_TYPE_WILL_CFG_PTR` Configures a pointer, if any.
- `MQTTc_PARAM_TYPE_CALLBACK_ON_COMPL` Generic callback on completion
- `MQTTc_PARAM_TYPE_CALLBACK_ON_CONNECT_CMPL` Callback on connection completion.
- `MQTTc_PARAM_TYPE_CALLBACK_ON_PUBLISH_CMPL` Callback on publish completion.
- `MQTTc_PARAM_TYPE_CALLBACK_ON_SUBSCRIBE_CMPL` Callback on subscribe completion.
- `MQTTc_PARAM_TYPE_CALLBACK_ON_UNSUBSCRIBE_CMPL` Callback on unsubscribe completion.
- `MQTTc_PARAM_TYPE_CALLBACK_ON_PINGREQ_CMPL` Callback on ping request completion.
- `MQTTc_PARAM_TYPE_CALLBACK_ON_DISCONNECT_CMPL` Callback on disconnection

- `MQTTc_PARAM_TYPE_CALLBACK_ON_PUBLISH_RX` Callback once publish is received.
- `MQTTc_PARAM_TYPE_CALLBACK_ARG_PTR` Pointer on an argument passed to callback.
- `MQTTc_PARAM_TYPE_TIMEOUT_MS` 'Open' timeout (in milliseconds).
- `MQTTc_PARAM_TYPE_PUBLISH_RX_MSG_PTR` Pointer on the message that is used to receive the publish.

`p_param`

Parameter's value.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

None.

MQTTc_ConnClose()

Description

Requests to close a MQTT client Connection.

Files

`mqtt_client.h/mqtt_client.c`

Prototype

```
void  MQTTc_ConnClose (MQTTc_CONN   *p_conn,
                 MQTTc_FLAGS   flags,
                 RTOS_ERR     *p_err)
```

Arguments

`p_conn`

Pointer to the MQTT client Connection to close.

`flags`

Configuration flags, reserved for future usage.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_OS_SCHED_LOCKED`

- RTOS_ERR_ABORT
- RTOS_ERR_TIMEOUT

Returned Value

None.

Notes / Warnings

None.

MQTTc_ConnClr()

Description

Clears an MQTT client Connection before its first usage.

Files

mqtt_client.h/mqtt_client.c

Prototype

```
void  MQTTc_ConnClr (MQTTc_CONN  *p_conn,
                     RTOS_ERR    *p_err)
```

Arguments

p_conn

Pointer to the MQTTc Connection.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function :

- RTOS_ERR_NONE

Returned Value

None.

Notes / Warnings

1. This function MUST be called before the MQTTc_CONN object is used for the first time.

MQTTc_Connect()

Description

Sends a 'Connect' message to the MQTT server.

Files

mqtt_client.h/mqtt_client.c

Prototype

```
void  MQTTc_Connect (MQTTc_CONN  *p_conn,
                     MQTTc_MSG   *p_msg,
                     RTOS_ERR    *p_err)
```

Arguments

`p_conn`

Pointer to the MQTT client Connection to use.

`p_msg`

Pointer to the MQTT client Message object to use.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_NULL_PTR`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

None.

Notes / Warnings

None.

MQTTc_ConnOpen()

Description

Opens a new MQTT Client connection.

Files

`mqtt_client.h/mqtt_client.c`

Prototype

```
void  MQTTc_ConnOpen (MQTTc_CONN    *p_conn,
                      MQTTc_FLAGS    flags,
                      RTOS_ERR      *p_err)
```

Arguments

`p_conn`

Pointer to the MQTT client Connection object to open.

`flags`

Configuration flags, reserved for future usage.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_TYPE`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_NOT_SUPPORTED`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_FAIL`

- RTOS_ERR_NET_INVALID_ADDR_SRC
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_NET_IF_LINK_DOWN
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_INVALID_STATE
- RTOS_ERR_ABORT
- RTOS_ERR_TIMEOUT
- RTOS_ERR_NET_OP_IN_PROGRESS
- RTOS_ERR_TX
- RTOS_ERR_ALREADY_EXISTS
- RTOS_ERR_NOT_FOUND
- RTOS_ERR_NET_INVALID_CONN
- RTOS_ERR_RX
- RTOS_ERR_NOT_READY
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_NET_ADDR_UNRESOLVED
- RTOS_ERR_NET_NEXT_HOP
- RTOS_ERR_NET_CONN_CLOSED_FAULT

Returned Value

None.

Notes / Warnings

None.

MQTTc_Disconnect()

Description

Sends a 'Disconnect' message to the MQTT server.

Files

mqtt_client.h/mqtt_client.c

Prototype

```
void  MQTTc_Disconnect (MQTTc_CONN  *p_conn,
                        MQTTc_MSG   *p_msg,
                        RTOS_ERR    *p_err)
```

Arguments

p_conn

Pointer to the MQTT client Connection object to use.

p_msg

Pointer to the MQTT client Message object to use.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function :

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_HANDLE

Returned Value

None.

Notes / Warnings

None.

MQTTc_MsgClr()

Description

Clears the Message object members.

Files

`mqtt_client.h/mqtt_client.c`

Prototype

```
void  MQTTc_MsgClr (MQTTc_MSG  *p_msg,
                    RTOS_ERR   *p_err)
```

Arguments

`p_msg`

Pointer to the message object to clear.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

None.

MQTTc_MsgSetParam()

Description

Sets the parameter related to a given MQTT Message.

Files

`mqtt_client.h/mqtt_client.c`

Prototype

```
void  MQTTc_MsgSetParam (MQTTc_MSG         *p_msg,
                         MQTTc_PARAM_TYPE   type,
                         void              *p_param,
                         RTOS_ERR          *p_err)
```

Arguments

`p_msg`

Pointer to a message object.

`type`

Parameter type :

- `MQTTc_PARAM_TYPE_MSG_BUF_PTR` Msg's buf ptr.
- `MQTTc_PARAM_TYPE_MSG_BUF_LEN` Msg's buf len.

`p_param`

Parameter's value.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

None.

MQTTc_PingReq()

Description

Sends a 'PingReq' message to the MQTT server.

Files

`mqtt_client.h/mqtt_client.c`

Prototype

```
void  MQTTc_PingReq (MQTTc_CONN  *p_conn,
                     MQTTc_MSG   *p_msg,
                     RTOS_ERR    *p_err)
```

Arguments

`p_conn`

Pointer to the MQTT client Connection object to use.

`p_msg`

Pointer to the MQTT client Message object to use.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

None.

Notes / Warnings

None.

MQTTc_Publish()

Description

Sends a 'Publish' message to the MQTT server.

Files

`mqtt_client.h/mqtt_client.c`

Prototype

```
void  MQTTc_Publish (       MQTTc_CONN    *p_conn,
                            MQTTc_MSG     *p_msg,
                      const CPU_CHAR      *topic_str,
                            CPU_INT08U     qos_lvl,
                            CPU_BOOLEAN    retain_flag,
                      const CPU_INT08U    *p_payload,
                            CPU_INT32U     payload_len,
                            RTOS_ERR      *p_err)
```

Arguments

`p_conn`

Pointer to the MQTT client Connection object to use.

`p_msg`

Pointer to the MQTT client Message object to use.

`topic_str`

String containing the topic on which to publish. Must stay valid until the message has been completely sent.

`qos_lvl`

Level of QoS at which to publish.

`retain_flag`

Flag that indicates if the retain flag in the PUBLISH header needs to be set.

`p_payload`

Pointer to the payload to publish. Must stay valid until the message has been completely sent.

`payload_len`

Length, in bytes, of the payload.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

None.

Notes / Warnings

None.

MQTTc_Subscribe()

Description

Sends a 'Subscribe' message to the MQTT server.

Files

`mqtt_client.h/mqtt_client.c`

Prototype

```
void  MQTTc_Subscribe (      MQTTc_CONN  *p_conn,
                             MQTTc_MSG   *p_msg,
                       const CPU_CHAR    *topic_str,
                             CPU_INT08U   req_qos,
                             RTOS_ERR    *p_err)
```

Arguments

`p_conn`

Pointer to the MQTT client Connection object to use.

`p_msg`

Pointer to the MQTT client Message object to use.

`topic_str`

String containing the topic at which to subscribe. Must stay valid until the message has been completely sent.

`req_qos`

Requested level of QoS for this subscription.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

None.

Notes / Warnings

None.

MQTTc_SubscribeMult()

Description

Sends a 'Subscribe' message containing multiple topics to MQTT server.

Files

`mqtt_client.h/mqtt_client.c`

Prototype

```
void MQTTc_SubscribeMult (      MQTTc_CONN   *p_conn,
                                 MQTTc_MSG    *p_msg,
                           const CPU_CHAR    **topic_str_tbl,
                                 CPU_INT08U   *req_qos_tbl,
                                 CPU_INT08U    topic_nbr,
                                 RTOS_ERR     *p_err)
```

Arguments

`p_conn`

Pointer to the MQTT client Connection object to use.

`p_msg`

Pointer to the MQTT client Message object to use.

`topic_str_tbl`

Table containing string of all the topic(s) at which to subscribe. Must all stay valid until the message has been completely sent.

`req_qos_tbl`

Table of the requested level of QoS for each subscription.

`topic_nbr`

Number of topics and QoS' contained in tables.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

None.

Notes / Warnings

None.

MQTTc_Unsubscribe()

Description

Sends an 'Unsubscribe' message to the MQTT server.

Files

`mqtt_client.h/mqtt_client.c`

Prototype

```
void MQTTc_Unsubscribe (      MQTTc_CONN  *p_conn,
                               MQTTc_MSG   *p_msg,
                         const CPU_CHAR    *topic_str,
                               RTOS_ERR    *p_err)
```

Arguments

`p_conn`

Pointer to the MQTT client Connection object to use.

`p_msg`

Pointer to the MQTT client Message object to use.

`topic_str`

String containing the topic at which to unsubscribe. Must stay valid until the message has been completely sent.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

None.

Notes / Warnings

None.

MQTTc_UnsubscribeMult()

Description

Sends an 'Unsubscribe' message for multiple topics to the MQTT server.

Files

`mqtt_client.h/mqtt_client.c`

Prototype

```
void  MQTTc_UnsubscribeMult (       MQTTc_CONN   *p_conn,
                                    MQTTc_MSG    *p_msg,
                              const  CPU_CHAR   **topic_str_tbl,
                                    CPU_INT08U    topic_nbr,
                                    RTOS_ERR     *p_err)
```

Arguments

`p_conn`

Pointer to the MQTT client Connection object to use.

`p_msg`

Pointer to the MQTT client Message object to use.

`topic_str_tbl`

Table containing string of all the topic(s) at which to unsubscribe. Must stay valid until the message has been completely sent.

`topic_nbr`

Number of topic contained in tables.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

None.

Notes / Warnings

None.

SMTP Client API

SMTP Client API

- SMTPc_ConfigureMemSeg()
- SMTPc_ConfigureAuthBufLen()
- SMTPc_Init()
- SMTPc_DfltCfgSet()
- SMTPc_MsgAlloc()
- SMTPc_MsgFree()
- SMTPc_MsgClr()
- SMTPc_MsgSetParam()
- SMTPc_SendMail()
- SMTPc_Connect()
- SMTPc_Disconnect()
- SMTPc_SendMsg()

SMTPc_ConfigureMemSeg()

Description

Configure the memory segment that will be used to allocate internal data needed by SMTP client module instead of the default memory segment.

Files

`smtp_client.h/smtp_client.c`

Prototype

```
void SMTPc_ConfigureMemSeg (MEM_SEG *p_mem_seg)
```

Arguments

`p_mem_seg`

Pointer to the memory segment from which the internal data will be allocated. If `DEF_NULL`, the internal data will be allocated from the global Heap.

Returned Value

None.

Notes / Warnings

1. This function is optional. If it is called, it must be called before `SMTPc_Init()`. If it is not called, default values will be used to initialize the module.

SMTPc_ConfigureAuthBufLen()

Description

Configure the size of internal authentication input buffer.

Files

smtp_client.h/smtp_client.c

Prototype

```
void SMTPc_ConfigureAuthBufLen (CPU_INT16U buf_len)
```

Arguments

buf_len

Input buffer length to configure.

Returned Value

None.

Notes / Warnings

1. This function is optional. If it is called, it must be called before SMTPc_Init() . If it is not called, default values will be used to initialize the module.
2. The maximum input buffer passed to the base 64 encoder depends of the configured maximum lengths for the username and password. Two additional characters are added to these values to account for the delimiter.
3. The size of the output buffer the base 64 encoder produces is typically bigger than the input buffer by a factor of (4 x 3). However, when padding is necessary, up to 3 additional characters could by appended. Finally, one more character is used to NULL terminate the buffer.

SMTPc_Init()

Description

Initialize the SMTP client module.

Files

smtp_client.h/smtp_client.c

Prototype

```
void SMTPc_Init (RTOS_ERR *p_err)
```

Arguments

p_err

Pointer to the variable that will receive one of the following error code(s) from this function.

Returned Value

None.

Notes / Warnings

None.

SMTPc_DfltCfgSet()

Description

Sets the Default Configuration that will be used for opening TCP connections inside the SMTP client module.

Files

smtp_client.h/smtp_client.c

Prototype

```
void SMTPc_DfltCfgSet (CPU_INT16U  timeout_conn_ms,
                       CPU_INT16U  timeout_rx_ms,
                       CPU_INT16U  timeout_tx_ms,
                       CPU_INT16U  timeout_close_ms,
                       RTOS_ERR   *p_err)
```

Arguments

`timeout_conn_ms`

Timeout on connection request.

`timeout_rx_ms`

Timeout on receive.

`timeout_tx_ms`

Timeout on transmit.

`timeout_close_ms`

Timeout on socket close

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function.

Returned Value

None.

Notes / Warnings

1. The new configuration value are applied only on new connection.

SMTPc_MsgAlloc()

Description

Get a message structure.

Files

`smtp_client.h/smtp_client.c`

Prototype

```
SMTPc_MSG  *SMTPc_MsgAlloc (RTOS_ERR  *p_err)
```

Arguments

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function.

Returned Value

Pointer to an allocated message structure.

Notes / Warnings

1. If the message structure is allocated from the application. The function `SMTPc_MsgClr()` must be called to free internal structure allocated during message processing.

SMTPc_MsgFree()

Description

Free a message structure.

Files

`smtp_client.h/smtp_client.c`

Prototype

```
void SMTPc_MsgFree (SMTPc_MSG *p_msg,
                    RTOS_ERR *p_err)
```

Arguments

`p_msg`

Pointer to the message structure to free.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function.

Returned Value

None.

Notes / Warnings

None.

SMTPc_MsgClr()

Description

Clear a message structure to be reused for another message.

Files

`smtp_client.h/smtp_client.c`

Prototype

```
void SMTPc_MsgClr (SMTPc_MSG *p_msg,
                   RTOS_ERR  *p_err)
```

Arguments

`p_msg`

Pointer to the message structure to clear.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function.

Returned Value

None.

Notes / Warnings

1. If the message structure is allocated from the application. The function `SMTPc_MsgClr()` must be called to free internal structure allocated during message processing.

SMTPc_MsgSetParam()

Description

Configure message parameter.

Files

`smtp_client.h/smtp_client.c`

Prototype

```
void SMTPc_MsgSetParam (SMTPc_MSG        *p_msg,
                        SMTPc_MSG_PARAM   param_type,
                        void             *p_val,
                        RTOS_ERR         *p_err)
```

Arguments

`p_msg`

Pointer to the message structure to configure.

`param_type`

Parameter to set in the message:

- `SMTPc_FROM_ADDR`
- `SMTPc_FROM_DISPL_NAME`
- `SMTPc_TO_ADDR`
- `SMTPc_CC_ADDR`
- `SMTPc_BCC_ADDR`
- `SMTPc_REPLY_TO_ADDR`
- `SMTPc_SENDER_ADDR`
- `SMTPc_ATTACHMENT`
- `SMTPc_MSG_SUBJECT`
- `SMTPc_MSG_BODY`

`p_val`

Pointer to the parameter value.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function.

Returned Value

None.

Notes / Warnings

None.

SMTPc_SendMail()

Description

Sends an email.

Files

`smtp_client.h/smtp_client.c`

Prototype

```
void  SMTPc_SendMail (CPU_CHAR                *p_host_name,
                      CPU_INT16U               port,
                      CPU_CHAR                *p_username,
                      CPU_CHAR                *p_pwd,
                      NET_APP_SOCK_SECURE_CFG *p_secure_cfg,
                      SMTPc_MSG               *p_msg,
                      RTOS_ERR                *p_err)
```

Arguments

`p_host_name`

Pointer to host name of the SMTP server to contact. Can be also an IP address.

`port`

TCP port to use, or '0' if `SMTPc_DFLT_PORT`.

`p_username`

Pointer to user name, if authentication enabled.

`p_pwd`

Pointer to password, if authentication enabled.

`p_secure_cfg`

Pointer to the secure configuration (TLS/SSL):

- `DEF_NULL`, if no security enabled.
- Pointer to a structure that contains the parameters.

`p_msg`

SMTPc_MSG structure encapsulating the message to send.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function.

Returned Value

None.

Notes / Warnings

1. The function `SMTPc_MsgSetParam()` must be called before it can send a message.

SMTPc_Connect()

Description

Establishes a TCP connection to the SMTP server and initiates the SMTP session.

Files

`smtp_client.h/smtp_client.c`

Prototype

```
NET_SOCK_ID  SMTPc_Connect (CPU_CHAR              *p_host_name,
                            CPU_INT16U             port,
                            CPU_CHAR              *p_username,
                            CPU_CHAR              *p_pwd,
                            NET_APP_SOCK_SECURE_CFG  *p_secure_cfg,
                            RTOS_ERR              *p_err)
```

Arguments

`p_host_name`

Pointer to hostname of the SMTP server to contact (can be an IP address).

`port`

TCP port to use, or enter '0' if `SMTPc_DFLT_PORT`.

`p_username`

Pointer to user name, if authentication is enabled.

`p_pwd`

Pointer to password, if authentication is enabled.

`p_secure_cfg`

Pointer to the secure configuration (TLS/SSL):

- `DEF_NULL`, if no security is enabled.
- Pointer to a structure that contains the parameters.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function.

Returned Value

- Socket descriptor/handle identifier, if NO error.
- -1, otherwise.

Notes / Warnings

1. Network security manager MUST be available and enabled to initialize the server in secure mode.
2. If anything goes wrong while trying to connect to the server, the socket is closed by calling `NetSock_Close`. In case of a failure to establish the TCP connection, all data structures are returned to their original state.
   If the failure occurs when initiating the session, the application is responsible to take the appropriate action(s).
3. If authentication is disabled (`SMTPc_CFG_AUTH_EN` set to `DEF_ENABLED`), the '`p_username`' and '`p_pwd`' parameters should be passed as a NULL pointer.
4. The server will send a 220 "Service ready" reply when the connection is completed. The SMTP protocol allows a server to formally reject a transaction, while still allowing the initial connection by responding with a 554 "Transaction failed" reply.
5. The Plain-text (PLAIN) authentication mechanism is implemented here. However, it takes some liberties from RFC #4964, section 4 'The AUTH Command', stating the "A server implementation MUST implement a configuration in which it does not permit any plaintext password mechanisms, unless either the STARTTLS command has been negotiated or some other mechanism that protects the session from password snooping has been provided".
   Since this client does not support SSL or TLS, nor any other protection against password snooping, it relies on the server to NOT fully follow RFC #4954 to be successful.

SMTPc_Disconnect()

Description

Closes the connection between the client and the server.

Files

`smtp_client.h/smtp_client.c`

Prototype

```
void  SMTPc_Disconnect (NET_SOCK_ID   sock_id,
                        RTOS_ERR     *p_err)
```

Arguments

`sock_id`

Socket ID.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function.

Returned Value

None.

Notes / Warnings

1. The receiver (client) MUST NOT intentionally close the transmission channel until it receives and replies to a QUIT command.
2. The receiver of the QUIT command MUST send an OK reply, then close the transmission channel.

SMTPc_SendMsg()

Description

Sends a message (an instance of the SMTPc_MSG structure) to the SMTP server.

Files

`smtp_client.h/smtp_client.c`

Prototype

```
void  SMTPc_SendMsg (NET_SOCK_ID   sock_id,
                     SMTPc_MSG    *p_msg,
                     RTOS_ERR     *p_err)
```

Arguments

`sock_id`

Socket ID.

`p_msg`

SMTPc_MSG structure encapsulating the message to send.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function.

Returned Value

None.

Notes / Warnings

1. The function `SMTPc_SetMsg` must be called before it can send a message.
2. The message must have at least one receiver: either "To", "CC", or "BCC".

SNTP Client API

SNTP Client API

- SNTPc_ConfigureMemSeg()
- SNTPc_Init()
- SNTPc_DfltCfgSet()
- SNTPc_ReqRemoteTime()
- SNTPc_GetRemoteTime()
- SNTPc_GetRoundTripDly_us()

SNTPc_ConfigureMemSeg()

Description

Configures the memory segment to use when allocating control data and buffers.

Files

`sntp_client.h/sntp_client.c`

Prototype

```
void  SNTPc_ConfigureMemSeg (MEM_SEG  *p_mem_seg)
```

Arguments

`p_mem_seg`

Pointer to memory segment to use when allocating control data. `DEF_NULL` means general purpose heap segment.

Returned Value

None.

Notes / Warnings

1. Calling this function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the SNTPc client is initialized via the `SNTPc_Init()` function.

SNTPc_Init()

Description

Initializes the SNTPc Module.

Files

`sntp_client.h/sntp_client.c`

Prototype

```
void  SNTPc_Init (RTOS_ERR  *p_err)
```

Arguments

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_OS_ILLEGAL_RUN_TIME
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_BLK_ALLOC_CALLBACK
- RTOS_ERR_SEG_OVF

Returned Value

None.

Notes / Warnings

None.

SNTPc_DfltCfgSet()

Description

Sets the default server configurations.

Files

sntp_client.h/sntp_client.c

Prototype

```
void  SNTPc_DfltCfgSet (NET_PORT_NBR        port_nbr,
                        NET_IP_ADDR_FAMILY  addr_family,
                        CPU_INT32U          rx_timeout_ms,
                        RTOS_ERR           *p_err)
```

Arguments

port_nbr

Port number to use.

addr_family

Address family. Use NET_IP_ADDR_FAMILY_NONE if you are unsure.

rx_timeout_ms

Timeout, in milliseconds, for the reception.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE

Returned Value

- DEF_TRUE , if the new default server configuration is successfully set.
- DEF_FALSE , otherwise.

Notes / Warnings

None.

SNTPc_ReqRemoteTime()

Description

Send a request to an NTP server and receive an SNTPc packet to compute.

Files

`sntp_client.h/sntp_client.c`

Prototype

```
void SNTPc_ReqRemoteTime (CPU_CHAR *hostname,
                         SNTP_PKT  *p_pkt,
                         RTOS_ERR  *p_err)
```

Arguments

`hostname`

String that contains the NTP server hostname.

`p_pkt`

Pointer to an `SNTP_PKT` variable that will contain the received SNTPc packet.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_TYPE`
- `RTOS_ERR_NET_RETRY_MAX`
- `RTOS_ERR_NET_SOCK_CLOSED`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_NOT_SUPPORTED`
- `RTOS_ERR_NET_CONN_CLOSE_RX`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_FAIL`
- `RTOS_ERR_NET_INVALID_ADDR_SRC`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_NET_IF_LINK_DOWN`
- `RTOS_ERR_CODE_GET(err_rtn`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_NET_OP_IN_PROGRESS`
- `RTOS_ERR_TX`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_ALREADY_EXISTS`
- `RTOS_ERR_CODE_GET(local_err`
- `RTOS_ERR_NET_INVALID_CONN`
- `RTOS_ERR_NET_STR_ADDR_INVALID`
- `RTOS_ERR_NOT_READY`
- `RTOS_ERR_POOL_EMPTY`

- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_NET_ADDR_UNRESOLVED
- RTOS_ERR_NET_NEXT_HOP
- RTOS_ERR_NET_CONN_CLOSED_FAULT

Returned Value

None.

Notes / Warnings

None.

SNTPc_GetRemoteTime()

Description

Gets the remote time (NTP timestamp) from a received NTP packet.

Files

sntp_client.h/sntp_client.c

Prototype

```
SNTP_TS  SNTPc_GetRemoteTime (SNTP_PKT  *p_pkt,
                             RTOS_ERR  *p_err)
```

Arguments

p_pkt

Pointer to received SNTP message packet.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE

Returned Value

NTP timestamp.

Notes / Warnings

None.

SNTPc_GetRoundTripDly_us()

Description

Gets SNTP packet round trip delay from a received SNTP message packet.

Files

sntp_client.h/sntp_client.c

Prototype

```
CPU_INT32U  SNTPc_GetRoundTripDly_us (SNTP_PKT   *p_pkt,
                                     RTOS_ERR   *p_err)
```

Arguments

p_kt

Pointer to received SNTP message packet.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE

Returned Value

SNTP packet round trip delay in microseconds.

Notes / Warnings

1. If the round trip delay is faster than the precision of the system clock, then the round trip delay is approximated to 0.
2. Only the integer part of the round trip delay is returned.

FTP Client API

FTP Client API

- FTPc_Open()
- FTPc_Close()
- FTPc_RecvBuf()
- FTPc_RecvFile()
- FTPc_SendBuf()
- FTPc_SendFile()

FTPc_Open()

Description

Open connection to an FTP server.

Files

`ftp_client.h/ftp_client.c`

Prototype

```
CPU_BOOLEAN FTPc_Open (     FTPc_CONN        *p_conn,
                    const FTPc_CFG        *p_cfg,
                    const FTPc_SECURE_CFG *p_secure_cfg,
                       CPU_CHAR         *p_host_server,
                       NET_PORT_NBR      port_nbr,
                       CPU_CHAR         *p_user,
                       CPU_CHAR         *p_pass,
                       RTOS_ERR         *p_err);
```

Arguments

`p_conn`

Pointer to FTP Client Connection object.

`p_cfg`

Pointer to FTPc Configuration object. `DEF_NULL` to use internal default configuration.

`p_secure_cfg`

Pointer to a secure configuration structure if secured connection is required or `DEF_NULL` if secured connection is not required.

`p_host_server`

Pointer to hostname/IP address string of the server.

`port_nbr`

IP port of the server.

`p_user`

Pointer to account username on the server. `DEF_NULL` for anonymous connection.

`p_pass`

Pointer to account password on the server. `DEF_NULL` for no password.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function.

Returned Value

- `DEF_FAIL` connection failed.
- `DEF_OK` connection successful.

Notes / Warnings

None.

FTPc_Close()

Description

Close FTP connection.

Files

`ftp_client.h/ftp_client.c`

Prototype

```
CPU_BOOLEAN FTPc_Close (FTPc_CONN  *p_conn,
                        RTOS_ERR   *p_err);
```

Arguments

`p_conn`

Pointer to FTP Client Connection object.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function.

Returned Value

- `DEF_FAIL` FTP connection close failed.
- `DEF_OK` FTP connection close successful.

Notes / Warnings

None.

FTPc_RecvBuf()

Description

Receive a file from an FTP server into a memory buffer.

Files

`ftp_client.h/ftp_client.c`

Prototype

```
CPU_BOOLEAN FTPc_RecvBuf (FTPc_CONN  *p_conn,
                          CPU_CHAR   *p_remote_file_name,
                          CPU_INT08U *p_buf,
                          CPU_INT32U  buf_len,
                          CPU_INT32U *p_file_size,
                          RTOS_ERR   *p_err);
```

Arguments

`p_conn`

Pointer to FTP Client Connection object.

`p_remote_file_name`

Pointer to name of the file in FTP server.

`p_buf`

Pointer to memory buffer to hold received file.

`buf_len`

Size of the memory buffer.

`port_nbr`

IP port of the server.

`p_file_size`

Pointer to a variable that will received the size of the file received.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function.

Returned Value

- `DEF_FAIL` reception failed.
- `DEF_OK` reception successful.

Notes / Warnings

None.

FTPc_RecvFile()

Description

Receive a file from an FTP server to the file system.

Files

`ftp_client.h/ftp_client.c`

Prototype

```
CPU_BOOLEAN FTPc_RecvFile (FTPc_CONN *p_conn,
                           CPU_CHAR  *p_remote_file_name,
                           CPU_CHAR  *p_local_file_name,
                           RTOS_ERR  *p_err);
```

Arguments

`p_conn`

Pointer to FTP Client Connection object.

`p_remote_file_name`

Remote File path.

`p_local_file_name`

Remote File path.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function.

Returned Value

- `DEF_FAIL` reception failed.
- `DEF_OK` reception successful.

Notes / Warnings

None.

FTPc_SendBuf()

Description

Send a memory buffer to an FTP server.

Files

`ftp_client.h/ftp_client.c`

Prototype

```
CPU_BOOLEAN FTPc_SendBuf (FTPc_CONN   *p_conn,
                          CPU_CHAR    *p_remote_file_name,
                          CPU_INT08U  *p_buf,
                          CPU_INT32U   buf_len,
                          CPU_BOOLEAN  append,
                          RTOS_ERR    *p_err)
```

Arguments

`p_conn`

Pointer to FTP Client Connection object.

`p_remote_file_name`

File path on the server

`p_buf`

Pointer to memory buffer to send.

`buf_len`

Size of the memory buffer.

`append`

if `DEF_YES` , existing file on FTP server will be appended with memory buffer. If file doesn't exist on FTP server, it will be created.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function.

Returned Value

- `DEF_FAIL` transmission failed.
- `DEF_OK` transmission successful.

Notes / Warnings

None.

FTPc_SendFile()

Description

Send a file located in the file system to an FTP server.

Files

`ftp_client.h/ftp_client.c`

Prototype

```
CPU_BOOLEAN FTPc_SendFile (FTPc_CONN   *p_conn,
                           CPU_CHAR    *p_remote_file_name,
                           CPU_CHAR    *p_local_file_name,
                           CPU_BOOLEAN  append,
                           RTOS_ERR    *p_err);
```

Arguments

`p_conn`

Pointer to FTP Client Connection object.

`p_remote_file_name`

Remote File path.

`p_local_file_name`

Local File path.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function.

Returned Value

- `DEF_FAIL` transmission failed.
- `DEF_OK` transmission successful.

Notes / Warnings

None.

TFTP Client API

TFTP Client API

- TFTPc_Get()
- TFTPc_Put()

TFTPc_Get()

Description

Gets a file from the TFTP server.

Files

`tftp_client.h/tftp_client.c`

Prototype

```
void TFTPc_Get (      CPU_CHAR      *p_host_server,
            const TFTPc_CFG    *p_cfg,

                  CPU_CHAR     *p_filename_local,
                  CPU_CHAR     *p_filename_remote,
                  TFTPc_MODE    mode,
                  RTOS_ERR     *p_err)
```

Arguments

`p_host_server`

Server host name.

`p_cfg`

- Pointer to TFTP client configuration to use.
- `DEF_NULL` , if the default configuration must be used.

`p_filename_local`

Pointer to name of the file to be written by the client.

`p_filename_remote`

Pointer to name of the file to be read from the server.

`mode`

TFTP transfer mode :

- `TFTPc_MODE_NETASCII` ASCII mode.
- `TFTPc_MODE_OCTET` Binary mode.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_TX
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_TIMEOUT

Returned Value

None.

Notes / Warnings

None.

TFTPc_Put()

Description

Puts a file on the TFTP server.

Files

tftp_client.h/tftp_client.c

Prototype

```
void  TFTPc_Put (      CPU_CHAR     *p_host_server,
            const  TFTPc_CFG    *p_cfg,
                 CPU_CHAR     *p_filename_local,
                 CPU_CHAR     *p_filename_remote,
                 TFTPc_MODE    mode,
                 RTOS_ERR     *p_err)
```

Arguments

p_host_server

Server host name.

p_cfg

Pointer to TFTP client configuration to use. DEF_NULL, if the default configuration must be used.

p_filename_local

Pointer to name of the file to be read by the client.

p_filename_remote

Pointer to name of the file to be written to the server.

mode

TFTP transfer mode :

- TFTPc_MODE_NETASCII ASCII mode.
- TFTPc_MODE_OCTET Binary mode.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_TX
- RTOS_ERR_WOULD_BLOCK

- `RTOS_ERR_TIMEOUT`

Returned Value

None.

Notes / Warnings

None.

TFTP Server API

TFTP Server API

- TFTPs_ConfigureTaskStk()
- TFTPs_ConfigureMemSeg()
- TFTPs_ConfigureConnParam()
- TFTPs_Init()
- TFTPs_TaskPrioSet()

TFTPs_ConfigureTaskStk()

Description

Configure the TFTP server task stack properties to use the parameters contained in the passed structure instead of the default parameters.

Files

`tftp_server.h/tftp_server.c`

Prototype

```
void  TFTPs_ConfigureTaskStk (CPU_STK_SIZE   stk_size_elements,
                              void          *p_stk_base)
```

Arguments

`stk_size_elements`

Size of the stack, in `CPU_STK` elements.

`p_stk_base`

Pointer to base of the stack.

Returned Value

None.

Notes / Warnings

1. This function is optional. If it is called, it must be called before `TFTPs_Init()`. If it is not called, default values will be used to initialize the module.

TFTPs_ConfigureMemSeg()

Description

Configure the memory segment that will be used to allocate internal data needed by the TFTP server module instead of the default memory segment.

Files

`tftp_server.h/tftp_server.c`

Prototype

```
void  TFTPs_ConfigureMemSeg (MEM_SEG  *p_mem_seg)
```

Arguments

`p_mem_seg`

Pointer to the memory segment from which the internal data will be allocated. If `DEF_NULL`, the internal data will be allocated from the global Heap.

Returned Value

None.

Notes / Warnings

1. This function is optional. If it is called, it must be called before `TFTPs_Init()`. If it is not called, default values will be used to initialize the module.

TFTPs_ConfigureConnParam()

Description

Overwrite the Connection configuration object for TFTP server.

Files

`tftp_server.h/tftp_server.c`

Prototype

```
void  TFTPs_ConfigureConnParam (TFTPs_CONN_CFG  *p_conn_cfg)
```

Arguments

`p_conn_cfg`

Pointer to structure containing the connection parameters.

Returned Value

None.

Notes / Warnings

1. This function is optional. If it is called, it must be called before `TFTPs_Init()`. If it is not called, default values will be used to initialize the module.

TFTPs_Init()

Description

Initializes and starts up the TFTP server.

Files

`tftp_server.h/tftp_server.c`

Prototype

```
void  TFTPs_Init (const  CPU_CHAR  *p_root_dir,
                  RTOS_ERR  *p_err)
```

Arguments

`p_root_dir`

Pointer to root directory name for the TFTP server.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_SUPPORTED`
- `RTOS_ERR_SEG_OVF`

Returned Value

None.

Notes / Warnings

None.

TFTPs_TaskPrioSet()

Description

Sets the priority of the given TFTP server task.

Files

`tftp_server.h/tftp_server.c`

Prototype

```
void  TFTPs_TaskPrioSet (RTOS_TASK_PRIO   prio,
                         RTOS_ERR        *p_err)
```

Arguments

`prio`

Priority of the TFTP server task.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_ARG`

Returned Value

None.

Notes / Warnings

None.

Telnet Server API

Telnet Server API

- TELNETs_ConfigureMemSeg()
- TELNETs_Init()
- TELNETs_InstanceInit()
- TELNETs_InstanceStart()

TELNETs_ConfigureMemSeg()

Description

Configure the memory segment that will be used to allocate internal data needed by the TELNET server module instead of the default memory segment.

Files

`telnet_server.h/telnet_server.c`

Prototype

```
void  TELNETs_ConfigureMemSeg (MEM_SEG  *p_mem_seg)
```

Arguments

`p_mem_seg`

Pointer to the memory segment from which the internal data will be allocated. If `DEF_NULL`, the internal data will be allocated from the global Heap.

Returned Value

None.

Notes / Warnings

1. This function is optional. If it is called, it must be called before `TELNETs_Init()`. If it is not called, default values will be used to initialize the module.

TELNETs_Init()

Description

Initializes the TELNET server module.

Files

`telnet_server.h/telnet_server.c`

Prototype

```
void TELNETs_Init (RTOS_ERR  *p_err)
```

Arguments

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_SEG_OVF

Returned Value

None.

Notes / Warnings

1. TELNETs_Init() MUST be called only AFTER the product's OS and network have been initialized.
2. TELNETs_Init() MUST ONLY be called ONCE from the product's application.

TELNETs_InstanceInit()

Description

Initializes a TELNET server instance.

Files

telnet_server.h/telnet_server.c

Prototype

```
TELNETs_INSTANCE *TELNETs_InstanceInit (const  TELNETs_CFG    *p_cfg,
                                       const  RTOS_TASK_CFG  *p_task_srv_cfg,
                                       const  RTOS_TASK_CFG  *p_task_session_cfg,
                                              RTOS_ERR       *p_err);
```

Arguments

p_cfg

Pointer to the instance configuration object.

p_task_srv_cfg

Pointer to the instance task server configuration object.

p_task_session_cfg

Pointer to the instance task session configuration object.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_INVALID_TYPE
- RTOS_ERR_NOT_FOUND
- RTOS_ERR_ALREADY_EXISTS
- RTOS_ERR_NET_INVALID_CONN
- RTOS_ERR_INVALID_STATE
- RTOS_ERR_NET_CONN_CLOSED_FAULT
- RTOS_ERR_OS_ILLEGAL_RUN_TIME

Returned Value

None.

Notes / Warnings

None.

TELNETs_InstanceStart()

Description

Starts a specific TELNET server instance which had been previously initialized.

Files

`telnet_server.h/telnet_server.c`

Prototype

```
void  TELNETs_InstanceStart(TELNETs_INSTANCE  *p_instance,
                            RTOS_ERR          *p_err)
```

Arguments

`p_instance`

Pointer to specific TELNET server instance handler.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`
- `RTOS_ERR_INVALID_TYPE`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_ALREADY_EXISTS`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_NET_INVALID_CONN`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_NET_CONN_CLOSED_FAULT`

Returned Value

None.

Notes / Warnings

None.

IPerf API

IPerf API

- IPerf_ConfigureCfg()
- IPerf_ConfigureMemSeg()
- IPerf_ConfigureTaskStk()
- IPerf_TaskPrioSet()
- IPerf_Init()
- IPerf_TestStart()
- IPerf_TestRelease()
- IPerf_TestGetStatus()
- IPerf_TestGetResults()
- IPerf_Reporter()

IPerf_ConfigureCfg()

Description

Configure the IPerf module parameters.

Files

`iperf.h/iperf.c`

Prototype

```
void IPerf_ConfigureCfg (const IPERF_CFG *p_cfg)
```

Arguments

`p_cfg`

Pointer to the structure containing the IPerf module parameters.

Returned Value

None.

Notes / Warnings

1. This function is optional. If it is called, it must be called before `IPerf_Init()`. If it is not called, default values will be used to initialize the module.

IPerf_ConfigureMemSeg()

Description

Configure the memory segment that will be used to allocate internal data needed by the IPerf module instead of the default memory segment.

Files

`iperf.h/iperf.c`

Prototype

```
void IPerf_ConfigureMemSeg (MEM_SEG *p_mem_seg)
```

Arguments

`p_mem_seg`

Pointer to the memory segment from which the internal data will be allocated. If `DEF_NULL`, the internal data will be allocated from the global Heap.

Returned Value

None.

Notes / Warnings

1. This function is optional. If it is called, it must be called before `IPerf_Init()`. If it is not called, default values will be used to initialize the module.

IPerf_ConfigureTaskStk()

Description

Configure IPerf's task stack size.

Files

`iperf.h/iperf.c`

Prototype

```
void IPerf_ConfigureTaskStk (CPU_STK_SIZE  stk_size_elements,
                             void         *p_stk_base)
```

Arguments

`stk_size_elements`

Size of the stack, in CPU_STK elements.

`p_stk_base`

Pointer to base of the stack.

Returned Value

None.

Notes / Warnings

1. This function is optional. If it is called, it must be called before `IPerf_Init()`. If it is not called, default values will be used to initialize the module.

IPerf_TaskPrioSet()

Description

Sets the priority of the IPerf task.

Files

`iperf.h/iperf.c`

Prototype

```
void IPerf_TaskPrioSet (RTOS_TASK_PRIO   prio,
                        RTOS_ERR        *p_err)
```

Arguments

`prio`

New priority for the IPerf task.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function.

Returned Value

None.

Notes / Warnings

None.

IPerf_Init()

Description

Initializes and starts the IPerf application.

Files

`iperf.h/iperf.c`

Prototype

```
void  IPerf_Init (IPERF_CFG      *p_cfg,
                  RTOS_TASK_CFG *p_task_cfg,
                  MEM_SEG       *p_mem_seg,
                  RTOS_ERR      *p_err)
```

Arguments

`p_cfg`

Pointer to the IPerf configuration.

`p_task_cfg`

Pointer to the IPerf task configuration.

`p_mem_seg`

Memory segment from which internal data will be allocated. If `DEF_NULL` , it will be allocated from the global heap.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_INVALID_CFG`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`

Returned Value

None.

Notes / Warnings

None.

IPerf_TestStart()

Description

Validates and schedules a new IPerf test.

Files

`iperf.h/iperf.c`

Prototype

```
IPERF_TEST_ID  IPerf_TestStart (CPU_CHAR         *argv,
                                IPERF_OUT_FNCT    p_out_fnct,
                                IPERF_OUT_PARAM  *p_out_param,
                                RTOS_ERR         *p_err)
```

Arguments

`argv`

Pointer to the string arguments values.

`p_out_fnct`

Pointer to the string output function.

`p_out_param`

Pointer to the output function parameters.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_NO_MORE_RSRC`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`

Returned Value

- Test ID, if no error(s).
- `IPERF_TEST_ID_NONE` , otherwise.

Notes / Warnings

None.

IPerf_TestRelease()

Description

Removes the test in ring array holding.

Files

`iperf.h/iperf.c`

Prototype

```
void  IPerf_TestRelease (IPERF_TEST_ID   test_id,
                         RTOS_ERR       *p_err)
```

Arguments

`test_id`

Test ID of the test to release.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_NET_OP_IN_PROGRESS`

Returned Value

None.

Notes / Warnings

None.

IPerf_TestGetStatus()

Description

Gets the test status.

Files

`iperf.h/iperf.c`

Prototype

```
IPERF_TEST_STATUS  IPerf_TestGetStatus (IPERF_TEST_ID   test_id,
                                        RTOS_ERR       *p_err)
```

Arguments

`test_id`

Test ID of the test to get status.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_FOUND`

Returned Value

Status of specified test.

Notes / Warnings

None.

IPerf_TestGetResults()

Description

Gets the test result.

Files

iperf.h/iperf.c

Prototype

```
void  IPerf_TestGetResults (IPERF_TEST_ID   test_id,
                            IPERF_TEST     *p_test_result,
                            RTOS_ERR       *p_err)
```

Arguments

test_id

Test ID of the test to Get result.

p_test_result

Pointer to structure that will receive the result of specified test.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function :

- RTOS_ERR_NONE
- RTOS_ERR_NOT_FOUND
- RTOS_ERR_INVALID_STATE

Returned Value

None.

Notes / Warnings

1. Test results can be obtained before, after, or even during a test run.

IPerf_Reporter()

Description

Prints (in a terminal) the IPerf test results before, during, and after the performance test.

Files

iperf_rep.h/iperf_rep.c

Prototype

```
void  IPerf_Reporter (IPERF_TEST_ID     test_id,
                      IPERF_OUT_FNCT    p_out_fnct,
                      IPERF_OUT_PARAM  *p_out_param)
```

Arguments

test_id

Test ID of the test to print.

`p_out_fnct`

Pointer to the string output function.

`p_out_param`

Pointer to the output function parameters.

Returned Value

None.

Notes / Warnings

None.

Mocana nanoSSL Certificate API

Mocana nanoSSL Certificate API

- NetSecure_CA_CertIntall()
- NetSecure_Log()

NetSecure_CA_CertIntall()

Description

Installs the certificate authority's certificate.

Files

`net_secure_mocana.h/net_secure_mocana.c`

Prototype

```
CPU_BOOLEAN  NetSecure_CA_CertIntall (const  void              *p_ca_cert,
                                      CPU_INT32U           ca_cert_len,
                                      NET_SECURE_CERT_FMT    fmt,
                                      RTOS_ERR             *p_err)
```

Arguments

`p_ca_cert`

Pointer to CA certificate.

`ca_cert_len`

Certificate length.

`fmt`

Certificate format:

- `NET_SOCK_SECURE_CERT_KEY_FMT_PEM`
- `NET_SOCK_SECURE_CERT_KEY_FMT_DER`

`p_err`

Pointer to variable that will receive the return error code from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_FAIL`
- `RTOS_ERR_INVALID_TYPE`

Returned Value

- `DEF_OK` , if certificate was installed successfully.
- `DEF_FAIL` , otherwise.

Notes / Warnings

None.

NetSecure_Log()

Description

Logs the given string.

Files

`net_secure_mocana.h/net_secure_mocana.c`

Prototype

```
void NetSecure_Log (CPU_CHAR *p_str)
```

Arguments

`p_str`

Pointer to string to log.

Returned Value

None.

Notes / Warnings

None.

USB Controller API

USB Controller API

NOTE: This documentation refers to a deprecated software component that will no longer be supported and removed in an future release. Consider using the latest Silicon Labs USB stack instead. For more information, see USB Device.

- USB_CTRLR_HW_INFO_REG()
- USB_CTRLR_HW_INFO_DEV_ONLY_REG()
- USB_CTRLR_HW_INFO_HOST_ONLY_REG()
- USB_CTRLR_HW_INFO_HOST_COMPANION_REG()

USB_CTRLR_HW_INFO_REG()

USB_CTRLR_HW_INFO_REG()

Description

Registers a USB controller to the platform manager. The USB controller implements both device and host functionalities.

Files

`usb_ctrlr.h`

Prototype

```
USB_CTRLR_HW_INFO_REG(name, dev_ptr, host_ptr)
```

Arguments

`name`

Unique name for the USB controller. It is recommended to follow the standard "usbX" where X is a digit.

`dev_ptr`

Pointer to the USB device hardware information structure of type `USBD_DEV_CTRLR_HW_INFO`.

`host_ptr`

Pointer to the USB host hardware information structure of type `USBH_HC_HW_INFO`.

Returned Value

None.

Notes / Warnings

1. This macro should normally be called from the BSP.

USB_CTRLR_HW_INFO_DEV_ONLY_REG()

USB_CTRLR_HW_INFO_DEV_ONLY_REG()

Description

Registers a USB Device controller to the platform manager.

Files

`usb_ctrlr.h`

Prototype

```
USB_CTRLR_HW_INFO_DEV_ONLY_REG(name, dev_ptr)
```

Arguments

`name`

Unique name for the USB device controller. It is recommended to follow the standard "usbX" where X is a digit.

`dev_ptr`

Pointer to the USB device hardware information structure of type `USBD_DEV_CTRLR_HW_INFO` .

Returned Value

None.

Notes / Warnings

1. This macro should normally be called from the BSP.

USB_CTRLR_HW_INFO_HOST_ONLY_REG()

USB_CTRLR_HW_INFO_HOST_ONLY_REG()

Description

Registers a USB Host controller to the platform manager.

Files

`usb_ctrlr.h`

Prototype

```
USB_CTRLR_HW_INFO_HOST_ONLY_REG(name, host_ptr)
```

Arguments

`name`

Unique name for the USB host controller. It is recommended to follow the standard "usbX" where X is a digit.

`host_ptr`

Pointer to the USB host hardware information structure of type `USBH_HC_HW_INFO` .

Returned Value

None.

Notes / Warnings

1. This macro should normally be called from the BSP.

USB_CTRLR_HW_INFO_HOST_COMPANION_REG()

USB_CTRLR_HW_INFO_HOST_COMPANION_REG()

Description

Registers a USB Host companion controller to the platform manager.

Files

`usb_ctrlr.h`

Prototype

```
USB_CTRLR_HW_INFO_HOST_COMPANION_REG(name, main_ctrlr_name, host_ptr)
```

Arguments

`name`

Unique name for the USB host controller. It is recommended to follow the standard "usbX" where X is a digit.

`main_ctrlr_name`

The name of the main USB controller for which this controller is a companion.

`host_ptr`

Pointer to the USB host hardware information structure of type `USBH_HC_HW_INFO` .

Returned Value

None.

Notes / Warnings

1. This macro should normally be called from the BSP.

SILICON LABS

USB Device API

USB Device API

NOTE: This documentation refers to a deprecated software component that will no longer be supported and removed in an future release. Consider using the latest Silicon Labs USB stack instead. For more information, see USB Device.

- USB Device Core API
- USB Device CDC API
- USB Device ACM API
- USB Device CDC EEM API
- USB Device HID API
- USB Device MSC API
- USB Device Vendor API
- USBDev_API API

USB Device Core API

USB Device Core API

- USBD_ConfigureBufAlignOctets()
- USBD_ConfigureMemSeg()
- USBD_Init()
- USBD_StdReqTimeoutSet()
- USBD_DevAdd()
- USBD_DevTaskPrioSet()
- USBD_DevNbrGetFromName()
- USBD_DevStart()
- USBD_DevStop()
- USBD_ConfigAdd()
- USBD_ConfigOtherSpeed()
- USBD_DevStateGet()
- USBD_DevSpdGet()
- USBD_DevSelfPwrSet()
- USBD_DevSetMS_VendorCode()
- USBD_DevGetCfg()
- USBD_IF_Add()
- USBD_IF_AltAdd()
- USBD_IF_Grp()
- USBD_DevFrameNbrGet()
- USBD_DescDevGet()
- USBD_DescConfigGet()
- USBD_DescStrGet()
- USBD_StrAdd()
- USBD_DescWr08()
- USBD_DescWr16()
- USBD_DescWr24()
- USBD_DescWr32()
- USBD_DescWr()
- USBD_BulkAdd()
- USBD_IntrAdd()
- USBD_IsocAdd()
- USBD_IsocSyncRefreshSet()
- USBD_IsocSyncAddrSet()
- USBD_EP_MaxPhyNbrGet()
- USBD_EventConn()
- USBD_EventDisconn()
- USBD_EventHS()
- USBD_EventReset()
- USBD_EventSuspend()
- USBD_EventResume()
- USBD_EventSetup()
- USBD_BulkRx()
- USBD_BulkRxAsync()
- USBD_BulkTx()
- USBD_BulkTxAsync()
- USBD_IntrRx()

- USBD_IntrRxAsync()
- USBD_IntrTx()
- USBD_IntrTxAsync()
- USBD_IsocRxAsync()
- USBD_IsocTxAsync()
- USBD_CtrlRx()
- USBD_CtrlTx()
- USBD_EP_MaxPktSizeGet()
- USBD_EP_MaxNbrOpenGet()
- USBD_EP_Abort()
- USBD_EP_Stall()
- USBD_EP_IsStalled()

USBD_ConfigureBufAlignOctets()

Description

Configures the alignment of the internal buffers.

Files

`usbd_core.h/usbd_core.c`

Prototype

```
void  USBD_ConfigureBufAlignOctets (CPU_SIZE_T  buf_align_octets)
```

Arguments

`buf_align_octets`

Buffer alignment, in octets.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the USB device core is initialized via the `USBD_Init()` function.

USBD_ConfigureMemSeg()

Description

Configures the memory segment to use when allocating control data and buffers.

Files

`usbd_core.h/usbd_core.c`

Prototype

```
void  USBD_ConfigureMemSeg (MEM_SEG  *p_mem_seg,
                            MEM_SEG  *p_mem_seg_buf)
```

Arguments

`p_mem_seg`

Pointer to memory segment to use when allocating control data. Can be the same segment used for `p_mem_seg_buf`. `DEF_NULL` means general purpose heap segment.

`p_mem_seg_buf`

Pointer to memory segment to use when allocating data buffers. Can be the same segment used for `p_mem_seg`. `DEF_NULL` means general purpose heap segment.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the USB device core is initialized via the `USBD_Init()` function.

USBD_Init()

Description

Initializes the USB device stack.

Files

`usbd_core.h/usbd_core.c`

Prototype

```
void USBD_Init (USBD_QTY_CFG  *p_qty_cfg,             RTOS_ERR     *p_err)
```

Arguments

`p_qty_cfg`

Pointer to the USBD configuration structure.

`p_err`

Pointer to variable that will receive the return error code from these functions:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_SEG_OVF`

Returned Value

None.

Notes / Warnings

`USBD_Init()` must be called:

- Only once from a product's application.
- With the following conditions:
- After the product's OS has been initialized.
- Before the product's application calls any USB device stack function(s).

USBD_StdReqTimeoutSet()

Description

Assigns a new timeout delay for the USB device standard requests.

Files

`usbd_core.h/usbd_core.c`

Prototype

```
void  USBD_StdReqTimeoutSet (CPU_INT32U   std_req_timeout_ms,
                             RTOS_ERR    *p_err)
```

Arguments

`std_req_timeout_ms`

New timeout, in milliseconds.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

None.

USBD_DevAdd()

Description

Adds a device to the stack and creates the default control endpoints.

Files

`usbd_core.h/usbd_core.c`

Prototype

```
CPU_INT08U  USBD_DevAdd (const  CPU_CHAR          *name,
                         const  RTOS_TASK_CFG     *p_task_cfg,
                         const  USBD_DEV_CFG      *p_dev_cfg,
                         const  USBD_DEV_DRV_CFG  *p_dev_drv_cfg,
                                USBD_BUS_FNCTS    *p_bus_fnct,
                                RTOS_ERR          *p_err)
```

Arguments

`name`

Name of the USB controller.

`p_task_cfg`

Pointer to the task configuration structure.

`p_dev_cfg`

Pointer to the specific USB device configuration.

`p_dev_drv_cfg`

Pointer to the device driver configuration structure.

`p_dev_hw_info`

Pointer to the device hardware information structure.

`p_bus_fnct`

Pointer to the specific USB device bus events callback functions. Content MUST be persistent.

`p_err`

Pointer to the variable that will receive the returned error code from these functions:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`
- `RTOS_ERR_ALLOC`
- `RTOS_ERR_DEV_ALLOC`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_EP_NONE_AVAIL`

Returned Value

- The device number, if no errors are returned.
- `USBD_DEV_NBR_NONE` , if any errors are returned.

Notes / Warnings

Certain driver functions are required for the driver to work correctly with the core. The pointers to these functions are checked in this function to ensure they are valid and can be used throughout the core.

USBD_DevTaskPrioSet()

Description

Sets priority of the given device's task.

Files

`usbd_core.h/usbd_core.c`

Prototype

```
void  USBD_DevTaskPrioSet (CPU_INT08U      dev_nbr,
                           RTOS_TASK_PRIO   prio,
                           RTOS_ERR        *p_err)
```

Arguments

`dev_nbr`

Device number.

`prio`

Priority of the device's task.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_ARG`

Returned Value

None.

Notes / Warnings

None.

USBD_DevNbrGetFromName()

Description

Get device number from the USB controller name.

Files

`usbd_core.h/usbd_core.c`

Prototype

```
CPU_INT08U  USBD_DevNbrGetFromName (const  CPU_CHAR  *name)
```

Arguments

`name`

Name of the USB controller.

Returned Value

Device number.

Notes / Warnings

None.

USBD_DevStart()

Description

Starts the device stack.

Files

`usbd_core.h/usbd_core.c`

Prototype

```
void USBD_DevStart (CPU_INT08U    dev_nbr,
                    RTOS_ERR     *p_err)
```

Arguments

`dev_nbr`

Device number.

`p_err`

Pointer to the variable that receives the following returned error codes from these functions:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_EP_NONE_AVAIL`
- `RTOS_ERR_FAIL`

Returned Value

None.

Notes / Warnings

The device stack can be only started if the device is in one of the following states:

- `USBD_DEV_STATE_NONE` : Device controller has not been initialized.
- `USBD_DEV_STATE_INIT` : Device controller is already initialized.

USBD_DevStop()

Description

Stops the device stack.

Files

`usbd_core.h/usbd_core.c`

Prototype

```
void USBD_DevStop (CPU_INT08U   dev_nbr,
                  RTOS_ERR    *p_err)
```

Arguments

`dev_nbr`

Device number.

`p_err`

Pointer to variable that will receive the following return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`

Returned Value

None.

Notes / Warnings

None.

USBD_ConfigAdd()

Description

Adds a configuration attribute to the device.

Files

`usbd_core.h/usbd_core.c`

Prototype

```
CPU_INT08U USBD_ConfigAdd ( CPU_INT08U      dev_nbr,
                            CPU_INT08U      attrib,
                            CPU_INT16U      max_pwr,
                            USBD_DEV_SPD    spd,
                            const CPU_CHAR *p_name,
                            RTOS_ERR       *p_err)
```

Arguments

`dev_nbr`

Device number.

`attrib`

Available configuration attributes:

- `USBD_DEV_ATTRIB_SELF_POWERED` : Power does not come from VBUS.
- `USBD_DEV_ATTRIB_REMOTE_WAKEUP` : Remote wake-up feature enabled.

`max_pwr`

Bus power required for this device (see **Note #1**).

`spd`

Available configuration speeds.

`USBD_DEV_SPD_FULL` : Configuration is added to the full-speed configuration set.

`USBD_DEV_SPD_HIGH` : Configuration is added to the high-speed configuration set.

`p_name`

Pointer to a string that describes the configuration (see Note #2).

`p_err`

Pointers to the variable that receives the returned error code from these functions:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`
- `RTOS_ERR_ALLOC`
- `RTOS_ERR_CONFIG_ALLOC`
- `RTOS_ERR_INVALID_ARG`

Returned Value

- Configuration number, if no errors are returned.
- `USBD_CONFIG_NBR_NONE` , if any errors are returned.

Notes / Warnings

1. USB spec 2.0, (section 7.2.1.3/4) defines power constraints for bus-powered devices:
    - A low-power function draws up to one unit load from the USB cable when operational.
    - A function is high-power if, when fully-powered, draws over one, but no more than five, unit loads from the USB cable.
    - A unit load is defined as 100mA, so the 'max_pwr' argument should be between 0mA and 500mA.
2. String support is optional: 'p_name' can be a NULL string pointer.
    - Configuration can only be added when the device is in the following states:
    - `USBD_DEV_STATE_NONE` : Device controller has not been initialized.
3. `USBD_DEV_STATE_INIT` : Device controller is already initialized.
4. A high-speed configuration can only be added if the device controller is high-speed.

USBD_ConfigOtherSpeed()

Description

Associate a configuration with its alternative-speed counterpart.

Files

usbd_core.h/usbd_core.c

Prototype

```
void USBD_ConfigOtherSpeed (CPU_INT08U   dev_nbr,
                     CPU_INT08U   config_nbr,
                     CPU_INT08U   config_other,
                     RTOS_ERR    *p_err)
```

Arguments

dev_nbr

Device number.

config_nbr

Configuration number.

config_other

Other-speed configuration number.

p_err

Pointer to the variable that receives the returned error code from these functions:

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_DEV_STATE

Returned Value

None.

Notes / Warnings

1. Configurations from high and full-speed can be associated with each other to provide comparable functionality regardless of speed.
2. Configuration can only be associated when the device is in the following states:
   - USBD_DEV_STATE_NONE Device controller has not been initialized.
   - USBD_DEV_STATE_INIT Device controller is already initialized.

USBD_DevStateGet()

Description

Gets the current device state.

Files

usbd_core.h/usbd_core.c

Prototype

```
USBD_DEV_STATE USBD_DevStateGet (CPU_INT08U   dev_nbr,
                         RTOS_ERR    *p_err)
```

Arguments

dev_nbr

Device number.

p_err

Pointer to the variable that receives the returned error code from this function :

- `RTOS_ERR_NONE`

Returned Value

- Current device state, if no errors are returned.
- `USBD_DEV_STATE_NONE` , if any errors are returned.

Notes / Warnings

None.

USBD_DevSpdGet()

Description

Gets the device speed.

Files

`usbd_core.h/usbd_core.c`

Prototype

```
USBD_DEV_SPD USBD_DevSpdGet (CPU_INT08U   dev_nbr,
                             RTOS_ERR    *p_err)
```

Arguments

`dev_nbr`

Device number.

`p_err`

Pointer to the variable that receives the returned error code from these functions:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`

Returned Value

- The current device speed, if successful.
- `USBD_DEV_SPD_INVALID` , if not successful.

Notes / Warnings

None.

USBD_DevSelfPwrSet()

Description

Sets the device's current power source.

Files

`usbd_core.h/usbd_core.c`

Prototype

```
void USBD_DevSelfPwrSet (CPU_INT08U    dev_nbr,
                         CPU_BOOLEAN   self_pwr,
                         RTOS_ERR     *p_err)
```

Arguments

`dev_nbr`

Device number.

`self_pwr`

The power source of the device :

- `DEF_TRUE` : device is self-powered.
- `DEF_FALSE` : device is bus-powered.

`p_err`

Pointer to the variable that receives the returned error code from this function :

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

None.

USBD_DevSetMS_VendorCode()

Description

Set the device's Microsoft vendor code.

Files

`usbd_core.h/usbd_core.c`

Prototype

```
void USBD_DevSetMS_VendorCode (CPU_INT08U    dev_nbr,
                               CPU_INT08U    vendor_code,
                               RTOS_ERR     *p_err)
```

Arguments

`dev_nbr`

Device number.

`vendor_code`

Microsoft vendor code.

`p_err`

Pointer to the variable that receives the returned error code from this function:

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

The vendor code used must be different from any vendor bRequest value.

USBD_DevGetCfg()

Description

Gets the device's configuration.

Files

`usbd_core.h/usbd_core.c`

Prototype

```
USBD_DEV_CFG *USBD_DevCfgGet (CPU_INT08U   dev_nbr,
                              RTOS_ERR    *p_err)
```

Arguments

`dev_nbr`

Device number.

`p_err`

Pointer to the variable that receives the returned error code from this function :

- `RTOS_ERR_NONE`

Returned Value

- Pointer to device configuration, if no errors are returned.
- Pointer to `NULL`, if any errors are returned.

Notes / Warnings

None.

USBD_IF_Add()

Description

Adds an interface to a specific configuration.

Files

`usbd_core.h/usbd_core.c`

Prototype

```
CPU_INT08U USBD_IF_Add ( CPU_INT08U       dev_nbr,
                        CPU_INT08U       cfg_nbr,
                        USBD_CLASS_DRV   *p_class_drv,
                        void             *p_if_class_arg,
                        void             *p_if_alt_class_arg,
                        CPU_INT08U       class_code,
                        CPU_INT08U       class_sub_code,
                        CPU_INT08U       class_protocol_code,
                  const CPU_CHAR         *p_name,
                        RTOS_ERR         *p_err)
```

Arguments

`dev_nbr`

Device number.

`cfg_nbr`

Configuration index to add the interface.

`p_class_drv`

Pointer to interface driver.

`p_if_class_arg`

Pointer to interface driver argument.

`p_if_alt_class_arg`

Pointer to alternate interface argument.

`class_code`

Class code assigned by the USB-IF.

`class_sub_code`

Subclass code assigned by the USB-IF.

`class_protocol_code`

Protocol code assigned by the USB-IF.

`p_name`

Pointer to string describing the Interface.

`p_err`

Pointer to variable that will receive the return error code from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`
- `RTOS_ERR_IF_ALT_ALLOC`
- `RTOS_ERR_ALLOC`
- `RTOS_ERR_IF_ALLOC`

Returned Value

- Interface number, if no errors are returned.
- `USBD_IF_NBR_NONE` , if any errors are returned.

Notes / Warnings

USB Spec 2.0 Interface (section 9.6.5) states: "An interface may include alternate settings that allow the endpoints and/or their characteristics to be varied after the device has been configured. The default setting for an interface is always an alternate setting of zero."

USBD_IF_AltAdd()

Description

Adds an alternate setting to a specific interface.

Files

usbd_core.h/usbd_core.c

Prototype

```
CPU_INT08U USBD_IF_AltAdd ( CPU_INT08U    dev_nbr,
                            CPU_INT08U    config_nbr,
                            CPU_INT08U    if_nbr,
                            void         *p_class_arg,
                      const CPU_CHAR     *p_name,
                            RTOS_ERR     *p_err)
```

Arguments

`dev_nbr`

Device number.

`config_nbr`

Configuration number.

`if_nbr`

Interface number.

`p_class_arg`

Pointer to alternate interface argument.

`p_name`

Pointer to alternate setting name.

`p_err`

Pointer to a variable that receives the returned error code from these functions:

- `RTOS_ERR_NONE`
- `RTOS_ERR_IF_ALT_ALLOC`
- `RTOS_ERR_ALLOC`

Returned Value

- Interface alternate setting number, if no errors are returned.
- `USBD_IF_ALT_NBR_NONE` , if any errors are returned.

Notes / Warnings

None.

USBD_IF_Grp()

Description

Creates an interface group.

Files

usbd_core.h/usbd_core.c

Prototype

```
CPU_INT08U USBD_IF_Grp ( CPU_INT08U    dev_nbr,
                         CPU_INT08U    config_nbr,
                         CPU_INT08U    class_code,
                         CPU_INT08U    class_sub_code,
                         CPU_INT08U    class_protocol_code,
                         CPU_INT08U    if_start,
                         CPU_INT08U    if_cnt,
                    const CPU_CHAR     *p_name,
                         RTOS_ERR     *p_err)
```

Arguments

`dev_nbr`

Device number

`config_nbr`

Configuration number.

`class_code`

Class code assigned by the USB-IF.

`class_sub_code`

Subclass code assigned by the USB-IF.

`class_protocol_code`

Protocol code assigned by the USB-IF.

`if_start`

Interface number of the first interface that is associated with this group.

`if_cnt`

Number of consecutive interfaces that are associated with this group.

`p_name`

Pointer to the string that describes interface group.

`p_err`

Pointer to the variable that receives the returned error code from these functions:

- `RTOS_ERR_NONE`
- `RTOS_ERR_ALLOC`
- `RTOS_ERR_IF_GRP_ALLOC`
- `RTOS_ERR_ALREADY_EXISTS`

Returned Value

- Interface group number, if no errors are returned.
- `USBD_IF_GRP_NBR_NONE` , if any errors are returned.

Notes / Warnings

None.

USBD_DevFrameNbrGet()

Description

Get the last frame number from the driver.

Files

`usbd_core.h/usbd_core.c`

Prototype

```
CPU_INT16U USBD_DevFrameNbrGet (CPU_INT08U    dev_nbr,
                                RTOS_ERR     *p_err)
```

Arguments

`dev_nbr`

Device number.

`p_err`

Pointer to variable that will receive the return error code from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`
- `RTOS_ERR_NOT_SUPPORTED`

Returned Value

The current frame number.

Notes / Warnings

1. The frame number will always be in the range of 0-2047 (11 bits).
2. Frame number returned to the caller contains the frame and microframe numbers. It is encoded following this 16-bit format:

```
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

| 0    0 | microframe |              frame                |
```

Caller must use the macros `USBD_FRAME_NBR_GET()` or `USBD_MICROFRAME_NBR_GET()` to get the frame or microframe number only.

USBD_DescDevGet()

Description

Gets the device descriptor.

Files

`usbd_core.h/usbd_core.c`

Prototype

```
CPU_INT08U USBD_DescDevGet (USBD_DRV     *p_drv,
                            CPU_INT08U   *p_buf,
                            CPU_INT08U    max_len,
                            RTOS_ERR     *p_err)
```

Arguments

`p_drv`

Pointer to device driver structure.

p_buf

Pointer to the destination buffer.

max_len

Maximum number of bytes to write in destination buffer.

p_err

Pointer to the variable that receives the returned error code from these functions:

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_DEV_STATE
- RTOS_ERR_NULL_PTR
- RTOS_ERR_EP_QUEUING
- RTOS_ERR_TX
- RTOS_ERR_NOT_READY
- RTOS_ERR_EP_INVALID
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_INVALID_ARG
- RTOS_ERR_IS_OWNER
- RTOS_ERR_INVALID_EP_STATE
- RTOS_ERR_ABORT
- RTOS_ERR_TIMEOUT

Returned Value

- Number of bytes actually in the descriptor, if no errors are returned.
- 0, if any errors are returned.

Notes / Warnings

This function should be used by drivers that support the standard requests auto-reply (during the initialization process).

USBD_DescConfigGet()

Description

Gets a configuration descriptor.

Files

usbd_core.h/usbd_core.c

Prototype

```
CPU_INT16U USBD_DescConfigGet (USBD_DRV     *p_drv,
                               CPU_INT08U   *p_buf,
                               CPU_INT16U    max_len,
                               CPU_INT08U    config_ix,
                               RTOS_ERR     *p_err)
```

Arguments

p_drv

Pointer to device driver structure.

`p_buf`

Pointer to the destination buffer.

`max_len`

Maximum number of bytes to write in the destination buffer.

`config_ix`

Index of the desired configuration descriptor.

`p_err`

Pointer to the variable that receives the returned the error code from these functions:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`
- `RTOS_ERR_NULL_PTR`
- `RTOS_ERR_EP_QUEUING`
- `RTOS_ERR_TX`
- `RTOS_ERR_NOT_READY`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_EP_STATE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

- Number of bytes actually in the descriptor, if no errors are returned.
- 0, if any errors are returned.

Notes / Warnings

This function should be used by drivers supporting a standard request's auto-reply, during the initialization process.

USBD_DescStrGet()

Description

Gets a string descriptor.

Files

`usbd_core.h/usbd_core.c`

Prototype

```
CPU_INT08U USBD_DescStrGet (USBD_DRV     *p_drv,
                            CPU_INT08U   *p_buf,
                            CPU_INT08U    max_len,
                            CPU_INT08U    str_ix,
                            RTOS_ERR     *p_err)
```

Arguments

`p_drv`

Pointer to the device driver structure.

`p_buf`

Pointer to the destination buffer.

`max_len`

Maximum number of bytes to write in destination buffer.

`str_ix`

Index of the desired string descriptor.

`p_err`

Pointer to variable that will receive the return error code from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`
- `RTOS_ERR_NULL_PTR`
- `RTOS_ERR_EP_QUEUING`
- `RTOS_ERR_TX`
- `RTOS_ERR_NOT_READY`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_EP_STATE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

- Number of bytes actually in the descriptor, if no errors are returned.
- 0, if any errors are returned.

Notes / Warnings

This function should be used by drivers supporting a standard request's auto-reply, during the initialization process.

USBD_StrAdd()

Description

Adds a string to a USB device.

Files

`usbd_core.h/usbd_core.c`

Prototype

```
void USBD_StrAdd ( CPU_INT08U    dev_nbr,
          const CPU_CHAR     *p_str,
              RTOS_ERR     *p_err)
```

Arguments

`dev_nbr`

Device number.

`p_str`

Pointer to the string to add (see **Notes**).

`p_err`

Pointer to variable that will receive the return error code from these functions:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`
- `RTOS_ERR_ALLOC`

Returned Value

None.

Notes / Warnings

USB spec 2.0 (chapter 9.5) states:

"Where appropriate, descriptors contain references to string descriptors that provide displayable information describing a descriptor in human-readable form. The inclusion of string descriptors is optional. However, the reference fields within descriptors are mandatory. If a device does not support string descriptors, string reference fields must be reset to zero to indicate no string descriptor is available".

Since string descriptors are optional, '`p_str`' could be a NULL pointer.

USBD_DescWr08()

Description

Writes an 8-bit value to the descriptor buffer.

Files

`usbd_core.h/usbd_core.c`

Prototype

```
void USBD_DescWr08 (CPU_INT08U   dev_nbr,
               CPU_INT08U   val)
```

Arguments

`dev_nbr`

Device number.

`val`

8-bit value to write in the descriptor buffer.

Returned Value

None.

Notes / Warnings

USB classes may use this function to append class-specific descriptors to the configuration descriptor.

USBD_DescWr16()

Description

Writes a 16-bit value to the descriptor buffer.

Files

`usbd_core.h/usbd_core.c`

Prototype

```
void USBD_DescWr16 (CPU_INT08U   dev_nbr,
                    CPU_INT16U   val)
```

Arguments

`dev_nbr`

Device number.

`val`

16-bit value to write in descriptor buffer.

Returned Value

none.

Notes / Warnings

1. USB classes may use this function to append class-specific descriptors to the configuration descriptor.
2. USB descriptors are in little-endian format.

USBD_DescWr24()

Description

Writes a 24-bit value to the descriptor buffer.

Files

`usbd_core.h/usbd_core.c`

Prototype

```
void USBD_DescWr24 (CPU_INT08U   dev_nbr,
                    CPU_INT32U   val)
```

Arguments

`dev_nbr`

Device number.

`val`

32-bit value containing 24 useful bits to write in the descriptor buffer.

Returned Value

none.

Notes / Warnings

1. USB classes may use this function to append class-specific descriptors to the configuration descriptor.
2. USB descriptors are in little-endian format.

USBD_DescWr32()

Description

Writes a 32-bit value to the descriptor buffer.

Files

`usbd_core.h/usbd_core.c`

Prototype

```
void USBD_DescWr32 (CPU_INT08U   dev_nbr,
                   CPU_INT32U   val)
```

Arguments

`dev_nbr`

Device number.

`val`

32-bit value to write in the descriptor buffer.

Returned Value

none.

Notes / Warnings

1. USB classes may use this function to append class-specific descriptors to the configuration descriptor.
2. USB descriptors are in little-endian format.

USBD_DescWr()

Description

Writes a buffer into the descriptor buffer.

Files

`usbd_core.h/usbd_core.c`

Prototype

```
void USBD_DescWr ( CPU_INT08U    dev_nbr,
            const CPU_INT08U   *p_buf,
                  CPU_INT16U    len)
```

Arguments

`dev_nbr`

Device number.

`p_buf`

Pointer to the buffer to write into the descriptor buffer.

`len`

Length of the buffer.

Returned Value

None.

Notes / Warnings

USB classes may use this function to append class-specific descriptors to the configuration descriptor.

USBD_BulkAdd()

Description

Adds a bulk endpoint to an alternate setting interface.

Files

`usbd_core.h/usbd_core.c`

Prototype

```
CPU_INT08U USBD_BulkAdd (CPU_INT08U     dev_nbr,
                         CPU_INT08U     config_nbr,
                         CPU_INT08U     if_nbr,
                         CPU_INT08U     if_alt_nbr,
                         CPU_BOOLEAN    dir_in,
                         CPU_INT16U     max_pkt_len,
                         RTOS_ERR      *p_err)
```

Arguments

`dev_nbr`

Device number.

`config_nbr`

Configuration number.

`if_nbr`

Interface number.

`if_alt_nbr`

Interface alternate setting number.

`dir_in`

DEF_NO OUT direction.

`max_pkt_len`

Endpoint maximum packet length (see **Notes**)

`p_err`

Pointer to the variable that receives the returned error code from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_EP_ALLOC`
- `RTOS_ERR_EP_NONE_AVAIL`

Returned Value

- Endpoint address, if no errors are returned.
- `USBD_EP_ADDR_NONE`, if any errors are returned.

Notes / Warnings

If the `max_pkt_len` argument is '0', the stack will allocate the first available `BULK` endpoint, regardless its maximum packet size.

USBD_IntrAdd()

Description

Adds an interrupt endpoint to an alternate setting interface.

Files

`usbd_core.h/usbd_core.c`

Prototype

```
CPU_INT08U USBD_IntrAdd (CPU_INT08U    dev_nbr,
                    CPU_INT08U    config_nbr,
                    CPU_INT08U    if_nbr,
                    CPU_INT08U    if_alt_nbr,
                    CPU_BOOLEAN   dir_in,
                    CPU_INT16U    max_pkt_len,
                    CPU_INT16U    interval,
                    RTOS_ERR     *p_err)
```

Arguments

`dev_nbr`

Device number.

`config_nbr`

Configuration number.

`if_nbr`

Interface number.

`if_alt_nbr`

Interface alternate setting number.

`dir_in`

- `DEF_NO` : OUT direction.
- `DEF_YES` : IN direction.

`max_pkt_len`

Endpoint maximum packet length. (see **Note #1**)

`interval`

Endpoint interval in frames or microframes.

`p_err`

Pointer to the variable that receives the returned error code from these functions:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_EP_ALLOC`
- `RTOS_ERR_EP_NONE_AVAIL`

Returned Value

- Endpoint address, if no errors are returned.
- `USBD_EP_ADDR_NONE`, if any errors are returned.

Notes / Warnings

1. If the `max_pkt_len` argument is '0', the stack will allocate the first available `INTERRUPT` endpoint, regardless its maximum packet size.
2. For high-speed interrupt endpoints, the bInterval values must be in the range from 1 to 16. The bInterval value is used as the exponent for a 2^(bInterval-1) value. Maximum polling interval value is 2^(16-1) = 32768 32768 microframes (i.e., 4096 frames) in high-speed.

USBD_IsocAdd()

Description

Adds an isochronous endpoint to alternate setting interface.

Files

`usbd_core.h/usbd_core.c`

Prototype

```
CPU_INT08U USBD_IsocAdd (CPU_INT08U     dev_nbr,
                    CPU_INT08U     config_nbr,
                    CPU_INT08U     if_nbr,
                    CPU_INT08U     if_alt_nbr,
                    CPU_BOOLEAN    dir_in,
                    CPU_INT08U     attrib,
                    CPU_INT16U     max_pkt_len,
                    CPU_INT08U     transaction_frame,
                    CPU_INT16U     interval,
                    RTOS_ERR      *p_err)
```

Arguments

`dev_nbr`

Device number.

`config_nbr`

Configuration number.

`if_nbr`

Interface number.

`if_alt_nbr`

Interface alternate setting number.

`dir_in`

DEF_NO, OUT direction.

`attrib`

Isochronous endpoint synchronization and usage type attributes.

`max_pkt_len`

Endpoint maximum packet length (see Note #1).

`transaction_frame`

Endpoint transactions per (micro)frame (see Note #2).

`interval`

Endpoint interval in frames or microframes.

`p_err`

Pointer to variable that will receive the return error code from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_EP_ALLOC`
- `RTOS_ERR_EP_NONE_AVAIL`

Returned Value

- Endpoint address, if no errors are returned.
- `USBD_EP_ADDR_NONE` , if any errors are returned.

Notes / Warnings

1. If the `max_pkt_len` argument is '0', the stack allocates the first available `ISOCHRONOUS` endpoint, regardless of its maximum packet size.
2. For full-speed endpoints, ' `transaction_frame` ' must be set to 1 since there is no support for high-bandwidth endpoints.
3. For full-/high-speed isochronous endpoints, bInterval values must be in the range from 1 to 16. The bInterval value is used as the exponent for a 2^(bInterval-1) value. Maximum polling interval value is 2^(16-1) = 32768 frames in full-speed and 32768 microframes (i.e., 4096 frames) in high-speed.

USBD_IsocSyncRefreshSet()

Description

Sets the synchronization feedback rate on the synchronization isochronous endpoint.

Files

`usbd_core.h/usbd_core.c`

Prototype

```
void USBD_IsocSyncRefreshSet (CPU_INT08U    dev_nbr,
                              CPU_INT08U    config_nbr,
                              CPU_INT08U    if_nbr,
                              CPU_INT08U    if_alt_nbr,
                              CPU_INT08U    synch_ep_addr,
                              CPU_INT08U    sync_refresh,
                              RTOS_ERR     *p_err)
```

Arguments

`dev_nbr`

Device number.

`config_nbr`

Configuration number.

`if_nbr`

Interface number.

`if_alt_nbr`

Interface alternate setting number.

`synch_ep_addr`

Synchronization endpoint address.

`sync_refresh`

Exponent of synchronization feedback rate (see Note #3).

`p_err`

Pointer to variable that will receive the return error code from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_EP_INVALID`

Returned Value

None.

Notes / Warnings

1. Synchronization endpoints can only be associated when the device is in the following states:
    - `USBD_DEV_STATE_NONE` : Device controller has not been initialized.
    - `USBD_DEV_STATE_INIT` :Device controller is already initialized.
2. For audio class 1.0, the interface class code must be `USBD_CLASS_CODE_AUDIO` and protocol '`zero`'.
3. If explicit synchronization mechanism is needed to maintain synchronization during transfers, the information carried over the synchronization path must be available every 2 ^ (10 - P) frames, with P ranging from 1 to 9 (512 ms down to 2 ms).
4. Table 4-22 "Standard AS Isochronous Synch Endpoint Descriptor" of Audio 1.0 specification indicates that for bmAttributes field no usage type for bits 5..4. But USB 2.0 specification, Table 9-13 "Standard Endpoint Descriptor" indicates that several types of usage. When an explicit feedback is defined for a asynchronous isochronous endpoint, the associated synch feedback should use the Usage type 'Feedback endpoint'.

USBD_IsocSyncAddrSet()

Description

Associates synchronization endpoint to isochronous endpoint.

Files

`usbd_core.h/usbd_core.c`

Prototype

```
void USBD_IsocSyncAddrSet (CPU_INT08U    dev_nbr,
                           CPU_INT08U    config_nbr,
                           CPU_INT08U    if_nbr,
                           CPU_INT08U    if_alt_nbr,
                           CPU_INT08U    data_ep_addr,
                           CPU_INT08U    sync_addr,
                           RTOS_ERR     *p_err)
```

Arguments

`dev_nbr`

Device number.

`config_nbr`

Configuration number.

`if_nbr`

Interface number.

`if_alt_nbr`

Interface alternate setting number.

`data_ep_addr`

Data endpoint address.

`sync_addr`

Associated synchronization endpoint.

`p_err`

Pointer to variable that will receive the return error code from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_EP_INVALID`

Returned Value

none.

Notes / Warnings

1. Synchronization endpoints can only be associated when the device is in the following states:
   - `USBD_DEV_STATE_NONE` Device controller has not been initialized.
   - `USBD_DEV_STATE_INIT` Device controller is already initialized.
2. For audio class 1.0, interface class code must be `USBD_CLASS_CODE_AUDIO` and protocol '`zero`'.

USBD_EP_MaxPhyNbrGet()

Description

Gets the maximum physical endpoint number.

Files

`usbd_core.h/usbd_core.c`

Prototype

```
CPU_INT08U USBD_EP_MaxPhyNbrGet (CPU_INT08U   dev_nbr)
```

Arguments

`dev_nbr`

Device number.

Returned Value

- Maximum physical endpoint number, if no errors are returned.
- `USBD_EP_NBR_NONE` , if any errors are returned.

Notes / Warnings

None.

USBD_EventConn()

Description

Notifies the USB connection bus events to the device stack.

Files

`usbd_core.h/usbd_core.c`

Prototype

```
void USBD_EventConn (USBD_DRV   *p_drv)
```

Arguments

`p_drv`

Pointer to device driver.

Returned Value

None.

Notes / Warnings

None.

USBD_EventDisconn()

Description

Notifies the USB disconnection bus events to the device stack.

Files

`usbd_core.h/usbd_core.c`

Prototype

```
void USBD_EventDisconn (USBD_DRV *p_drv)
```

Arguments

`p_drv`

Pointer to device driver.

Returned Value

None.

Notes / Warnings

None.

USBD_EventHS()

Description

Notifies the USB High-Speed bus events to the device stack.

Files

`usbd_core.h/usbd_core.c`

Prototype

```
void USBD_EventHS (USBD_DRV *p_drv)
```

Arguments

`p_drv`

Pointer to device driver.

Returned Value

None.

Notes / Warnings

None.

USBD_EventReset()

Description

Notifies the USB reset bus events to the device stack.

Files

`usbd_core.h/usbd_core.c`

Prototype

```
void USBD_EventReset (USBD_DRV *p_drv)
```

Arguments

`p_drv`

Pointer to device driver.

Returned Value

None.

Notes / Warnings

None.

USBD_EventSuspend()

Description

Notifies the USB suspend bus events to the device stack.

Files

`usbd_core.h/usbd_core.c`

Prototype

```
void USBD_EventSuspend (USBD_DRV   *p_drv)
```

Arguments

`p_drv`

Pointer to device driver.

Returned Value

None.

Notes / Warnings

None.

USBD_EventResume()

Description

Notifies the USB resume bus events to the device stack.

Files

`usbd_core.h/usbd_core.c`

Prototype

```
void USBD_EventResume (USBD_DRV   *p_drv)
```

Arguments

`p_drv`

Pointer to device driver.

Returned Value

None.

Notes / Warnings

None.

USBD_EventSetup()

Description

Sends a USB setup event to the core task.

Files

`usbd_core.h/usbd_core.c`

Prototype

```
void USBD_EventSetup (USBD_DRV   *p_drv,
                      void       *p_buf)
```

Arguments

`p_drv`

Pointer to device driver.

`p_buf`

Pointer to the setup packet.

Returned Value

None.

Notes / Warnings

None.

USBD_BulkRx()

Description

Receives data on Bulk OUT endpoint.

Files

`usbd_ep.h/usbd_ep.c`

Prototype

```
CPU_INT32U USBD_BulkRx (CPU_INT08U    dev_nbr,
                        CPU_INT08U    ep_addr,
                        void         *p_buf,
                        CPU_INT32U    buf_len,
                        CPU_INT16U    timeout_ms,
                        RTOS_ERR     *p_err)
```

Arguments

`dev_nbr`

Device number.

`ep_addr`

Endpoint address.

`p_buf`

Pointer to the destination buffer to receive data (see Note #1).

`buf_len`

Number of octets to receive.

`timeout_ms`

Timeout in milliseconds.

`p_err`

Pointer to the variable that receives the following returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`
- `RTOS_ERR_NULL_PTR`
- `RTOS_ERR_EP_QUEUING`
- `RTOS_ERR_RX`
- `RTOS_ERR_NOT_READY`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_FAIL`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_EP_STATE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

- Number of octets received, if no errors are returned.
- 0, if any errors returned.

Notes / Warnings

1. Receive buffer must be aligned with a single word (minimum).

USBD_BulkRxAsync()

Description

Receives data on Bulk OUT endpoint asynchronously.

Files

`usbd_ep.h/usbd_ep.c`

Prototype

```
void USBD_BulkRxAsync (CPU_INT08U        dev_nbr,
                       CPU_INT08U        ep_addr,
                       void             *p_buf,
                       CPU_INT32U        buf_len,
                       USBD_ASYNC_FNCT   async_fnct,
                       void             *p_async_arg,
                       RTOS_ERR         *p_err)
```

Arguments

`dev_nbr`

Device number.

`ep_addr`

Endpoint address.

`p_buf`

Pointer to the destination buffer to receive data (see Note #1).

`buf_len`

Number of octets to receive.

`async_fnct`

Function that will be invoked upon completion of receive operation.

`p_async_arg`

Pointer to the argument that will be passed as parameter of 'async_fnct'.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`
- `RTOS_ERR_NULL_PTR`
- `RTOS_ERR_EP_QUEUING`
- `RTOS_ERR_RX`
- `RTOS_ERR_NOT_READY`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_EP_STATE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

None.

Notes / Warnings

1. Receive buffer must be aligned with a single word (minimum).

USBD_BulkTx()

Description

Sends data on a Bulk IN endpoint.

Files

`usbd_ep.h/usbd_ep.c`

Prototype

```
CPU_INT32U USBD_BulkTx (CPU_INT08U     dev_nbr,
                        CPU_INT08U     ep_addr,
                        void          *p_buf,
                        CPU_INT32U     buf_len,
                        CPU_INT16U     timeout_ms,
                        CPU_BOOLEAN    end,
                        RTOS_ERR      *p_err)
```

Arguments

`dev_nbr`

Device number.

`ep_addr`

Endpoint address.

`p_buf`

Pointer to buffer of data that will be transmitted (see Note #2).

`buf_len`

Number of octets to transmit.

`timeout_ms`

Timeout in milliseconds.

`end`

End-of-transfer flag (see Note #3).

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`
- `RTOS_ERR_NULL_PTR`
- `RTOS_ERR_EP_QUEUING`
- `RTOS_ERR_TX`
- `RTOS_ERR_NOT_READY`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_EP_STATE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

- Number of octets transmitted, if no errors are returned.
- 0, if any errors returned.

Notes / Warnings

1. This function SHOULD NOT be called from interrupt service routine (ISR).
2. Transmit buffer must be aligned with a single word (minimum).
3. If end-of-transfer is set and transfer length is multiple of maximum packet sizes, a zero-length packet is transferred to indicate a short transfer to the host.

USBD_BulkTxAsync()

Description

Sends data on the Bulk IN endpoint asynchronously.

Files

`usbd_ep.h/usbd_ep.c`

Prototype

```
void USBD_BulkTxAsync (CPU_INT08U        dev_nbr,
                       CPU_INT08U        ep_addr,
                       void             *p_buf,
                       CPU_INT32U        buf_len,
                       USBD_ASYNC_FNCT   async_fnct,
                       void             *p_async_arg,
                       CPU_BOOLEAN       end,
                       RTOS_ERR         *p_err)
```

Arguments

`dev_nbr`

Device number.

`ep_addr`

Endpoint address.

`p_buf`

Pointer to the buffer of data that will be transmitted (see Note #1).

`buf_len`

Number of octets to transmit.

`async_fnct`

Function that will be invoked upon completion of transmit operation.

`p_async_arg`

Pointer to the argument that will be passed as parameter of 'async_fnct'.

`end`

End-of-transfer flag (see Note #2).

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`
- `RTOS_ERR_NULL_PTR`
- `RTOS_ERR_EP_QUEUING`

- RTOS_ERR_TX
- RTOS_ERR_NOT_READY
- RTOS_ERR_EP_INVALID
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_IS_OWNER
- RTOS_ERR_INVALID_EP_STATE
- RTOS_ERR_ABORT
- RTOS_ERR_TIMEOUT

Returned Value

None.

Notes / Warnings

1. Transmit buffer must be aligned with a single word (minimum).
2. If end-of-transfer is set and transfer length is multiple of maximum packet size, a zero-length packet is transferred to indicate a short transfer to the host.

USBD_IntrRx()

Description

Receives data on the Interrupt OUT endpoint.

Files

`usbd_ep.h/usbd_ep.c`

Prototype

```
CPU_INT32U USBD_IntrRx (CPU_INT08U    dev_nbr,
                        CPU_INT08U    ep_addr,
                        void         *p_buf,
                        CPU_INT32U    buf_len,
                        CPU_INT16U    timeout_ms,
                        RTOS_ERR     *p_err)
```

Arguments

`dev_nbr`

Device number.

`ep_addr`

Endpoint address.

`p_buf`

Pointer to the destination buffer to receive data (see Note #2).

`buf_len`

Number of octets to receive.

`timeout_ms`

Timeout in milliseconds.

p_err

Pointer to the variable that will receive one of these returned error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_DEV_STATE
- RTOS_ERR_NULL_PTR
- RTOS_ERR_EP_QUEUING
- RTOS_ERR_RX
- RTOS_ERR_NOT_READY
- RTOS_ERR_EP_INVALID
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_IS_OWNER
- RTOS_ERR_INVALID_EP_STATE
- RTOS_ERR_ABORT
- RTOS_ERR_TIMEOUT

Returned Value

- Number of octets received, if no errors are returned.
- 0, if any errors returned.

Notes / Warnings

1. This function SHOULD NOT be called from interrupt service routine (ISR).
2. Receive buffer must be aligned with a single word (minimum).

USBD_IntrRxAsync()

Description

Receives data on Interrupt OUT endpoint asynchronously.

Files

usbd_ep.h/usbd_ep.c

Prototype

```
void USBD_IntrRxAsync (CPU_INT08U        dev_nbr,
                       CPU_INT08U        ep_addr,
                       void             *p_buf,
                       CPU_INT32U        buf_len,
                       USBD_ASYNC_FNCT   async_fnct,
                       void             *p_async_arg,
                       RTOS_ERR         *p_err)
```

Arguments

dev_nbr

Device number.

ep_addr

Endpoint address.

`p_buf`

Pointer to the destination buffer to receive data (see Note #1).

`buf_len`

Number of octets to receive.

`async_fnct`

Function that will be invoked upon completion of receive operation.

`p_async_arg`

Pointer to argument that will be passed as parameter of 'async_fnct'.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`
- `RTOS_ERR_NULL_PTR`
- `RTOS_ERR_EP_QUEUING`
- `RTOS_ERR_RX`
- `RTOS_ERR_NOT_READY`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_EP_STATE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

None.

Notes / Warnings

1. Receive buffer must be aligned with a single word (minimum).

USBD_IntrTx()

Description

Sends data on Interrupt IN endpoint.

Files

`usbd_ep.h/usbd_ep.c`

Prototype

```
CPU_INT32U USBD_IntrTx (CPU_INT08U     dev_nbr,
                        CPU_INT08U     ep_addr,
                        void          *p_buf,
                        CPU_INT32U     buf_len,
                        CPU_INT16U     timeout_ms,
                        CPU_BOOLEAN    end,
                        RTOS_ERR      *p_err)
```

Arguments

`dev_nbr`

Device number.

`ep_addr`

Endpoint address.

`p_buf`

Pointer to the buffer of data that will be transmitted (see Note #2).

`buf_len`

Number of octets to transmit.

`timeout_ms`

Timeout in milliseconds.

`end`

End-of-transfer flag (see Note #3).

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`
- `RTOS_ERR_NULL_PTR`
- `RTOS_ERR_EP_QUEUING`
- `RTOS_ERR_TX`
- `RTOS_ERR_NOT_READY`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_EP_STATE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

- Number of octets transmitted, if no errors are returned.
- 0, if any errors returned.

Notes / Warnings

1. This function SHOULD NOT be called from interrupt service routine (ISR).
2. Transmit buffer must be aligned with a single word (minimum).
3. If end-of-transfer is set and transfer length is multiple of maximum packet size, a zero-length packet is transferred to indicate a short transfer to the host.

USBD_IntrTxAsync()

Description

Sends data on the Interrupt IN endpoint asynchronously.

Files

`usbd_ep.h/usbd_ep.c`

Prototype

```
void USBD_IntrTxAsync (CPU_INT08U        dev_nbr,
                       CPU_INT08U        ep_addr,
                       void             *p_buf,
                       CPU_INT32U        buf_len,
                       USBD_ASYNC_FNCT   async_fnct,
                       void             *p_async_arg,
                       CPU_BOOLEAN       end,
                       RTOS_ERR         *p_err)
```

Arguments

`dev_nbr`

Device number.

`ep_addr`

Endpoint address.

`p_buf`

Pointer to the buffer of data that will be transmitted (see Note #1).

`buf_len`

Number of octets to transmit.

`async_fnct`

Function that will be invoked upon completion of transmit operation.

`p_async_arg`

Pointer to the argument that will be passed as parameter of '`async_fnct`'.

`end`

End-of-transfer flag (see Note #2).

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`
- `RTOS_ERR_NULL_PTR`
- `RTOS_ERR_EP_QUEUING`

- RTOS_ERR_TX
- RTOS_ERR_NOT_READY
- RTOS_ERR_EP_INVALID
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_IS_OWNER
- RTOS_ERR_INVALID_EP_STATE
- RTOS_ERR_ABORT
- RTOS_ERR_TIMEOUT

Returned Value

None.

Notes / Warnings

1. Transmit buffer must be aligned with a single word (minimum).
2. If end-of-transfer is set and transfer length is multiple of maximum packet size, a zero-length packet is transferred to indicate a short transfer to the host.

USBD_IsocRxAsync()

Description

Receives data on an isochronous OUT endpoint asynchronously.

Files

usbd_ep.h/usbd_ep.c

Prototype

```
void USBD_IsocRxAsync (CPU_INT08U       dev_nbr,
                       CPU_INT08U       ep_addr,
                       void            *p_buf,
                       CPU_INT32U       buf_len,
                       USBD_ASYNC_FNCT  async_fnct,
                       void            *p_async_arg,
                       RTOS_ERR        *p_err)
```

Arguments

dev_nbr

Device number.

ep_addr

Endpoint address.

p_buf

Pointer to the destination buffer to receive data (see Note #1).

buf_len

Number of octets to receive.

async_fnct

Function that will be invoked upon completion of a receive operation.

`p_async_arg`

Pointer to the argument that will be passed as parameter of ' `async_fnct` '.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`
- `RTOS_ERR_NULL_PTR`
- `RTOS_ERR_EP_QUEUING`
- `RTOS_ERR_RX`
- `RTOS_ERR_NOT_READY`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_EP_STATE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

None.

Notes / Warnings

1. Receive buffer must be aligned with a single word (minimum).

USBD_IsocTxAsync()

Description

Sends data on an isochronous IN endpoint asynchronously.

Files

`usbd_ep.h/usbd_ep.c`

Prototype

```
void USBD_IsocTxAsync (CPU_INT08U       dev_nbr,
                       CPU_INT08U       ep_addr,
                       void            *p_buf,
                       CPU_INT32U       buf_len,
                       USBD_ASYNC_FNCT  async_fnct,
                       void            *p_async_arg,
                       RTOS_ERR        *p_err)
```

Arguments

`dev_nbr`

Device number.

`ep_addr`

Endpoint address.

`p_buf`

Pointer to the buffer of data that will be transmitted (see Note #1).

`buf_len`

Number of octets to transmit.

`async_fnct`

Function that will be invoked upon completion of transmit operation.

`p_async_arg`

Pointer to the argument that will be passed as parameter of ' `async_fnct` '.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`
- `RTOS_ERR_NULL_PTR`
- `RTOS_ERR_EP_QUEUING`
- `RTOS_ERR_TX`
- `RTOS_ERR_NOT_READY`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_EP_STATE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

None.

Notes / Warnings

1. Transmit buffer must be aligned with a single word (minimum).

USBD_CtrlRx()

Description

Receives data on the Control OUT endpoint.

Files

`usbd_ep.h/usbd_ep.c`

Prototype

```
CPU_INT32U USBD_CtrlRx (CPU_INT08U    dev_nbr,
                        void          *p_buf,
                        CPU_INT32U    buf_len,
                        CPU_INT16U    timeout_ms,
                        RTOS_ERR      *p_err)
```

Arguments

`dev_nbr`

Device number.

`p_buf`

Pointer to the destination buffer to receive data.

`buf_len`

Number of octets to receive.

`timeout_ms`

Timeout in milliseconds.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`
- `RTOS_ERR_NULL_PTR`
- `RTOS_ERR_EP_QUEUING`
- `RTOS_ERR_RX`
- `RTOS_ERR_NOT_READY`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_EP_STATE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

- Number of octets received, if no errors are returned.
- 0, if any errors returned.

Notes / Warnings

None.

USBD_CtrlTx()

Description

Sends data on the Control IN endpoint.

Files

usbd_ep.h/usbd_ep.c

Prototype

```
CPU_INT32U USBD_CtrlTx (CPU_INT08U    dev_nbr,
                        void         *p_buf,
                        CPU_INT32U    buf_len,
                        CPU_INT16U    timeout_ms,
                        CPU_BOOLEAN   end,
                        RTOS_ERR     *p_err)
```

Arguments

dev_nbr

Device number.

p_buf

Pointer to the buffer of data that will be sent.

buf_len

Number of octets to transmit.

timeout_ms

Timeout in milliseconds.

end

End-of-transfer flag (see Note #1).

p_err

Pointer to the variable that will receive one of these returned error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_DEV_STATE
- RTOS_ERR_NULL_PTR
- RTOS_ERR_EP_QUEUING
- RTOS_ERR_TX
- RTOS_ERR_NOT_READY
- RTOS_ERR_EP_INVALID
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_INVALID_ARG
- RTOS_ERR_IS_OWNER
- RTOS_ERR_INVALID_EP_STATE
- RTOS_ERR_ABORT
- RTOS_ERR_TIMEOUT

Returned Value

- Number of octets transmitted, if no errors are returned.
- 0, if any errors returned.

Notes / Warnings

1. If end-of-transfer is set and transfer length is multiple of maximum packet size, a zero-length packet is transferred to indicate a short transfer to the host.

USBD_EP_MaxPktSizeGet()

Description

Retrieves the endpoint maximum packet size.

Files

`usbd_ep.h/usbd_ep.c`

Prototype

```
CPU_INT16U USBD_EP_MaxPktSizeGet (CPU_INT08U    dev_nbr,
                                  CPU_INT08U    ep_addr,
                                  RTOS_ERR     *p_err)
```

Arguments

`dev_nbr`

Device number.

`ep_addr`

Endpoint address.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_EP_INVALID`

Returned Value

- Maximum packet size, if no errors are returned.
- 0, if any errors returned.

Notes / Warnings

None.

USBD_EP_MaxNbrOpenGet()

Description

Retrieves the maximum number of opened endpoints.

Files

`usbd_ep.h/usbd_ep.c`

Prototype

```
CPU_INT08U USBD_EP_MaxNbrOpenGet (CPU_INT08U   dev_nbr)
```

Arguments

`dev_nbr`

Device number.

Returned Value

- Maximum number of opened endpoints, if no errors are returned.
- 0, if any errors returned.

Notes / Warnings

None.

USBD_EP_Abort()

Description

Aborts I/O transfer on the endpoint.

Files

`usbd_ep.h/usbd_ep.c`

Prototype

```
void USBD_EP_Abort (CPU_INT08U   dev_nbr,
                    CPU_INT08U   ep_addr,
                    RTOS_ERR    *p_err)
```

Arguments

`dev_nbr`

Device number.

`ep_addr`

Endpoint address.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_NONE_WAITING`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_EP_STATE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

None.

Notes / Warnings

None.

USBD_EP_Stall()

Description

Stalls the non-control endpoint.

Files

`usbd_ep.h/usbd_ep.c`

Prototype

```
void USBD_EP_Stall (CPU_INT08U    dev_nbr,
                    CPU_INT08U    ep_addr,
                    CPU_BOOLEAN   state,
                    RTOS_ERR     *p_err)
```

Arguments

`dev_nbr`

Device number.

`ep_addr`

Endpoint address.

`state`

Endpoint stall state.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_NONE_WAITING`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_FAIL`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_EP_STATE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

None.

Notes / Warnings

None.

USBD_EP_IsStalled()

Description

Gets the stall status of a non-control endpoint.

Files

`usbd_ep.h/usbd_ep.c`

Prototype

```
CPU_BOOLEAN USBD_EP_IsStalled (CPU_INT08U    dev_nbr,
                               CPU_INT08U    ep_addr,
                               RTOS_ERR     *p_err)
```

Arguments

`dev_nbr`

Device number.

`ep_addr`

Endpoint address.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_EP_INVALID`

Returned Value

- `DEF_TRUE` , if endpoint is stalled.
- `DEF_FALSE` , if endpoint is not stalled.

Notes / Warnings

None.

USB Device CDC API

USB Device CDC API

- USBD_CDC_ConfigureMemSeg()
- USBD_CDC_Init()
- USBD_CDC_Add()
- USBD_CDC_ConfigAdd()
- USBD_CDC_IsConn()
- USBD_CDC_DataIF_Add()
- USBD_CDC_DataRx()
- USBD_CDC_DataTx()
- USBD_CDC_Notify()

USBD_CDC_ConfigureMemSeg()

Description

Configures the memory segment to use when allocating control data.

Files

`usbd_cdc.h/usbd_cdc.c`

Prototype

```
void USBD_CDC_ConfigureMemSeg (MEM_SEG *p_mem_seg)
```

Arguments

`p_mem_seg`

Pointer to memory segment to use when allocating control data. `DEF_NULL` means general purpose heap segment.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the CDC class is initialized via the `USBD_CDC_Init()` function.

USBD_CDC_Init()

Description

Initializes CDC class.

Files

`usbd_cdc.h/usbd_cdc.c`

Prototype

```
void USBD_CDC_Init (USBD_CDC_QTY_CFG   *p_qty_cfg,
                    RTOS_ERR           *p_err)
```

Arguments

`p_qty_cfg`

Pointer to CDC bas class configuration structure.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_SEG_OVF`

Returned Value

None.

Notes / Warnings

None.

USBD_CDC_Add()

Description

Adds a new instance of the CDC class.

Files

`usbd_cdc.h/usbd_cdc.c`

Prototype

```
CPU_INT08U USBD_CDC_Add (CPU_INT08U             subclass,
                         USBD_CDC_SUBCLASS_DRV  *p_subclass_drv,
                         void                   *p_subclass_arg,
                         CPU_INT08U             protocol,
                         CPU_BOOLEAN            notify_en,
                         CPU_INT16U             notify_interval,
                         RTOS_ERR               *p_err)
```

Arguments

`subclass`

Communication class subclass subcode (see Note #1).

`p_subclass_drv`

Pointer to the CDC subclass driver.

`p_subclass_arg`

Pointer to the CDC subclass driver argument.

`protocol`

Communication class protocol code.

`notify_en`

Notification enabled :

DEF_ENABLED Enable CDC class notifications.

DEF_DISABLED Disable CDC class notifications.

`notify_interval`

Notification interval in milliseconds (must be a power of 2).

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_CLASS_INSTANCE_ALLOC`

Returned Value

- CDC class instance number, if no errors are returned.
- `USBD_CDC_NBR_NONE` , if any errors returned.

Notes / Warnings

1. Communication class subclass codes are defined in ' `usbd_cdc.h` ' ' `USBD_CDC_SUBCLASS_XXXX` '.

USBD_CDC_ConfigAdd()

Description

Adds a CDC instance into the USB device configuration.

Files

`usbd_cdc.h/usbd_cdc.c`

Prototype

```
CPU_BOOLEAN USBD_CDC_ConfigAdd (CPU_INT08U    class_nbr,
                                CPU_INT08U    dev_nbr,
                                CPU_INT08U    config_nbr,
                                RTOS_ERR     *p_err)
```

Arguments

`class_nbr`

Class instance number.

`dev_nbr`

Device number.

`config_nbr`

Configuration index to which to add the new CDC class interface.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`
- `RTOS_ERR_IF_ALT_ALLOC`
- `RTOS_ERR_ALLOC`

- RTOS_ERR_IF_GRP_ALLOC
- RTOS_ERR_CLASS_INSTANCE_ALLOC
- RTOS_ERR_ALREADY_EXISTS
- RTOS_ERR_IF_ALLOC
- RTOS_ERR_INVALID_ARG
- RTOS_ERR_EP_ALLOC
- RTOS_ERR_EP_NONE_AVAIL

Returned Value

- DEF_YES , if the CDC class instance was added to USB device configuration successfully.
- DEF_NO , if the instance was not added successfully.

Notes / Warnings

None.

USBD_CDC_IsConn()

Description

Gets the CDC class connection state.

Files

usbd_cdc.h/usbd_cdc.c

Prototype

```
CPU_BOOLEAN USBD_CDC_IsConn (CPU_INT08U   class_nbr)
```

Arguments

class_nbr

Class instance number.

Returned Value

- DEF_YES , if CDC class is connected.
- DEF_NO , if CDC class is not connected.

Notes / Warnings

None.

USBD_CDC_DataIF_Add()

Description

Adds a data interface class to the CDC communication interface class.

Files

usbd_cdc.h/usbd_cdc.c

Prototype

```
CPU_INT08U USBD_CDC_DataIF_Add (CPU_INT08U     class_nbr,
                                CPU_BOOLEAN    isoc_en,
                                CPU_INT08U     protocol,
                                RTOS_ERR      *p_err)
```

Arguments

`class_nbr`

Class instance number.

`isoc_en`

Data interface isochronous enable (see Note #1) :

- `DEF_ENABLED` Data interface uses isochronous EPs.
- `DEF_DISABLED` Data interface uses bulk EPs.

`protocol`

Data interface protocol code.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ALLOC`

Returned Value

Data interface number.

Notes / Warnings

1. The value of '`isoc_en`' must be `DEF_DISABLED`. Isochronous EPs are not supported.

USBD_CDC_DataRx()

Description

Receives data on the CDC data interface.

Files

`usbd_cdc.h/usbd_cdc.c`

Prototype

```
CPU_INT32U USBD_CDC_DataRx (CPU_INT08U   class_nbr,
                            CPU_INT08U   data_if_nbr,
                            CPU_INT08U  *p_buf,
                            CPU_INT32U   buf_len,
                            CPU_INT16U   timeout,
                            RTOS_ERR    *p_err)
```

Arguments

`class_nbr`

Class instance number.

`data_if_nbr`

CDC data interface number.

`p_buf`

Pointer to the destination buffer to receive data.

buf_len

Number of octets to receive.

timeout

Timeout in milliseconds.

p_err

Pointer to the variable that will receive one of these returned error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_DEV_STATE
- RTOS_ERR_NULL_PTR
- RTOS_ERR_EP_QUEUING
- RTOS_ERR_RX
- RTOS_ERR_NOT_SUPPORTED
- RTOS_ERR_NOT_READY
- RTOS_ERR_EP_INVALID
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_INVALID_ARG
- RTOS_ERR_IS_OWNER
- RTOS_ERR_INVALID_CLASS_STATE
- RTOS_ERR_INVALID_EP_STATE
- RTOS_ERR_ABORT
- RTOS_ERR_TIMEOUT

Returned Value

- Number of octets received, if no errors are returned.
- 0, if any errors returned.

Notes / Warnings

None.

USBD_CDC_DataTx()

Description

Sends data on the CDC data interface.

Files

usbd_cdc.h/usbd_cdc.c

Prototype

```
CPU_INT32U USBD_CDC_DataTx (CPU_INT08U    class_nbr,
                            CPU_INT08U    data_if_nbr,
                            CPU_INT08U   *p_buf,
                            CPU_INT32U    buf_len,
                            CPU_INT16U    timeout,
                            RTOS_ERR     *p_err)
```

Arguments

`class_nbr`

Class instance number.

`data_if_nbr`

CDC data interface number.

`p_buf`

Pointer to the buffer of data that will be transmitted.

`buf_len`

Number of octets to transmit.

`timeout`

Timeout in milliseconds.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`
- `RTOS_ERR_NULL_PTR`
- `RTOS_ERR_EP_QUEUING`
- `RTOS_ERR_TX`
- `RTOS_ERR_NOT_SUPPORTED`
- `RTOS_ERR_NOT_READY`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_CLASS_STATE`
- `RTOS_ERR_INVALID_EP_STATE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

- Number of octets transmitted, if no errors are returned.
- 0, if any errors returned.

Notes / Warnings

None.

USBD_CDC_Notify()

Description

Sends a communication interface class notification to the host.

Files

`usbd_cdc.h/usbd_cdc.c`

Prototype

```
CPU_BOOLEAN USBD_CDC_Notify (CPU_INT08U   class_nbr,
                             CPU_INT08U   notification,
                             CPU_INT16U   value,
                             CPU_INT08U  *p_buf,
                             CPU_INT16U   data_len,
                             RTOS_ERR    *p_err)
```

Arguments

`class_nbr`

Class instance number.

`notification`

Notification code.

`value`

Notification value.

`p_buf`

Pointer to the notification buffer (see Note #1).

`data_len`

Length of the data portion of the notification.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`
- `RTOS_ERR_NULL_PTR`
- `RTOS_ERR_EP_QUEUING`
- `RTOS_ERR_TX`
- `RTOS_ERR_NOT_READY`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_CLASS_STATE`
- `RTOS_ERR_INVALID_EP_STATE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

None.

Notes / Warnings

1. The notification buffer MUST contain space for the notification header ' `USBD_CDC_NOTIFICATION_HEADER` ' plus the variable-length data portion.

USB Device ACM API

USB Device ACM API

- USBD_ACM_SerialConfigureBufAlignOctets()
- USBD_ACM_SerialConfigureMemSeg()
- USBD_ACM_SerialInit()
- USBD_ACM_SerialAdd()
- USBD_ACM_SerialConfigAdd()
- USBD_ACM_SerialIsConn()
- USBD_ACM_SerialRx()
- USBD_ACM_SerialTx()
- USBD_ACM_SerialLineCtrlGet()
- USBD_ACM_SerialLineCtrlReg()
- USBD_ACM_SerialLineCodingGet()
- USBD_ACM_SerialLineCodingSet()
- USBD_ACM_SerialLineCodingReg()
- USBD_ACM_SerialLineStateSet()
- USBD_ACM_SerialLineStateClr()

USBD_ACM_SerialConfigureBufAlignOctets()

Description

Configures the alignment of the internal buffers.

Files

`usbd_cdc_acm_serial.h/usbd_cdc_acm_serial.c`

Prototype

```
void USBD_ACM_SerialConfigureBufAlignOctets (CPU_SIZE_T  buf_align_octets)
```

Arguments

`buf_align_octets`

Buffer alignment, in octets.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the CDC ACM class is initialized via the `USBD_ACM_SerialInit()` function.

USBD_ACM_SerialConfigureMemSeg()

Description

Configures the memory segment to use when allocating control data and buffers.

Files

`usbd_cdc_acm_serial.h/usbd_cdc_acm_serial.c`

Prototype

```
void  USBD_ACM_SerialConfigureMemSeg (MEM_SEG  *p_mem_seg,
                                      MEM_SEG  *p_mem_seg_buf)
```

Arguments

`p_mem_seg`

Pointer to memory segment to use when allocating control data. Can be the same segment used for `p_mem_seg_buf`. `DEF_NULL` means general purpose heap segment.

`p_mem_seg_buf`

Pointer to memory segment to use when allocating data buffers. Can be the same segment used for `p_mem_seg`. `DEF_NULL` means general purpose heap segment.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the `CDC ACM` class is initialized via the `USBD_ACM_SerialInit()` function.

USBD_ACM_SerialInit()

Description

Initializes the CDC ACM serial emulation subclass.

Files

`usbd_cdc_acm_serial.h/usbd_cdc_acm_serial.c`

Prototype

```
`void USBD_ACM_SerialInit (CPU_INT08U   subclass_instance_qty,                    RTOS_ERR    *p_err)`
```

Arguments

`subclass_instance_qty`

Quantity of CDC ACM subclass instances.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_SEG_OVF`

Returned Value

None.

Notes / Warnings

None.

USBD_ACM_SerialAdd()

Description

Adds a new instance of the CDC ACM serial emulation subclass.

Files

`usbd_cdc_acm_serial.h/usbd_cdc_acm_serial.c`

Prototype

```
CPU_INT08U USBD_ACM_SerialAdd (CPU_INT16U    line_state_interval,
                               CPU_INT16U    call_mgmt_capabilities,
                               RTOS_ERR     *p_err)
```

Arguments

`line_state_interval`

Line state notification interval in milliseconds (value must be a power of 2).

`call_mgmt_capabilities`

Call Management Capabilities bitmap. OR'ed of the following flags:

- `USBD_ACM_SERIAL_CALL_MGMT_DEV` Device handles call management itself.
- `USBD_ACM_SERIAL_CALL_MGMT_DATA_CCI_DCI` Device can send/receive call management information over a Data Class interface.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ALLOC`
- `RTOS_ERR_SUBCLASS_INSTANCE_ALLOC`
- `RTOS_ERR_CLASS_INSTANCE_ALLOC`

Returned Value

CDC ACM serial emulation subclass instance number.

Notes / Warnings

None.

USBD_ACM_SerialConfigAdd()

Description

Adds a CDC ACM subclass class instance into USB device configuration.

Files

`usbd_cdc_acm_serial.h/usbd_cdc_acm_serial.c`

Prototype

```
CPU_BOOLEAN USBD_ACM_SerialConfigAdd (CPU_INT08U    subclass_nbr,
                                      CPU_INT08U    dev_nbr,
                                      CPU_INT08U    config_nbr,
                                      RTOS_ERR     *p_err)
```

Arguments

`subclass_nbr`

CDC ACM serial emulation subclass instance number.

`dev_nbr`

Device number.

`config_nbr`

Configuration index to add new test class interface to.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`
- `RTOS_ERR_IF_ALT_ALLOC`
- `RTOS_ERR_ALLOC`
- `RTOS_ERR_IF_GRP_ALLOC`
- `RTOS_ERR_CLASS_INSTANCE_ALLOC`
- `RTOS_ERR_ALREADY_EXISTS`
- `RTOS_ERR_IF_ALLOC`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_EP_ALLOC`
- `RTOS_ERR_EP_NONE_AVAIL`

Returned Value

- `DEF_YES` , if the CDC ACM serial subclass instance was added to USB device configuration successfully.
- `DEF_NO` , if the instance was not added.

Notes / Warnings

None.

USBD_ACM_SerialIsConn()

Description

Gets the CDC ACM serial emulation subclass connection state.

Files

`usbd_cdc_acm_serial.h/usbd_cdc_acm_serial.c`

Prototype

```
CPU_BOOLEAN USBD_ACM_SerialIsConn (CPU_INT08U   subclass_nbr)
```

Arguments

`subclass_nbr`

CDC ACM serial emulation subclass instance number.

Returned Value

- `DEF_YES` , CDC ACM serial emulation is connected.
- `DEF_NO` , CDC ACM serial emulation is not connected.

Notes / Warnings

None.

USBD_ACM_SerialRx()

Description

Receives data on the CDC ACM serial emulation subclass.

Files

`usbd_cdc_acm_serial.h/usbd_cdc_acm_serial.c`

Prototype

```
CPU_INT32U USBD_ACM_SerialRx (CPU_INT08U    subclass_nbr,
                              CPU_INT08U   *p_buf,
                              CPU_INT32U    buf_len,
                              CPU_INT16U    timeout,
                              RTOS_ERR     *p_err)
```

Arguments

`subclass_nbr`

CDC ACM serial emulation subclass instance number.

`p_buf`

Pointer to the destination buffer to receive data.

`buf_len`

Number of octets to receive.

`timeout`

Timeout, in milliseconds.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`
- `RTOS_ERR_NULL_PTR`
- `RTOS_ERR_EP_QUEUING`
- `RTOS_ERR_RX`
- `RTOS_ERR_NOT_SUPPORTED`
- `RTOS_ERR_NOT_READY`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_CLASS_STATE`
- `RTOS_ERR_INVALID_EP_STATE`

- RTOS_ERR_ABORT
- RTOS_ERR_TIMEOUT

Returned Value

- Number of octets received, if no errors are returned.
- 0, if any errors are returned.

Notes / Warnings

None.

USBD_ACM_SerialTx()

Description

Sends data on the CDC ACM serial emulation subclass.

Files

usbd_cdc_acm_serial.h/usbd_cdc_acm_serial.c

Prototype

```
CPU_INT32U USBD_ACM_SerialTx (CPU_INT08U   subclass_nbr,
                              CPU_INT08U  *p_buf,
                              CPU_INT32U   buf_len,
                              CPU_INT16U   timeout,
                              RTOS_ERR    *p_err)
```

Arguments

subclass_nbr

CDC ACM serial emulation subclass instance number.

p_buf

Pointer to the buffer of data that will be transmitted.

buf_len

Number of octets to transmit.

timeout

Timeout in milliseconds.

p_err

Pointer to the variable that will receive one of these returned error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_DEV_STATE
- RTOS_ERR_NULL_PTR
- RTOS_ERR_EP_QUEUING
- RTOS_ERR_TX
- RTOS_ERR_NOT_SUPPORTED
- RTOS_ERR_NOT_READY
- RTOS_ERR_EP_INVALID
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_WOULD_OVF

- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_INVALID_ARG
- RTOS_ERR_IS_OWNER
- RTOS_ERR_INVALID_CLASS_STATE
- RTOS_ERR_INVALID_EP_STATE
- RTOS_ERR_ABORT
- RTOS_ERR_TIMEOUT

Returned Value

- Number of octets transmitted, if no errors are returned.
- 0, if any errors are returned.

Notes / Warnings

None.

USBD_ACM_SerialLineCtrlGet()

Description

Returns the state of control lines.

Files

usbd_cdc_acm_serial.h/usbd_cdc_acm_serial.c

Prototype

```
CPU_INT08U USBD_ACM_SerialLineCtrlGet (CPU_INT08U    subclass_nbr,
                                       RTOS_ERR     *p_err)
```

Arguments

subclass_nbr

CDC ACM serial emulation subclass instance number.

p_err

Pointer to the variable that will receive one of these returned error codes from this function :

- RTOS_ERR_NONE

Returned Value

Bit-field with the state of the control line.

- USBD_ACM_SERIAL_CTRL_BREAK
- USBD_ACM_SERIAL_CTRL_RTS
- USBD_ACM_SERIAL_CTRL_DTR

Notes / Warnings

None.

USBD_ACM_SerialLineCtrlReg()

Description

Sets the line control change notification callback.

Files

`usbd_cdc_acm_serial.h/usbd_cdc_acm_serial.c`

Prototype

```
void USBD_ACM_SerialLineCtrlReg (CPU_INT08U                    subclass_nbr,
                                 USBD_ACM_SERIAL_LINE_CTRL_CHNGD   line_ctrl_chngd,
                                 void                          *p_arg,
                                 RTOS_ERR                      *p_err)
```

Arguments

`subclass_nbr`

CDC ACM serial emulation subclass instance number.

`line_ctrl_chngd`

Line control change notification callback (see Note #1).

`p_arg`

Pointer to the callback argument.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`

Returned Value

ACM serial emulation subclass device number.

Notes / Warnings

1. The callback specified by ' `line_ctrl_chngd` ' argument notifies of changes in the control signals to the application.

The line control notification function uses the following prototype:

```
void   AppLineCtrlChngd   (CPU_INT08U  subclass_nbr,
                          CPU_INT08U  events,
                          CPU_INT08U  events_chngd,
                          void       *p_arg);

    Argument(s) : subclass_nbr      CDC ACM serial emulation subclass instance number.

                  events            Current line state. The line state is a OR'ed of the following flags:

                                    USBD_ACM_SERIAL_CTRL_BREAK
                                    USBD_ACM_SERIAL_CTRL_RTS
                                    USBD_ACM_SERIAL_CTRL_DTR

                  events_chngd      Line state flags that have changed.

                  p_arg             Pointer to callback argument.
```

USBD_ACM_SerialLineCodingGet()

Description

Gets the current state of the line coding.

Files

`usbd_cdc_acm_serial.h/usbd_cdc_acm_serial.c`

Prototype

```
void USBD_ACM_SerialLineCodingGet (CPU_INT08U                   subclass_nbr,
                                   USBD_ACM_SERIAL_LINE_CODING  *p_line_coding,
                                   RTOS_ERR                     *p_err)
```

Arguments

`subclass_nbr`

CDC ACM serial emulation subclass instance number.

`p_line_coding`

Pointer to the structure where the current line coding will be stored.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

None.

USBD_ACM_SerialLineCodingSet()

Description

Sets a new line coding.

Files

`usbd_cdc_acm_serial.h/usbd_cdc_acm_serial.c`

Prototype

```
void USBD_ACM_SerialLineCodingSet (CPU_INT08U                   subclass_nbr,
                                   USBD_ACM_SERIAL_LINE_CODING  *p_line_coding,
                                   RTOS_ERR                     *p_err)
```

Arguments

`subclass_nbr`

CDC ACM serial emulation subclass instance number.

`p_line_coding`

Pointer to the structure that contains the new line coding.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`

- RTOS_ERR_INVALID_ARG

Returned Value

None.

Notes / Warnings

None.

USBD_ACM_SerialLineCodingReg()

Description

Sets the line coding change notification callback.

Files

usbd_cdc_acm_serial.h/usbd_cdc_acm_serial.c

Prototype

```
void USBD_ACM_SerialLineCodingReg (CPU_INT08U                         subclass_nbr,
                                   USBD_ACM_SERIAL_LINE_CODING_CHNGD   line_coding_chngd,
                                   void                               *p_arg,
                                   RTOS_ERR                           *p_err)
```

Arguments

subclass_nbr

CDC ACM serial emulation subclass instance number.

line_coding_chngd

Line coding change notification callback (see Note #1).

p_arg

Pointer to the callback argument.

p_err

Pointer to the variable that will receive one of these returned error codes from this function :

- RTOS_ERR_NONE

Returned Value

None.

Notes / Warnings

1. This callback notifies of changes in the line coding to the application.
   The line coding change notification function has the following prototype:

```
CPU_BOOLEAN  AppLineCodingChngd  (CPU_INT08U              subclass_nbr,
                                  USBD_ACM_SERIAL_LINE_CODING  *p_line_coding,
                                  void                    *p_arg);

   Argument(s) : subclass_nbr       CDC ACM serial emulation subclass instance number.

              p_line_coding Pointer to the line coding structure.

              p_arg Pointer to the callback argument.

Return(s) :     DEF_OK, if line coding is supported by the application.

              DEF_FAIL, if line coding is NOT supported by the application.
```

USBD_ACM_SerialLineStateSet()

Description

Sets a line state event(s).

Files

`usbd_cdc_acm_serial.h/usbd_cdc_acm_serial.c`

Prototype

```
void USBD_ACM_SerialLineStateSet (CPU_INT08U   subclass_nbr,
                                  CPU_INT08U   events,
                                  RTOS_ERR    *p_err)
```

Arguments

`subclass_nbr`

CDC ACM serial emulation subclass instance number.

`events`

Line state event(s) to set. OR'ed of the following flags:

- `USBD_ACM_SERIAL_STATE_DCD`
- `USBD_ACM_SERIAL_STATE_DSR`
- `USBD_ACM_SERIAL_STATE_BREAK`
- `USBD_ACM_SERIAL_STATE_RING`
- `USBD_ACM_SERIAL_STATE_FRAMING`
- `USBD_ACM_SERIAL_STATE_PARITY`
- `USBD_ACM_SERIAL_STATE_OVERUN`

`p_err`

Pointer to variable that will receive the return error code from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`
- `RTOS_ERR_NULL_PTR`
- `RTOS_ERR_EP_QUEUING`
- `RTOS_ERR_TX`
- `RTOS_ERR_NOT_READY`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`

- RTOS_ERR_WOULD_OVF
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_IS_OWNER
- RTOS_ERR_INVALID_CLASS_STATE
- RTOS_ERR_INVALID_EP_STATE
- RTOS_ERR_ABORT
- RTOS_ERR_TIMEOUT

Returned Value

None.

Notes / Warnings

None.

USBD_ACM_SerialLineStateClr()

Description

Clears a line state event(s).

Files

usbd_cdc_acm_serial.h/usbd_cdc_acm_serial.c

Prototype

```
void USBD_ACM_SerialLineStateClr (CPU_INT08U    subclass_nbr,
                                  CPU_INT08U    events,
                                  RTOS_ERR     *p_err)
```

Arguments

subclass_nbr

CDC ACM serial emulation subclass instance number.

events

Line state event(s) set to be cleared. OR'ed of the following flags (see Note #1) :

- USBD_ACM_SERIAL_STATE_DCD Set DCD signal (Rx carrier).
- USBD_ACM_SERIAL_STATE_DSR Set DSR signal (Tx carrier).

p_err

Pointer to the variable that will receive return error code from this function :

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_DEV_STATE
- RTOS_ERR_NULL_PTR
- RTOS_ERR_EP_QUEUING
- RTOS_ERR_TX
- RTOS_ERR_NOT_READY
- RTOS_ERR_EP_INVALID
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_WOULD_OVF

- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_CLASS_STATE`
- `RTOS_ERR_INVALID_EP_STATE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

None.

Notes / Warnings

1. USB PSTN spec ver 1.20 states: "For the irregular signals like break, the incoming ring signal, or the overrun error state, this will reset their values to zero and again will not send another notification until their state changes."

The irregular events are automatically cleared by the ACM serial emulation subclass.

USB Device CDC EEM API

USB Device CDC EEM API

- USBD_CDC_EEM_ConfigureBufAlignOctets()
- USBD_CDC_EEM_ConfigureEchoBufLen()
- USBD_CDC_EEM_ConfigureMemSeg()
- USBD_CDC_EEM_ConfigureRxBuf()
- USBD_CDC_EEM_Init()
- USBD_CDC_EEM_NetIF_Reg()
- USBD_CDC_EEM_Add()
- USBD_CDC_EEM_ConfigAdd()
- USBD_CDC_EEM_IsConn()
- USBD_CDC_EEM_DevNbrGet

USBD_CDC_EEM_ConfigureBufAlignOctets()

Description

Configures the alignment of the internal buffers.

Files

`usbd_cdc_eem.h/usbd_cdc_eem.c`

Prototype

```
void  USBD_CDC_EEM_ConfigureBufAlignOctets (CPU_SIZE_T  buf_align_octets)
```

Arguments

`buf_align_octets`

Buffer alignment, in octets.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the CDC EEM class is initialized via the function `USBD_CDC_EEM_Init()` .

USBD_CDC_EEM_ConfigureEchoBufLen()

Description

Configures the length, in octets, of the buffer used for echo requests.

Files

`usbd_cdc_eem.h/usbd_cdc_eem.c`

Prototype

```
void USBD_CDC_EEM_ConfigureEchoBufLen (CPU_INT16U echo_buf_len)
```

Arguments

`echo_buf_len`

Length, in octets, of the echo buffer.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the CDC EEM class is initialized via the function `USBD_CDC_EEM_Init()`.

USBD_CDC_EEM_ConfigureMemSeg()

Description

Configures the memory segment to use when allocating control data and buffers.

Files

`usbd_cdc_eem.h/usbd_cdc_eem.c`

Prototype

```
void USBD_CDC_EEM_ConfigureMemSeg (MEM_SEG *p_mem_seg,
                                   MEM_SEG *p_mem_seg_buf)
```

Arguments

`p_mem_seg`

Pointer to memory segment to use when allocating control data. Can be the same segment used for `p_mem_seg_buf`. `DEF_NULL` means general purpose heap segment.

`p_mem_seg_buf`

Pointer to memory segment to use when allocating data buffers. Can be the same segment used for `p_mem_seg`. `DEF_NULL` means general purpose heap segment.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the CDC EEM class is initialized via the function `USBD_CDC_EEM_Init()`.

USBD_CDC_EEM_ConfigureRxBuf()

Description

Configures the quantity CDC EEM receive buffer(s).

Files

`usbd_cdc_eem.h/usbd_cdc_eem.c`

Prototype

```
void  USBD_CDC_EEM_ConfigureRxBuf (CPU_INT08U  rx_buf_qty,
                                   CPU_INT32U  rx_buf_len)
```

Arguments

`rx_buf_qty`

Quantity of receive buffer. Unless your USB driver supports URQ queuing, always set to 1.

`rx_buf_len`

Length, in octets, of the receive buffer(s).

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the `CDC EEM` class is initialized via the function `USBD_CDC_EEM_Init()`.

USBD_CDC_EEM_Init()

Description

Initializes the internal structures and variables used by the CDC EEM class.

Files

`usbd_cdc_eem.h/usbd_cdc_eem.c`

Prototype

```
void USBD_CDC_EEM_Init (USBD_CDC_EEM_QTY_CFG   *p_qty_cfg,
                        RTOS_ERR               *p_err)
```

Arguments

`p_qty_cfg`

Pointer to the configuration structure.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`

Returned Value

None.

Notes / Warnings

None.

USBD_CDC_EEM_NetIF_Reg()

Description

Adds a new instance of the CDC EEM class AND registers a new Ethernet controller to the platform manager.

Files

`usbd_cdc_eem.h/usbd_cdc_eem.c`

Prototype

```
CPU_INT08U  USBD_CDC_EEM_NetIF_Reg (CPU_CHAR                    **p_name,
                                    USBD_CDC_EEM_NET_IF_ETHER_CFG  *p_net_if_ether_cfg,
                                    RTOS_ERR                      *p_err)
```

Arguments

`p_name`

Pointer to a string that will receive the Network Interface Name as registered in the platform manager.

`p_net_if_ether_cfg`

Pointer to structure containing the configurations related to the Ethernet Interface.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_SUBCLASS_INSTANCE_ALLOC`
- `RTOS_ERR_ALREADY_EXISTS`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_SEG_OVF`

Returned Value

- Class instance number, if no errors are returned.
- `USBD_CLASS_NBR_NONE` , if any errors are returned.

Notes / Warnings

1. Micrium OS Net Ethernet module MUST be available in your product to be able to use this function.

USBD_CDC_EEM_Add()

Description

Adds a new instance of the CDC EEM class.

Files

`usbd_cdc_eem.h/usbd_cdc_eem.c`

Prototype

```
CPU_INT08U USBD_CDC_EEM_Add (RTOS_ERR   *p_err)
```

Arguments

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`

- RTOS_ERR_SUBCLASS_INSTANCE_ALLOC

Returned Value

- Class instance number, if no errors are returned.
- USBD_CLASS_NBR_NONE , if any errors are returned.

Notes / Warnings

None.

USBD_CDC_EEM_ConfigAdd()

Description

Adds the CDC-EEM class instance into the specified configuration.

Files

usbd_cdc_eem.h/usbd_cdc_eem.c

Prototype

```
void USBD_CDC_EEM_ConfigAdd ( CPU_INT08U    class_nbr,
                              CPU_INT08U    dev_nbr,
                              CPU_INT08U    config_nbr,
                         const CPU_CHAR    *p_if_name,
                              RTOS_ERR     *p_err)
```

Arguments

class_nbr

Class instance number.

dev_nbr

Device number.

config_nbr

Configuration index to which to add the CDC-EEM class instance.

p_if_name

Pointer to the string that contains name of the CDC-EEM interface. Can be DEF_NULL .

p_err

Pointer to the variable that will receive one of these returned error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_DEV_STATE
- RTOS_ERR_IF_ALT_ALLOC
- RTOS_ERR_ALLOC
- RTOS_ERR_IF_ALLOC
- RTOS_ERR_INVALID_ARG
- RTOS_ERR_EP_ALLOC
- RTOS_ERR_EP_NONE_AVAIL

Returned Value

None.

Notes / Warnings

None.

USBD_CDC_EEM_IsConn()

Description

Gets the CDC-EEM class instance connection state.

Files

`usbd_cdc_eem.h/usbd_cdc_eem.c`

Prototype

```
CPU_BOOLEAN USBD_CDC_EEM_IsConn (CPU_INT08U   class_nbr)
```

Arguments

`class_nbr`

Class instance number.

Returned Value

- `DEF_YES` , if the class instance is connected.
- `DEF_NO` , if the class instance is NOT connected.

Notes / Warnings

None.

USBD_CDC_EEM_DevNbrGet

Description

Gets the device number associated to this CDC-EEM class instance.

Files

`usbd_cdc_eem.h/usbd_cdc_eem.c`

Prototype

```
CPU_INT08U USBD_CDC_EEM_DevNbrGet (CPU_INT08U  class_nbr,
                                   RTOS_ERR   *p_err)
```

Arguments

`class_nbr`

Class instance number.

`p_err`

Pointer to the variable that will receive the returned error code from this function:

- `RTOS_ERR_NONE`

Returned Value

Device number.

Notes / Warnings

None.

USB Device HID API

USB Device HID API

- USBD_HID_ConfigureBufAlignOctets()
- USBD_HID_ConfigureReportID_Qty()
- USBD_HID_ConfigurePushPopItemsQty()
- USBD_HID_ConfigureMemSeg()
- USBD_HID_ConfigureTmrTaskStk()
- USBD_HID_Init()
- USBD_HID_TmrTaskPrioSet()
- USBD_HID_Add()
- USBD_HID_ConfigAdd()
- USBD_HID_IsConn()
- USBD_HID_Rd()
- USBD_HID_Wr()

USBD_HID_ConfigureBufAlignOctets()

Description

Configures the alignment of the internal buffers.

Files

`usbd_hid.h/usbd_hid.c`

Prototype

```
void  USBD_HID_ConfigureBufAlignOctets (CPU_SIZE_T  buf_align_octets)
```

Arguments

`buf_align_octets`

Buffer alignment, in octets.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the HID class is initialized via the `USBD_HID_Init()` function.

USBD_HID_ConfigureReportID_Qty()

Description

Configures the quantity of report IDs.

Files

`usbd_hid.h/usbd_hid.c`

Prototype

```
void USBD_HID_ConfigureReportID_Qty (CPU_INT08U report_id_qty)
```

Arguments

`report_id_qty`

Quantity of report IDs.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the HID class is initialized via the USBD_HID_Init() function.

USBD_HID_ConfigurePushPopItemsQty()

Description

Configures the quantity of push/pop items.

Files

`usbd_hid.h/usbd_hid.c`

Prototype

```
void USBD_HID_ConfigurePushPopItemsQty (CPU_INT08U push_pop_items_qty)
```

Arguments

`push_pop_items_qty`

Quantity of Push/pop items.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the HID class is initialized via the `USBD_HID_Init()` function.

USBD_HID_ConfigureMemSeg()

Description

Configures the memory segment to use when allocating control data and buffers.

Files

`usbd_hid.h/usbd_hid.c`

Prototype

```
void USBD_HID_ConfigureMemSeg (MEM_SEG *p_mem_seg,
                               MEM_SEG *p_mem_seg_buf)
```

Arguments

`p_mem_seg`

Pointer to memory segment to use when allocating control data. Can be the same segment used for `p_mem_seg_buf`. `DEF_NULL` means general purpose heap segment.

`p_mem_seg_buf`

Pointer to memory segment to use when allocating data buffers. Can be the same segment used for `p_mem_seg`. `DEF_NULL` means general purpose heap segment.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the HID class is initialized via the `USBD_HID_Init()` function.

USBD_HID_ConfigureTmrTaskStk()

Description

Configures the timer task stack.

Files

`usbd_hid.h/usbd_hid.c`

Prototype

```
void  USBD_HID_ConfigureTmrTaskStk (CPU_INT32U   stk_size_elements,
                                    void        *p_stk)
```

Arguments

`stk_size_elements`

Size of the stack, in stack elements.

`p_stk`

Pointer to base of the stack.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the HID class is initialized via the `USBD_HID_Init()` function.

USBD_HID_Init()

Description

Initialize HID class.

Files

`usbd_hid.h/usbd_hid.c`

Prototype

```
void USBD_HID_Init (const  USBD_HID_QTY_CFG  *p_qty_cfg,
                    RTOS_ERR        *p_err)
```

Arguments

`p_qty_cfg`

Pointer to HID class configuration structure.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_INVALID_CFG`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`

Returned Value

None.

Notes / Warnings

None.

USBD_HID_TmrTaskPrioSet()

Description

Sets priority of the HID timer task.

Files

`usbd_hid.h/usbd_hid.c`

Prototype

```
void  USBD_HID_TmrTaskPrioSet (RTOS_TASK_PRIO   prio,
                               RTOS_ERR       *p_err)
```

Arguments

`prio`

Priority of the HID timer task.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_ARG`

Returned Value

None.

Notes / Warnings

None.

USBD_HID_Add()

Description

Adds a new instance of the HID class.

Files

`usbd_hid.h/usbd_hid.c`

Prototype

```
CPU_INT08U USBD_HID_Add ( CPU_INT08U           subclass,
                          CPU_INT08U           protocol,
                          USBD_HID_COUNTRY_CODE    country_code,
                    const CPU_INT08U          *p_report_desc,
                          CPU_INT16U           report_desc_len,
                    const CPU_INT08U          *p_phy_desc,
                          CPU_INT16U           phy_desc_len,
                          CPU_INT16U           interval_in,
                          CPU_INT16U           interval_out,
                          CPU_BOOLEAN          ctrl_rd_en,
                          USBD_HID_CALLBACK   *p_hid_callback,
                          RTOS_ERR            *p_err)
```

Arguments

`subclass`

Subclass code.

`protocol`

Protocol code.

`country_code`

Country code ID.

`p_report_desc`

Pointer to the report descriptor structure. Content MUST be persistent.

`report_desc_len`

Report descriptor length.

`p_phy_desc`

Pointer to the physical descriptor structure. Content MUST be persistent.

`phy_desc_len`

Physical descriptor length.

`interval_in`

Polling interval for input transfers, in milliseconds. It must be a power of 2.

`interval_out`

Polling interval for output transfers, in milliseconds. It must be a power of 2. Used only when read operations are not through control transfers.

`ctrl_rd_en`

Enable read operations through the control transfers.

`p_hid_callback`

Pointer to HID descriptor and request callback structure. Content MUST be persistent.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ALLOC`
- `RTOS_ERR_CLASS_INSTANCE_ALLOC`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_SEG_OVF`

Returned Value

- Class instance number, if no errors are returned.
- `USBD_CLASS_NBR_NONE` , if any errors are returned.

Notes / Warnings

None.

USBD_HID_ConfigAdd()

Description

Adds the HID class instance to the USB device configuration (see Note #1).

Files

`usbd_hid.h/usbd_hid.c`

Prototype

```
CPU_BOOLEAN USBD_HID_ConfigAdd (CPU_INT08U    class_nbr,
                                CPU_INT08U    dev_nbr,
                                CPU_INT08U    config_nbr,
                                RTOS_ERR     *p_err)
```

Arguments

`class_nbr`

Class instance number.

`dev_nbr`

Device number.

`config_nbr`

Configuration index to add HID class instance to.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`

- `RTOS_ERR_IF_ALT_ALLOC`
- `RTOS_ERR_ALLOC`
- `RTOS_ERR_CLASS_INSTANCE_ALLOC`
- `RTOS_ERR_IF_ALLOC`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_EP_ALLOC`
- `RTOS_ERR_EP_NONE_AVAIL`

Returned Value

- `DEF_YES`, if the HID class instance was added to USB device configuration successfully.
- `DEF_NO`, if it fails to be added.

Notes / Warnings

1. Called several times, it creates multiple instances and configurations. For example, the following architecture could be created:

```
HS
|-- Configuration 0 (HID class 0)
                (HID class 1)
                (HID class 2)
    |-- Interface 0
|-- Configuration 1 (HID class 0)
    |-- Interface 0
```

2. Configuration Descriptor corresponding to a HID device uses the following format:

```
Configuration Descriptor
|-- Interface Descriptor (HID class)
    |-- Endpoint Descriptor (Interrupt IN)
    |-- Endpoint Descriptor (Interrupt OUT) - optional
```

USBD_HID_IsConn()

Description

Gets the HID class connection state.

Files

`usbd_hid.h/usbd_hid.c`

Prototype

```
CPU_BOOLEAN USBD_HID_IsConn (CPU_INT08U   class_nbr)
```

Arguments

`class_nbr`

Class instance number.

Returned Value

- `DEF_YES`, if HID class is connected.
- `DEF_NO`, it fails to connect.

Notes / Warnings

None.

USBD_HID_Rd()

Description

Receives data from the host through the Interrupt OUT endpoint. This function is blocking.

Files

`usbd_hid.h/usbd_hid.c`

Prototype

```
CPU_INT32U USBD_HID_Rd (CPU_INT08U    class_nbr,
                        void          *p_buf,
                        CPU_INT32U    buf_len,
                        CPU_INT16U    timeout,
                        RTOS_ERR      *p_err)
```

Arguments

`class_nbr`

Class instance number.

`p_buf`

Pointer to the receive buffer.

`buf_len`

Receive the buffer length, in octets.

`timeout`

Timeout, in milliseconds.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`
- `RTOS_ERR_NULL_PTR`
- `RTOS_ERR_EP_QUEUING`
- `RTOS_ERR_RX`
- `RTOS_ERR_NOT_READY`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_FAIL`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_CLASS_STATE`
- `RTOS_ERR_INVALID_EP_STATE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

- Number of octets received, if no errors are returned.
- 0, if any errors are returned.

Notes / Warnings

None.

USBD_HID_Wr()

Description

Sends data to the host through the Interrupt IN endpoint. This function is blocking.

Files

`usbd_hid.h/usbd_hid.c`

Prototype

```
CPU_INT32U USBD_HID_Wr (CPU_INT08U    class_nbr,
                        void          *p_buf,
                        CPU_INT32U    buf_len,
                        CPU_INT16U    timeout,
                        RTOS_ERR      *p_err)
```

Arguments

`class_nbr`

Class instance number.

`p_buf`

Pointer to the transmit buffer. If more than one input report exists, the first byte must represent the Report ID.

`buf_len`

Transmit buffer length, in octets.

`timeout`

Timeout in milliseconds.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`
- `RTOS_ERR_NULL_PTR`
- `RTOS_ERR_EP_QUEUING`
- `RTOS_ERR_TX`
- `RTOS_ERR_NOT_READY`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_IS_OWNER`

- `RTOS_ERR_INVALID_CLASS_STATE`
- `RTOS_ERR_INVALID_EP_STATE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

- Number of octets sent, if no errors are returned.
- 0, if any errors are returned.

Notes / Warnings

None.

USB Device MSC API

USB Device MSC API

- USBD_MSC_ConfigureBufAlignOctets()
- USBD_MSC_ConfigureDataBufLen()
- USBD_MSC_ConfigureMemSeg()
- USBD_MSC_Init()
- USBD_MSC_Add()
- USBD_MSC_TaskPrioSet()
- USBD_MSC_ConfigAdd()
- USBD_MSC_SCSI_LunAdd()
- USBD_MSC_SCSI_LunAttach()
- USBD_MSC_SCSI_LunDetach()
- USBD_MSC_IsConn()

USBD_MSC_ConfigureBufAlignOctets()

Description

Configures the alignment of the internal buffers.

Files

`usbd_msc.h/usbd_msc.c`

Prototype

```
void USBD_MSC_ConfigureBufAlignOctets (CPU_SIZE_T  buf_align_octets)
```

Arguments

`buf_align_octets`

Buffer alignment, in octets.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the MSC class is initialized via the `USBD_MSC_Init()` function.

USBD_MSC_ConfigureDataBufLen()

Description

Configures the length, in octets, of the buffer used to exchange MSC data with the USB host.

Files

`usbd_msc.h/usbd_msc.c`

Prototype

```
void USBD_MSC_ConfigureDataBufLen (CPU_INT32U data_buf_len)
```

Arguments

`data_buf_len`

Data buffer length, in octets.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the MSC class is initialized via the `USBD_MSC_Init()` function.

USBD_MSC_ConfigureMemSeg()

Description

Configures the memory segment to use when allocating control data and buffers.

Files

`usbd_msc.h/usbd_msc.c`

Prototype

```
void USBD_MSC_ConfigureMemSeg (MEM_SEG *p_mem_seg,
                               MEM_SEG *p_mem_seg_buf)
```

Arguments

`p_mem_seg`

Pointer to memory segment to use when allocating control data. Can be the same segment used for `p_mem_seg_buf`. `DEF_NULL` means general purpose heap segment.

`p_mem_seg_buf`

Pointer to memory segment to use when allocating data buffers. Can be the same segment used for `p_mem_seg`. `DEF_NULL` means general purpose heap segment.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the MSC class is initialized via the `USBD_MSC_Init()` function.

USBD_MSC_Init()

Description

Initializes internal structures and variables used by the Mass Storage Class Bulk Only Transport.

Files

`usbd_msc.h/usbd_msc.c`

Prototype

```
void USBD_MSC_Init (USBD_MSC_CFG  *p_qty_cfg,
                    RTOS_ERR      *p_err)
```

Arguments

`p_qty_cfg`

Pointer to the MSC configuration structure.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`

Returned Value

None.

Notes / Warnings

None.

USBD_MSC_Add()

Description

Adds a new instance of the Mass Storage Class.

Files

`usbd_msc.h/usbd_msc.c`

Prototype

```
CPU_INT08U USBD_MSC_Add (RTOS_TASK_CFG    *p_msc_task_cfg,
                         RTOS_ERR         *p_err)
```

Arguments

`p_msc_task_cfg`

Pointer to the configuration structure of the task to be created for this MSC instance.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`
- `RTOS_ERR_CLASS_INSTANCE_ALLOC`
- `RTOS_ERR_SEG_OVF`

Returned Value

- Class instance number, if no errors are returned.
- `USBD_CLASS_NBR_NONE` , if any errors are returned.

Notes / Warnings

None.

USBD_MSC_TaskPrioSet()

Description

Sets priority of the given MSC class instance.

Files

`usbd_msc.h/usbd_msc.c`

Prototype

```
void  USBD_MSC_TaskPrioSet (CPU_INT08U       class_nbr,
                            RTOS_TASK_PRIO   prio,
                            RTOS_ERR        *p_err)
```

Arguments

`class_nbr`

MSC instance number.

`prio`

Priority of the MSC instance's task.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_ARG`

Returned Value

None.

Notes / Warnings

None.

USBD_MSC_ConfigAdd()

Description

Adds an existing MSC instance to the specified configuration and device.

Files

`usbd_msc.h/usbd_msc.c`

Prototype

```
CPU_BOOLEAN USBD_MSC_ConfigAdd (CPU_INT08U    class_nbr,
                                CPU_INT08U    dev_nbr,
                                CPU_INT08U    config_nbr,
                                RTOS_ERR     *p_err)
```

Arguments

`class_nbr`

MSC instance number.

`dev_nbr`

Device number.

`config_nbr`

Configuration index to which to add the existing MSC interface.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`
- `RTOS_ERR_IF_ALT_ALLOC`
- `RTOS_ERR_ALLOC`
- `RTOS_ERR_CLASS_INSTANCE_ALLOC`
- `RTOS_ERR_IF_ALLOC`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_EP_ALLOC`
- `RTOS_ERR_EP_NONE_AVAIL`

Returned Value

- `DEF_YES` , if the MSC instance is added to USB device configuration successfully.
- `DEF_NO` , if it fails to be added.

Notes / Warnings

`USBD_MSC_ConfigAdd()` adds an Interface descriptor and its associated Endpoint descriptor(s) to the Configuration descriptor. One call to `USBD_MSC_ConfigAdd()` builds the Configuration descriptor corresponding to an MSC device with the following format:

```
Configuration Descriptor
|-- Interface Descriptor (MSC)
|-- Endpoint Descriptor (Bulk OUT)
|-- Endpoint Descriptor (Bulk IN)
```

If `USBD_MSC_ConfigAdd()` is called several times from the application, it creates multiple instances and configurations. For example, the following architecture could be created for a high-speed device:

```
High-speed
|-- Configuration 0
|-- Interface 0 (MSC 0)
|-- Configuration 1
|-- Interface 0 (MSC 0)
|-- Interface 1 (MSC 1)
```

In this example, there are two instances of MSC: 'MSC 0' and 'MSC 1', and two possible configurations for the device: 'Configuration 0' and 'Configuration 1'. 'Configuration 1' is composed of two interfaces.

Each class instance has an association with one of the interfaces. If 'Configuration 1' is activated by the host, it allows the host to access two different functionalities offered by the device.

USBD_MSC_SCSI_LunAdd()

Description

Adds a SCSI logical unit to the MSC interface.

Files

`usbd_msc.h/usbd_msc.c`

Prototype

```
CPU_INT08U USBD_MSC_SCSI_LunAdd (CPU_INT08U           class_nbr,
                                 USBD_SCSI_LU_INFO    *p_lu_info,
                                 USBD_SCSI_LU_FNCTS   *p_lu_fncts,
                                 RTOS_ERR             *p_err)
```

Arguments

`class_nbr`

MSC instance number.

`p_lu_info`

Pointer to the logical unit information structure.

`p_lu_fncts`

Pointer to the structure of callbacks associated to this logical unit. Can be `DEF_NULL`.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ALLOC`

Returned Value

Logical Unit Number.

Notes / Warnings

None.

USBD_MSC_SCSI_LunAttach()

Description

Attach a storage media to the given LUN.

Files

`usbd_msc.h/usbd_msc.c`

Prototype

```
void USBD_MSC_SCSI_LunAttach (CPU_INT08U    class_nbr,
                              CPU_INT08U    lu_nbr,
                              CPU_CHAR     *media_name,
                              RTOS_ERR     *p_err)
```

Arguments

`class_nbr`

MSC instance number.

`lu_nbr`

Logical unit number.

`media_name`

Name of the storage media to use as registered in the platform manager.

`p_err`

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_ALREADY_EXISTS`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

None.

Notes / Warnings

1. As soon as this function is called, the media will be shown as available to the host. In case you are using a removable media (such as an SD card), you MUST call `USBD_MSC_SCSI_LunDetach()` once the removable media is removed to mark it as unavailable for the host.
2. If the LUN has been ejected from the host, calling `USBD_MSC_SCSI_LunDetach()` and `USBD_MSC_SCSI_LunAttach()` will make it re-appear.
3. Use this function with care as ejecting a logical unit in the middle of a transfer may corrupt the file system. The only scenarios where the usage of this function is safe are:
    1. Host: Read, Embedded app: Read
    2. Host: Read, Embedded app: Write

    The following scenarios are not considered safe. Use at your own risk.
    1. Host: Write, Embedded app: Read
    2. Host: Write, Embedded app: Write

USBD_MSC_SCSI_LunDetach()

Description

Detach a storage media from a LUN.

Files

`usbd_msc.h/usbd_msc.c`

Prototype

```
void USBD_MSC_SCSI_LunDetach (CPU_INT08U    class_nbr,
                              CPU_INT08U    lu_nbr,
                              RTOS_ERR     *p_err)
```

Arguments

`class_nbr`

MSC instance number.

`lu_nbr`

Logical unit number.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

None.

Notes / Warnings

1. After a call to this function, the media will be available to the embedded application. The standard file API can be used.
2. Use this function with care as ejecting a logical unit in the middle of a transfer may corrupt the file system. The only scenarios where the usage of this function is considered safe are:
    1. Host: Read, Embedded app: Read
    2. Host: Read, Embedded app: Write

    The following scenarios are not considered safe. Use at your own risk.
    1. Host: Write, Embedded app: Read
    2. Host: Write, Embedded app: Write

USBD_MSC_IsConn()

Description

Gets the MSC connection state of the device.

Files

`usbd_msc.h/usbd_msc.c`

Prototype

```
CPU_BOOLEAN USBD_MSC_IsConn (CPU_INT08U   class_nbr)
```

Arguments

`class_nbr`

MSC instance number.

Returned Value

- `DEF_YES` , if MSC class is connected.
- `DEF_NO` , if the MSC class fails to connect.

Notes / Warnings

None.

USB Device Vendor API

USB Device Vendor API

- USBD_Vendor_ConfigureMsExtPropertiesQty()
- USBD_Vendor_ConfigureMemSeg()
- USBD_Vendor_Init()
- USBD_Vendor_Add()
- USBD_Vendor_ConfigAdd()
- USBD_Vendor_IsConn()
- USBD_Vendor_IntrRd()
- USBD_Vendor_IntrRdAsync()
- USBD_Vendor_IntrWr()
- USBD_Vendor_IntrWrAsync()
- USBD_Vendor_MS_ExtPropertyAdd()
- USBD_Vendor_Rd()
- USBD_Vendor_RdAsync()
- USBD_Vendor_Wr()
- USBD_Vendor_WrAsync()

USBD_Vendor_ConfigureMsExtPropertiesQty()

Description

Configures the quantity of Microsoft extended properties. Ignored when `USBD_CFG_MS_OS_DESC_EN` is set do `DEF_DISABLED`.

Files

`usbd_vendor.h/usbd_vendor.c`

Prototype

```
void USBD_Vendor_ConfigureMsExtPropertiesQty (CPU_INT08U ms_ext_properties_qty)
```

Arguments

`ms_ext_properties_qty`

Quantity of Microsoft extended properties.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the Vendor class is initialized via the `USBD_Vendor_Init()` function.

USBD_Vendor_ConfigureMemSeg()

Description

Configures the memory segment to use when allocating control data.

Files

`usbd_vendor.h/usbd_vendor.c`

Prototype

```
void  USBD_Vendor_ConfigureMemSeg (MEM_SEG  *p_mem_seg)
```

Arguments

`p_mem_seg`

Pointer to memory segment to use when allocating control data. DEF_NULL means general purpose heap segment.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the Vendor class is initialized via the `USBD_Vendor_Init()` function.

USBD_Vendor_Init()

Description

Initializes the internal structures and variables used by the Vendor class.

Files

`usbd_vendor.h/usbd_vendor.c`

Prototype

```
void USBD_Vendor_Init (USBD_VENDOR_QTY_CFG  *p_qty_cfg,                RTOS_ERR           *p_err)
```

Arguments

`p_qty_cfg`

Pointer to the vendor class configuration structure.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_SEG_OVF`

Returned Value

None.

Notes / Warnings

None.

USBD_Vendor_Add()

Description

Adds a new instance of the Vendor class.

Files

usbd_vendor.h/usbd_vendor.c

Prototype

```
CPU_INT08U USBD_Vendor_Add (CPU_BOOLEAN           intr_en,
                            CPU_INT16U            interval,
                            USBD_VENDOR_REQ_FNCT  req_callback,
                            RTOS_ERR             *p_err)
```

Arguments

intr_en

Interrupt endpoints IN and OUT flag:

- DEF_TRUE Pair of interrupt endpoints added to interface.
- DEF_FALSE Pair of interrupt endpoints not added to interface.

interval

Endpoint interval in milliseconds (must be a power of 2).

req_callback

Vendor-specific request callback.

p_err

Pointer to the variable that will receive one of these returned error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_CLASS_INSTANCE_ALLOC

Returned Value

- Class instance number, if no errors are returned.
- USBD_CLASS_NBR_NONE , otherwise.

Notes / Warnings

None.

USBD_Vendor_ConfigAdd()

Description

Adds the Vendor class instance into the specified configuration (see Note #1).

Files

usbd_vendor.h/usbd_vendor.c

Prototype

```
void USBD_Vendor_ConfigAdd (CPU_INT08U   class_nbr,
                            CPU_INT08U   dev_nbr,
                            CPU_INT08U   config_nbr,
                            RTOS_ERR    *p_err)
```

Arguments

class_nbr

Class instance number.

`dev_nbr`

Device number.

`config_nbr`

Configuration index to which to add the Vendor class instance.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`
- `RTOS_ERR_IF_ALT_ALLOC`
- `RTOS_ERR_ALLOC`
- `RTOS_ERR_CLASS_INSTANCE_ALLOC`
- `RTOS_ERR_IF_ALLOC`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_EP_ALLOC`
- `RTOS_ERR_EP_NONE_AVAIL`

Returned Value

None.

Notes / Warnings

1. Called several times, it creates multiple instances and configurations. For instance, the following architecture could be created:

```
HS

|-- Configuration 0
|-- Interface 0 (Vendor 0)
|-- Configuration 1
|-- Interface 0 (Vendor 0)
|-- Interface 1 (Vendor 1)
```

   In this example, there are two instances of Vendor class: 'Vendor 0' and '1', and two possible configurations: 'Configuration 0' and '1'. 'Configuration 1' is composed of two interfaces.
   Each class instance has an association with one of the interfaces. If 'Configuration 1' is activated by the host, it allows the host to access two different functionalities offered by the device.
2. Configuration Descriptor corresponding to a Vendor-specific device has the following format:

```
Configuration Descriptor
|-- Interface Descriptor (Vendor class)
|-- Endpoint Descriptor (Bulk OUT)
|-- Endpoint Descriptor (Bulk IN)
|-- Endpoint Descriptor (Interrupt OUT) - optional
|-- Endpoint Descriptor (Interrupt IN) - optional
```

USBD_Vendor_IsConn()

Description

Gets the vendor class connection state.

Files

`usbd_vendor.h/usbd_vendor.c`

Prototype

```
CPU_BOOLEAN USBD_Vendor_IsConn (CPU_INT08U   class_nbr)
```

Arguments

`class_nbr`

Class instance number.

Returned Value

- `DEF_YES` , if the Vendor class is connected.
- `DEF_NO` , if the Vendor class is not connected.

Notes / Warnings

None.

USBD_Vendor_IntrRd()

Description

Receives the data from the host through the Interrupt OUT endpoint. This function is blocking.

Files

`usbd_vendor.h/usbd_vendor.c`

Prototype

```
CPU_INT32U USBD_Vendor_IntrRd (CPU_INT08U    class_nbr,
                               void         *p_buf,
                               CPU_INT32U    buf_len,
                               CPU_INT16U    timeout,
                               RTOS_ERR     *p_err)
```

Arguments

`class_nbr`

Class instance number.

`p_buf`

Pointer to the receive buffer.

`buf_len`

Receive buffer length in octets.

`timeout`

Timeout in milliseconds.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`
- `RTOS_ERR_NULL_PTR`
- `RTOS_ERR_EP_QUEUING`
- `RTOS_ERR_RX`
- `RTOS_ERR_NOT_READY`

- RTOS_ERR_EP_INVALID
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_IS_OWNER
- RTOS_ERR_INVALID_CLASS_STATE
- RTOS_ERR_INVALID_EP_STATE
- RTOS_ERR_ABORT
- RTOS_ERR_TIMEOUT

Returned Value

- Number of octets received, if no errors are returned.
- 0, if any errors are returned.

Notes / Warnings

None.

USBD_Vendor_IntrRdAsync()

Description

Receives the data from the host through Interrupt OUT endpoint. This function is non-blocking and returns immediately after transfer preparation. Upon transfer completion, a callback provided by the application will be called to finalize the transfer.

Files

usbd_vendor.h/usbd_vendor.c

Prototype

```
void USBD_Vendor_IntrRdAsync (CPU_INT08U            class_nbr,
                              void                  *p_buf,
                              CPU_INT32U             buf_len,
                              USBD_VENDOR_ASYNC_FNCT async_fnct,
                              void                  *p_async_arg,
                              RTOS_ERR              *p_err)
```

Arguments

class_nbr

Class instance number.

p_buf

Pointer to the receive buffer.

buf_len

Receive the buffer length in octets.

async_fnct

Receive the callback.

p_async_arg

Additional argument provided by application for the receive callback.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`
- `RTOS_ERR_NULL_PTR`
- `RTOS_ERR_EP_QUEUING`
- `RTOS_ERR_RX`
- `RTOS_ERR_NOT_READY`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_CLASS_STATE`
- `RTOS_ERR_INVALID_EP_STATE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

None.

Notes / Warnings

None.

USBD_Vendor_IntrWr()

Description

Sends data to the host through the Interrupt IN endpoint. This function is blocking.

Files

`usbd_vendor.h/usbd_vendor.c`

Prototype

```
CPU_INT32U USBD_Vendor_IntrWr (CPU_INT08U     class_nbr,
                               void          *p_buf,
                               CPU_INT32U     buf_len,
                               CPU_INT16U     timeout,
                               CPU_BOOLEAN    end,
                               RTOS_ERR      *p_err)
```

Arguments

`class_nbr`

Class instance number.

`p_buf`

Pointer to the transmit buffer.

`buf_len`

Transmit buffer length in octets.

`timeout`

Timeout in milliseconds.

`end`

End-of-transfer flag (see Note #1).

`p_err`

- `RTOS_ERR_INVALID_DEV_STATE`
- `RTOS_ERR_NULL_PTR`
- `RTOS_ERR_EP_QUEUING`
- `RTOS_ERR_TX`
- `RTOS_ERR_NOT_READY`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_CLASS_STATE`
- `RTOS_ERR_INVALID_EP_STATE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

- Number of octets sent, if no errors are returned.
- 0, if any errors are returned.

Notes / Warnings

1. If the end-of-transfer is set and the transfer length is a multiple of the maximum packet size, a zero-length packet is transferred to signal the end of transfer to the host.

USBD_Vendor_IntrWrAsync()

Description

Send the data to the host through the Interrupt IN endpoint. This function is non-blocking and returns immediately after transfer preparation. Upon transfer completion, a callback provided by the application will be called to finalize the transfer.

Files

`usbd_vendor.h/usbd_vendor.c`

Prototype

```
void USBD_Vendor_IntrWrAsync (CPU_INT08U             class_nbr,
                        void                  *p_buf,
                        CPU_INT32U             buf_len,
                        USBD_VENDOR_ASYNC_FNCT  async_fnct,
                        void                  *p_async_arg,
                        CPU_BOOLEAN            end,
                        RTOS_ERR              *p_err)
```

Arguments

`class_nbr`

Class instance number.

`p_buf`

Pointer to the transmit buffer.

`buf_len`

Transmit buffer length in octets.

`async_fnct`

Transmit callback.

`p_async_arg`

Additional argument provided by the application for the transmit callback.

`end`

End-of-transfer flag (see Note #1).

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`
- `RTOS_ERR_NULL_PTR`
- `RTOS_ERR_EP_QUEUING`
- `RTOS_ERR_TX`
- `RTOS_ERR_NOT_READY`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_CLASS_STATE`
- `RTOS_ERR_INVALID_EP_STATE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

None.

Notes / Warnings

1. If the end-of-transfer is set and the transfer length is a multiple of the maximum packet size, a zero-length packet is transferred to signal the end of transfer to the host.

USBD_Vendor_MS_ExtPropertyAdd()

Description

Adds a Microsoft OS extended property to this vendor class instance.

Files

usbd_vendor.h/usbd_vendor.c

Prototype

```
void USBD_Vendor_MS_ExtPropertyAdd ( CPU_INT08U    class_nbr,
                                     CPU_INT08U    property_type,
                               const CPU_INT08U   *p_property_name,
                                     CPU_INT16U    property_name_len,
                               const CPU_INT08U   *p_property,
                                     CPU_INT32U    property_len,
                                     RTOS_ERR     *p_err)
```

Arguments

class_nbr

Class instance number.

property_type

Property type (see Note #2).

- USBD_MS_OS_PROPERTY_TYPE_REG_SZ
- USBD_MS_OS_PROPERTY_TYPE_REG_EXPAND_SZ
- USBD_MS_OS_PROPERTY_TYPE_REG_BINARY
- USBD_MS_OS_PROPERTY_TYPE_REG_DWORD_LITTLE_ENDIAN
- USBD_MS_OS_PROPERTY_TYPE_REG_DWORD_BIG_ENDIAN
- USBD_MS_OS_PROPERTY_TYPE_REG_LINK
- USBD_MS_OS_PROPERTY_TYPE_REG_MULTI_SZ

p_property_name

Pointer to the buffer that contains the property name.

---- Buffer assumed to be persistent ----

property_name_len

Length of the property name in octets.

p_property

Pointer to the buffer that contains the property.

---- Buffer assumed to be persistent ----

property_len

Length of the property in octets.

p_err

Pointer to the variable that will receive this return error code from this function :

- RTOS_ERR_NONE

Returned Value

None.

Notes / Warnings

1. For more information on Microsoft OS descriptors, see http://msdn.microsoft.com/en-us/library/windows/hardware/gg463179.aspx .
2. For more information on property types, refer to "Table 3. Property Data Types" of "Extended Properties OS Feature Descriptor Specification" document provided by Microsoft available at:

http://msdn.microsoft.com/en-us/library/windows/hardware/gg463179.aspx

USBD_Vendor_Rd()

Description

Receive the data from the host through the Bulk OUT endpoint. This function is blocking.

Files

`usbd_vendor.h/usbd_vendor.c`

Prototype

```
CPU_INT32U USBD_Vendor_Rd (CPU_INT08U    class_nbr,
                           void         *p_buf,
                           CPU_INT32U    buf_len,
                           CPU_INT16U    timeout,
                           RTOS_ERR     *p_err)
```

Arguments

`class_nbr`

Class instance number.

`p_buf`

Pointer to the receive buffer.

`buf_len`

Receive the buffer length in octets.

`timeout`

Timeout in milliseconds.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`
- `RTOS_ERR_NULL_PTR`
- `RTOS_ERR_EP_QUEUING`
- `RTOS_ERR_RX`
- `RTOS_ERR_NOT_READY`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_CLASS_STATE`
- `RTOS_ERR_INVALID_EP_STATE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

- Number of octets received, if no errors are returned.
- 0, if any errors are returned.

Notes / Warnings

None.

USBD_Vendor_RdAsync()

Description

Receive the data from the host through the Bulk OUT endpoint. This function is non-blocking are returns immediately after transfer preparation. Upon transfer completion, a callback provided by the application will be called to finalize the transfer.

Files

`usbd_vendor.h/usbd_vendor.c`

Prototype

```
void USBD_Vendor_RdAsync (CPU_INT08U            class_nbr,
                          void                 *p_buf,
                          CPU_INT32U            buf_len,
                          USBD_VENDOR_ASYNC_FNCT  async_fnct,
                          void                 *p_async_arg,
                          RTOS_ERR             *p_err)
```

Arguments

`class_nbr`

Class instance number.

`p_buf`

Pointer to the receive buffer.

`buf_len`

Receive buffer length in octets.

`async_fnct`

Receive the the callback.

`p_async_arg`

Additional argument provided by the application for the receive callback.

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`
- `RTOS_ERR_NULL_PTR`
- `RTOS_ERR_EP_QUEUING`
- `RTOS_ERR_RX`
- `RTOS_ERR_NOT_READY`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`

- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_IS_OWNER
- RTOS_ERR_INVALID_CLASS_STATE
- RTOS_ERR_INVALID_EP_STATE
- RTOS_ERR_ABORT
- RTOS_ERR_TIMEOUT

Returned Value

None.

Notes / Warnings

None.

USBD_Vendor_Wr()

Description

Sends the data to host through Bulk IN endpoint. This function is blocking.

Files

usbd_vendor.h/usbd_vendor.c

Prototype

```
CPU_INT32U USBD_Vendor_Wr (CPU_INT08U     class_nbr,
                           void          *p_buf,
                           CPU_INT32U     buf_len,
                           CPU_INT16U     timeout,
                           CPU_BOOLEAN    end,
                           RTOS_ERR      *p_err)
```

Arguments

class_nbr

Class instance number.

p_buf

Pointer to the transmit buffer.

buf_len

Transmit the buffer length in octets.

timeout

Timeout in milliseconds.

end

End-of-transfer flag (see Note #1).

p_err

Pointer to the variable that will receive one of these returned error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_DEV_STATE

- RTOS_ERR_NULL_PTR
- RTOS_ERR_EP_QUEUING
- RTOS_ERR_TX
- RTOS_ERR_NOT_READY
- RTOS_ERR_EP_INVALID
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_IS_OWNER
- RTOS_ERR_INVALID_CLASS_STATE
- RTOS_ERR_INVALID_EP_STATE
- RTOS_ERR_ABORT
- RTOS_ERR_TIMEOUT

Returned Value

- Number of octets sent, if no errors are returned.
- 0, if any errors are returned.

Notes / Warnings

1. If the end-of-transfer is set and the transfer length is a multiple of the maximum packet size, a zero-length packet is transferred to signal the end of transfer to the host.

USBD_Vendor_WrAsync()

Description

Sends the data to host through Bulk IN endpoint. This function is non-blocking and returns immediately after transfer preparation. Upon transfer completion, a callback provided by the application will be called to finalize the transfer.

Files

usbd_vendor.h/usbd_vendor.c

Prototype

```
void USBD_Vendor_WrAsync (CPU_INT08U             class_nbr,
                    void                  *p_buf,
                    CPU_INT32U             buf_len,
                    USBD_VENDOR_ASYNC_FNCT  async_fnct,
                    void                  *p_async_arg,
                    CPU_BOOLEAN            end,
                    RTOS_ERR              *p_err)
```

Arguments

class_nbr

Class instance number.

p_buf

Pointer to the transmit buffer.

buf_len

Transmit buffer length in octets.

`async_fnct`

Transmit the callback.

`p_async_arg`

Additional argument provided by the application for the transmit callback.

`end`

End-of-transfer flag (see Note #1).

`p_err`

Pointer to the variable that will receive one of these returned error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_DEV_STATE`
- `RTOS_ERR_NULL_PTR`
- `RTOS_ERR_EP_QUEUING`
- `RTOS_ERR_TX`
- `RTOS_ERR_NOT_READY`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_CLASS_STATE`
- `RTOS_ERR_INVALID_EP_STATE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

None.

Notes / Warnings

1. If the end-of-transfer is set and the transfer length is a multiple of the maximum packet size, a zero-length packet is transferred to signal the end of transfer to the host.

USBDev_API API

USBDev_API API

USBDev_API is a library implemented under Windows operating system. Functions return values and parameters use Windows data types such as DWORD, HANDLE, ULONG.

Refer to MSDN online documentation for more details about Windows data types (http://msdn.microsoft.com/en-us/library/aa383751(v=VS.85).aspx).

- USBDev_DevQtyGet
- USBDev_Open
- USBDev_Close
- USBDev_AltSettingQtyGet
- USBDev_AssociatedIF_QtyGet
- USBDev_AltSettingSet
- USBDev_AltSettingCurGet
- USBDev_IsHighSpeed
- USBDev_BulkIn_Open
- USBDev_BulkOut_Open
- USBDev_IntrIn_Open
- USBDev_IntrOut_Open
- USBDev_PipeAddrGet
- USBDev_PipeClose
- USBDev_PipeStall
- USBDev_PipeAbort
- USBDev_CtrlReq
- USBDev_PipeWr
- USBDev_PipeWrExt
- USBDev_PipeRd
- USBDev_PipeRdAsync

USBDev_DevQtyGet

Description

Get number of devices belonging to the specified GUID.

Files

`usbdev_api.c`

Prototype

```
DWORD  USBDev_DevQtyGet (const  GUID    guid_dev_if,
                         DWORD  *p_err);
```

Arguments

`guid_dev_if`

Device interface class GUID.

`p_err`

Pointer to variable that will receive the return error code from this function.

- ERROR_SUCCESS

Returned Value

- Number of devices for the provided GUID, if NO error(s).
- 0, otherwise.

Callers

Application.

Notes / Warnings

The function USBDev_DevQtyGet() uses the concept of device information set. A device information set consists of device information elements for all the devices that belong to some device setup class or device interface class. The GUID passed to USBDev_DevQtyGet() function is a device interface class. Internally by using some control options the function retrieves the device information set which represents a list of all devices present in the system and registered under the specified GUID. More details about the device information set can be found at http://msdn.microsoft.com/en-us/library/ff541247(VS.85).aspx .

USBDev_Open

Description

Open a device by retrieving a general device handle.

Files

usbdev_api.c

Prototype

```
HANDLE  USBDev_Open (const  GUID    guid_dev_if,
                     DWORD   dev_nbr,
                     DWORD  *p_err);
```

Arguments

guid_dev_if

Device interface class GUID.

dev_nbr

Device number.

p_err

Pointer to variable that will receive the return error code from this function:

- ERROR_SUCCESS
- ERROR_INVALID_PARAMETER
- ERROR_NOT_ENOUGH_MEMORY
- ERROR_BAD_DEVICE

Returned Value

- Handle to device, if NO error(s).
- INVALID_HANDLE_VALUE , otherwise.

Callers

Application.

Notes / Warnings

None.

USBDev_Close

Description

Close a device by freeing any allocated resources and by releasing any created handles.

Files

`usbdev_api.c`

Prototype

```
void  USBDev_Close (HANDLE   dev,
                    DWORD   *p_err);
```

Arguments

`dev`

General handle to device.

`p_err`

Pointer to variable that will receive the return error code from this function:

- `ERROR_SUCCESS`
- `ERROR_INVALID_HANDLE`

Returned Value

None.

Callers

Application.

Notes / Warnings

`USBDev_Close()` closes any remaining open pipes. The open pipes are usually closed from the application by calling the function USBDev_PipeClose.

USBDev_AltSettingQtyGet

Description

Get number of alternate settings for the specified interface.

Files

`usbdev_api.c`

Prototype

```
UCHAR  USBDev_AltSettingQtyGet (HANDLE   dev,
                                UCHAR    if_nbr,
                                DWORD   *p_err);
```

Arguments

`dev`

General handle to device.

`if_nbr`

Interface number.

`p_err`

Pointer to variable that will receive the return error code from this function:

- `ERROR_SUCCESS`
- `ERROR_INVALID_HANDLE`
- `ERROR_INVALID_PARAMETER`

Returned Value

- Number of alternate setting, if NO error(s).
- 0, otherwise.

Callers

Application.

Notes / Warnings

1. An interface may include alternate settings that allow the endpoints and/or their characteristics to be varied after the device has been configured. The default setting for an interface is always alternate setting zero. Alternate settings allow a portion of the device configuration to be varied while other interfaces remain in operation.
2. The number of alternate settings gotten can be used to open a pipe associated with a certain alternate interface.

USBDev_AssociatedIF_QtyGet

Description

Get number of associated interfaces with the default interface. That is all the interfaces besides the default interface managed by WinUSB.sys and registered under the same GUID.

Files

`usbdev_api.c`

Prototype

```
UCHAR  USBDev_AssociatedIF_QtyGet (HANDLE   dev,
                                   DWORD   *p_err);
```

Arguments

`dev`

General handle to device.

`p_err`

Pointer to variable that will receive the return error code from this function:

- `ERROR_SUCCESS`
- `ERROR_INVALID_HANDLE`

Returned Value

- Number of associated interfaces, if NO error(s).

- 0, otherwise.

Callers

Application.

Notes / Warnings

Let's assume that a device has three interfaces managed by WinUSB.sys driver and belonging to the same GUID: Interface #0, #1 and #2. Interface #0 is the default interface. Interfaces #1 and #2 are the associated interfaces. In that example calling `USBDev_AssociatedIF_QtyGet()` will return 2 associated interfaces.

USBDev_AltSettingSet

Description

Set the alternate setting of an interface.

Files

`usbdev_api.c`

Prototype

```
void  USBDev_AltSettingSet (HANDLE   dev,
                            UCHAR    if_nbr,
                            UCHAR    alt_set,
                            DWORD   *p_err);
```

Arguments

`dev`

General handle to device.

`if_nbr`

Interface number.

`alt_set`

Alternate setting number.

`p_err`

Pointer to variable that will receive the return error code from this function:

- `ERROR_SUCCESS`
- `ERROR_INVALID_HANDLE`
- `ERROR_INVALID_PARAMETER`

Returned Value

None.

Callers

Application.

Notes / Warnings

This function sends a SET_INTERFACE request to the device to set the alternate setting number and updates the one used internally by WinUSB.

USBDev_AltSettingCurGet

Description

Get the current alternate setting for the specified interface.

Files

`usbdev_api.c`

Prototype

```
UCHAR  USBDev_AltSettingCurGet (HANDLE   dev,
                                UCHAR    if_nbr,
                                DWORD   *p_err);
```

Arguments

`dev`

General handle to device.

`if_nbr`

Interface number.

`p_err`

Pointer to variable that will receive the return error code from this function:

- `ERROR_SUCCESS`
- `ERROR_INVALID_HANDLE`
- `ERROR_INVALID_PARAMETER`

Returned Value

- Current alternate setting number, if NO error(s).
- 0, otherwise.

Callers

Application.

Notes / Warnings

This function gets the current alternate setting number used internally by WinUSB and gets the one from the device by sending a `GET_INTERFACE` request. Both alternate setting numbers are compared. If they match, the current alternate setting is returned.

USBDev_IsHighSpeed

Description

Specify if the device attached to PC is high speed or not.

Files

`usbdev_api.c`

Prototype

```
BOOL  USBDev_IsHighSpeed (HANDLE   dev,
                          DWORD   *p_err);
```

Arguments

`dev`

General handle to device.

`p_err`

Pointer to variable that will receive the return error code from this function:

- `ERROR_SUCCESS`
- `ERROR_INVALID_HANDLE`
- `ERROR_INVALID_PARAMETER`

Returned Value

- `TRUE` , if device is high-speed.
- `FALSE` , otherwise.

Callers

Application.

Notes / Warnings

None.

USBDev_BulkIn_Open

Description

Open a Bulk IN pipe.

Files

`usbdev_api.c`

Prototype

```
HANDLE  USBDev_BulkIn_Open (HANDLE   dev,
                            UCHAR    if_nbr,
                            UCHAR    alt_set,
                            DWORD   *p_err);
```

Arguments

`dev`

General handle to device.

`if_nbr`

Interface number.

`alt_set`

Alternate setting number for specified interface.

`p_err`

Pointer to variable that will receive the return error code from this function:

- `ERROR_SUCCESS`
- `ERROR_INVALID_HANDLE`
- `ERROR_NO_MORE_ITEMS`

Returned Value

- Handle to Bulk IN pipe, if NO error(s).
- `INVALID_HANDLE_VALUE` , otherwise.

Callers

Application.

Notes / Warnings

None.

USBDev_BulkOut_Open

Description

Open a Bulk OUT pipe.

Files

`usbdev_api.c`

Prototype

```
HANDLE  USBDev_BulkOut_Open (HANDLE   dev,
                             UCHAR    if_nbr,
                             UCHAR    alt_set,
                             DWORD   *p_err);
```

Arguments

`dev`

General handle to device.

`if_nbr`

Interface number.

`alt_set`

Alternate setting number for specified interface.

`p_err`

Pointer to variable that will receive the return error code from this function:

- `ERROR_SUCCESS`
- `ERROR_INVALID_HANDLE`
- `ERROR_NO_MORE_ITEMS`

Returned Value

- Handle to Bulk OUT pipe, if NO error(s).
- `INVALID_HANDLE_VALUE` , otherwise.

Callers

Application.

Notes / Warnings

None.

USBDev_IntrIn_Open

Description

Open a Interrupt IN pipe.

Files

`usbdev_api.c`

Prototype

```
HANDLE  USBDev_IntrIn_Open (HANDLE   dev,
                   UCHAR    if_nbr,
                   UCHAR    alt_set,
                   DWORD   *p_err);
```

Arguments

`dev`

General handle to device.

`if_nbr`

Interface number.

`alt_set`

Alternate setting number for specified interface.

`p_err`

Pointer to variable that will receive the return error code from this function:

- `ERROR_SUCCESS`
- `ERROR_INVALID_HANDLE`
- `ERROR_NO_MORE_ITEMS`

Returned Value

- Handle to Interrupt IN pipe, if NO error(s).
- `INVALID_HANDLE_VALUE` , otherwise.

Callers

Application.

Notes / Warnings

None.

USBDev_IntrOut_Open

Description

Open a Interrupt OUT pipe.

Files

`usbdev_api.c`

Prototype

```
HANDLE  USBDev_IntrOut_Open (HANDLE   dev,
                    UCHAR    if_nbr,
                    UCHAR    alt_set,
                    DWORD   *p_err);
```

Arguments

`dev`

General handle to device.

`if_nbr`

Interface number.

`alt_set`

Alternate setting number for specified interface.

`p_err`

Pointer to variable that will receive the return error code from this function:

- `ERROR_SUCCESS`
- `ERROR_INVALID_HANDLE`
- `ERROR_NO_MORE_ITEMS`

Returned Value

- Handle to Interrupt OUT pipe, if NO error(s).
- `INVALID_HANDLE_VALUE` , otherwise.

Callers

Application.

Notes / Warnings

None.

USBDev_PipeAddrGet

Description

Get pipe address.

Files

`usbdev_api.c`

Prototype

```
UCHAR  USBDev_PipeAddrGet (HANDLE   pipe,
                           DWORD   *p_err);
```

Arguments

`pipe`

Pipe handle.

`p_err`

Pointer to variable that will receive the return error code from this function:

- `ERROR_SUCCESS`
- `ERROR_INVALID_HANDLE`

Returned Value

- Pipe address, if NO error(s).
- 0, otherwise.

Callers

Application.

Notes / Warnings

The pipe address is composed of the pipe direction and the pipe logical address. The pipe direction is located at bit 7. A value of '1' indicates an IN pipe and '0' an OUT pipe. The pipe logical address sits in bit 3 to bit 0.

USBDev_PipeClose

Description

Close a pipe.

Files

`usbdev_api.c`

Prototype

```
void  USBDev_PipeClose (HANDLE   pipe,
                        DWORD   *p_err);
```

Arguments

`pipe`

Pipe handle.

`p_err`

Pointer to variable that will receive the return error code from this function:

- `ERROR_SUCCESS`
- `ERROR_INVALID_HANDLE`

Returned Value

None

Callers

Application.

Notes / Warnings

None.

USBDev_PipeStall

Description

Stall a pipe or clear the stall condition of a pipe.

Files

`usbdev_api.c`

Prototype

```
void  USBDev_PipeStall (HANDLE   pipe,
                        BOOL     stall,
                        DWORD   *p_err);
```

Arguments

`pipe`

Pipe handle.

`stall`

Indicate which action to do:

- `TRUE`
  Stall pipe.
- `FALSE`
  Clear stall condition of the pipe.

`p_err`

Pointer to variable that will receive the return error code from this function:

- `ERROR_SUCCESS`
- `ERROR_INVALID_HANDLE`
- `ERROR_NOT_ENOUGH_MEMORY`

Returned Value

None.

Callers

Application.

Notes / Warnings

The `SET_FEATURE` standard request is sent to the device to stall the pipe. The `CLEAR_FEATURE` standard request is sent to the device to clear the stall condition of the pipe.

USBDev_PipeAbort

Description

Aborts all of the pending transfers for a pipe.

Files

`usbdev_api.c`

Prototype

```
void  USBDev_PipeAbort (HANDLE   pipe,
                        DWORD   *p_err);
```

Arguments

`pipe`

Pipe handle.

`p_err`

Pointer to variable that will receive the return error code from this function:

- `ERROR_SUCCESS`
- `ERROR_INVALID_HANDLE`

Returned Value

None.

Callers

Application.

Notes / Warnings

None.

USBDev_CtrlReq

Description

Send control data over the default control endpoint.

Files

`usbdev_api.c`

Prototype

```
ULONG  USBDev_CtrlReq (HANDLE   dev,
                       UCHAR    bm_req_type,
                       UCHAR    b_request,
                       USHORT   w_value,
                       USHORT   w_index,
                       UCHAR   *p_buf,
                       USHORT   buf_len,
                       DWORD   *p_err);
```

Arguments

`dev`

General handle to device

`bm_req_type`

Variable representing bmRequestType of setup packet. bmRequestType is a bitmap with the following characteristics:

- `D7`
  Data transfer direction:
  - '0': USB_DIR_HOST_TO_DEVICE
  - '1': USB_DIR_DEVICE_TO_HOST
- D6...5
  Request type:
  - '00': USB_REQUEST_TYPE_STD (standard)
  - '01': USB_REQUEST_TYPE_CLASS
  - '10': USB_REQUEST_TYPE_VENDOR
- `D4...0`
  Recipient:
  - '0000': USB_RECIPIENT_DEV (device)
  - '0001': USB_RECIPIENT_IF (interface)
  - '0010': USB_RECIPIENT_ENDPOINT

`b_request`

Variable representing bRequest of setup packet. Possible values are:

bRequest	Description
`GET_STATUS`	Returns status for the specified recipient.

bRequest	Description
CLEAR_FEATURE	Clear or disable a specific feature.
SET_FEATURE	Set or enable a specific feature.
SET_ADDRESS	Set the device address for all future device accesses.
GET_DESCRIPTOR	Return the specified descriptor if the descriptor exists.
SET_DESCRIPTOR	Update existing descriptors or new descriptors may be added.
GET_CONFIGURATION	Return the current device configuration value.
SET_CONFIGURATION	Set the device configuration.
GET_INTERFACE	Return the selected alternate setting for the specified interface.
SET_INTERFACE	Select an alternate setting for the specified interface.
SYNCH_FRAME	Set and then report an endpoint's synchronization frame.

`w_value`

Variable representing wValue of setup packet.

`w_index`

Variable representing wIndex of setup packet.

`p_buf`

Pointer to transmit or receive buffer for data phase of control transfer.

`buf_len`

Length of transmit or receive buffer.

`p_err`

Pointer to variable that will receive the return error code from this function:

- `ERROR_SUCCESS`
- `ERROR_INVALID_HANDLE`
- `ERROR_NOT_ENOUGH_MEMORY`
- `ERROR_GEN_FAILURE`

Returned Value

None

Callers

Application.

Notes / Warnings

1. The value of `w_value` and `w_index` arguments vary according to the specific request defined by `b_request` argument.
    1. When the request's recipient is an interface or an endpoint, w_index must be properly configured. For an interface recipient, w_index will contain the interface number. For an endpoint recipient, w_index will contain the endpoint address. This one can be retrieved using the function USBDev_PipeAddrGet().
2. The following code shows an example using USBDev_CtrlReq() to send the `SET_INTERFACE` request:

```
DWORD  err;
                    /* Select alternate setting #1 for default interface. */
USBDev_CtrlReq ( dev_handle,
           (USB_DIR_HOST_TO_DEVICE | USB_REQUEST_TYPE_STD | USB_RECIPIENT_IF),
```

```
1,/* Alternate setting #1.                    */0,/* Interface #0 inside active configuration.          */0,/* No data phase.
*/0,&err);if(err != ERROR_SUCCESS){printf("[ERROR #%d] SET_INTERFACE(1) request failed.\n", err);}
```

More details about USB device requests can be found in “Universal Serial Bus Specification, Revision 2.0, April 27, 2000”, section 9.3.

3. A control transfer is composed of 3 stages: Setup, Data (IN, OUT or no data stage) and Status. The table below presents the parameters of USBDev_CtrlReq() involved in each specific stage.

Control Transfer Stage	Parameter Involved	Note
Setup	`bm_req_type` , `b_request` , `w_value` , `w_index`	This parameters are used to build the Setup packet sent by the host to the device.
Data	`p_buf` , `buf_len`	If no data stage is required, you just need to set `p_buf` to null pointer and `buf_len` to the value 0.
Status	None	NO parameter is involved for the Status stage. It is managed automatically by the Windows Host stack.

USBDev_PipeWr

Description

Write data to device over the specified pipe.

Files

`usbdev_api.c`

Prototype

```
DWORD  USBDev_PipeWr (HANDLE   pipe,
                      UCHAR   *p_buf,
                      DWORD    buf_len,
                      DWORD    timeout,
                      DWORD   *p_err);
```

Arguments

`pipe`

Pipe handle.

`p_buf`

Pointer to transmit buffer.

`buf_len`

Transmit buffer length.

`timeout`

Timeout in milliseconds. A value of 0 indicates a wait forever.

`p_err`

Pointer to variable that will receive the return error code from this function:

- `ERROR_SUCCESS`
- `ERROR_INVALID_HANDLE`
- `ERROR_INVALID_USER_BUFFER`
- `ERROR_BAD_PIPE`
- `ERROR_INVALID_PARAMETER`

- ERROR_NOT_ENOUGH_MEMORY
- ERROR_SEM_TIMEOUT

Returned Value

- Number of bytes written, if NO error(s).
- 0, otherwise.

Callers

Application.

Notes / Warnings

None.

USBDev_PipeWrExt

Description

Write data to device over the specified pipe.

Files

usbdev_api.cs

Prototype

```
DWORD  USBDev_PipeWrExt (HANDLE   pipe,
                         UCHAR   *p_buf,
                         DWORD    buf_len,
                         BOOL     end,
                         DWORD    timeout,
                         DWORD   *p_err);
```

Arguments

pipe

Pipe handle.

p_buf

Pointer to transmit buffer.

buf_len

Transmit buffer length.

end

End-of-transfer flag (see Note #1).

timeout

Timeout in milliseconds. A value of 0 indicates a wait forever.

p_err

Pointer to variable that will receive the return error code from this function:

- ERROR_SUCCESS
- ERROR_INVALID_HANDLE
- ERROR_INVALID_USER_BUFFER
- ERROR_BAD_PIPE

- ERROR_INVALID_PARAMETER
- ERROR_NOT_ENOUGH_MEMORY
- ERROR_SEM_TIMEOUT

Returned Value

- Number of bytes written, if NO error(s).
- 0, otherwise.

Callers

Application.

Notes / Warnings

If end-of-transfer is set and transfer length is multiple of maximum packet size, a zero-length packet is transferred to indicate a short transfer to the device.

USBDev_PipeRd

Description

Read data from device over the specified pipe.

Files

usbdev_api.c

Prototype

```
DWORD  USBDev_PipeRd (HANDLE   pipe,
                      UCHAR   *p_buf,
                      DWORD    buf_len,
                      DWORD    timeout,
                      DWORD   *p_err);
```

Arguments

pipe

Pipe handle.

p_buf

Pointer to receive buffer.

buf_len

Receive buffer length.

timeout

Timeout in milliseconds. A value of 0 indicates a wait forever.

p_err

Pointer to variable that will receive the return error code from this function:

- ERROR_SUCCESS
- ERROR_INVALID_HANDLE
- ERROR_INVALID_USER_BUFFER
- ERROR_BAD_PIPE
- ERROR_INVALID_PARAMETER
- ERROR_NOT_ENOUGH_MEMORY

- ERROR_SEM_TIMEOUT7

Returned Value

- Number of bytes received, if NO error(s).
- 0, otherwise.

Callers

Application.

Notes / Warnings

None.

USBDev_PipeRdAsync

Description

Read data from device over the specified pipe. This function returns immediately if data is not present. The data will be retrieved later.

Files

usbdev_api.c

Prototype

```
void  USBDev_PipeRdAsync (HANDLE                  pipe,
                          UCHAR                  *p_buf,
                          DWORD                   buf_len,
                          USBDEV_PIPE_RD_CALLBACK callback,
                          void                   *p_callback_arg,
                          DWORD                  *p_err);
```

Arguments

pipe

Pipe handle.

p_buf

Pointer to receive buffer.

buf_len

Receive buffer length.

callback

Pointer to application callback called by Asynchronous thread upon completion.

p_callback_arg

Pointer to argument which can carry private information passed by application. This argument is used when the callback is called.

p_err

Pointer to variable that will receive the return error code from this function:

- ERROR_SUCCESS
- ERROR_INVALID_HANDLE

- `ERROR_INVALID_USER_BUFFER`
- `ERROR_BAD_PIPE`
- `ERROR_NOT_ENOUGH_MEMORY`
- `ERROR_SEM_TIMEOUT`

Returned Value

None.

Callers

Application.

Notes / Warnings

1. When a IN pipe is open with one of the open functions `USBDev_xxxxIn_Open()` , a thread is automatically created. This thread is in charge of informing the application about a completed asynchronous IN transfer. Upon completion of an asynchronous transfer, the thread is waken up and calls the application callback provided to `USBDev_API` library using the `callback` argument.
2. `USBDev_API` library allows to queue several asynchronous IN transfers for the same pipe.

USB Host API

USB Host API

NOTE: This documentation refers to a deprecated software component that will no longer be supported and removed in an future release. Consider using the latest Silicon Labs USB stack instead. For more information, see USB Device.

- USB Host Core API
- USB Host PBHCI API
- USB Host Shell Commands API
- USB Host HUB API
- USB Host AOAP API
- USB Host CDC API
- USB Host ACM API
- USB Host HID API
- USB Host MSC API
- USB Host USB2SER API

USB Host Core API

USB Host Core API

- USBH_ConfigureBufAlignOctets()
- USBH_ConfigureMaxDescLen()
- USBH_ConfigureEventFncts()
- USBH_ConfigureOptimizeSpdCfg()
- USBH_ConfigureInitAllocCfg()
- USBH_ConfigureHubTaskStk()
- USBH_ConfigureAsyncTaskStk()
- USBH_ConfigureMemSeg()
- USBH_Init()
- USBH_PreferredStrLangID_Set()
- USBH_StdReqTimeoutSet()
- USBH_AsyncTaskPrioSet()
- USBH_UnInit()
- USBH_HC_Add()
- USBH_HC_HandleGetFromName()
- USBH_HC_Start()
- USBH_HC_Stop()
- USBH_HC_Suspend()
- USBH_HC_Resume()
- USBH_HC_FrameNbrGet()
- USBH_DevSpdGet()
- USBH_DevAddrGet()
- USBH_DevHostNbrGet()
- USBH_DevHC_NbrGet()
- USBH_DevSpecNbrGet()
- USBH_DevClassGet()
- USBH_DevSubclassGet()
- USBH_DevProtocolGet()
- USBH_DevVendorID_Get()
- USBH_DevProductID_Get()
- USBH_DevRelNbrGet()
- USBH_DevManufacturerStrGet()
- USBH_DevProductStrGet()
- USBH_DevSerNbrStrGet()
- USBH_DevConfigQtyGet()
- USBH_DevConfigSet()
- USBH_DevConfigGet()
- USBH_DevPortNbrGet()
- USBH_DevHubHandleGet()
- USBH_DevHS_HubNearestHandleGet()
- USBH_DevDescRd()
- USBH_ConfigAttribGet()
- USBH_ConfigFnctQtyGet()
- USBH_ConfigMaxPwrGet()
- USBH_ConfigStrGet()
- USBH_FnctIF_QtyGet()
- USBH_FnctClassGet()

- USBH_FnctSubclassGet()
- USBH_FnctProtocolGet()
- USBH_FnctStrGet()
- USBH_IF_NbrGet()
- USBH_IF_AltNbrGet()
- USBH_IF_AltIxCurGet()
- USBH_IF_EP_QtyGet()
- USBH_IF_ClassGet()
- USBH_IF_SubclassGet()
- USBH_IF_ProtocolGet()
- USBH_IF_StrGet()
- USBH_IF_DescExtraGet()
- USBH_IF_EP_AddrNextGet()
- USBH_IF_AltSet()
- USBH_EP_AttribGet()
- USBH_EP_MaxPktSizeGet()
- USBH_EP_IntervalGet()
- USBH_EP_CtrlXfer()

USBH_ConfigureBufAlignOctets()

Description

Configures the alignment of the internal buffers.

Files

`usbh_core.h/usbh_core.c`

Prototype

```
void  USBH_ConfigureBufAlignOctets (CPU_SIZE_T  buf_align_octets)
```

Arguments

`buf_align_octets`

Buffer alignment, in octets.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the USB Host module is initialized via the `USBH_Init()` function.

USBH_ConfigureMaxDescLen()

Description

Configures the length of the buffer used to retrieve the USB descriptors from the devices.

Files

`usbh_core.h/usbh_core.c`

Prototype

```
void  USBH_ConfigureMaxDescLen (CPU_INT16U  max_desc_len)
```

Arguments

`max_desc_len`

Length of the descriptor buffer.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the USB Host module is initialized via the `USBH_Init()` function.

USBH_ConfigureEventFncts()

Description

Sets the structure of callback that will be used by the USB host module to notify the application of certain events.

Files

`usbh_core.h/usbh_core.c`

Prototype

```
void USBH_ConfigureEventFncts(const USBH_EVENT_FNCTS *p_event_fncts)
```

Arguments

`p_event_fncts`

Pointer to a structure containing the event functions to call. Content MUST be persistent.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the USB Host module is initialized via the `USBH_Init()` function.

USBH_ConfigureOptimizeSpdCfg()

Description

Sets the configurations required when optimize speed mode is enabled.

Files

`usbh_core.h/usbh_core.c`

Prototype

```
void USBH_ConfigureOptimizeSpdCfg(const USBH_CFG_OPTIMIZE_SPD *p_optimize_spd_cfg)
```

Arguments

`p_optimize_spd_cfg`

Pointer to the structure containing the configurations for the optimize speed mode.

Returned Value

None.

Notes / Warnings

1. This function MUST be called before the USB Host module is initialized via the `USBH_Init()` function.
2. This function MUST be called when the `USBH_CFG_OPTIMIZE_SPD_EN` configuration is set to `DEF_ENABLED`.

USBH_ConfigureInitAllocCfg()

Description

Sets the configurations required when allocation at initialization mode is enabled.

Files

`usbh_core.h/usbh_core.c`

Prototype

```
void  USBH_ConfigureInitAllocCfg(const  USBH_CFG_INIT_ALLOC  *p_init_alloc_cfg)
```

Arguments

`p_init_alloc_cfg`

Pointer to the structure containing the configurations for the allocation at initialization mode.

Returned Value

None.

Notes / Warnings

1. This function MUST be called before the USB Host module is initialized via the `USBH_Init()` function.
2. This function MUST be called when the `USBH_CFG_INIT_ALLOC_EN` configuration is set to `DEF_ENABLED`.

USBH_ConfigureHubTaskStk()

Description

Configures the USB host hub task's stack.

Files

`usbh_core.h/usbh_core.c`

Prototype

```
void  USBH_ConfigureHubTaskStk (CPU_INT32U   stk_size_elements,
                                void        *p_stk)
```

Arguments

`stk_size_elements`

Size, in stack elements, of the task's stack.

`p_stk`

Pointer to base of the task's stack. If `DEF_NULL`, stack will be allocated from KAL's memory segment.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the USB Host module is initialized via the `USBH_Init()` function.
3. In order to change the priority of the USB host hub task, use the function `USBH_HUB_TaskPrioSet()` available in file `usbh_core_hub.h` .

USBH_ConfigureAsyncTaskStk()

Description

Configures the USB host asynchronous task's stack.

Files

`usbh_core.h/usbh_core.c`

Prototype

```
void  USBH_ConfigureAsyncTaskStk (CPU_INT32U   stk_size_elements,
                                  void        *p_stk)
```

Arguments

`stk_size_elements`

Size, in stack elements, of the task's stack.

`p_stk`

Pointer to base of the task's stack. If `DEF_NULL` , stack will be allocated from Common's memory segment.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the USB Host module is initialized via the `USBH_Init()` function.
3. In order to change the priority of the USB host hub task, use the function `USBH_AsyncTaskPrioSet()` .

USBH_ConfigureMemSeg()

Description

Configures the memory segment to use when allocating control data and buffers.

Files

`usbh_core.h/usbh_core.c`

Prototype

```
void  USBH_ConfigureMemSeg (MEM_SEG  *p_mem_seg,
                            MEM_SEG  *p_mem_seg_buf)
```

Arguments

`p_mem_seg`

Pointer to memory segment to use when allocating control data. Can be the same segment used for `p_mem_seg_buf`. `DEF_NULL` means general purpose heap segment.

`p_mem_seg_buf`

Pointer to memory segment to use when allocating data buffers. Can be the same segment used for `p_mem_seg`. `DEF_NULL` means general purpose heap segment.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the USB Host module is initialized via the `USBH_Init()` function.

USBH_Init()

Description

Initializes USB Host stack.

Files

`usbh_core.h/usbh_core.c`

Prototype

```
void USBH_Init (CPU_INT08U   host_qty,
                RTOS_ERR    *p_err)
```

Arguments

`host_qty`

Quantity of USB host.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_NOT_AVAIL`

Returned Value

None.

Notes / Warnings

1. USBH_Init() must be called:
   (a) Only once from a product's application.
   (b) After product's OS has been initialized.

USBH_PreferredStrLangID_Set()

Description

Sets the preferred language ID to use when retrieving strings from the device.

Files

usbh_core.h/usbh_core.c

Prototype

```
void USBH_PreferredStrLangID_Set(CPU_INT16U  preferred_str_lang_id,
                                 RTOS_ERR   *p_err)
```

Arguments

preferred_str_lang_id

ID of the preferred language. See file usbh_core_langid.h for a list of language IDs.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE

Returned Value

None.

Notes / Warnings

1. This function cannot be called before the USB Host module has been initialized via the USBH_Init() function.

USBH_StdReqTimeoutSet()

Description

Assigns a new timeout delay for the USB standard requests.

Files

usbh_core.h/usbh_core.c

Prototype

```
void USBH_StdReqTimeoutSet(CPU_INT32U  std_req_timeout_ms,
                           RTOS_ERR   *p_err)
```

Arguments

std_req_timeout_ms

New timeout, in milliseconds.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE

Returned Value

None.

Notes / Warnings

1. This function cannot be called before the USB Host module has been initialized via the USBH_Init() function.

USBH_AsyncTaskPrioSet()

Description

Assigns a new priority to the USB host async task.

Files

`usbh_core.h/usbh_core.c`

Prototype

```
void  USBH_AsyncTaskPrioSet(CPU_INT08U   prio,
                    RTOS_ERR    *p_err)
```

Arguments

`prio`

New priority of the the async task.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_ARG`

Returned Value

None.

Notes / Warnings

1. This function cannot be called before the USB Host module has been initialized via the `USBH_Init()` function.

USBH_UnInit()

Description

Un-initializes the USB Host stack.

Files

`usbh_core.h/usbh_core.c`

Prototype

```
void USBH_UnInit (RTOS_ERR    *p_err)
```

Arguments

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_POOL_UNLIMITED`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_INVALID_STATE`

Returned Value

None.

Notes / Warnings

1. USBH_UnInit() must be called after all the HCs have been stopped.
2. Once the USB host has been uninitialized, all the used memory segments can be cleared and re-used for other purposes.

USBH_HC_Add()

Description

Adds the host controller.

Files

`usbh_core.h/usbh_core.c`

Prototype

```
USBH_HC_HANDLE USBH_HC_Add (const  CPU_CHAR        *name,
                            const  USBH_HC_CFG_EXT  *p_hc_cfg_ext,
                                   RTOS_ERR         *p_err)
```

Arguments

`name`

USB controller name.

`p_hc_cfg_ext`

Pointer to the extended USB host controller configuration. Can be null when `USBH_CFG_OPTIMIZE_SPD_EN` and `USBH_CFG_INIT_ALLOC_EN` are set to `DEF_DISABLED`.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_ALLOC`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_SEG_OVF`

Returned Value

Host Controller handle, if host controller successfully added. `USBH_HC_HANDLE_INVALID`, if host controller was not added.

Notes / Warnings

None.

USBH_HC_HandleGetFromName()

Description

Gets Host Controller handle from USB controller name.

Files

`usbh_core.h/usbh_core.c`

Prototype

```
USBH_HC_HANDLE  USBH_HC_HandleGetFromName (const  CPU_CHAR  *name)
```

Arguments

`name`

USB controller name.

Returned Value

Host Controller handle, if host controller successfully retrieved. `USBH_HC_HANDLE_INVALID`, if host controller was not retrieved.

Notes / Warnings

None.

USBH_HC_Start()

Description

Starts the given host controller.

Files

`usbh_core.h/usbh_core.c`

Prototype

```
void USBH_HC_Start (USBH_HC_HANDLE   hc_handle,
                    RTOS_ERR        *p_err)
```

Arguments

`hc_handle`

Host controller handle.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_FAIL`
- `RTOS_ERR_IO`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_URB_ALLOC`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_TX`
- `RTOS_ERR_NOT_AVAIL`

- RTOS_ERR_WOULD_OVF

Returned Value

None.

Notes / Warnings

None.

USBH_HC_Stop()

Description

Stops the given host controller.

Files

usbh_core.h/usbh_core.c

Prototype

```
void USBH_HC_Stop (USBH_HC_HANDLE     hc_handle,
                   USBH_HC_OPER_CMPL  callback_fnct,
                   void              *p_arg,
                   RTOS_ERR          *p_err)
```

Arguments

hc_handle

Handle to host controller.

callback_fnct

Function that will be called after operation is completed. Can be null.

p_arg

Pointer to the argument that will be passed to the callback function.

p_err

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_ALLOC
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_NOT_AVAIL

Returned Value

None.

Notes / Warnings

None.

USBH_HC_Suspend()

Description

Suspends the host controller.

Files

usbh_core.h/usbh_core.c

Prototype

```
void USBH_HC_Suspend (USBH_HC_HANDLE      hc_handle,
                      USBH_HC_OPER_CMPL   callback_fnct,
                      void               *p_arg,
                      RTOS_ERR           *p_err)
```

Arguments

hc_handle

Handle to host controller.

callback_fnct

Pointer to the function that will be called after the operation is completed.

p_arg

Pointer to the argument that will be passed to the callback function.

p_err

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_ALLOC
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_NOT_AVAIL

Returned Value

None.

Notes / Warnings

None.

USBH_HC_Resume()

Description

Resumes the host controller.

Files

usbh_core.h/usbh_core.c

Prototype

```
void USBH_HC_Resume (USBH_HC_HANDLE      hc_handle,
                     USBH_HC_OPER_CMPL   callback_fnct,
                     void               *p_arg,
                     RTOS_ERR           *p_err)
```

Arguments

hc_handle

Handle to host controller.

callback_fnct

Function that will be called after the operation is completed.

`p_arg`

Pointer to the argument that will be passed to the callback function.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ALLOC`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_NOT_AVAIL`

Returned Value

None.

Notes / Warnings

None.

USBH_HC_FrameNbrGet()

Description

Retrieves the current frame number for the given host controller.

Files

`usbh_core.h/usbh_core.c`

Prototype

```
CPU_INT16U USBH_HC_FrameNbrGet (USBH_HC_HANDLE   hc_handle,
                                RTOS_ERR        *p_err)
```

Arguments

`hc_handle`

Handle to host controller.

`p_err`

Pointer to the variable that will receive this return error code from this function :

- `RTOS_ERR_NONE`

Returned Value

- Current frame number processed by Host Controller, if successful.
- 0, if unsuccessful.

Notes / Warnings

None.

USBH_DevSpdGet()

Description

Gets the device speed.

Files

`usbh_core_dev.h/usbh_core_dev.c`

Prototype

```
USBH_DEV_SPD USBH_DevSpdGet (USBH_DEV_HANDLE   dev_handle,
                             RTOS_ERR         *p_err)
```

Arguments

`dev_handle`

Handle on the device.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_INVALID_STATE`

Returned Value

Device speed, if successful. `USBH_DEV_SPD_NONE` , if unsuccessful.

Notes / Warnings

None.

USBH_DevAddrGet()

Description

Gets the device address.

Files

`usbh_core_dev.h/usbh_core_dev.c`

Prototype

```
CPU_INT08U USBH_DevAddrGet (USBH_DEV_HANDLE   dev_handle,
                            RTOS_ERR         *p_err)
```

Arguments

`dev_handle`

Handle on the device.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_INVALID_STATE`

Returned Value

- Device address, if successful.
- `USBH_DEV_ADDR_INVALID` , if unsuccessful.

Notes / Warnings

None.

USBH_DevHostNbrGet()

Description

Gets the host number to which the device is connected.

Files

`usbh_core_dev.h/usbh_core_dev.c`

Prototype

```
CPU_INT08U USBH_DevHostNbrGet (USBH_DEV_HANDLE   dev_handle,
                               RTOS_ERR         *p_err)
```

Arguments

`dev_handle`

Handle on the device.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

Returned Value

- Host number, if successful.
- 0, if unsuccessful.

Notes / Warnings

None.

USBH_DevHC_NbrGet()

Description

Gets the host controller number to which the device is connected.

Files

`usbh_core_dev.h/usbh_core_dev.c`

Prototype

```
CPU_INT08U USBH_DevHC_NbrGet (USBH_DEV_HANDLE   dev_handle,
                              RTOS_ERR         *p_err)
```

Arguments

`dev_handle`

Handle on the device.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_HANDLE

Returned Value

- Host controller number, if successful.
- USBH_HC_NBR_NONE , if unsuccessful.

Notes / Warnings

None.

USBH_DevSpecNbrGet()

Description

Gets the device specification number.

Files

usbh_core_dev.h/usbh_core_dev.c

Prototype

```
typedef CPU_INT32U USBH_DevSpdGet (USBH_DEV_HANDLE   dev_handle,
                                  RTOS_ERR         *p_err)
```

Arguments

dev_handle

Handle on the device.

p_err

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_INVALID_STATE

Returned Value

- Device specification number, if successful.
- 0, if unsuccessful.

Notes / Warnings

1. The value returned by this function corresponds to the 'bcdUSB' field of the device descriptor. For more information, see 'Universal Serial Bus specification, revision 2.0, section 9.6.1'.

USBH_DevClassGet()

Description

Gets the device class code.

Files

usbh_core_dev.h/usbh_core_dev.c

Prototype

```
CPU_INT08U USBH_DevClassGet (USBH_DEV_HANDLE   dev_handle,                     RTOS_ERR       *p_err)
```

Arguments

`dev_handle`

Handle on the device.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_INVALID_STATE`

Returned Value

Device class code.

Notes / Warnings

1. The value returned by this function corresponds to the bDeviceClass field of the device descriptor. For more information, see 'Universal Serial Bus specification, revision 2.0, section 9.6.1'.

USBH_DevSubclassGet()

Description

Gets the device subclass code.

Files

`usbh_core_dev.h/usbh_core_dev.c`

Prototype

```
typedef CPU_INT32U USBH_DevSpdGet (USBH_DEV_HANDLE   dev_handle,
                                   RTOS_ERR       *p_err)
```

Arguments

`dev_handle`

Handle on the device.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_INVALID_STATE`

Returned Value

Device subclass code.

Notes / Warnings

1. The value returned by this function corresponds to the bDeviceSubClass field of the device descriptor. For more information, see 'Universal Serial Bus specification, revision 2.0, section 9.6.1'.

USBH_DevProtocolGet()

Description

Gets the device protocol code.

Files

`usbh_core_dev.h/usbh_core_dev.c`

Prototype

```
typedef CPU_INT32U USBH_DevSpdGet (USBH_DEV_HANDLE    dev_handle,                         RTOS_ERR        *p_err)
```

Arguments

`dev_handle`

Handle on the device.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_INVALID_STATE`

Returned Value

Device protocol code.

Notes / Warnings

1. The value returned by this function corresponds to the bDeviceProtocol field of the device descriptor. For more information, see 'Universal Serial Bus specification, revision 2.0, section 9.6.1' .

USBH_DevVendorID_Get()

Description

Gets the device vendor ID.

Files

`usbh_core_dev.h/usbh_core_dev.c`

Prototype

```
typedef CPU_INT32U USBH_DevSpdGet (USBH_DEV_HANDLE   dev_handle,
                                   RTOS_ERR        *p_err)
```

Arguments

`dev_handle`

Handle on the device.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

- `RTOS_ERR_INVALID_STATE`

Returned Value

- Device protocol code, if successful.
- 0, if unsuccessful.

Notes / Warnings

1. The value returned by this function corresponds to the idVendor field of the device descriptor. For more information, see 'Universal Serial Bus specification, revision 2.0, section 9.6.1'.

USBH_DevProductID_Get()

Description

Gets the device product ID.

Files

`usbh_core_dev.h/usbh_core_dev.c`

Prototype

```
typedef CPU_INT32U USBH_DevSpdGet (USBH_DEV_HANDLE   dev_handle,
                                   RTOS_ERR         *p_err)
```

Arguments

`dev_handle`

Handle on the device.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_INVALID_STATE`

Returned Value

- Device protocol code, if successful.
- 0, if unsuccessful.

Notes / Warnings

1. The value returned by this function corresponds to the idProduct field of the device descriptor. For more information, see 'Universal Serial Bus specification, revision 2.0, section 9.6.1'.

USBH_DevRelNbrGet()

Description

Gets the device release number.

Files

`usbh_core_dev.h/usbh_core_dev.c`

Prototype

```
typedef CPU_INT32U USBH_DevSpdGet (USBH_DEV_HANDLE   dev_handle,
                                   RTOS_ERR         *p_err)
```

Arguments

`dev_handle`

Handle on the device.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_INVALID_STATE`

Returned Value

- Device release number, if successful.
- 0, if unsuccessful.

Notes / Warnings

1. The value returned by this function corresponds to the bcdDevice field of the device descriptor. For more information, see 'Universal Serial Bus specification, revision 2.0, section 9.6.1'.

USBH_DevManufacturerStrGet()

Description

Gets the device manufacturer string.

Files

`usbh_core_dev.h/usbh_core_dev.c`

Prototype

```
CPU_INT08U USBH_DevManufacturerStrGet (USBH_DEV_HANDLE   dev_handle,
                                       CPU_CHAR         *p_str_buf,
                                       CPU_INT08U        str_buf_len,
                                       RTOS_ERR         *p_err)
```

Arguments

`dev_handle`

Handle on the device.

`p_str_buf`

Buffer that will receive the string descriptor.

`str_buf_len`

String buffer length in octets.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ABORT`

- RTOS_ERR_INVALID_DESC
- RTOS_ERR_BLK_ALLOC_CALLBACK
- RTOS_ERR_NOT_FOUND
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_IS_OWNER
- RTOS_ERR_INVALID_STATE
- RTOS_ERR_URB_ALLOC
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_SEG_OVF
- RTOS_ERR_EP_INVALID
- RTOS_ERR_TIMEOUT
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_TX
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_WOULD_OVF

Returned Value

String length in octets, if successful. 0, if unsuccessful.

Notes / Warnings

1. The string returned by this function is retrieved via the `GetDescriptor(String)` standard request using the 'iManufacturer' field of the device descriptor. For more information, see 'Universal Serial Bus specification, revision 2.0, section 9.6.1'.
2. The string retrieved by this function is in unicode format.
3. This function can determine if a device manufacturer string is present by passing a `str_buf_len` of 0. In this case, `p_err` will be set to `RTOS_ERR_NOT_FOUND` if no string is available.

USBH_DevProductStrGet()

Description

Gets the device product string.

Files

`usbh_core_dev.h/usbh_core_dev.c`

Prototype

```
CPU_INT08U USBH_DevProductStrGet (USBH_DEV_HANDLE   dev_handle,
                                  CPU_CHAR         *p_str_buf,
                                  CPU_INT08U        str_buf_len,
                                  RTOS_ERR         *p_err)
```

Arguments

`dev_handle`

Handle on the device.

`p_str_buf`

Buffer that will receive the string descriptor.

`str_buf_len`

String buffer length in octets.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_INVALID_DESC`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_URB_ALLOC`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_TX`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`

Returned Value

String length in octets, if successful. 0, if unsuccessful.

Notes / Warnings

1. The string returned by this function is retrieved via the `GetDescriptor(String)` standard request using the iProduct field of the device descriptor. For more information, see 'Universal Serial Bus specification, revision 2.0, section 9.6.1'.
2. The string retrieved by this function is in unicode format.
3. This function can be used to determine if a device product string is present or not by passing a `str_buf_len` of 0. In this case, `p_err` will be set to `RTOS_ERR_NOT_FOUND` if no string is available.

USBH_DevSerNbrStrGet()

Description

Gets the device serial number string.

Files

`usbh_core_dev.h/usbh_core_dev.c`

Prototype

```
CPU_INT08U USBH_DevSerNbrStrGet (USBH_DEV_HANDLE   dev_handle,
                                 CPU_CHAR         *p_str_buf,
                                 CPU_INT08U        str_buf_len,
                                 RTOS_ERR         *p_err)
```

Arguments

`dev_handle`

Handle on the device.

`p_str_buf`

Buffer that will receive the string descriptor.

`str_buf_len`

String buffer length in octets.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_INVALID_DESC`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_URB_ALLOC`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_TX`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`

Returned Value

String length in octets, if successful. 0, if unsuccessful.

Notes / Warnings

1. The string returned by this function is retrieved via the `GetDescriptor(String)` standard request using the iSerialNumber field of the device descriptor. For more information, see 'Universal Serial Bus specification, revision 2.0, section 9.6.1'.
2. The string retrieved by this function is in unicode format.
3. This function can be used to determine if a device serial number string is present or not by passing a `str_buf_len` of 0. In this case, `p_err` will be set to `RTOS_ERR_NOT_FOUNDif` no string is available.

USBH_DevConfigQtyGet()

Description

Gets the quantity of configuration contained in the device.

Files

`usbh_core_dev.h/usbh_core_dev.c`

Prototype

```
CPU_INT08U USBH_DevConfigQtyGet (USBH_DEV_HANDLE   dev_handle,
                                 RTOS_ERR         *p_err)
```

Arguments

`dev_handle`

Handle on the device.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_INVALID_STATE`

Returned Value

Number of configuration, if successful. 0, if unsuccessful.

Notes / Warnings

1. The value returned by this function corresponds to the bNumConfigurations field of the device descriptor. For more information, see 'Universal Serial Bus specification, revision 2.0, section 9.6.1'.

USBH_DevConfigSet()

Description

Sets the given device configuration.

Files

`usbh_core_dev.h/usbh_core_dev.c`

Prototype

```
void USBH_DevConfigSet (USBH_DEV_HANDLE   dev_handle,
                        CPU_INT08U        cfg_nbr,
                        RTOS_ERR         *p_err)
```

Arguments

`dev_handle`

Handle on the device.

`cfg_nbr`

Configuration number to set.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_ALLOC`
- `RTOS_ERR_NOT_SUPPORTED`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_INVALID_STATE`

Returned Value

none.

Notes / Warnings

1. Set configuration is not supported on hub devices.
2. Two consecutive calls to this function will result in a failure of the second call, since the device handle passed as argument will necessarily be invalid.

USBH_DevConfigGet()

Description

Gets the currently selected configuration.

Files

`usbh_core_dev.h/usbh_core_dev.c`

Prototype

```
CPU_INT08U USBH_DevConfigGet (USBH_DEV_HANDLE   dev_handle,
                              RTOS_ERR         *p_err)
```

Arguments

`dev_handle`

Handle on the device.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_INVALID_STATE`

Returned Value

Device current configuration number.

Notes / Warnings

None.

USBH_DevPortNbrGet()

Description

Gets the device hub port number.

Files

`usbh_core_dev.h/usbh_core_dev.c`

Prototype

```
CPU_INT08U USBH_DevPortNbrGet (USBH_DEV_HANDLE   dev_handle,
                               RTOS_ERR         *p_err)
```

Arguments

`dev_handle`

Handle on the device.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_INVALID_STATE

Returned Value

Device current configuration number.

Notes / Warnings

None.

USBH_DevHubHandleGet()

Description

Gets handle on device's hub.

Files

usbh_core_dev.h/usbh_core_dev.c

Prototype

```
USBH_DEV_HANDLE USBH_DevHubHandleGet (USBH_DEV_HANDLE   dev_handle,
                                     RTOS_ERR         *p_err)
```

Arguments

dev_handle

Handle on device.

p_err

Pointer to the variable that will receive the return error code from this function :

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_INVALID_STATE

Returned Value

Device's hub handle, if successful. USBH_DEV_HANDLE_RH , if nearest hub is root hub. USBH_DEV_HANDLE_INVALID , if an error occured.

Notes / Warnings

None.

USBH_DevHS_HubNearestHandleGet()

Description

Gets the handle of device's nearest hub working at high-speed.

Files

usbh_core_dev.h/usbh_core_dev.c

Prototype

```
USBH_DEV_HANDLE USBH_DevHS_HubNearestHandleGet (USBH_DEV_HANDLE   dev_handle,
                                                CPU_INT08U       *p_port_nbr,
                                                RTOS_ERR         *p_err)
```

Arguments

`dev_handle`

Handle on the device.

`p_port_nbr`

Pointer to the variable that will receive the port number of the high-speed hub (if not null).

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_INVALID_DEV_SPD`
- `RTOS_ERR_INVALID_STATE`

Returned Value

Device's nearest high-speed hub handle, if successful. `USBH_DEV_HANDLE_RH` , if the nearest high-speed hub is the root hub. `USBH_DEV_HANDLE_INVALID` , if an error occurred.

Notes / Warnings

None.

USBH_DevDescRd()

Description

Reads the descriptor from the device.

Files

`usbh_core_dev.h/usbh_core_dev.c`

Prototype

```
CPU_INT16U USBH_DevDescRd (USBH_DEV_HANDLE dev_handle,
                           CPU_INT08U   recipient,
                           CPU_INT08U   type,
                           CPU_INT08U   desc_type,
                           CPU_INT08U   desc_ix,
                           CPU_INT16U   desc_len_req,
                           CPU_INT08U  *p_desc_buf,
                           CPU_INT16U   desc_buf_len,
                           RTOS_ERR    *p_err)
```

Arguments

`dev_handle`

Handle on the device.

`recipient`

Recipient of the Get descriptor request.

- USBH_REQ_RECIPIENT_DEV
- USBH_REQ_RECIPIENT_IF
- USBH_REQ_RECIPIENT_EP

type

Request type.

- USBH_REQ_TYPE_STD
- USBH_REQ_TYPE_CLASS
- USBH_REQ_TYPE_VENDOR

desc_type

Descriptor type.

- USBH_DESC_TYPE_DEV
- USBH_DESC_TYPE_CONFIG
- USBH_DESC_TYPE_STR
- USBH_DESC_TYPE_DEV_QUALIFIER
- USBH_DESC_TYPE_OTHER_SPD_CONFIG
- USBH_DESC_TYPE_IF_PWR
- USBH_DESC_TYPE_OTG

desc_ix

wIndex value that should be used in the setup request.

desc_len_req

wLength value to use in the setup request.

p_desc_buf

Pointer to the buffer that will receive the descriptor.

desc_buf_len

Length in octets of the descriptor buffer.

p_err

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_ABORT
- RTOS_ERR_BLK_ALLOC_CALLBACK
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_IS_OWNER
- RTOS_ERR_INVALID_STATE
- RTOS_ERR_URB_ALLOC
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_SEG_OVF
- RTOS_ERR_EP_INVALID
- RTOS_ERR_TIMEOUT
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_TX
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_WOULD_OVF

Returned Value

Descriptor length in octets.

Notes / Warnings

1. The GET_DESCRIPTOR request is described in 'Universal Serial Bus Specification Revision 2.0, section 9.4.3'.

USBH_ConfigAttribGet()

Description

Gets the configuration's attributes.

Files

`usbh_core_config.h/usbh_core_config.c`

Prototype

```
CPU_INT08U  USBH_ConfigAttribGet (USBH_DEV_HANDLE    dev_handle,
                                  RTOS_ERR          *p_err)
```

Arguments

`dev_handle`

Handle on the device.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_INVALID_STATE`

Returned Value

Configuration's attributes.

Notes / Warnings

1. The value returned by this function corresponds to the `bmAttributes` field of the configuration descriptor. For more information, see 'Universal Serial Bus specification, revision 2.0, section 9.6.3'.

USBH_ConfigFnctQtyGet()

Description

Gets the quantity of function contained in device's current configuration.

Files

`usbh_core_config.h/usbh_core_config.c`

Prototype

```
CPU_INT08U  USBH_ConfigFnctQtyGet (USBH_DEV_HANDLE    dev_handle,
                                   RTOS_ERR          *p_err)
```

Arguments

`dev_handle`

Handle on the the device.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_INVALID_STATE`

Returned Value

Quantity of function.

Notes / Warnings

None.

USBH_ConfigMaxPwrGet()

Description

Gets the configuration's maximum power consumption.

Files

`usbh_core_config.h/usbh_core_config.c`

Prototype

```
CPU_INT08U  USBH_ConfigMaxPwrGet (USBH_DEV_HANDLE    dev_handle,
                                  RTOS_ERR          *p_err)
```

Arguments

`dev_handle`

Handle on the device.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_INVALID_STATE`

Returned Value

Configuration's maximum power consumption.

Notes / Warnings

1. The value returned by this function corresponds to the `bMaxPower` field of the configuration descriptor. For more information, see 'Universal Serial Bus specification, revision 2.0, section 9.6.3'.

USBH_ConfigStrGet()

Description

Gets the configuration's string.

Files

`usbh_core_config.h/usbh_core_config.c`

Prototype

```
CPU_INT08U  USBH_ConfigStrGet (USBH_DEV_HANDLE    dev_handle,
                               CPU_CHAR          *p_str_buf,
                               CPU_INT08U         str_buf_len,
                               RTOS_ERR          *p_err)
```

Arguments

`dev_handle`

Handle on the device.

`p_str_buf`

Buffer that will receive the string descriptor.

`str_buf_len`

String buffer length in octets.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_INVALID_DESC`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_URB_ALLOC`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_TX`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`

Returned Value

String length in octets.

Notes / Warnings

1. The string returned by this function is retrieved via the `GetDescriptor(String)` standard request using the `iConfiguration` field of the configuration descriptor. For more information, see 'Universal Serial Bus specification, revision 2.0, section 9.6.3'.
2. The string retrieved by this function is in unicode format.
3. This function can be used to determine if a configuration string is present by passing a `str_buf_len` of 0. In this case, `p_err` will be set to `RTOS_ERR_NOT_FOUND` if no string is available.

USBH_FnctIF_QtyGet()

Description

Gets the quantity of interface contained in given function.

Files

`usbh_core_fnct.h/usbh_core_fnct.c`

Prototype

```
CPU_INT08U USBH_FnctIF_QtyGet (USBH_DEV_HANDLE    dev_handle,
                               USBH_FNCT_HANDLE   fnct_handle,
                               RTOS_ERR          *p_err)
```

Arguments

`dev_handle`

Handle on the device.

`fnct_handle`

Handle on the function.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_INVALID_STATE`

Returned Value

- Quantity of interface, if successful.
- 0, if unsuccessful.

Notes / Warnings

None.

USBH_FnctClassGet()

Description

Get the function's class code.

Files

`usbh_core_fnct.h/usbh_core_fnct.c`

Prototype

```
CPU_INT08U USBH_FnctClassGet (USBH_DEV_HANDLE    dev_handle,
                              USBH_FNCT_HANDLE   fnct_handle,
                              RTOS_ERR          *p_err)
```

Arguments

`dev_handle`

Handle on the device.

`fnct_handle`

Handle on the function.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_INVALID_STATE`

Returned Value

Class code, if successful. 0, if unsuccessful.

Notes / Warnings

1. The value returned by this function corresponds to the class code.
   (a) If the function has only one interface, it corresponds to the `bInterfaceClass` field of the interface descriptor. For more information, see 'Universal Serial Bus specification, revision 2.0, section 9.6.5'.
   (b) If the function has more than one interface, it corresponds to the `bFunctionClass` field of the Interface Association Descriptor (IAD). For more information, see 'USB ECN: Interface Association Descriptor'.
   (c) If the class code is defined at device level, it corresponds to the `bDeviceClass` field of the device descriptor. For more information, see 'Universal Serial Bus specification, revision 2.0, section 9.6.1'.

USBH_FnctSubclassGet()

Description

Get the function's subclass code.

Files

`usbh_core_fnct.h/usbh_core_fnct.c`

Prototype

```
typedef CPU_INT32U USBH_DevSpdGet (USBH_DEV_HANDLE   dev_handle,
                                  RTOS_ERR         *p_err)
```

Arguments

`dev_handle`

Handle on the device.

`fnct_handle`

Handle on the function.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_INVALID_STATE`

Returned Value

Subclass code, if successful. 0, if unsuccessful.

Notes / Warnings

1. The value returned by this function corresponds to the subclass code.
   (a) If the function has only one interface, it corresponds to the `bInterfaceSubClass` field of the interface descriptor. For more information, see 'Universal Serial Bus specification, revision 2.0, section 9.6.5'.

(b) If the function has more than one interface, it corresponds to the `bFunctionSubClass` field of the Interface Association Descriptor (IAD). For more information, see 'USB ECN: Interface Association Descriptor'.
(c) If the class code is defined at device level, it corresponds to the `bDeviceSubClass` field of the device descriptor. For more information, see 'Universal Serial Bus specification, revision 2.0, section 9.6.1'.

USBH_FnctProtocolGet()

Description

Get the function's protocol code.

Files

`usbh_core_fnct.h/usbh_core_fnct.c`

Prototype

```
typedef CPU_INT32U USBH_DevSpdGet (USBH_DEV_HANDLE   dev_handle,
                                   RTOS_ERR        *p_err)
```

Arguments

`dev_handle`

Handle on the device.

`fnct_handle`

Handle on the function.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_INVALID_STATE`

Returned Value

- Protocol code, if successful.
- 0, if unsuccessful.

Notes / Warnings

1. The value returned by this function corresponds to the protocol code.
(a) If the function has only one interface, it corresponds to the `bInterfaceProtocol` field of the interface descriptor. For more information, see 'Universal Serial Bus specification, revision 2.0, section 9.6.5'.
(b) If the function has more than one interface, it corresponds to the `bFunctionProtocol` field of the Interface Association Descriptor (IAD). See For more information, see 'USB ECN: Interface Association Descriptor'.
(c) If the class code is defined at device level, it corresponds to the `bDeviceProtocol` field of the device descriptor. For more information, see 'Universal Serial Bus specification, revision 2.0, section 9.6.1'.

USBH_FnctStrGet()

Description

Gets the given function's string descriptor.

Files

`usbh_core_fnct.h/usbh_core_fnct.c`

Prototype

```
CPU_INT08U USBH_FnctStrGet (USBH_DEV_HANDLE     dev_handle,
                            USBH_FNCT_HANDLE    fnct_handle,
                            CPU_CHAR           *p_str_buf,
                            CPU_INT08U          str_buf_len,
                            RTOS_ERR           *p_err)
```

Arguments

`dev_handle`

Handle on the device.

`fnct_handle`

Handle on the function.

`p_str_buf`

Pointer to the buffer that will receive the string descriptor.

`str_buf_len`

String buffer length in octets.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_NOT_FOUND`

Returned Value

String descriptor length in octets.

Notes / Warnings

1. The string returned by this function is retrieved via the `GetDescriptor(String)` standard request using the iConfiguration field of the configuration descriptor. For more information, see 'Universal Serial Bus specification, revision 2.0, section 9.6.3'.
2. The string retrieved by this function is in unicode format.
3. This function can determine if a function string is present or not by passing a `str_buf_len` of 0. In this case, `p_err` will be set to `RTOS_ERR_NOT_FOUND` if no string is available.

USBH_IF_NbrGet()

Description

Gets the interface number.

Files

`usbh_core_if.h/usbh_core_if.c`

Prototype

```
CPU_INT08U USBH_IF_NbrGet (USBH_DEV_HANDLE     dev_handle,
                           USBH_FNCT_HANDLE    fnct_handle,
                           CPU_INT08U          if_ix,
                           RTOS_ERR           *p_err)
```

Arguments

dev_handle

Handle on the device.

fnct_handle

Handle on the function.

if_ix

Index of the interface.

p_err

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_INVALID_ARG
- RTOS_ERR_INVALID_STATE

Returned Value

Interface number.

Notes / Warnings

1. The value returned by this function corresponds to the 'bInterfaceNumber' field of the interface descriptor. For more information, see 'Universal Serial Bus specification, revision 2.0, section 9.6.5'.

USBH_IF_AltNbrGet()

Description

Gets the interface alternate number.

Files

usbh_core_if.h/usbh_core_if.c

Prototype

```
CPU_INT08U USBH_IF_AltNbrGet (USBH_DEV_HANDLE    dev_handle,
                              USBH_FNCT_HANDLE   fnct_handle,
                              CPU_INT08U         if_ix,
                              RTOS_ERR          *p_err)
```

Arguments

dev_handle

Handle on the device.

fnct_handle

Handle on the function.

if_ix

Index of interface.

p_err

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_INVALID_ARG
- RTOS_ERR_INVALID_STATE

Returned Value

Interface alternate number.

Notes / Warnings

1. The value returned by this function corresponds to the `bAlternateSetting` field of the interface descriptor. For more information, see 'Universal Serial Bus specification, revision 2.0, section 9.6.5'.

USBH_IF_AltIxCurGet()

Description

Gets the interface's currently selected alternate number.

Files

`usbh_core_if.h/usbh_core_if.c`

Prototype

```
CPU_INT08U USBH_IF_AltIxCurGet (USBH_DEV_HANDLE    dev_handle,
                                USBH_FNCT_HANDLE   fnct_handle,
                                CPU_INT08U         if_ix,
                                RTOS_ERR          *p_err)
```

Arguments

`dev_handle`

Handle on the device.

`fnct_handle`

Handle on the function.

`if_ix`

Index of the interface.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_HANDLE
- TOS_ERR_INVALID_ARG
- RTOS_ERR_INVALID_STATE

Returned Value

Currently selected interface alternate number.

Notes / Warnings

None.

USBH_IF_EP_QtyGet()

Description

Gets the quantity of endpoint contained in interface.

Files

`usbh_core_if.h/usbh_core_if.c`

Prototype

```
CPU_INT08U USBH_IF_EP_QtyGet (USBH_DEV_HANDLE    dev_handle,
                              USBH_FNCT_HANDLE   fnct_handle,
                              CPU_INT08U         if_ix,
                              RTOS_ERR          *p_err)
```

Arguments

`dev_handle`

Handle on the device.

`fnct_handle`

Handle on the function.

`if_ix`

Index of the interface.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_INVALID_STATE`

Returned Value

- Quantity of endpoint, if successful.
- 0, if unsuccessful.

Notes / Warnings

None.

USBH_IF_ClassGet()

Description

Gets the interface class code number.

Files

`usbh_core_if.h/usbh_core_if.c`

Prototype

```
typedef CPU_INT32U USBH_DevSpdGet (USBH_DEV_HANDLE   dev_handle,
                                   RTOS_ERR         *p_err)
```

Arguments

dev_handle

Handle on the device.

fnct_handle

Handle on the function.

if_ix

Index of the interface.

p_err

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_INVALID_ARG
- RTOS_ERR_INVALID_STATE

Returned Value

- Interface class code, if successful.
- 0, if unsuccessful.

Notes / Warnings

1. The value returned by this function corresponds to the bInterfaceClass field of the interface descriptor. For more information, see 'Universal Serial Bus specification, revision 2.0, section 9.6.5'.

USBH_IF_SubclassGet()

Description

Gets the interface subclass code number.

Files

usbh_core_if.h/usbh_core_if.c

Prototype

```
typedef CPU_INT32U USBH_DevSpdGet (USBH_DEV_HANDLE    dev_handle,
                                   RTOS_ERR           *p_err)
```

Arguments

dev_handle

Handle on the device.

fnct_handle

Handle on the function.

if_ix

Index of the interface.

p_err

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE

- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_INVALID_ARG
- RTOS_ERR_INVALID_STATE

Returned Value

- Interface subclass code, if successful.
- 0, if unsuccessful.

Notes / Warnings

1. The value returned by this function corresponds to the bInterfaceSubClass field of the interface descriptor. For more information, see 'Universal Serial Bus specification, revision 2.0, section 9.6.5'.

USBH_IF_ProtocolGet()

Description

Gets the interface protocol code number.

Files

usbh_core_if.h/usbh_core_if.c

Prototype

```
typedef CPU_INT32U USBH_DevSpdGet (USBH_DEV_HANDLE    dev_handle,
                                  RTOS_ERR          *p_err)
```

Arguments

dev_handle

Handle on the device.

fnct_handle

Handle on the function.

if_ix

Index of the interface.

p_err

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_INVALID_ARG
- RTOS_ERR_INVALID_STATE

Returned Value

- Interface protocol code, if successful.
- 0, if unsuccessful.

Notes / Warnings

1. The value returned by this function corresponds to the bInterfaceProtcol field of the interface descriptor. For more information, see 'Universal Serial Bus specification, revision 2.0, section 9.6.5'.

USBH_IF_StrGet()

Description

Gets the interfaces's string.

Files

`usbh_core_if.h/usbh_core_if.c`

Prototype

```
CPU_INT08U USBH_IF_StrGet (USBH_DEV_HANDLE    dev_handle,
                           USBH_FNCT_HANDLE   fnct_handle,
                           CPU_INT08U         if_ix,
                           CPU_CHAR          *p_str_buf,
                           CPU_INT08U         str_buf_len,
                           RTOS_ERR          *p_err)
```

Arguments

`dev_handle`

Handle on the device.

`fnct_handle`

Handle on the function.

`if_ix`

Index of the interface.

`p_str_buf`

Buffer that will receive the string descriptor.

`str_buf_len`

String buffer length in octets.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_NOT_FOUND`

Returned Value

String length in octets.

Notes / Warnings

1. The string returned by this function is retrieved via the `GetDescriptor(String)` standard request using the ' `iInterface` ' field of the interface descriptor. For more information, see 'Universal Serial Bus specification, revision 2.0, section 9.6.5'.
2. The string retrieved by this function is in unicode format.
3. This function can be used to determine if an interface string is present or not by passing a `str_buf_len` of 0. In that case `p_err` will be set to `RTOS_ERR_NOT_FOUND` if no string is available.

USBH_IF_DescExtraGet()

Description

Gets the interfaces's extra descriptor(s).

Files

`usbh_core_if.h/usbh_core_if.c`

Prototype

```
CPU_INT08U *USBH_IF_DescExtraGet (USBH_FNCT_HANDLE   fnct_handle,
                                  CPU_INT08U         if_ix,
                                  CPU_INT16U        *p_desc_extra_len,
                                  RTOS_ERR          *p_err)
```

Arguments

`fnct_handle`

Handle on the function.

`if_ix`

Index of the interface.

`p_desc_extra_len`

Pointer to the variable that will receive total extra descriptors length in octets.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_NULL_PTR`
- `RTOS_ERR_INVALID_STATE`

Returned Value

Pointer to the buffer that contains interface's extra descriptor(s).

Notes / Warnings

1. First descriptor returned by this function will always be the interface descriptor itself.

USBH_IF_EP_AddrNextGet()

Description

Retrieves the address of the next endpoint available on interface.

Files

`usbh_core_if.h/usbh_core_if.c`

Prototype

```
CPU_INT08U USBH_IF_EP_AddrNextGet (USBH_DEV_HANDLE     dev_handle,
                                   USBH_FNCT_HANDLE    fnct_handle,
                                   CPU_INT08U          if_ix,
                                   CPU_INT08U          ep_addr_prev,
                                   RTOS_ERR           *p_err)
```

Arguments

`dev_handle`

Handle on the device.

`fnct_handle`

Handle on the function.

`if_ix`

Index of the interface.

`ep_addr_prev`

Previous endpoint address, or 0 u to obtain first endpoint address.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_INVALID_STATE`

Returned Value

- Next interface's available endpoint's address, if successful.
- 0 u, if an error occurred or if the last endpoint was reached.

Notes / Warnings

None.

USBH_IF_AltSet()

Description

Sets the alternate interface and attempt to open related endpoints.

Files

`usbh_core_if.h/usbh_core_if.c`

Prototype

```
void USBH_IF_AltSet (USBH_DEV_HANDLE       dev_handle,
                     USBH_FNCT_HANDLE      fnct_handle,
                     CPU_INT08U            if_ix,
                     CPU_INT08U            if_alt_ix,
                     USBH_IF_ALT_SET_CMPL  callback_fnct,
                     void                 *p_arg,
                     RTOS_ERR             *p_err)
```

Arguments

`dev_handle`

Handle on the device.

`fnct_handle`

Handle on the function.

`if_ix`

Index of the interface.

`if_alt_ix`

Index of the alternate interface.

`callback_fnct`

Function to call when operation is completed.

`p_arg`

Pointer to the argument that will be passed to the callback function.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_ALLOC`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_INVALID_STATE`

Returned Value

None.

Notes / Warnings

None.

USBH_EP_AttribGet()

Description

Gets the endpoint attributes.

Files

`usbh_core_ep.h/usbh_core_ep.c`

Prototype

```
CPU_INT08U USBH_EP_AttribGet (USBH_DEV_HANDLE    dev_handle,
                              USBH_FNCT_HANDLE   fnct_handle,
                              CPU_INT08U         if_ix,
                              CPU_INT08U         ep_addr,
                              RTOS_ERR          *p_err)
```

Arguments

`dev_handle`

Handle on the device.

`fnct_handle`

Handle on the function.

`if_ix`

Interface index.

`ep_addr`

Endpoint address.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_INVALID_STATE`

Returned Value

- Endpoint attributes, if successful.
- 0, if unsuccessful.

Notes / Warnings

1. The value returned by this function corresponds to the '`bmAttributes`' field of the endpoint descriptor. See 'Universal Serial Bus specification, revision 2.0, section 9.6.6' for more information.

USBH_EP_MaxPktSizeGet()

Description

Gets the endpoint maximum packet size.

Files

`usbh_core_ep.h/usbh_core_ep.c`

Prototype

```
CPU_INT16U USBH_EP_MaxPktSizeGet (USBH_DEV_HANDLE    dev_handle,
                                  USBH_FNCT_HANDLE   fnct_handle,
                                  CPU_INT08U         if_ix,
                                  CPU_INT08U         ep_addr,
                                  RTOS_ERR          *p_err)
```

Arguments

`dev_handle`

Handle on device.

`fnct_handle`

Handle on function.

`if_ix`

Interface index.

`ep_addr`

Endpoint address.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_HANDLE`

- RTOS_ERR_INVALID_ARG
- RTOS_ERR_INVALID_STATE

Returned Value

- Endpoint maximum packet size, if successful.
- 0, if unsuccessful.

Notes / Warnings

1. The value returned by this function corresponds to the 'wMaxPacketSize' field of the endpoint descriptor. For more information, see 'Universal Serial Bus specification, revision 2.0, section 9.6.6'.

USBH_EP_IntervalGet()

Description

Gets the endpoint interval.

Files

usbh_core_ep.h/usbh_core_ep.c

Prototype

```
CPU_INT08U USBH_EP_IntervalGet (USBH_DEV_HANDLE    dev_handle,
                                USBH_FNCT_HANDLE   fnct_handle,
                                CPU_INT08U         if_ix,
                                CPU_INT08U         ep_addr,
                                RTOS_ERR          *p_err)
```

Arguments

dev_handle

Handle on device.

fnct_handle

Handle on function.

if_ix

Interface index.

ep_addr

Endpoint address.

p_err

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_INVALID_ARG
- RTOS_ERR_INVALID_STATE

Returned Value

- Endpoint interval, if successful.
- 0, if unsuccessful.

Notes / Warnings

1. The value returned by this function corresponds to the ' `bInterval` ' field of the endpoint descriptor. For more information, see 'Universal Serial Bus specification, revision 2.0, section 9.6.6' .

USBH_EP_CtrlXfer()

Description

Perform the synchronous control transfer on endpoint.

Files

`usbh_core_ep.h/usbh_core_ep.c`

Prototype

```
CPU_INT16U USBH_EP_CtrlXfer (USBH_DEV_HANDLE    dev_handle,
                             CPU_INT08U         req,
                             CPU_INT08U         req_type,
                             CPU_INT16U         val,
                             CPU_INT16U         ix,
                             CPU_INT08U        *p_buf,
                             CPU_INT16U         len,
                             CPU_INT16U         buf_len,
                             CPU_INT32U         timeout,
                             RTOS_ERR          *p_err)
```

Arguments

`dev_handle`

Handle on the device.

`req`

bRequest value of the setup packet.

`req_type`

bmRequestType value of the setup packet.

`val`

wValue value of the setup packet.

`ix`

wIndex value of the setup packet.

`p_buf`

Pointer to the data buffer for the data phase.

`len`

wLength value of the setup packet that specifies the number of data bytes in data stage.

`buf_len`

Buffer length in octets.

`timeout`

Timeout, in milliseconds.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_ABORT
- RTOS_ERR_BLK_ALLOC_CALLBACK
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_IS_OWNER
- RTOS_ERR_INVALID_STATE
- RTOS_ERR_URB_ALLOC
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_SEG_OVF
- RTOS_ERR_EP_INVALID
- RTOS_ERR_TIMEOUT
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_TX
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_WOULD_OVF

Returned Value

Number of octets transfered in data stage (if any).

Notes / Warnings

None.

USB Host PBHCI API

USB Host PBHCI API

- USBH_PBHCI_ConfigureSOF_EventQty()
- USBH_PBHCI_ConfigureSchedTaskStk()
- USBH_PBHCI_ConfigureMemSeg()
- USBH_PBHCI_Init()
- USBH_PBHCI_SchedTaskPrioSet()
- USBH_PBHCI_UnInit()

USBH_PBHCI_ConfigureSOF_EventQty()

Description

Configures the quantity of Start-Of-Frame events.

Files

`usbh_pbhci.h/usbh_pbhci.c`

Prototype

```
void  USBH_PBHCI_ConfigureSOF_EventQty (CPU_INT16U  sof_event_qty)
```

Arguments

`sof_event_qty`

Quantity of start-of-frame events.

Returned Value

None.

Notes / Warnings

1. Calling this function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the USB Host PBHCI module is initialized via the `USBH_PBHCI_Init()` function.

USBH_PBHCI_ConfigureSchedTaskStk()

Description

Configures the USB host PBHCI scheduler task's stack.

Files

`usbh_pbhci.h/usbh_pbhci.c`

Prototype

```
void  USBH_PBHCI_ConfigureSchedTaskStk (CPU_INT32U   stk_size_elements,
                                        void        *p_stk)
```

Arguments

`stk_size_elements`

Size, in stack elements, of the task's stack.

`p_stk`

Pointer to base of the task's stack. If DEF_NULL, stack will be allocated from KAL's memory segment.

Returned Value

None.

Notes / Warnings

1. Calling this function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the USB Host PBHCI module is initialized via the `USBH_PBHCI_Init()` function.
3. In order to change the priority of the USB host PBHCI scheduler task, use the function `USBH_PBHCI_SchedTaskPrioSet()`.

USBH_PBHCI_ConfigureMemSeg()

Description

Configures the memory segment to use when allocating control data and buffers.

Files

`usbh_pbhci.h/usbh_pbhci.c`

Prototype

```
void USBH_PBHCI_ConfigureMemSeg (MEM_SEG *p_mem_seg)
```

Arguments

`p_mem_seg`

Pointer to memory segment to use when allocating control data.

`DEF_NULL` means general purpose heap segment.

Returned Value

None.

Notes / Warnings

1. Calling this function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the PBHCI module is initialized via the `USBH_PBHCI_Init()` function.

USBH_PBHCI_Init()

Description

Initializes the Pipe-Based Host Controller Interface (PBHCI).

Files

`usbh_pbhci.h/usbh_pbhci.c`

Prototype

```
void USBH_PBHCI_Init (USBH_PBHCI_CFG    *p_cfg,
                      RTOS_TASK_CFG     *p_sched_task_cfg,
                      MEM_SEG           *p_mem_seg,
                      RTOS_ERR          *p_err)
```

Arguments

`p_cfg`

Pointer to the PBHCI configuration structure.

`p_sched_task_cfg`

Pointer to the configuration structure for scheduler task.

`p_mem_seg`

Pointer to the memory segment where to allocate PBHCI data. Passing `DEF_NULL` will make the PBHCI use the general-purpose heap.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_NOT_AVAIL`

Returned Value

None.

Notes / Warnings

None.

USBH_PBHCI_SchedTaskPrioSet()

Description

Assigns a new priority to the USB host PBHCI scheduler task.

Files

`usbh_pbhci.h/usbh_pbhci.c`

Prototype

```
void  USBH_PBHCI_SchedTaskPrioSet(CPU_INT08U   prio,
                                  RTOS_ERR    *p_err)
```

Arguments

`prio`

New priority of the the scheduler task.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_ARG

Returned Value

None.

Notes / Warnings

None.

USBH_PBHCI_UnInit()

Description

Un-initializes Pipe-Based Host Controller Interface (PBHCI).

Files

usbh_pbhci.h/usbh_pbhci.c

Prototype

```
void USBH_PBHCI_UnInit (RTOS_ERR *p_err)
```

Arguments

p_err

Pointer to the variable that will receive this return error code from this function :

- RTOS_ERR_NONE

Returned Value

None.

Notes / Warnings

None.

USB Host Shell Commands API

USB Host Shell Commands API

- USBH_ShellCmdInit()

USBH_ShellCmdInit()

Description

Initializes the USBH's Shell command list.

Files

`usbh_cmd.h/usbh_cmd.c`

Prototype

```
void USBH_ShellCmdInit (RTOS_ERR  *p_err)
```

Arguments

None.

Returned Value

- `DEF_OK` , if an interface was opened.
- `DEF_FAIL` , if no interfaces were opened.

Notes / Warnings

None.

USB Host HUB API

USB Host HUB API

- USBH_HUB_TaskPrioSet()
- USBH_HUB_PortSuspendReq()
- USBH_HUB_PortResumeReq()
- USBH_HUB_PortResetReq()
- USBH_HUB_PortDisconnReq()

USBH_HUB_TaskPrioSet()

Description

Assigns a new priority to the USB host HUB task.

Files

`usbh_core_hub.h/usbh_core_hub.c`

Prototype

```
void  USBH_HUB_TaskPrioSet(CPU_INT08U   prio,
                           RTOS_ERR    *p_err)
```

Arguments

`prio`

New priority of the the hub task.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_INVALID_ARG`

Returned Value

None.

Notes / Warnings

1. This function cannot be called before the USB Host module has been initialized via the `USBH_Init()` function.

USBH_HUB_PortSuspendReq()

Description

Requests a suspend on the specified port and hub.

Files

`usbh_core_hub.h/usbh_core_hub.c`

Prototype

```
void USBH_HUB_PortSuspendReq (USBH_DEV_HANDLE      hub_dev_handle,
                              CPU_INT08U           port_nbr,
                              USBH_PORT_REQ_CMPL   cmpl_callback,
                              void                *p_arg,
                              RTOS_ERR            *p_err)
```

Arguments

`hub_dev_handle`

Handle on the hub device.

`port_nbr`

Port number, can be USBH_HUB_PORT_ALL.

`cmpl_callback`

Callback to notify caller that the operation is complete. Can be DEF_NULL.

`p_arg`

Argument that will be passed to callback.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ALLOC`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_NOT_AVAIL`

Returned Value

None.

Notes / Warnings

None.

USBH_HUB_PortResumeReq()

Description

Request resume on the specified port and hub.

Files

`usbh_core_hub.h/usbh_core_hub.c`

Prototype

```
void USBH_HUB_PortResumeReq (USBH_DEV_HANDLE      hub_dev_handle,
                             CPU_INT08U           port_nbr,
                             USBH_PORT_REQ_CMPL   cmpl_callback,
                             void                *p_arg,
                             RTOS_ERR            *p_err)
```

Arguments

`hub_dev_handle`

Handle on the hub device.

port_nbr

Port number, can be USBH_HUB_PORT_ALL.

cmpl_callback

Callback to notify caller that the operation is complete. Can be DEF_NULL.

p_arg

Argument that will be passed to the callback.

p_err

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_ALLOC
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_NOT_AVAIL

Returned Value

None.

Notes / Warnings

None.

USBH_HUB_PortResetReq()

Description

Requests the reset on specified port and hub.

Files

usbh_core_hub.h/usbh_core_hub.c

Prototype

```
void USBH_HUB_PortResetReq (USBH_DEV_HANDLE       hub_dev_handle,
                            CPU_INT08U            port_nbr,
                            USBH_PORT_REQ_CMPL    cmpl_callback,
                            void                 *p_arg,
                            RTOS_ERR             *p_err)
```

Arguments

hub_dev_handle

Handle on the hub device.

port_nbr

Port number, can be USBH_HUB_PORT_ALL.

cmpl_callback

Callback to notify the caller that the operation is complete. Can be DEF_NULL.

p_arg

Argument that will be passed to the callback.

p_err

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_ALLOC
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_NOT_AVAIL

Returned Value

None.

Notes / Warnings

None.

USBH_HUB_PortDisconnReq()

Description

Request a disconnect on the specified port and hub.

Files

usbh_core_hub.h/usbh_core_hub.c

Prototype

```
void USBH_HUB_PortDisconnReq (USBH_DEV_HANDLE      hub_dev_handle,
                              CPU_INT08U           port_nbr,
                              USBH_PORT_REQ_CMPL cmpl_callback,
                              void                *p_arg,
                              RTOS_ERR            *p_err)
```

Arguments

hub_dev_handle

Handle on the hub device.

port_nbr

Port number, can be USBH_HUB_PORT_ALL.

cmpl_callback

Callback to notify the caller that the operation is complete. Can be DEF_NULL.

p_arg

Argument that will be passed to the callback.

p_err

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_ALLOC
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_NOT_AVAIL

Returned Value

None.

Notes / Warnings

None.

USB Host AOAP API

USB Host AOAP API

- USBH_AOAP_ConfigureBufAlignOctets()
- USBH_AOAP_ConfigureOptimizeSpdCfg()
- USBH_AOAP_ConfigureInitAllocCfg()
- USBH_AOAP_ConfigureMemSeg()
- USBH_AOAP_Init()
- USBH_AOAP_StdReqTimeoutSet()
- USBH_AOAP_DevHandleGet()
- USBH_AOAP_AccDataRx()
- USBH_AOAP_AccDataTx()

USBH_AOAP_ConfigureBufAlignOctets()

Description

Configures the alignment of the internal buffers.

Files

`usbh_aoap.h/usbh_aoap.c`

Prototype

```
void  USBH_AOAP_ConfigureBufAlignOctets (CPU_SIZE_T  buf_align_octets)
```

Arguments

`buf_align_octets`

Buffer alignment, in octets.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the AOAP class is initialized via the `USBH_AOAP_Init()` function.

USBH_AOAP_ConfigureOptimizeSpdCfg()

Description

Sets the configurations required when optimize speed mode is enabled.

Files

`usbh_aoap.h/usbh_aoap.c`

Prototype

```
void USBH_AOAP_ConfigureOptimizeSpdCfg(const USBH_AOAP_CFG_OPTIMIZE_SPD *p_optimize_spd_cfg)
```

Arguments

`p_optimize_spd_cfg`

Pointer to the structure containing the configurations for the optimize speed mode.

Returned Value

None.

Notes / Warnings

1. This function MUST be called before the AOAP class is initialized via the `USBH_AOAP_Init()` function.
2. This function MUST be called when the `USBH_CFG_OPTIMIZE_SPD_EN` configuration is set to `DEF_ENABLED`.

USBH_AOAP_ConfigureInitAllocCfg()

Description

Sets the configurations required when allocation at initialization mode is enabled.

Files

`usbh_aoap.h/usbh_aoap.c`

Prototype

```
void USBH_AOAP_ConfigureInitAllocCfg (const USBH_AOAP_CFG_INIT_ALLOC *p_init_alloc_cfg)
```

Arguments

`p_init_alloc_cfg`

Pointer to the structure containing the configurations for the allocation at initialization mode.

Returned Value

None.

Notes / Warnings

1. This function MUST be called before the AOAP class is initialized via the `USBH_AOAP_Init()` function.
2. This function MUST be called when the `USBH_CFG_INIT_ALLOC_EN` configuration is set to `DEF_ENABLED`.

USBH_AOAP_ConfigureMemSeg()

Description

Configures the memory segment to use when allocating control data and buffers.

Files

`usbh_aoap.h/usbh_aoap.c`

Prototype

```
void USBH_AOAP_ConfigureMemSeg (MEM_SEG *p_mem_seg,
                                MEM_SEG *p_mem_seg_buf)
```

Arguments

`p_mem_seg`

- Pointer to memory segment to use when allocating control data. Can be the same segment used for `p_mem_seg_buf`.
- `DEF_NULL` means general purpose heap segment.

`p_mem_seg_buf`

- Pointer to memory segment to use when allocating data buffers. Can be the same segment used for `p_mem_seg`.
- `DEF_NULL` means general purpose heap segment.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the AOAP class is initialized via the `USBH_AOAP_Init()` function.

USBH_AOAP_Init()

Description

Initializes the AOAP Class.

Files

`usbh_aoap.h/usbh_aoap.c`

Prototype

```
void USBH_AOAP_Init (USBH_AOAP_STR_CFG    *p_str_cfg,
                     USBH_AOAP_APP_FNCTS  *p_app_fncts,
                     RTOS_ERR             *p_err)
```

Arguments

`p_str_cfg`

Pointer to the configuration structure containing the AOAP strings. Content MUST be persistent.

`p_app_fncts`

Pointer to the callback functions structure. Content MUST be persistent.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`

Returned Value

None.

Notes / Warnings

None.

USBH_AOAP_StdReqTimeoutSet()

Description

Assigns a new timeout delay for the AOAP standard requests.

Files

`usbh_aoap.h/usbh_aoap.c`

Prototype

```
void  USBH_AOAP_StdReqTimeoutSet(CPU_INT32U   std_req_timeout_ms,
                                 RTOS_ERR    *p_err)
```

Arguments

`std_req_timeout_ms`

New timeout, in milliseconds.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

None.

USBH_AOAP_DevHandleGet()

Description

Retrieves the device handle associated with the AOAP device.

Files

`usbh_aoap.h/usbh_aoap.c`

Prototype

```
USBH_DEV_HANDLE USBH_AOAP_DevHandleGet (USBH_AOAP_FNCT_HANDLE   aoap_fnct_handle,
                                        RTOS_ERR               *p_err)
```

Arguments

`aoap_fnct_handle`

Handle on the AOAP function.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_NOT_AVAIL`

Returned Value

Device handle.

Notes / Warnings

None.

USBH_AOAP_AccDataRx()

Description

Receives the data from android accessory.

Files

`usbh_aoap.h/usbh_aoap.c`

Prototype

```
CPU_INT32U USBH_AOAP_AccDataRx (USBH_AOAP_FNCT_HANDLE   aoap_fnct_handle,
                                CPU_INT08U             *p_buf,
                                CPU_INT32U              buf_len,
                                CPU_INT32U              timeout,
                                RTOS_ERR               *p_err)
```

Arguments

`aoap_fnct_handle`

Handle on the AOAP function.

`p_buf`

Pointer to the buffer that will receive the data.

`buf_len`

Buffer length in octets.

`timeout`

Timeout in milliseconds.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_URB_ALLOC`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_NOT_AVAIL`

Returned Value

Received length, in octets.

Notes / Warnings

None.

USBH_AOAP_AccDataTx()

Description

Send data to the android accessory.

Files

usbh_aoap.h/usbh_aoap.c

Prototype

```
CPU_INT32U USBH_AOAP_AccDataTx (USBH_AOAP_FNCT_HANDLE   aoap_fnct_handle,
                                CPU_INT08U             *p_buf,
                                CPU_INT32U              buf_len,
                                CPU_INT32U              timeout,
                                RTOS_ERR               *p_err)
```

Arguments

aoap_fnct_handle

Handle on the AOAP function.

p_buf

Pointer to the buffer that contains the data to send.

buf_len

Buffer length in octets.

timeout

Timeout in milliseconds.

p_err

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_ABORT
- RTOS_ERR_BLK_ALLOC_CALLBACK
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_INVALID_STATE
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_URB_ALLOC
- RTOS_ERR_EP_INVALID
- RTOS_ERR_SEG_OVF
- RTOS_ERR_OS_ILLEGAL_RUN_TIME
- RTOS_ERR_TIMEOUT
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_NOT_AVAIL

Returned Value

Transfer length, in octets.

Notes / Warnings

None.

USB Host CDC API

USB Host CDC API

- USBH_CDC_ConfigureBufAlignOctets()
- USBH_CDC_ConfigureEventURB_Qty()
- USBH_CDC_ConfigureOptimizeSpdCfg()
- USBH_CDC_ConfigureInitAllocCfg()
- USBH_CDC_ConfigureMemSeg()
- USBH_CDC_Init()
- USBH_CDC_StdReqTimeoutSet()
- USBH_CDC_PostInit()
- USBH_CDC_DevHandleGet()
- USBH_CDC_FnctHandleGet()
- USBH_CDC_DCI_QtyGet()
- USBH_CDC_RelNbrGet()
- USBH_CDC_EncapsulatedCmdTx()
- USBH_CDC_EncapsulatedRespRx()
- USBH_CDC_CommFeatureSet()
- USBH_CDC_CommFeatureGet()
- USBH_CDC_CommFeatureClr()
- USBH_CDC_LineCodingSet()
- USBH_CDC_LineCodingGet()
- USBH_CDC_CtrlLineStateSet()
- USBH_CDC_BrkSend()

USBH_CDC_ConfigureBufAlignOctets()

Description

Configures the alignment of the internal buffers.

Files

`usbh_cdc.h/usbh_cdc.c`

Prototype

```
void  USBH_CDC_ConfigureBufAlignOctets (CPU_SIZE_T  buf_align_octets)
```

Arguments

`buf_align_octets`

Buffer alignment, in octets.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the CDC class is initialized via the `USBH_CDC_Init()` function.

USBH_CDC_ConfigureEventURB_Qty()

Description

Configures the quantity of URBs used to retrieve CDC status to allocate/submit.

Files

`usbh_cdc.h/usbh_cdc.c`

Prototype

```
void USBH_CDC_ConfigureEventURB_Qty (CPU_INT08U event_urb_qty)
```

Arguments

`event_urb_qty`

Quantity of event URBs.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the CDC class is initialized via the `USBH_CDC_Init()` function.

USBH_CDC_ConfigureOptimizeSpdCfg()

Description

Sets the configurations required when optimize speed mode is enabled.

Files

`usbh_cdc.h/usbh_cdc.c`

Prototype

```
void USBH_CDC_ConfigureOptimizeSpdCfg(const USBH_CDC_CFG_OPTIMIZE_SPD *p_optimize_spd_cfg)
```

Arguments

`p_optimize_spd_cfg`

Pointer to the structure containing the configurations for the optimize speed mode.

Returned Value

None.

Notes / Warnings

1. This function MUST be called before the CDC class is initialized via the `USBH_CDC_Init()` function.
2. This function MUST be called when the `USBH_CFG_OPTIMIZE_SPD_EN` configuration is set to `DEF_ENABLED`.

USBH_CDC_ConfigureInitAllocCfg()

Description

Sets the configurations required when allocation at initialization mode is enabled.

Files

`usbh_cdc.h/usbh_cdc.c`

Prototype

```
void  USBH_CDC_ConfigureInitAllocCfg  (const  USBH_CDC_CFG_INIT_ALLOC  *p_init_alloc_cfg)
```

Arguments

`p_init_alloc_cfg`

Pointer to the structure containing the configurations for the allocation at initialization mode.

Returned Value

None.

Notes / Warnings

1. This function MUST be called before the CDC class is initialized via the `USBH_CDC_Init()` function.
2. This function MUST be called when the `USBH_CFG_INIT_ALLOC_EN` configuration is set to `DEF_ENABLED`.

USBH_CDC_ConfigureMemSeg()

Description

Configures the memory segment to use when allocating control data and buffers.

Files

`usbh_cdc.h/usbh_cdc.c`

Prototype

```
void  USBH_CDC_ConfigureMemSeg (MEM_SEG  *p_mem_seg,
                                MEM_SEG  *p_mem_seg_buf)
```

Arguments

`p_mem_seg`

- Pointer to memory segment to use when allocating control data. Can be the same segment used for `p_mem_seg_buf`.
- `DEF_NULL` means general purpose heap segment.

`p_mem_seg_buf`

- Pointer to memory segment to use when allocating data buffers. Can be the same segment used for `p_mem_seg`.
- `DEF_NULL` means general purpose heap segment.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the CDC class is initialized via the `USBH_CDC_Init()` function.

USBH_CDC_Init()

Description

Initializes the Communication Device Class (CDC) driver.

Files

`usbh_cdc.h/usbh_cdc.c`

Prototype

```
void USBH_CDC_Init ( RTOS_ERR *p_err)
```

Arguments

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_NOT_AVAIL`

Returned Value

None.

Notes / Warnings

None.

USBH_CDC_StdReqTimeoutSet()

Description

Assigns a new timeout delay for the CDC standard requests.

Files

`usbh_cdc.h/usbh_cdc.c`

Prototype

```
void  USBH_CDC_StdReqTimeoutSet (CPU_INT32U   std_req_timeout_ms,
                                 RTOS_ERR    *p_err)
```

Arguments

`std_req_timeout_ms`

New timeout, in milliseconds.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

None.

USBH_CDC_PostInit()

Description

Post-initializes the Communication Device Class (CDC) driver once all the subclass drivers added.

Files

`usbh_cdc.h/usbh_cdc.c`

Prototype

```
void USBH_CDC_PostInit (RTOS_ERR *p_err)
```

Arguments

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`

Returned Value

None.

Notes / Warnings

1. This function MUST be called by the application once all the subclass drivers are initialized and BEFORE starting the USB Host product.

USBH_CDC_DevHandleGet()

Description

Gets the device handle of a given CDC function.

Files

`usbh_cdc.h/usbh_cdc.c`

Prototype

```
USBH_DEV_HANDLE USBH_CDC_DevHandleGet (USBH_CDC_FNCT_HANDLE   cdc_fnct_handle,
                                       RTOS_ERR              *p_err)
```

Arguments

`cdc_fnct_handle`

Handle on the CDC function.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_OS_OBJ_DEL`

- RTOS_ERR_NOT_AVAIL

Returned Value

Device handle.

Notes / Warnings

None.

USBH_CDC_FnctHandleGet()

Description

Gets the function handle of a given CDC function.

Files

usbh_cdc.h/usbh_cdc.c

Prototype

```
USBH_FNCT_HANDLE USBH_CDC_FnctHandleGet (USBH_CDC_FNCT_HANDLE    cdc_fnct_handle,
                                         RTOS_ERR               *p_err)
```

Arguments

cdc_fnct_handle

Handle on the CDC function.

p_err

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_ABORT
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_TIMEOUT
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_NOT_AVAIL

Returned Value

Function handle.

Notes / Warnings

None.

USBH_CDC_DCI_QtyGet()

Description

Gets the quantity of Data Class Interface (DCI) for a given CDC function.

Files

usbh_cdc.h/usbh_cdc.c

Prototype

```
CPU_INT08U USBH_CDC_DCI_QtyGet (USBH_CDC_FNCT_HANDLE cdc_fnct_handle,
                                RTOS_ERR            *p_err)
```

Arguments

`cdc_fnct_handle`

Handle on CDC function.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_NOT_AVAIL`

Returned Value

Number of DCI.

Notes / Warnings

None.

USBH_CDC_RelNbrGet()

Description

Gets the CDC function release number (bcdCDC).

Files

`usbh_cdc.h/usbh_cdc.c`

Prototype

```
CPU_INT16U USBH_CDC_RelNbrGet (USBH_CDC_FNCT_HANDLE   cdc_fnct_handle,
                               RTOS_ERR              *p_err)
```

Arguments

`cdc_fnct_handle`

Handle on CDC function.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_NOT_AVAIL`

Returned Value

Release number.

Notes / Warnings

1. For more information on CDC release number, see 'USB Class Definitions for Communication Devices Specification", version 1.2, Section 5.2.3.1'.

USBH_CDC_EncapsulatedCmdTx()

Description

Sends the CDC encapsulated command.

Files

usbh_cdc.h/usbh_cdc.c

Prototype

```
CPU_INT16U USBH_CDC_EncapsulatedCmdTx (USBH_CDC_FNCT_HANDLE   cdc_fnct_handle,
                                       CPU_INT08U            *p_buf,
                                       CPU_INT16U             buf_len,
                                       CPU_INT32U             timeout,
                                       RTOS_ERR              *p_err)
```

Arguments

cdc_fnct_handle

Handle on the CDC function.

p_buf

Pointer to the buffer that contains the command.

buf_len

Buffer length in octets.

timeout

Timeout in milliseconds.

p_err

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_ABORT
- RTOS_ERR_BLK_ALLOC_CALLBACK
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_IS_OWNER
- RTOS_ERR_INVALID_STATE
- RTOS_ERR_URB_ALLOC
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_SEG_OVF
- RTOS_ERR_EP_INVALID
- RTOS_ERR_INVALID_ARG
- RTOS_ERR_TIMEOUT
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_TX
- RTOS_ERR_WOULD_OVF

Returned Value

Number of octets transferred.

Notes / Warnings

1. For more information on `SendEncapsulatedCommand`, see 'USB Class Definitions for Communication Devices Specification", version 1.2, Section 6.2.1'.

USBH_CDC_EncapsulatedRespRx()

Description

Send the CDC encapsulated command.

Files

`usbh_cdc.h/usbh_cdc.c`

Prototype

```
CPU_INT16U USBH_CDC_EncapsulatedRespRx (USBH_CDC_FNCT_HANDLE   cdc_fnct_handle,
                                        CPU_INT08U            *p_buf,
                                        CPU_INT16U             buf_len,
                                        CPU_INT32U             timeout,
                                        RTOS_ERR              *p_err)
```

Arguments

`cdc_fnct_handle`

Handle on the CDC function.

`p_buf`

Pointer to the buffer that will receive data.

`buf_len`

Buffer length in octets.

`timeout`

Timeout in milliseconds.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_URB_ALLOC`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_TIMEOUT`

- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_TX
- RTOS_ERR_WOULD_OVF

Returned Value

Number of octets received.

Notes / Warnings

1. For more information on GetEncapsulatedCommand, see 'USB Class Definitions for Communication Devices Specification", version 1.2, Section 6.2.2'.

USBH_CDC_CommFeatureSet()

Description

Configures the CDC communication feature.

Files

usbh_cdc.h/usbh_cdc.c

Prototype

```
void USBH_CDC_CommFeatureSet (USBH_CDC_FNCT_HANDLE   cdc_fnct_handle,
                              CPU_INT08U             feature,
                              CPU_INT16U             data,
                              CPU_INT32U             timeout,
                              RTOS_ERR              *p_err)
```

Arguments

cdc_fnct_handle

Handle on CDC function.

feature

Feature to configure, which includes:

- USBH_CDC_COMM_FEATURE_ABSTRACT_STATE
- USBH_CDC_COMM_FEATURE_COUNTRY_SETTING

data

Data of the configured communication feature request.

timeout

Timeout in milliseconds.

p_err

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_ABORT
- RTOS_ERR_BLK_ALLOC_CALLBACK
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_OS_SCHED_LOCKED

- RTOS_ERR_IS_OWNER
- RTOS_ERR_INVALID_STATE
- RTOS_ERR_URB_ALLOC
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_SEG_OVF
- RTOS_ERR_EP_INVALID
- RTOS_ERR_INVALID_ARG
- RTOS_ERR_TIMEOUT
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_TX
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_WOULD_OVF

Returned Value

None.

Notes / Warnings

1. For more information on `SetCommFeature` request, see 'Communication Class Subclass Specification for PSTN Devices, version 1.2, Section 6.3.1'.

USBH_CDC_CommFeatureGet()

Description

Gets CDC function line coding.

Files

usbh_cdc.h/usbh_cdc.c

Prototype

```
CPU_INT16U USBH_CDC_CommFeatureGet (USBH_CDC_FNCT_HANDLE   cdc_fnct_handle,
                                    CPU_INT08U             feature,
                                    CPU_INT32U             timeout,
                                    RTOS_ERR              *p_err)
```

Arguments

cdc_fnct_handle

Handle on the CDC function.

feature

Feature to Get, which includes:

- USBH_CDC_COMM_FEATURE_ABSTRACT_STATE
- USBH_CDC_COMM_FEATURE_COUNTRY_SETTING

timeout

Timeout in milliseconds.

p_err

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_ABORT

- RTOS_ERR_BLK_ALLOC_CALLBACK
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_IS_OWNER
- RTOS_ERR_INVALID_STATE
- RTOS_ERR_URB_ALLOC
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_SEG_OVF
- RTOS_ERR_EP_INVALID
- RTOS_ERR_INVALID_ARG
- RTOS_ERR_TIMEOUT
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_TX
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_WOULD_OVF

Returned Value

Data of Get communication feature request.

Notes / Warnings

1. For more information on `GetCommFeature` request, see 'Communication Class Subclass Specification for PSTN Devices, version 1.2, Section 6.3.2'.

USBH_CDC_CommFeatureClr()

Description

Clears the CDC function communication feature.

Files

`usbh_cdc.h/usbh_cdc.c`

Prototype

```
void USBH_CDC_CommFeatureClr (USBH_CDC_FNCT_HANDLE    cdc_fnct_handle,
                              CPU_INT08U              feature,
                              CPU_INT32U              timeout,
                              RTOS_ERR               *p_err)
```

Arguments

`cdc_fnct_handle`

Handle on CDC function.

`feature`

Feature to clear, which includes:

- USBH_CDC_COMM_FEATURE_ABSTRACT_STATE
- USBH_CDC_COMM_FEATURE_COUNTRY_SETTING

`timeout`

Timeout in milliseconds.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_ABORT
- RTOS_ERR_BLK_ALLOC_CALLBACK
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_IS_OWNER
- RTOS_ERR_INVALID_STATE
- RTOS_ERR_URB_ALLOC
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_SEG_OVF
- RTOS_ERR_EP_INVALID
- RTOS_ERR_INVALID_ARG
- RTOS_ERR_TIMEOUT
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_TX
- RTOS_ERR_WOULD_OVF

Returned Value

None.

Notes / Warnings

1. For more information on `ClearCommFeature` request, see 'Communication Class Subclass Specification for PSTN Devices, version 1.2, Section 6.3.3'.

USBH_CDC_LineCodingSet()

Description

Sets the CDC function line coding.

Files

`usbh_cdc.h/usbh_cdc.c`

Prototype

```
void USBH_CDC_LineCodingSet (USBH_CDC_FNCT_HANDLE cdc_fnct_handle,
                             USBH_CDC_LINECODING   *p_line_coding,
                             CPU_INT32U             timeout,
                             RTOS_ERR              *p_err)
```

Arguments

`cdc_fnct_handle`

Handle on the CDC function.

`p_line_coding`

Pointer to the structure that contains line coding to set.

`timeout`

Timeout in milliseconds.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_ABORT
- RTOS_ERR_BLK_ALLOC_CALLBACK
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_IS_OWNER
- RTOS_ERR_INVALID_STATE
- RTOS_ERR_URB_ALLOC
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_SEG_OVF
- RTOS_ERR_EP_INVALID
- RTOS_ERR_INVALID_ARG
- RTOS_ERR_TIMEOUT
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_TX
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_WOULD_OVF

Returned Value

None.

Notes / Warnings

1. For more information on `SetLineCoding` command, see 'Communication Class Subclass Specification for PSTN Devices, version 1.2, Section 6.3.10'.

USBH_CDC_LineCodingGet()

Description

Gets the CDC function line coding.

Files

usbh_cdc.h/usbh_cdc.c

Prototype

```
void USBH_CDC_LineCodingGet (USBH_CDC_FNCT_HANDLE    cdc_fnct_handle,
                             USBH_CDC_LINECODING    *p_line_coding,
                             CPU_INT32U              timeout,
                             RTOS_ERR               *p_err)
```

Arguments

cdc_fnct_handle

Handle on the CDC function.

p_line_coding

Pointer to the structure that will receive line coding.

timeout

Timeout in milliseconds.

p_err

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_ABORT
- RTOS_ERR_BLK_ALLOC_CALLBACK
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_IS_OWNER
- RTOS_ERR_INVALID_STATE
- RTOS_ERR_URB_ALLOC
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_SEG_OVF
- RTOS_ERR_EP_INVALID
- RTOS_ERR_INVALID_ARG
- RTOS_ERR_TIMEOUT
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_TX
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_WOULD_OVF

Returned Value

None.

Notes / Warnings

1. For more information on `GetLineCoding` command, see 'Communication Class Subclass Specification for PSTN Devices, version 1.2, Section 6.3.11'.

USBH_CDC_CtrlLineStateSet()

Description

Sets the CDC function control line state.

Files

usbh_cdc.h/usbh_cdc.c

Prototype

```
void USBH_CDC_CtrlLineStateSet (USBH_CDC_FNCT_HANDLE    cdc_fnct_handle,
                                CPU_INT16U              ctrl_signal,
                                CPU_INT32U              timeout,
                                RTOS_ERR               *p_err)
```

Arguments

cdc_fnct_handle

Handle on the CDC function.

ctrl_signal

Control signal, which includes:

- USBH_CDC_CTRL_LINE_STATE_CARRIER
- USBH_CDC_CTRL_LINE_STATE_DTE

timeout

Timeout in milliseconds.

p_err

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_ABORT
- RTOS_ERR_BLK_ALLOC_CALLBACK
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_IS_OWNER
- RTOS_ERR_INVALID_STATE
- RTOS_ERR_URB_ALLOC
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_SEG_OVF
- RTOS_ERR_EP_INVALID
- RTOS_ERR_INVALID_ARG
- RTOS_ERR_TIMEOUT
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_TX
- RTOS_ERR_WOULD_OVF

Returned Value

None.

Notes / Warnings

1. For more information on SetControlLineState request, see 'Communication Class Subclass Specification for PSTN Devices, version 1.2, Section 6.3.12'.

USBH_CDC_BrkSend()

Description

Sends the break signal to CDC function.

Files

usbh_cdc.h/usbh_cdc.c

Prototype

```
void USBH_CDC_BrkSend (USBH_CDC_FNCT_HANDLE   cdc_fnct_handle,
                       CPU_INT16U             dur,
                       CPU_INT32U             timeout,
                       RTOS_ERR              *p_err)
```

Arguments

cdc_fnct_handle

Handle on the CDC function.

dur

Duration of break.

timeout

Timeout in milliseconds.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_URB_ALLOC`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_TX`
- `RTOS_ERR_WOULD_OVF`

Returned Value

None.

Notes / Warnings

1. For more information on `endBreak` request, see 'Communication Class Subclass Specification for PSTN Devices, version 1.2, Section 6.3.13'.

USB Host ACM API

USB Host ACM API

- USBH_ACM_ConfigureMemSeg()
- USBH_ACM_ConfigureOptimizeSpdCfg()
- USBH_ACM_ConfigureInitAllocCfg()
- USBH_ACM_Init()
- USBH_ACM_StdReqTimeoutSet()
- USBH_ACM_CapabilitiesGet()
- USBH_ACM_EncapsulatedCmdTx()
- USBH_ACM_EncapsulatedCmdRx()
- USBH_ACM_CommFeatureSet()
- USBH_ACM_CommFeatureGet()
- USBH_ACM_CommFeatureClr()
- USBH_ACM_LineCodingSet()
- USBH_ACM_LineCodingGet()
- USBH_ACM_CtrlLineStateSet()
- USBH_ACM_BrkSend()
- USBH_ACM_RxAsync()
- USBH_ACM_TxAsync()

USBH_ACM_ConfigureMemSeg()

Description

Configures the memory segment to use when allocating control data and buffers.

Files

`usbh_acm.h/usbh_acm.c`

Prototype

```
void USBH_ACM_ConfigureMemSeg (MEM_SEG *p_mem_seg)
```

Arguments

`p_mem_seg`

- Pointer to memory segment to use when allocating control data. Can be the same segment used for `p_mem_seg_buf`.
- `DEF_NULL` means general purpose heap segment.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the ACM class is initialized via the `USBH_ACM_Init()` function.

USBH_ACM_ConfigureOptimizeSpdCfg()

Description

Sets the configurations required when optimize speed mode is enabled.

Files

`usbh_acm.h/usbh_acm.c`

Prototype

```
void USBH_ACM_ConfigureOptimizeSpdCfg(const USBH_ACM_CFG_OPTIMIZE_SPD *p_optimize_spd_cfg)
```

Arguments

`p_optimize_spd_cfg`

Pointer to the structure containing the configurations for the optimize speed mode.

Returned Value

None.

Notes / Warnings

1. This function MUST be called before the ACM class is initialized via the `USBH_ACM_Init()` function.
2. This function MUST be called when the `USBH_CFG_OPTIMIZE_SPD_EN` configuration is set to `DEF_ENABLED`.

USBH_ACM_ConfigureInitAllocCfg()

Description

Sets the configurations required when allocation at initialization mode is enabled.

Files

`usbh_acm.h/usbh_acm.c`

Prototype

```
void USBH_ACM_ConfigureInitAllocCfg (const USBH_ACM_CFG_INIT_ALLOC *p_init_alloc_cfg)
```

Arguments

`p_init_alloc_cfg`

Pointer to the structure containing the configurations for the allocation at initialization mode.

Returned Value

None.

Notes / Warnings

1. This function MUST be called before the ACM class is initialized via the `USBH_ACM_Init()` function.
2. This function MUST be called when the `USBH_CFG_INIT_ALLOC_EN` configuration is set to `DEF_ENABLED`.

USBH_ACM_Init()

Description

Initializes the Communication Device Class (CDC) Abstract Control Model (ACM) subclass driver.

Files

`usbh_acm.h/usbh_acm.c`

Prototype

```
void USBH_ACM_Init (USBH_ACM_APP_FNCTS  *p_acm_app_fncts,
                    RTOS_ERR            *p_err)
```

Arguments

`p_acm_app_fncts`

Pointer to the ACM application callback functions. Content MUST be persistent.

`p_err`

Pointer to the variable that will receive the return error code from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`

Returned Value

None.

Notes / Warnings

None.

USBH_ACM_StdReqTimeoutSet()

Description

Assigns a new timeout delay for the ACM standard requests.

Files

`usbh_acm.h/usbh_acm.c`

Prototype

```
void  USBH_ACM_StdReqTimeoutSet (CPU_INT32U   std_req_timeout_ms,
                                 RTOS_ERR    *p_err)
```

Arguments

`std_req_timeout_ms`

New timeout, in milliseconds.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

None.

USBH_ACM_CapabilitiesGet()

Description

Gets the CDC ACM function capabilities.

Files

`usbh_acm.h/usbh_acm.c`

Prototype

```
CPU_INT08U USBH_ACM_CapabilitiesGet (USBH_ACM_FNCT_HANDLE    acm_fnct_handle,
                                     RTOS_ERR                *p_err)
```

Arguments

`acm_fnct_handle`

Handle on the CDC ACM function.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_NOT_AVAIL`

Returned Value

Bitmap that represents the following CDC ACM features:

Notes / Warnings

1. This function returns a bitmap representing the features supported by the CDC ACM function. This information comes from the Call Management and Abstract Control Model functional descriptors.

USBH_ACM_EncapsulatedCmdTx()

Description

Sends the CDC ACM encapsulated command.

Files

`usbh_acm.h/usbh_acm.c`

Prototype

```
CPU_INT16U USBH_ACM_EncapsulatedCmdTx (USBH_ACM_FNCT_HANDLE    acm_fnct_handle,
                                       CPU_INT08U              *p_buf,
                                       CPU_INT16U               buf_len,
                                       CPU_INT32U               timeout,
                                       RTOS_ERR                *p_err)
```

Arguments

`acm_fnct_handle`

Handle on the CDC ACM function.

`p_buf`

Pointer to the buffer that contains command.

`buf_len`

Buffer length in octets.

`timeout`

Timeout in milliseconds.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_URB_ALLOC`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_TX`
- `RTOS_ERR_WOULD_OVF`

Returned Value

Number of bytes transferred.

Notes / Warnings

1. For more information on `SendEncapsulatedCommand`, see 'USB Class Definitions for Communication Devices Specification", version 1.2, Section 6.2.1'.

USBH_ACM_EncapsulatedCmdRx()

Description

Receives the CDC ACM encapsulated command.

Files

`usbh_acm.h/usbh_acm.c`

Prototype

```
CPU_INT16U USBH_ACM_EncapsulatedRespRx (USBH_ACM_FNCT_HANDLE acm_fnct_handle,                              CPU_INT08U         *p_buf,
CPU_INT16U          buf_len,                                  CPU_INT32U          timeout,                             RTOS_ERR           *p_err)
```

Arguments

`acm_fnct_handle`

Handle on the CDC ACM function.

`p_buf`

Pointer to the buffer that will receive data.

`buf_len`

Buffer length in octets.

`timeout`

Timeout in milliseconds.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_URB_ALLOC`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_TX`
- `RTOS_ERR_WOULD_OVF`

Returned Value

Number of octets received.

Notes / Warnings

1. For more information on `GetEncapsulatedCommand`, see 'USB Class Definitions for Communication Devices Specification", version 1.2, Section 6.2.2'.

USBH_ACM_CommFeatureSet()

Description

Configures the CDC ACM communication feature.

Files

`usbh_acm.h/usbh_acm.c`

Prototype

```
void USBH_ACM_CommFeatureSet (USBH_ACM_FNCT_HANDLE    acm_fnct_handle,
                              CPU_INT08U              feature,
                              CPU_INT16U              data,
                              CPU_INT32U              timeout,
                              RTOS_ERR               *p_err)
```

Arguments

`acm_fnct_handle`

Handle on the CDC ACM function.

`feature`

Feature to configure using the following:

- `USBH_CDC_COMM_FEATURE_ABSTRACT_STATE`
- `USBH_CDC_COMM_FEATURE_COUNTRY_SETTING`

`data`

Data of the communication feature request.

`timeout`

Timeout in milliseconds.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_URB_ALLOC`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_NOT_SUPPORTED`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_TX`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`

Returned Value

None.

Notes / Warnings

1. For more information on `SetCommFeature` request, see 'Communication Class Subclass Specification for PSTN Devices, version 1.2, Section 6.3.1'.

USBH_ACM_CommFeatureGet()

Description

Gets the CDC ACM function line coding.

Files

`usbh_acm.h/usbh_acm.c`

Prototype

```
CPU_INT16U USBH_ACM_CommFeatureGet (USBH_ACM_FNCT_HANDLE    acm_fnct_handle,
                                    CPU_INT08U              feature,
                                    CPU_INT32U              timeout,
                                    RTOS_ERR               *p_err)
```

Arguments

`acm_fnct_handle`

Handle on the CDC ACM function.

`feature`

Feature to Get using the following:.

- `USBH_CDC_COMM_FEATURE_ABSTRACT_STATE`
- `USBH_CDC_COMM_FEATURE_COUNTRY_SETTING`

`timeout`

Timeout in milliseconds.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_URB_ALLOC`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_NOT_SUPPORTED`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_TX`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`

Returned Value

Data of the Get communication feature request.

Notes / Warnings

1. For more information on `GetCommFeature` request, see 'Communication Class Subclass Specification for PSTN Devices, version 1.2, Section 6.3.2'.

USBH_ACM_CommFeatureClr()

Description

Clears the CDC ACM function communication feature.

Files

`usbh_acm.h/usbh_acm.c`

Prototype

```
void USBH_ACM_CommFeatureClr (USBH_ACM_FNCT_HANDLE   acm_fnct_handle,
                              CPU_INT08U             feature,
                              CPU_INT32U             timeout,
                              RTOS_ERR              *p_err)
```

Arguments

`acm_fnct_handle`

Handle on the CDC ACM function.

`feature`

Feature to clear including the following:

- `USBH_CDC_COMM_FEATURE_ABSTRACT_STATE`
- `USBH_CDC_COMM_FEATURE_COUNTRY_SETTING`

`timeout`

Timeout in milliseconds.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_URB_ALLOC`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_NOT_SUPPORTED`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_TX`
- `RTOS_ERR_WOULD_OVF`

Returned Value

None.

Notes / Warnings

1. For more information on ClearCommFeature request, see 'Communication Class Subclass Specification for PSTN Devices, version 1.2, Section 6.3.3'.

USBH_ACM_LineCodingSet()

Description

Sets the CDC ACM function line coding.

Files

`usbh_acm.h/usbh_acm.c`

Prototype

```
void USBH_ACM_LineCodingSet (USBH_ACM_FNCT_HANDLE    acm_fnct_handle,
                             USBH_CDC_LINECODING    *p_line_coding,
                             CPU_INT32U              timeout,
                             RTOS_ERR               *p_err)
```

Arguments

`acm_fnct_handle`

Handle on the CDC ACM function.

`p_line_coding`

Pointer to the structure that contains line coding to set.

`timeout`

Timeout in milliseconds.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_URB_ALLOC`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_NOT_SUPPORTED`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_TX`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`

Returned Value

None.

Notes / Warnings

1. For more information on `SetLineCoding` command, see 'Communication Class Subclass Specification for PSTN Devices, version 1.2, Section 6.3.10'.

USBH_ACM_LineCodingGet()

Description

Gets the CDC ACM function line coding.

Files

`usbh_acm.h/usbh_acm.c`

Prototype

```
void USBH_ACM_LineCodingGet (USBH_ACM_FNCT_HANDLE    acm_fnct_handle,
                             USBH_CDC_LINECODING    *p_line_coding,
                             CPU_INT32U              timeout,
                             RTOS_ERR               *p_err)
```

Arguments

`acm_fnct_handle`

Handle on the CDC ACM function.

`p_line_coding`

Pointer to the structure that will receive line coding.

`timeout`

Timeout in milliseconds.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_URB_ALLOC`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_NOT_SUPPORTED`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_TX`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`

Returned Value

None.

Notes / Warnings

1. For more information on `GetLineCoding` command, see 'Communication Class Subclass Specification for PSTN Devices, version 1.2, Section 6.3.11'.

USBH_ACM_CtrlLineStateSet()

Description

Sets the CDC ACM function control line state.

Files

`usbh_acm.h/usbh_acm.c`

Prototype

```
void  USBH_ACM_CtrlLineStateSet (USBH_ACM_FNCT_HANDLE    acm_fnct_handle,
                                 CPU_INT16U              ctrl_signal,
                                 CPU_INT32U              timeout,
                                 RTOS_ERR               *p_err)
```

Arguments

`acm_fnct_handle`

Handle on the CDC ACM function.

`ctrl_signal`

Configures the control signal using the following:

- `USBH_CDC_CTRL_LINE_STATE_CARRIER`
- `USBH_CDC_CTRL_LINE_STATE_DTE`

`timeout`

Timeout in milliseconds.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_STATE`
- `RTOS_ERR_URB_ALLOC`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_EP_INVALID`
- `RTOS_ERR_NOT_SUPPORTED`
- `RTOS_ERR_INVALID_ARG`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_TX`

- RTOS_ERR_WOULD_OVF

Returned Value

None.

Notes / Warnings

1. For more information on SetControlLineState request, see 'Communication Class Subclass Specification for PSTN Devices, version 1.2, Section 6.3.12'.

USBH_ACM_BrkSend()

Description

Sends the break signal to CDC ACM function.

Files

usbh_acm.h/usbh_acm.c

Prototype

```
void  USBH_ACM_BrkSend (USBH_ACM_FNCT_HANDLE    acm_fnct_handle,
                        CPU_INT16U              dur,
                        CPU_INT32U              timeout,
                        RTOS_ERR               *p_err)
```

Arguments

acm_fnct_handle

Handle on the CDC ACM function.

dur

Duration of break.

timeout

Timeout in milliseconds.

p_err

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_ABORT
- RTOS_ERR_BLK_ALLOC_CALLBACK
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_IS_OWNER
- RTOS_ERR_INVALID_STATE
- RTOS_ERR_URB_ALLOC
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_SEG_OVF
- RTOS_ERR_EP_INVALID
- RTOS_ERR_NOT_SUPPORTED
- RTOS_ERR_INVALID_ARG
- RTOS_ERR_TIMEOUT
- RTOS_ERR_OS_OBJ_DEL

- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_TX
- RTOS_ERR_WOULD_OVF

Returned Value

None.

Notes / Warnings

1. For more information on SendBreak request, see 'Communication Class Subclass Specification for PSTN Devices, version 1.2, Section 6.3.13'.

USBH_ACM_RxAsync()

Description

Receives the data from CDC ACM device. This function is asynchronous.

Files

usbh_acm.h/usbh_acm.c

Prototype

```
void  USBH_ACM_RxAsync (USBH_ACM_FNCT_HANDLE    acm_fnct_handle,
                        CPU_INT08U             *p_buf,
                        CPU_INT32U              buf_len,
                        USBH_ACM_ASYNC_FNCT     async_fnct,
                        void                   *p_async_arg,
                        RTOS_ERR               *p_err)
```

Arguments

acm_fnct_handle

Handle on the CDC ACM function.

p_buf

Pointer to the destination buffer to receive data.

buf_len

Buffer length in octets.

async_fnct

Function that will be invoked upon completion of receive operation.

p_async_arg

Pointer to the argument that will be passed as parameter of 'async_fnct'.

p_err

Pointer to the variable that will receive one of these return error codes from this function ::

- RTOS_ERR_NONE
- RTOS_ERR_ABORT
- RTOS_ERR_BLK_ALLOC_CALLBACK
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_INVALID_STATE
- RTOS_ERR_INVALID_HANDLE

- RTOS_ERR_URB_ALLOC
- RTOS_ERR_EP_INVALID
- RTOS_ERR_SEG_OVF
- RTOS_ERR_POOL_FULL
- RTOS_ERR_INVALID_ARG
- RTOS_ERR_TIMEOUT
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_NOT_AVAIL

Returned Value

None.

Notes / Warnings

None.

USBH_ACM_TxAsync()

Description

Transmits the data to CDC ACM device. This function is asynchronous.

Files

usbh_acm.h/usbh_acm.c

Prototype

```
void USBH_ACM_TxAsync (USBH_ACM_FNCT_HANDLE   acm_fnct_handle,
                       CPU_INT08U            *p_buf,
                       CPU_INT32U             buf_len,
                       USBH_ACM_ASYNC_FNCT    async_fnct,
                       void                  *p_async_arg,
                       RTOS_ERR              *p_err)
```

Arguments

acm_fnct_handle

Handle on the CDC ACM function.

p_buf

Pointer to the buffer of data that will be transmitted.

buf_len

Buffer length in octets.

async_fnct

Function that will be invoked upon completion of transmit operation.

p_async_arg

Pointer to the argument that will be passed as parameter of ' async_fnct '.

p_err

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_ABORT

- RTOS_ERR_BLK_ALLOC_CALLBACK
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_INVALID_STATE
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_URB_ALLOC
- RTOS_ERR_EP_INVALID
- RTOS_ERR_SEG_OVF
- RTOS_ERR_POOL_FULL
- RTOS_ERR_INVALID_ARG
- RTOS_ERR_TIMEOUT
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_NOT_AVAIL

Returned Value

None.

Notes / Warnings

None.

USB Host HID API

USB Host HID API

- USBH_HID_ConfigureBufAlignOctets()
- USBH_HID_ConfigureRxBuf()
- USBH_HID_ConfigureUsageMaxNbrPerItem()
- USBH_HID_ConfigureReportDescMaxLen()
- USBH_HID_ConfigureOptimizeSpdCfg()
- USBH_HID_ConfigureInitAllocCfg()
- USBH_HID_ConfigureMemSeg()
- USBH_HID_Init()
- USBH_HID_StdReqTimeoutSet()
- USBH_HID_UsageGet()
- USBH_HID_IsBootCapable()
- USBH_HID_ReportTx()
- USBH_HID_ProtocolSet()
- USBH_HID_ProtocolGet()
- USBH_HID_IdleSet()
- USBH_HID_IdleGet()

USBH_HID_ConfigureBufAlignOctets()

Description

Configures the alignment of the internal buffers.

Files

`usbh_hid.h/usbh_hid.c`

Prototype

```
void  USBH_HID_ConfigureBufAlignOctets (CPU_SIZE_T  buf_align_octets)
```

Arguments

`buf_align_octets`

Buffer alignment, in octets.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the HID class is initialized via the `USBH_HID_Init()` function.

USBH_HID_ConfigureRxBuf()

Description

Configures the receive buffers.

Files

`usbh_hid.h/usbh_hid.c`

Prototype

```
void  USBH_HID_ConfigureRxBuf (CPU_INT08U  rx_buf_qty,
                               CPU_INT08U  rx_buf_len)
```

Arguments

`rx_buf_qty`

Quantity of buffers available for report reception.

`rx_buf_len`

Len of buffers, in octets, used for report reception.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the HID class is initialized via the `USBH_HID_Init()` function.

USBH_HID_ConfigureUsageMaxNbrPerItem()

Description

Configures the maximum quantity of usages per item.

Files

`usbh_hid.h/usbh_hid.c`

Prototype

```
void  USBH_HID_ConfigureUsageMaxNbrPerItem(CPU_INT08U  usage_max_nbr_per_item)
```

Arguments

`usage_max_nbr_per_item`

Maximum number of usages associated with a given item.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the HID class is initialized via the `USBH_HID_Init()` function.

USBH_HID_ConfigureReportDescMaxLen()

Description

Configures the report descriptor buffer.

Files

`usbh_hid.h/usbh_hid.c`

Prototype

```
void USBH_HID_ConfigureReportDescMaxLen (CPU_INT16U report_desc_max_len)
```

Arguments

`report_desc_max_len`

Maximum length, in octets, of report desc.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the HID class is initialized via the `USBH_HID_Init()` function.

USBH_HID_ConfigureOptimizeSpdCfg()

Description

Sets the configurations required when optimize speed mode is enabled.

Files

`usbh_hid.h/usbh_hid.c`

Prototype

```
void USBH_HID_ConfigureOptimizeSpdCfg(const USBH_HID_CFG_OPTIMIZE_SPD *p_optimize_spd_cfg)
```

Arguments

`p_optimize_spd_cfg`

Pointer to the structure containing the configurations for the optimize speed mode.

Returned Value

None.

Notes / Warnings

1. This function MUST be called before the HID class is initialized via the `USBH_HID_Init()` function.
2. This function MUST be called when the `USBH_CFG_OPTIMIZE_SPD_EN` configuration is set to `DEF_ENABLED` .

USBH_HID_ConfigureInitAllocCfg()

Description

Sets the configurations required when allocation at initialization mode is enabled.

Files

`usbh_hid.h/usbh_hid.c`

Prototype

```
void USBH_HID_ConfigureInitAllocCfg (const USBH_HID_CFG_INIT_ALLOC *p_init_alloc_cfg)
```

Arguments

`p_init_alloc_cfg`

Pointer to the structure containing the configurations for the allocation at initialization mode.

Returned Value

None.

Notes / Warnings

1. This function MUST be called before the HID class is initialized via the `USBH_HID_Init()` function.
2. This function MUST be called when the `USBH_CFG_INIT_ALLOC_EN` configuration is set to `DEF_ENABLED`.

USBH_HID_ConfigureMemSeg()

Description

Configures the memory segment to use when allocating control data and buffers.

Files

`usbh_hid.h/usbh_hid.c`

Prototype

```
void USBH_HID_ConfigureMemSeg (MEM_SEG *p_mem_seg,
                               MEM_SEG *p_mem_seg_buf)
```

Arguments

`p_mem_seg`

- Pointer to memory segment to use when allocating control data. Can be the same segment used for `p_mem_seg_buf`.
- `DEF_NULL` means general purpose heap segment.

`p_mem_seg_buf`

- Pointer to memory segment to use when allocating data buffers. Can be the same segment used for `p_mem_seg`.
- `DEF_NULL` means general purpose heap segment.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the HID class is initialized via the `USBH_HID_Init()` function.

USBH_HID_Init()

Description

Initializes the HID class.

Files

`usbh_hid.h/usbh_hid.c`

Prototype

```
void USBH_HID_Init (USBH_HID_APP_FNCTS *p_hid_app_fncts,            RTOS_ERR        *p_err)
```

Arguments

`p_hid_app_fncts`

Pointer to the HID application callback functions structure. Content MUST be persistent.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_INVALID_CFG`

Returned Value

None.

Notes / Warnings

None.

USBH_HID_StdReqTimeoutSet()

Description

Assigns a new timeout delay for the HID standard requests.

Files

`usbh_hid.h/usbh_hid.c`

Prototype

```
void USBH_HID_StdReqTimeoutSet (CPU_INT32U   std_req_timeout_ms,
                                RTOS_ERR    *p_err)
```

Arguments

`std_req_timeout_ms`

New timeout, in milliseconds.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

None.

USBH_HID_UsageGet()

Description

Gets the global usage associated to HID function.

Files

`usbh_hid.h/usbh_hid.c`

Prototype

```
CPU_INT32U  USBH_HID_UsageGet (USBH_HID_FNCT_HANDLE    hid_fnct_handle,
                               RTOS_ERR               *p_err)
```

Arguments

`hid_fnct_handle`

Handle to the HID function.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_NOT_AVAIL`

Returned Value

Usage.

Notes / Warnings

None.

USBH_HID_IsBootCapable()

Description

Tests whether HID interface belongs to boot subclass.

Files

`usbh_hid.h/usbh_hid.c`

Prototype

```
CPU_BOOLEAN  USBH_HID_IsBootCapable (USBH_HID_FNCT_HANDLE    hid_fnct_handle,
                                     RTOS_ERR               *p_err)
```

Arguments

`hid_fnct_handle`

Handle to the HID function.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ABORT`

- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_TIMEOUT
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_INVALID_STATE

Returned Value

- DEF_YES , if the function belongs to the boot subclass,
- DEF_NO , if the function does not belong to the boot subclass.

Notes / Warnings

None.

USBH_HID_ReportTx()

Description

Sends the report to the device.

Files

usbh_hid.h/usbh_hid.c

Prototype

```
CPU_INT16U  USBH_HID_ReportTx (USBH_HID_FNCT_HANDLE    hid_fnct_handle,
                               CPU_INT08U              report_id,
                               void                   *p_buf,
                               CPU _INT16U             buf_len,
                               CPU _INT32U             timeout_ms,
                               RTOS_ERR               *p_err)
```

Arguments

hid_fnct_handle

Handle to the HID function.

report_id

ID of the Report.

p_buf

Pointer to the the buffer that contains the report.

buf_len

Buffer length, in octets.

timeout_ms

Timeout, in milliseconds.

p_err

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_ABORT
- RTOS_ERR_BLK_ALLOC_CALLBACK
- RTOS_ERR_WOULD_BLOCK

- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_IS_OWNER
- RTOS_ERR_INVALID_STATE
- RTOS_ERR_URB_ALLOC
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_EP_INVALID
- RTOS_ERR_SEG_OVF
- RTOS_ERR_OS_ILLEGAL_RUN_TIME
- RTOS_ERR_TIMEOUT
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_NULL_PTR
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_TX
- RTOS_ERR_NOT_AVAIL

Returned Value

Number of octets sent.

Notes / Warnings

1. Do not add the report id to `p_buf`, it will be added automatically.

USBH_HID_ProtocolSet()

Description

Sets the protocol (boot/report descriptor) of HID function.

Files

`usbh_hid.h/usbh_hid.c`

Prototype

```
void  USBH_HID_ProtocolSet (USBH_HID_FNCT_HANDLE    hid_fnct_handle,
                            CPU_INT16U              protocol,
                            RTOS_ERR               *p_err)
```

Arguments

`hid_fnct_handle`

Handle to the HID function.

`protocol`

Protocol to set.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_ABORT
- RTOS_ERR_BLK_ALLOC_CALLBACK
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_IS_OWNER

- RTOS_ERR_INVALID_STATE
- RTOS_ERR_URB_ALLOC
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_SEG_OVF
- RTOS_ERR_EP_INVALID
- RTOS_ERR_TIMEOUT
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_TX
- RTOS_ERR_WOULD_OVF

Returned Value

None.

Notes / Warnings

None.

USBH_HID_ProtocolGet()

Description

Gets the protocol (boot/report) of the HID function.

Files

usbh_hid.h/usbh_hid.c

Prototype

```
void  USBH_HID_ProtocolGet (USBH_HID_FNCT_HANDLE    hid_fnct_handle,
                            CPU_INT16U             *p_protocol,
                            RTOS_ERR               *p_err)
```

Arguments

hid_fnct_handle

Handle the HID function.

p_protocol

Variable that receives protocol of the device.

p_err

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_ABORT
- RTOS_ERR_BLK_ALLOC_CALLBACK
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_IS_OWNER
- RTOS_ERR_INVALID_STATE
- RTOS_ERR_URB_ALLOC
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_SEG_OVF
- RTOS_ERR_EP_INVALID

- RTOS_ERR_TIMEOUT
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_TX
- RTOS_ERR_WOULD_OVF

Returned Value

None.

Notes / Warnings

None.

USBH_HID_IdleSet()

Description

Sets the idle duration for the given report ID.

Files

usbh_hid.h/usbh_hid.c

Prototype

```
void  USBH_HID_IdleSet (USBH_HID_FNCT_HANDLE    hid_fnct_handle,
                        CPU_INT08U              report_id,
                        CPU_INT32U              dur,
                        RTOS_ERR               *p_err)
```

Arguments

hid_fnct_handle

Handle to the HID function.

report_id

ID of the Report.

dur

Idle duration, in milliseconds.

p_err

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_ABORT
- RTOS_ERR_BLK_ALLOC_CALLBACK
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_IS_OWNER
- RTOS_ERR_INVALID_STATE
- RTOS_ERR_URB_ALLOC
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_SEG_OVF
- RTOS_ERR_EP_INVALID
- RTOS_ERR_TIMEOUT

- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_TX
- RTOS_ERR_WOULD_OVF

Returned Value

None.

Notes / Warnings

None.

USBH_HID_IdleGet()

Description

Gets the idle duration for the given report ID.

Files

usbh_hid.h/usbh_hid.c

Prototype

```
CPU_INT32U  USBH_HID_IdleGet (USBH_HID_FNCT_HANDLE    hid_fnct_handle,
                              CPU_INT08U              report_id,
                              RTOS_ERR               *p_err)
```

Arguments

hid_fnct_handle

Handle to the HID function.

report_id

ID of the Report.

p_err

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_ABORT
- RTOS_ERR_BLK_ALLOC_CALLBACK
- RTOS_ERR_POOL_EMPTY
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_IS_OWNER
- RTOS_ERR_INVALID_STATE
- RTOS_ERR_URB_ALLOC
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_SEG_OVF
- RTOS_ERR_EP_INVALID
- RTOS_ERR_TIMEOUT
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_TX
- RTOS_ERR_WOULD_OVF

Returned Value

Idle duration in milliseconds.

Notes / Warnings

None.

USB Host MSC API

USB Host MSC API

- USBH_MSC_ConfigureBufAlignOctets()
- USBH_MSC_ConfigureOptimizeSpdCfg()
- USBH_MSC_ConfigureInitAllocCfg()
- USBH_MSC_ConfigureMemSeg()
- USBH_MSC_Init()
- USBH_MSC_StdReqTimeoutSet()

USBH_MSC_ConfigureBufAlignOctets()

Description

Configures the alignment of the internal buffers.

Files

`usbh_msc.h/usbh_msc.c`

Prototype

```
void USBH_MSC_ConfigureBufAlignOctets (CPU_SIZE_T buf_align_octets)
```

Arguments

`buf_align_octets`

Buffer alignment, in octets.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the MSC class is initialized via the `USBH_MSC_Init()` function.

USBH_MSC_ConfigureOptimizeSpdCfg()

Description

Sets the configurations required when optimize speed mode is enabled.

Files

`usbh_msc.h/usbh_msc.c`

Prototype

```
void USBH_MSC_ConfigureOptimizeSpdCfg(const USBH_MSC_CFG_OPTIMIZE_SPD *p_optimize_spd_cfg)
```

Arguments

`p_optimize_spd_cfg`

Pointer to the structure containing the configurations for the optimize speed mode.

Returned Value

None.

Notes / Warnings

1. This function MUST be called before the MSC class is initialized via the `USBH_MSC_Init()` function.
2. This function MUST be called when the `USBH_CFG_OPTIMIZE_SPD_EN` configuration is set to `DEF_ENABLED`.

USBH_MSC_ConfigureInitAllocCfg()

Description

Sets the configurations required when allocation at initialization mode is enabled.

Files

`usbh_msc.h/usbh_msc.c`

Prototype

```
void USBH_MSC_ConfigureInitAllocCfg (const USBH_MSC_CFG_INIT_ALLOC *p_init_alloc_cfg)
```

Arguments

`p_init_alloc_cfg`

Pointer to the structure containing the configurations for the allocation at initialization mode.

Returned Value

None.

Notes / Warnings

1. This function MUST be called before the MSC class is initialized via the `USBH_MSC_Init()` function.
2. This function MUST be called when the `USBH_CFG_INIT_ALLOC_EN` configuration is set to `DEF_ENABLED`.

USBH_MSC_ConfigureMemSeg()

Description

Configures the memory segment to use when allocating control data and buffers.

Files

`usbh_msc.h/usbh_msc.c`

Prototype

```
void USBH_MSC_ConfigureMemSeg (MEM_SEG *p_mem_seg,
                               MEM_SEG *p_mem_seg_buf)
```

Arguments

`p_mem_seg`

- Pointer to memory segment to use when allocating control data. Can be the same segment used for `p_mem_seg_buf`.
- `DEF_NULL` means general purpose heap segment.

`p_mem_seg_buf`

- Pointer to memory segment to use when allocating data buffers. Can be the same segment used for `p_mem_seg`.
- `DEF_NULL` means general purpose heap segment.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the MSC class is initialized via the `USBH_MSC_Init()` function.

USBH_MSC_Init()

Description

Initializes The Mass Storage Class (MSC) driver.

Files

`usbh_msc.h/usbh_msc.c`

Prototype

```
void USBH_MSC_Init (USBH_MSC_CMD_BLK_FNCTS  *p_cmd_blk_fncts,
                    RTOS_ERR                *p_err)
```

Arguments

`p_cmd_blk_fncts`

Pointer to the Command Block layer API structure. Content MUST be persistent.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`

Returned Value

None.

Notes / Warnings

1. MSC layer queries the Command Block layer for the maximum response buffer length that may be needed during the Data IN phase processing of the Bulk-Only Transport protocol.

USBH_MSC_StdReqTimeoutSet()

Description

Assigns a new timeout delay for the MSC standard requests.

Files

`usbh_msc.h/usbh_msc.c`

Prototype

```
void  USBH_MSC_StdReqTimeoutSet (CPU_INT32U   std_req_timeout_ms,
                                 RTOS_ERR    *p_err)
```

Arguments

`std_req_timeout_ms`

New timeout, in milliseconds.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

None.

USB Host USB2SER API

USB Host USB2SER API

1. USBH_USB2SER_ConfigureBufAlignOctets()
2. USBH_USB2SER_ConfigureDrvEntryTbl()
3. USBH_USB2SER_ConfigureHS_En()
4. USBH_USB2SER_ConfigureRxBufQty()
5. USBH_USB2SER_ConfigureOptimizeSpdCfg()
6. USBH_USB2SER_ConfigureInitAllocCfg()
7. USBH_USB2SER_ConfigureMemSeg()
8. USBH_USB2SER_Init()
9. USBH_USB2SER_StdReqTimeoutSet()
10. USBH_USB2SER_DevHandleGet()
11. USBH_USB2SER_PortNbrGet()
12. USBH_USB2SER_Reset()
13. USBH_USB2SER_BaudRateSet()
14. USBH_USB2SER_BaudRateGet()
15. USBH_USB2SER_DataSet()
16. USBH_USB2SER_DataGet()
17. USBH_USB2SER_BreakSignalSet()
18. USBH_USB2SER_ModemDTR_Set()
19. USBH_USB2SER_ModemDTR_Get()
20. USBH_USB2SER_ModemRTS_Set()
21. USBH_USB2SER_ModemRTS_Get()
22. USBH_USB2SER_HW_FlowCtrlSet()
23. USBH_USB2SER_HW_FlowCtrlGet()
24. USBH_USB2SER_SW_FlowCtrlSet()
25. USBH_USB2SER_SW_FlowCtrlGet()
26. USBH_USB2SER_StatusGet()
27. USBH_USB2SER_TxAsync()

USBH_USB2SER_ConfigureBufAlignOctets()

Description

Configures the alignment of the internal buffers.

Files

`usbh_usb2ser.h/usbh_usb2ser.c`

Prototype

```
void  USBH_USB2SER_ConfigureBufAlignOctets (CPU_SIZE_T   buf_align_octets)
```

Arguments

`buf_align_octets`

Buffer alignment, in octets.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the `USB2SER` class is initialized via the `USBH_USB2SER_Init()` function.

USBH_USB2SER_ConfigureDrvEntryTbl()

Description

Configures the adapter drivers table.

Files

`usbh_usb2ser.h/usbh_usb2ser.c`

Prototype

```
void  USBH_USB2SER_ConfigureDrvEntryTbl (USBH_USB2SER_ADAPTER_DRV_ENTRY  *p_tbl)
```

Arguments

`p_tbl`

Pointer to table that contains the adapter driver entries. Table MUST be null-terminated. Content MUST be persistent.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the USB2SER class is initialized via the `USBH_USB2SER_Init()` function.

USBH_USB2SER_ConfigureHS_En()

Description

Configures the support for high-speed adapter devices.

Files

`usbh_usb2ser.h/usbh_usb2ser.c`

Prototype

```
void  USBH_USB2SER_ConfigureHS_En (CPU_BOOLEAN  hs_en)
```

Arguments

`hs_en`

- `DEF_ENABLED` , if support for high-speed adapter devices is enabled.
- `DEF_DISABLED` , if support for high-speed adapter devices is disabled.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.

2. This function MUST be called before the `USB2SER` class is initialized via the `USBH_USB2SER_Init()` function.

USBH_USB2SER_ConfigureRxBufQty()

Description

Configures the quantity of receive buffers per adapter.

Files

`usbh_usb2ser.h/usbh_usb2ser.c`

Prototype

```
void USBH_USB2SER_ConfigureRxBufQty (CPU_INT08U rx_buf_qty)
```

Arguments

`rx_buf_qty`

Quantity of receive buffers.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the `USB2SER` class is initialized via the `USBH_USB2SER_Init()` function.

USBH_USB2SER_ConfigureOptimizeSpdCfg()

Description

Sets the configurations required when optimize speed mode is enabled.

Files

`usbh_usb2ser.h/usbh_usb2ser.c`

Prototype

```
void USBH_USB2SER_ConfigureOptimizeSpdCfg(const USBH_USB2SER_CFG_OPTIMIZE_SPD *p_optimize_spd_cfg)
```

Arguments

`p_optimize_spd_cfg`

Pointer to the structure containing the configurations for the optimize speed mode.

Returned Value

None.

Notes / Warnings

1. This function MUST be called before the `USB2SER` class is initialized via the `USBH_USB2SER_Init()` function.
2. This function MUST be called when the `USBH_CFG_OPTIMIZE_SPD_EN` configuration is set to `DEF_ENABLED` .

USBH_USB2SER_ConfigureInitAllocCfg()

Description

Sets the configurations required when allocation at initialization mode is enabled.

Files

`usbh_usb2ser.h/usbh_usb2ser.c`

Prototype

```
void  USBH_USB2SER_ConfigureInitAllocCfg  (const  USBH_USB2SER_CFG_INIT_ALLOC  *p_init_alloc_cfg)
```

Arguments

`p_init_alloc_cfg`

Pointer to the structure containing the configurations for the allocation at initialization mode.

Returned Value

None.

Notes / Warnings

1. This function MUST be called before the USB2SER class is initialized via the `USBH_USB2SER_Init()` function.
2. This function MUST be called when the `USBH_CFG_INIT_ALLOC_EN` configuration is set to `DEF_ENABLED`.

USBH_USB2SER_ConfigureMemSeg()

Description

Configures the memory segment to use when allocating control data and buffers.

Files

`usbh_usb2ser.h/usbh_usb2ser.c`

Prototype

```
void  USBH_USB2SER_ConfigureMemSeg (MEM_SEG  *p_mem_seg,
                                    MEM_SEG  *p_mem_seg_buf)
```

Arguments

`p_mem_seg`

- Pointer to memory segment to use when allocating control data. Can be the same segment used for `p_mem_seg_buf`.
- `DEF_NULL` means general purpose heap segment.

`p_mem_seg_buf`

- Pointer to memory segment to use when allocating data buffers. Can be the same segment used for `p_mem_seg`.
- `DEF_NULL` means general purpose heap segment.

Returned Value

None.

Notes / Warnings

1. This function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the USB2SER class is initialized via the `USBH_USB2SER_Init()` function.

USBH_USB2SER_Init()

Description

Initializes the USB-to-serial Class.

Files

`usbh_usb2ser.h/usbh_usb2ser.c`

Prototype

```
void USBH_USB2SER_Init (USBH_USB2SER_APP_FNCTS  *p_app_fncts,
                    RTOS_ERR            *p_err)
```

Arguments

`p_app_fncts`

Pointer to the callback functions structure that will be used to notify application of change in USB to Serial device status. Content MUST be persistent.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_OS_ILLEGAL_RUN_TIME`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_NOT_AVAIL`

Returned Value

None.

Notes / Warnings

None.

USBH_USB2SER_StdReqTimeoutSet()

Description

Assigns a new timeout delay for the USB2SER standard requests.

Files

`usbh_usb2ser.h/usbh_usb2ser.c`

Prototype

```
void  USBH_USB2SER_StdReqTimeoutSet (CPU_INT32U   std_req_timeout_ms,
                              RTOS_ERR    *p_err)
```

Arguments

`std_req_timeout_ms`

New timeout, in milliseconds.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`

Returned Value

None.

Notes / Warnings

None.

USBH_USB2SER_DevHandleGet()

Description

Retrieves the device handle associated to the USB-to-serial adapter.

Files

`usbh_usb2ser.h/usbh_usb2ser.c`

Prototype

```
USBH_DEV_HANDLE USBH_USB2SER_DevHandleGet (USBH_USB2SER_FNCT_HANDLE    usb2ser_fnct_handle,
                                           RTOS_ERR                    *p_err)
```

Arguments

`usb2ser_fnct_handle`

Handle on the USB-to-serial function.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_NOT_AVAIL`

Returned Value

Device handle.

Notes / Warnings

None.

USBH_USB2SER_PortNbrGet()

Description

Retrieves the port number associated with the usb-to-serial function handle.

Files

`usbh_usb2ser.h/usbh_usb2ser.c`

Prototype

```
CPU_INT08U USBH_USB2SER_PortNbrGet (USBH_USB2SER_FNCT_HANDLE   usb2ser_fnct_handle,
                                    RTOS_ERR                   *p_err)
```

Arguments

`usb2ser_fnct_handle`

Handle on the USB-to-serial function.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_NOT_AVAIL`

Returned Value

Port number.

Notes / Warnings

None.

USBH_USB2SER_Reset()

Description

Purges the buffers on the device.

Files

`usbh_usb2ser.h/usbh_usb2ser.c`

Prototype

```
void USBH_USB2SER_Reset (USBH_USB2SER_FNCT_HANDLE   usb2ser_fnct_handle,
                    USBH_USB2SER_RESET_SEL     sel,
                    RTOS_ERR                  *p_err)
```

Arguments

`usb2ser_fnct_handle`

Handle on USB-to-serial function.

`sel`

Type of reset.

- `USBH_USB2SER_RESET_SEL_TX`
- `USBH_USB2SER_RESET_SEL_RX`
- `USBH_USB2SER_RESET_SEL_ALL`

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ABORT`

- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_TIMEOUT
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_IS_OWNER
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_NOT_AVAIL

Returned Value

None.

Notes / Warnings

None.

USBH_USB2SER_BaudRateSet()

Description

Sets the baud rate of the communication port.

Files

usbh_usb2ser.h/usbh_usb2ser.c

Prototype

```
void USBH_USB2SER_BaudRateSet (USBH_USB2SER_FNCT_HANDLE   usb2ser_fnct_handle,
                               CPU_INT32U                 baudrate,
                               RTOS_ERR                  *p_err)
```

Arguments

usb2ser_fnct_handle

Handle on the USB-to-serial function.

baudrate

Baud rate to configure (in bauds/second).

p_err

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_ABORT
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_TIMEOUT
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_IS_OWNER
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_NOT_AVAIL

Returned Value

None.

Notes / Warnings

None.

USBH_USB2SER_BaudRateGet()

Description

Gets the current baud rate of the communication port.

Files

`usbh_usb2ser.h/usbh_usb2ser.c`

Prototype

```
CPU_INT32U USBH_USB2SER_BaudRateGet (USBH_USB2SER_FNCT_HANDLE   usb2ser_fnct_handle,
                                    RTOS_ERR                   *p_err)
```

Arguments

`usb2ser_fnct_handle`

Handle on the USB-to-serial function.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_NOT_AVAIL`

Returned Value

Current baud rate in bauds/sec.

Notes / Warnings

None.

USBH_USB2SER_DataSet()

Description

Sets the data characteristics of the communication port.

Files

`usbh_usb2ser.h/usbh_usb2ser.c`

Prototype

```
void USBH_USB2SER_DataSet (USBH_USB2SER_FNCT_HANDLE    usb2ser_fnct_handle,
                          CPU_INT08U                  data_size,
                          USBH_USB2SER_PARITY         parity,
                          USBH_USB2SER_STOP_BITS      stop_bits,
                          RTOS_ERR                   *p_err)
```

Arguments

`usb2ser_fnct_handle`

Handle on the USB-to-serial function.

`data_size`

Number of data bits.

`parity`

Define the Parity to use, as follows:

- `USBH_USB2SER_PARITY_NONE` , Do not use the parity bit.
- `USBH_USB2SER_PARITY_ODD` , Use odd parity bit.
- `USBH_USB2SER_PARITY_EVEN` , Use even parity bit.
- `USBH_USB2SER_PARITY_MARK` , Use mark parity bit.
- `USBH_USB2SER_PARITY_SPACE` , Use space parity bit.

`stop_bits`

Define the Number of stop bits, as follows:

- `USBH_USB2SER_STOP_BITS_1` , Use 1 stop bit.
- `USBH_USB2SER_STOP_BITS_1_5` , Use 1.5 stop bit.
- `USBH_USB2SER_STOP_BITS_2` , Use 2 stop bits.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_NOT_AVAIL`

Returned Value

None.

Notes / Warnings

None.

USBH_USB2SER_DataGet()

Description

Gets the data characteristics of the communication port.

Files

`usbh_usb2ser.h/usbh_usb2ser.c`

Prototype

```
void USBH_USB2SER_DataGet (USBH_USB2SER_FNCT_HANDLE   usb2ser_fnct_handle,
                           CPU_INT08U                 *p_data_size,
                           USBH_USB2SER_PARITY        *p_parity,
                           USBH_USB2SER_STOP_BITS     *p_stop_bits,
                           RTOS_ERR                   *p_err)
```

Arguments

`usb2ser_fnct_handle`

Handle on the USB-to-serial function.

`p_data_size`

Pointer to the variable that will receive the number of data bits.

`p_parity`

Pointer to the variable that will receive the parity check used, as follows:

- `USBH_USB2SER_PARITY_NONE` , Do not use the parity bit.
- `USBH_USB2SER_PARITY_ODD` , Use odd parity bit.
- `USBH_USB2SER_PARITY_EVEN` , Use even parity bit.
- `USBH_USB2SER_PARITY_MARK` , Use mark parity bit.
- `USBH_USB2SER_PARITY_SPACE` , Use space parity bit.

`p_stop_bits`

Pointer to the variable that will receive the number of stop bits, as follows:

- `USBH_USB2SER_STOP_BITS_1` , Use 1 stop bit.
- `USBH_USB2SER_STOP_BITS_1_5` , Use 1.5 stop bit.
- `USBH_USB2SER_STOP_BITS_2` , Use 2 stop bits.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_NOT_AVAIL`

Returned Value

None.

Notes / Warnings

None.

USBH_USB2SER_BreakSignalSet()

Description

Sets the break signal.

Files

`usbh_usb2ser.h/usbh_usb2ser.c`

Prototype

```
void USBH_USB2SER_BreakSignalSet (USBH_USB2SER_FNCT_HANDLE   usb2ser_fnct_handle,
                                  CPU_BOOLEAN                set,
                                  RTOS_ERR                  *p_err)
```

Arguments

`usb2ser_fnct_handle`

Handle on the USB-to-serial function.

`set`

Boolean that indicates if the break signal should be set or cleared.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_NOT_AVAIL`

Returned Value

None.

Notes / Warnings

None.

USBH_USB2SER_ModemDTR_Set()

Description

Controls the modem DTR pin on a port.

Files

`usbh_usb2ser.h/usbh_usb2ser.c`

Prototype

```
void USBH_USB2SER_ModemDTR_Set (USBH_USB2SER_FNCT_HANDLE   usb2ser_fnct_handle,
                               CPU_BOOLEAN              set,
                               RTOS_ERR                *p_err)
```

Arguments

`usb2ser_fnct_handle`

Handle on USB-to-serial function.

`set`

Sets or clears the Data Terminal Ready (DTR) pin.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_USB2SER_FLOW_CTRL_EN`

Returned Value

None.

Notes / Warnings

1. If the current flow control is set to DTR/DSR, the DTR pin cannot be set manually.

USBH_USB2SER_ModemDTR_Get()

Description

Gets the DTR pin state on a port.

Files

`usbh_usb2ser.h/usbh_usb2ser.c`

Prototype

```
CPU_BOOLEAN USBH_USB2SER_ModemDTR_Get (USBH_USB2SER_FNCT_HANDLE   usb2ser_fnct_handle,                          RTOS_ERR
*p_err)
```

Arguments

`usb2ser_fnct_handle`

Handle on the USB-to-serial function.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_ABORT
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_TIMEOUT
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_IS_OWNER
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_USB2SER_FLOW_CTRL_EN

Returned Value

DTR pin state.

Notes / Warnings

None.

USBH_USB2SER_ModemRTS_Set()

Description

Controls the modem RTS pin on a port.

Files

usbh_usb2ser.h/usbh_usb2ser.c

Prototype

```
void USBH_USB2SER_ModemRTS_Set (USBH_USB2SER_FNCT_HANDLE   usb2ser_fnct_handle,
                                CPU_BOOLEAN                set,
                                RTOS_ERR                  *p_err)
```

Arguments

usb2ser_fnct_handle

Handle on the USB-to-serial function.

set

Sets or clears the Ready To Send (RTS) pin.

p_err

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_ABORT
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_TIMEOUT
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_IS_OWNER
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_NOT_AVAIL
- RTOS_ERR_USB2SER_FLOW_CTRL_EN

Returned Value

None.

Notes / Warnings

1. If the current flow control is set to RTS/CTS, the RTS pin cannot be set.

USBH_USB2SER_ModemRTS_Get()

Description

Gets the RTS pin state on a port.

Files

`usbh_usb2ser.h/usbh_usb2ser.c`

Prototype

```
CPU_BOOLEAN USBH_USB2SER_ModemRTS_Get (USBH_USB2SER_FNCT_HANDLE   usb2ser_fnct_handle,                          RTOS_ERR
*p_err)
```

Arguments

`usb2ser_fnct_handle`

Handle on USB-to-serial function.

`p_err`

Pointer to the variable that will receive the return error code from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_USB2SER_FLOW_CTRL_EN`

Returned Value

RTS pin status.

Notes / Warnings

None.

USBH_USB2SER_HW_FlowCtrlSet()

Description

Sets the hardware flow control protocol on the serial port.

Files

`usbh_usb2ser.h/usbh_usb2ser.c`

Prototype

```
void USBH_USB2SER_HW_FlowCtrlSet (USBH_USB2SER_FNCT_HANDLE           usb2ser_fnct_handle,
                                 USBH_USB2SER_HW_FLOW_CTRL_PROTOCOL   protocol,
                                 RTOS_ERR                           *p_err)
```

Arguments

`usb2ser_fnct_handle`

Handle on the USB-to-serial function.

`protocol`

Defines which hardware flow control protocol to use, as follows:

- `USBH_USB2SER_HW_FLOW_CTRL_PROTOCOL_RTS_CTS` , Use RTS/CTS protocol.
- `USBH_USB2SER_HW_FLOW_CTRL_PROTOCOL_DTR_DSR` , Use DTR/DSR protocol.
- `USBH_USB2SER_HW_FLOW_CTRL_PROTOCOL_NONE` , No HW protocol.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_NOT_AVAIL`

Returned Value

None.

Notes / Warnings

None.

USBH_USB2SER_HW_FlowCtrlGet()

Description

Gets the hardware flow control protocol on the serial port.

Files

`usbh_usb2ser.h/usbh_usb2ser.c`

Prototype

```
USBH_USB2SER_HW_FLOW_CTRL_PROTOCOL USBH_USB2SER_HW_FlowCtrlGet (USBH_USB2SER_FNCT_HANDLE    usb2ser_fnct_handle,
                                                               RTOS_ERR                  *p_err)
```

Arguments

`usb2ser_fnct_handle`

Handle on the USB-to-serial function.

p_err

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_ABORT
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_TIMEOUT
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_IS_OWNER
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_NOT_AVAIL

Returned Value

Current hardware flow control protocol.

Notes / Warnings

None.

USBH_USB2SER_SW_FlowCtrlSet()

Description

Sets the software flow control on the serial port.

Files

usbh_usb2ser.h/usbh_usb2ser.c

Prototype

```
void USBH_USB2SER_SW_FlowCtrlSet (USBH_USB2SER_FNCT_HANDLE   usb2ser_fnct_handle,
                                  CPU_BOOLEAN                 en,
                                  CPU_INT08U                  xon_char,
                                  CPU_INT08U                  xoff_char,
                                  RTOS_ERR                   *p_err)
```

Arguments

usb2ser_fnct_handle

Handle on the USB-to-serial function.

en

Boolean that indicates if the software flow control should be enabled.

xon_char

Xon character to use.

xoff_char

Xoff character to use.

p_err

Pointer to the variable that will receive the return error code from this function :

- RTOS_ERR_NONE

- RTOS_ERR_ABORT
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_TIMEOUT
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_IS_OWNER
- RTOS_ERR_WOULD_OVF
- RTOS_ERR_NOT_AVAIL

Returned Value

None.

Notes / Warnings

None.

USBH_USB2SER_SW_FlowCtrlGet()

Description

Gets the software flow control state on the serial port.

Files

usbh_usb2ser.h/usbh_usb2ser.c

Prototype

```
CPU_BOOLEAN USBH_USB2SER_SW_FlowCtrlGet (USBH_USB2SER_FNCT_HANDLE    usb2ser_fnct_handle,
                                         CPU_INT08                  *p_xon_char,
                                         CPU_INT08U                 *p_xoff_char,
                                         RTOS_ERR                   *p_err)
```

Arguments

usb2ser_fnct_handle

Handle on the USB-to-serial function.

p_xon_char

Pointer to the variable that will receive the Xon character.

p_xoff_char

Pointer to the variable that will receive the Xoff character.

p_err

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_ABORT
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_WOULD_BLOCK
- RTOS_ERR_TIMEOUT
- RTOS_ERR_OS_SCHED_LOCKED
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_IS_OWNER
- RTOS_ERR_WOULD_OVF

- RTOS_ERR_NOT_AVAIL

Returned Value

State of the software flow control.

Notes / Warnings

None.

USBH_USB2SER_StatusGet()

Description

Retrieves the current serial status.

Files

usbh_usb2ser.h/usbh_usb2ser.c

Prototype

```
USBH_USB2SER_SERIAL_STATUS USBH_USB2SER_StatusGet (USBH_USB2SER_FNCT_HANDLE    usb2ser_fnct_handle,
                                                   RTOS_ERR                    *p_err)
```

Arguments

usb2ser_fnct_handle

Handle on the USB-to-serial function.

p_err

Pointer to the variable that will receive one of these return error codes from this function :

- RTOS_ERR_NONE
- RTOS_ERR_ABORT
- RTOS_ERR_INVALID_HANDLE
- RTOS_ERR_TIMEOUT
- RTOS_ERR_OS_OBJ_DEL
- RTOS_ERR_NOT_AVAIL

Returned Value

Current serial status.

Notes / Warnings

1. The following masks determine if part of the modem status is set, as follows:

- USBH_USB2SER_MODEM_STATUS_CTS
- USBH_USB2SER_MODEM_STATUS_DSR
- USBH_USB2SER_MODEM_STATUS_RING
- USBH_USB2SER_MODEM_STATUS_CARRIER

1. The following masks may be used to determine if part of the line status is set, as follows:

- USBH_USB2SER_LINE_STATUS_RX_OVERFLOW_ERR
- USBH_USB2SER_LINE_STATUS_PARITY_ERR
- USBH_USB2SER_LINE_STATUS_FRAMING_ERR
- USBH_USB2SER_LINE_STATUS_BRK_INT

USBH_USB2SER_TxAsync()

Description

Sends the data on the serial port. This function is non-blocking.

Files

`usbh_usb2ser.h/usbh_usb2ser.c`

Prototype

```
void USBH_USB2SER_TxAsync (USBH_USB2SER_FNCT_HANDLE     usb2ser_fnct_handle,
                           CPU_INT08U                   *p_buf,
                           CPU_INT32U                   buf_len,
                           USBH_USB2SER_ASYNC_TX_FNCT   tx_cmpl_notify,
                           void                         *p_arg,
                           RTOS_ERR                     *p_err)
```

Arguments

`usb2ser_fnct_handle`

Handle on the USB-to-serial function.

`p_buf`

Pointer to the buffer of data that will be sent.

`buf_len`

Buffer length in bytes.

`tx_cmpl_notify`

Function that will be invoked upon completion of transmit operation.

`p_arg`

Pointer to the argument that will be passed as parameter of tx_cmpl_notify.

`p_err`

Pointer to the variable that will receive one of these return error codes from this function :

- `RTOS_ERR_NONE`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_BLK_ALLOC_CALLBACK`
- `RTOS_ERR_POOL_EMPTY`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_INVALID_HANDLE`
- `RTOS_ERR_SEG_OVF`
- `RTOS_ERR_TIMEOUT`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`

Returned Value

None.

Notes / Warnings

None.

CAN API

CAN API

CAN API :

Contents:

- CAN API: Functions and Macros
- CAN API: Structures and Data Types

CAN API Functions And Macros

CAN API: Functions and Macros

Contents:

- CAN Bus API: Functions and Macros
- CANopen API: Functions and Macros

CAN Bus API: Functions and Macros

CAN Bus API : Functions and Macros

Contents:

- CAN_CTRLR_REG()

CAN_CTRLR_REG()

Description

Registers a CAN controller to the platform manager.

Files

`can_bus.h`

Prototype

```
CAN_CTRLR_REG (name            p_drv_info)
```

Arguments

`name`

Unique name for the CAN controller. It is recommended to follow the standard "canX", where X is a digit.

`p_drv_info`

Pointer to the CAN Bus driver hardware information structure of type `CAN_CTRLR_DRV_INFO`.

Returned Value

None.

Notes / Warnings

1. This macro should normally be called from the BSP.

CANopen API: Functions and Macros

CANopen API : Functions and Macros

Contents:

- CANopen Core API: Functions and Macros
- CANopen Communication Object API: Functions and Macros

CANopen Core API: Functions and Macros

CANopen Core API : Functions and Macros

Contents:

- CANopen_ConfigureMemSeg()
- CANopen_ConfigureEventQty()
- CANopen_ConfigureSvcTaskStk()
- CANopen_ConfigureTmrPeriod()
- CANopen_Init()
- CANopen_NodeAdd()
- CANopen_NodeStart()
- CANopen_NodeStop()
- CANopen_NodeParamLoad()
- CANopen_NodeLockTimeoutSet()
- CANopen_SvcTaskPrioSet()
- CANopen_DictByteRd()
- CANopen_DictWordRd()
- CANopen_DictLongRd()
- CANopen_DictByteWr()
- CANopen_DictWordWr()
- CANopen_DictLongWr()
- CANopen_DictBufRd()
- CANopen_DictBufWr()
- CANOPEN_KEY()
- CANOPEN_DEV()
- CANOPEN_OBJ_MAPPING_LINK()
- CANOPEN_OBJ_GET_DEV()
- CANOPEN_OBJ_GET_SUBIX()
- CANOPEN_OBJ_GET_IX()
- CANOPEN_OBJ_GET_SIZE()
- CANOPEN_OBJ_IS_PDOMAP()
- CANOPEN_OBJ_IS_NODEID()
- CANOPEN_OBJ_IS_DIRECT()
- CANOPEN_OBJ_IS_RD()
- CANOPEN_OBJ_IS_WR()
- CANOPEN_OBJ_IS_RD_ONLY()

CANopen_ConfigureMemSeg()

Description

Configures the memory segment where CANopen module data structures will be allocated.

Files

`canopen_core.h/canopen_core.c`

Prototype

```
void CANopen_ConfigureMemSeg (MEM_SEG   *p_seg)
```

Arguments

`p_seg`

Pointer to memory segment to use when allocating control data. DEF_NULL means general purpose heap segment.

Returned Value

None.

Notes / Warnings

1. Calling this function is optional, if it is not called, the default value will be used.
2. This function MUST be called after the `CANopen_Init()` function.

CANopen_ConfigureEventQty()

Description

Configures the maximum number of events for all the CANopen busses.

Files

`canopen_core.h/canopen_core.c`

Prototype

```
void CANopen_ConfigureEventQty (CPU_SIZE_T    event_qty)
```

Arguments

`event_qty`

Quantity of events.

Returned Value

None.

Notes / Warnings

1. Calling this function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the `CANopen_Init()` function.

CANopen_ConfigureSvcTaskStk()

Description

Configures the CANopen service task's stack.

Files

`canopen_core.h/canopen_core.c`

Prototype

```
void CANopen_ConfigureSvcTaskStk (CPU_INT32U    stk_size_elements,
                                  void         *p_stk)
```

Arguments

`stk_size_elements`

Size, in stack elements, of the task's stack.

`p_stk`

Pointer to base of the task's stack. If `DEF_NULL`, stack will be allocated from KAL's memory segment.

Returned Value

None.

Notes / Warnings

1. Calling this function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the CANopen module is initialized via the `CANopen_Init()` function.
3. In order to change the priority of the CANopen core task, use the function `CANopen_SvcTaskPrioSet()`.

CANopen_ConfigureTmrPeriod()

Description

Configures the hardware timer time base.

Files

`canopen_core.h/canopen_core.c`

Prototype

```
void CANopen_ConfigureTmrPeriod (CPU_INT32U   tmr_period)
```

Arguments

`tmr_period`

Timer period in microsecond.

Returned Value

None.

Notes / Warnings

1. Calling this function is optional, if it is not called, the default value will be used.
2. This function MUST be called before the `CANopen_Init()` function.

CANopen_Init()

Description

Initializes CANopen resources and service task.

Files

`canopen_core.h/canopen_core.c`

Prototype

```
void CANopen_Init (RTOS_ERR   *p_err)
```

Arguments

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_SEG_OVF`

Returned Value

None.

Notes / Warnings

None.

CANopen_NodeAdd()

Description

Adds a node to the stack and configures it with the given specifications.

Files

`canopen_core.h/canopen_core.c`

Prototype

```
CANOPEN_NODE_HANDLE CANopen_NodeAdd (const CPU_CHAR            *p_name,
                                     const CANOPEN_NODE_SPEC    *p_spec,
                                     const CANOPEN_EVENT_FNCTS  *p_event_fctns,
                                           RTOS_ERR             *p_err)
```

Arguments

`p_name`

Pointer to a CAN Bus controller name.

`p_spec`

Pointer to the node's specifications.

`p_event_fncts`

Pointer to event functions callback structure.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NULL_PTR`
- `RTOS_ERR_SEG_OVF`

Returned Value

Handle to the added node, if successful. DEF_NULL, if there was a problem adding the node.

Notes / Warnings

1. The node is still in INIT state after this function call. To finalize the initialization phase (e.g., profile-specific or application actions, etc.), see `CANopen_NodeStart()` .

CANopen_NodeStart()

Description

Starts a node by starting the CAN controller operations.

Files

`canopen_core.h/canopen_core.c`

Prototype

```
void CANopen_NodeStart (CANOPEN_NODE_HANDLE    node_handle,
                        RTOS_ERR              *p_err)
```

Arguments

`node_handle`

Handle to CANopen node object.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NULL_PTR`

Returned Value

None.

Notes / Warnings

None.

CANopen_NodeStop()

Description

Stops a node by stopping the CAN controller operations.

Files

`canopen_core.h/canopen_core.c`

Prototype

```
void CANopen_NodeStop (CANOPEN_NODE_HANDLE    node_handle,
                       RTOS_ERR              *p_err)
```

Arguments

`node_handle`

Handle to CANopen node object.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NULL_PTR`

Returned Value

None.

Notes / Warnings

None.

CANopen_NodeParamLoad()

Description

Loads all parameter groups with the given type.

Files

`canopen_core.h/canopen_core.c`

Prototype

```
void CANopen_NodeParamLoad (CANOPEN_NODE_HANDLE   node_handle,
                            CANOPEN_NMT_RESET     reset_type,
                            RTOS_ERR             *p_err)
```

Arguments

`node_handle`

Handle to CANopen node object.

`reset_type`

Reset type:

- `CANOPEN_RESET_COMM`
- `CANOPEN_RESET_NODE`

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NULL_PTR`
- `RTOS_ERR_SIZE_INVALID`
- `RTOS_ERR_NOT_FOUND`
- `RTOS_ERR_FAIL`

Returned Value

None.

Notes / Warnings

1. The single parameter group(s) will be loaded from non-volatile memory by calling the user application callback function set through `CANOPEN_EVENT_FNCTS` / `ParaOnLoad`.
2. This function considers all parameter groups, which are linked to the parameter store index (`1010h`) within the object directory. Every not linked parameter group is not in the scope of this function and must be handled within the application.

CANopen_NodeLockTimeoutSet()

Description

Sets the node object directory lock timeout.

Files

`canopen_core.h/canopen_core.c`

Prototype

```
void CANopen_NodeLockTimeoutSet (CANOPEN_NODE_HANDLE    node_handle,
                                 CPU_INT32U             timeout,
                                 RTOS_ERR              *p_err)
```

Arguments

`node_handle`

Handle to CANopen node object.

`timeout`

Timeout value for the object dictionary lock.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NULL_PTR`

Returned Value

None.

Notes / Warnings

None.

CANopen_SvcTaskPrioSet()

Description

Assigns a new priority to the CANopen service task.

Files

`canopen_core.h/canopen_core.c`

Prototype

```
void CANopen_SvcTaskPrioSet (CPU_INT08U    prio,
                             RTOS_ERR     *p_err)
```

Arguments

`prio`

New priority of the service task.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_INVALID_ARG

Returned Value

None.

Notes / Warnings

1. This function MUST be called before the CANopen_Init() function.

CANopen_DictByteRd()

Description

Reads a 8-bit value from the given object dictionary.

Files

canopen_dict.h/canopen_dict.c

Prototype

```
void CANopen_DictByteRd (CANOPEN_NODE_HANDLE   node_handle,
                         CPU_INT32U            key,
                         CPU_INT08U           *p_val,
                         RTOS_ERR             *p_err)
```

Arguments

node_handle

Handle to CANopen node object.

key

Object entry key; should be generated with the macro CANOPEN_DEV() .

p_val

Pointer to the value destination.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_SIZE_INVALID
- RTOS_ERR_OBJ_RD

Returned Value

None.

Notes / Warnings

1. The object entry is addressed with the given key and the value will be written to the given destination pointer.

CANopen_DictWordRd()

Description

Reads a 16-bit value from the given object dictionary.

Files

`canopen_dict.h/canopen_dict.c`

Prototype

```
void CANopen_DictWordRd (CANOPEN_NODE_HANDLE   node_handle,
                         CPU_INT32U            key,
                         CPU_INT16U           *p_val,
                         RTOS_ERR             *p_err)
```

Arguments

`node_handle`

Handle to CANopen node object.

`key`

Object entry key; should be generated with the macro `CANOPEN_DEV()` .

`p_val`

Pointer to the value destination.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_SIZE_INVALID`
- `RTOS_ERR_OBJ_RD`

Returned Value

None.

Notes / Warnings

1. The object entry is addressed with the given key and the value will be written to the given destination pointer.

CANopen_DictLongRd()

Description

Reads a 32-bit value from the given object dictionary.

Files

`canopen_dict.h/canopen_dict.c`

Prototype

```
void CANopen_DictLongRd (CANOPEN_NODE_HANDLE   node_handle,
                         CPU_INT32U            key,
                         CPU_INT32U           *p_val,
                         RTOS_ERR             *p_err)
```

Arguments

`node_handle`

Handle to CANopen node object.

`key`

Object entry key; should be generated with the macro `CANOPEN_DEV()` .

`p_val`

Pointer to the value destination.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_SIZE_INVALID`
- `RTOS_ERR_OBJ_RD`

Returned Value

None.

Notes / Warnings

1. The object entry is addressed with the given key and the value will be written to the given destination pointer.

CANopen_DictByteWr()

Description

Writes a 8-bit value to the given object dictionary.

Files

`canopen_dict.h/canopen_dict.c`

Prototype

```
void CANopen_DictByteWr (CANOPEN_NODE_HANDLE    node_handle,
                         CPU_INT32U             key,
                         CPU_INT08U             val,
                         RTOS_ERR              *p_err)
```

Arguments

`node_handle`

Handle to CANopen node object.

`key`

Object entry key; should be generated with the macro `CANOPEN_DEV()` .

`val`

The source value.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_SIZE_INVALID`

- RTOS_ERR_OBJ_WD

Returned Value

None.

Notes / Warnings

1. The object entry is addressed with the given key and the value will be read from the given source pointer.

CANopen_DictWordWr()

Description

Writes a 16-bit value to the given object dictionary.

Files

canopen_dict.h/canopen_dict.c

Prototype

```
void CANopen_DictWordWr (CANOPEN_NODE_HANDLE   node_handle,
                         CPU_INT32U            key,
                         CPU_INT16U            val,
                         RTOS_ERR             *p_err)
```

Arguments

node_handle

Handle to CANopen node object.

key

Object entry key; should be generated with the macro CANOPEN_DEV() .

val

The source value.

p_err

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_SIZE_INVALID
- RTOS_ERR_OBJ_WD

Returned Value

None.

Notes / Warnings

1. The object entry is addressed with the given key and the value will be read from the given source pointer.

CANopen_DictLongWr()

Description

This function writes a 32-bit value to the given object dictionary.

Files

`canopen_dict.h/canopen_dict.c`

Prototype

```
void CANopen_DictLongWr (CANOPEN_NODE_HANDLE    node_handle,
                         CPU_INT32U             key,
                         CPU_INT32U             val,
                         RTOS_ERR              *p_err)
```

Arguments

`node_handle`

Handle to CANopen node object.

`key`

Object entry key; should be generated with the macro `CANOPEN_DEV()` .

`val`

The source value.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_SIZE_INVALID`
- `RTOS_ERR_OBJ_WD`

Returned Value

None.

Notes / Warnings

1. The object entry is addressed with the given key and the value will be read from the given source pointer.

CANopen_DictBufRd()

Description

Reads a buffer byte stream from the given object dictionary.

Files

`canopen_dict.h/canopen_dict.c`

Prototype

```
void CANopen_DictBufRd (CANOPEN_NODE_HANDLE    node_handle,
                        CPU_INT32U             key,
                        CPU_INT08U            *p_buf,
                        CPU_INT32U             len,
                        RTOS_ERR              *p_err)
```

Arguments

`node_handle`

Handle to CANopen node object.

`key`

Object entry key; should be generated with the macro `CANOPEN_DEV()` .

`p_buf`

Pointer to the destination buffer.

`len`

Length of destination buffer.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

None.

Notes / Warnings

1. The object entry is addressed with the given key and the bytes to read will be written to the given destination buffer of the given length.

CANopen_DictBufWr()

Description

Writes a buffer byte stream to the given object dictionary.

Files

`canopen_dict.h/canopen_dict.c`

Prototype

```
void CANopen_DictBufWr (CANOPEN_NODE_HANDLE    node_handle,
                        CPU_INT32U             key,
                        CPU_INT08U            *p_buf,
                        CPU_INT32U             len,
                        RTOS_ERR              *p_err)
```

Arguments

`node_handle`

Handle to CANopen node object.

`key`

Object entry key; should be generated with the macro `CANOPEN_DEV()` .

`p_buf`

Pointer to the source buffer.

`len`

Length of source buffer.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_AVAIL`
- `RTOS_ERR_WOULD_OVF`
- `RTOS_ERR_OS_OBJ_DEL`
- `RTOS_ERR_WOULD_BLOCK`
- `RTOS_ERR_IS_OWNER`
- `RTOS_ERR_OS_SCHED_LOCKED`
- `RTOS_ERR_ABORT`
- `RTOS_ERR_TIMEOUT`

Returned Value

None.

Notes / Warnings

1. The object entry is addressed with the given key and the bytes to write will be read from to the given source buffer of the given length.

CANOPEN_KEY()

Description

This macro helps to build the CANopen object member variable 'key'.

Files

`canopen_obj.h`

Prototype

```
CANOPEN_KEY (idx,
             sub,
             flags)
```

Arguments

`idx`

CANopen object index [0x0000..0xFFFF]

`sub`

CANopen object subindex [0x00..0xFF]

`flags`

The additional object property flags

Returned Value

None.

Notes / Warnings

The resulting variable is used to hold the unique address of the object entry with index:subindex. Furthermore the lower 8 bits of the key are used to describe the way of accessing this object entry.

CANOPEN_DEV()

Description

This macro helps to build the CANopen object member device identifier.

Files

`canopen_obj .h`

Prototype

```
CANOPEN_DEV (idx            sub)
```

Arguments

`idx`

CANopen object index [0x0000..0xFFFF]

`sub`

CANopen object subindex [0x00..0xFF]

Returned Value

None.

Notes / Warnings

This identifier is used to search the unique address of the object entry (index:subindex).

CANOPEN_OBJ_MAPPING_LINK()

Description

This macro helps to build the CANopen object entry for a PDO mapping.

Files

`canopen_obj .h`

Prototype

```
CANOPEN_OBJ_MAPPING_LINK (idx                    sub                    bit)
```

Arguments

`idx`

CANopen object index [0x0000..0xFFFF]

`sub`

CANopen object subindex [0x00..0xFF]

`bit`

Length of mapped signal in bits [8,16 or 32]

Returned Value

None.

Notes / Warnings

This entry is used to specify the linked signal within a PDO.

CANOPEN_OBJ_GET_DEV()

Description

This macro helps to extract the index and subindex out of the CANopen object entry member 'key'.

Files

`canopen_obj .h`

Prototype

```
CANOPEN_OBJ_GET_DEV (key)
```

Arguments

`key`

CANopen object member variable 'key'.

Returned Value

None.

Notes / Warnings

None.

CANOPEN_OBJ_GET_SUBIX()

Description

This macro helps to extract the subindex out of the CANopen object entry member 'key'.

Files

`canopen_obj .h`

Prototype

```
CANOPEN_OBJ_GET_SUBIX (key)
```

Arguments

`key`

CANopen object member variable ' `key` '.

Returned Value

None.

Notes / Warnings

None.

CANOPEN_OBJ_GET_IX()

Description

This macro helps to extract the index out of the CANopen object entry member ' `key` '.

Files

`canopen_obj .h`

Prototype

```
CANOPEN_OBJ_GET_IX (key)
```

Arguments

`key`

CANopen object member variable 'key'.

Returned Value

None.

Notes / Warnings

None.

CANOPEN_OBJ_GET_SIZE()

Description

This macro helps to extract the object entry size in bytes out of the CANopen object entry member ' `key` '.

Files

`canopen_obj.h`

Prototype

```
CANOPEN_OBJ_GET_SIZE (key)
```

Arguments

`key`

CANopen object member variable ' `key` '.

Returned Value

None.

Notes / Warnings

If this result is 0, the size must be calculated with the function `Size()` , referenced in the linked type structure.

CANOPEN_OBJ_IS_PDOMAP()

Description

This macro helps to determine, if the object entry is PDO mappable.

Files

`canopen_obj.h`

Prototype

```
CANOPEN_OBJ_IS_PDOMAP (key)
```

Arguments

`key`

CANopen object member variable 'key'.

Returned Value

None.

Notes / Warnings

None.

CANOPEN_OBJ_IS_NODEID()

Description

This macro helps to determine, if the object entry value depends on the node-ID.

Files

`canopen_obj.h`

Prototype

```
CANOPEN_OBJ_IS_NODEID (key)
```

Arguments

`key`

CANopen object member variable 'key'.

Returned Value

None.

Notes / Warnings

None.

CANOPEN_OBJ_IS_DIRECT()

Description

This macro helps to determine, if the object entry value is a direct value.

Files

`canopen_obj.h`

Prototype

```
CANOPEN_OBJ_IS_DIRECT (key)
```

Arguments

`key`

CANopen object member variable 'key'.

Returned Value

None.

Notes / Warnings

None.

CANOPEN_OBJ_IS_RD()

Description

This macro helps to determine, if the object entry value is readable.

Files

`canopen_obj.h`

Prototype

```
CANOPEN_OBJ_IS_RD (key)
```

Arguments

`key`

CANopen object member variable 'key'.

Returned Value

None.

Notes / Warnings

None.

CANOPEN_OBJ_IS_WR()

Description

This macro helps to determine, if the object entry value is writeable.

Files

`canopen_obj.h`

Prototype

```
CANOPEN_OBJ_IS_WR (key)
```

Arguments

`key`

CANopen object member variable 'key'.

Returned Value

None.

Notes / Warnings

None.

CANOPEN_OBJ_IS_RD_ONLY()

Description

This macro helps to determine, if the object entry value is read-only.

Files

`canopen_obj.h`

Prototype

```
CANOPEN_OBJ_IS_RD_ONLY (key)
```

Arguments

`key`

CANopen object member variable 'key'.

Returned Value

None.

Notes / Warnings

None.

CANopen Communication Object API: Functions and Macros

CANopen Communication object API : Functions and Macros

Contents:

- CANopen_EmcySet()
- CANopen_EmcyClr()
- CANopen_EmcyGet()
- CANopen_EmcyCnt()
- CANopen_EmcyReset()
- CANopen_EmcyHistReset()
- CANopen_NmtReset()
- CANopen_NmtStateSet()
- CANopen_NmtStateGet()
- CANopen_NmtHbConsEventsGet()
- CANopen_NmtHbConsLastStateGet()

CANopen_EmcySet()

Description

Sets an emergency message and sends an event to the Service task to trigger emergency processing.

Files

`canopen_emcy.h/canopen_emcy.c`

Prototype

```
void CANopen_EmcySet (CANOPEN_NODE_HANDLE    node_handle,
                      CPU_INT08U             err_code_ix,
                      CANOPEN_EMCY_USR      *p_user,
                      RTOS_ERR              *p_err)
```

Arguments

`node_handle`

Handle to CANopen node object.

`err_code_ix`

EMCY error index in User EMCY table.

`p_user`

Vendor-specific fields in EMCY history and/or EMCY message.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NULL_PTR`

Returned Value

None.

Notes / Warnings

None.

CANopen_EmcyClr()

Description

Clears an emergency message and sends an event to the Service task to trigger emergency processing.

Files

`canopen_emcy.h/canopen_emcy.c`

Prototype

```
void CANopen_EmcyClr (CANOPEN_NODE_HANDLE    node_handle,
                      CPU_INT08U             err_code_ix,
                      RTOS_ERR              *p_err)
```

Arguments

`node_handle`

Handle to CANopen node object.

`err_code_ix`

EMCY error index in User EMCY table.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NULL_PTR`

Returned Value

None.

Notes / Warnings

None.

CANopen_EmcyGet()

Description

Returns the current EMCY error status.

Files

`canopen_emcy.h/canopen_emcy.c`

Prototype

```
CPU_INT16S CANopen_EmcyGet (CANOPEN_NODE_HANDLE    node_handle,
                            CPU_INT08U             err_code_ix,
                            RTOS_ERR              *p_err)
```

Arguments

`node_handle`

Handle to CANopen node object.

`err_code_ix`

EMCY error index in User EMCY table.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_INIT`
- `RTOS_ERR_NULL_PTR`

Returned Value

Current EMCY error status.

Notes / Warnings

None.

CANopen_EmcyCnt()

Description

Returns the number of currently detected EMCY errors.

Files

`canopen_emcy.h/canopen_emcy.c`

Prototype

```
CPU_INT16S CANopen_EmcyCnt (CANOPEN_NODE_HANDLE    node_handle,
                            RTOS_ERR              *p_err)
```

Arguments

`node_handle`

Handle to CANopen node object.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NOT_INIT`
- `RTOS_ERR_NULL_PTR`

Returned Value

Number of EMCY errors.

Notes / Warnings

None.

CANopen_EmcyReset()

Description

Clears all EMCY errors.

Files

`canopen_emcy.h/canopen_emcy.c`

Prototype

```
void CANopen_EmcyReset (CANOPEN_NODE_HANDLE    node_handle,
                        RTOS_ERR              *p_err)
```

Arguments

`node_handle`

Handle to CANopen node object.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- RTOS_ERR_NONE
- RTOS_ERR_NOT_INIT
- RTOS_ERR_NULL_PTR

Returned Value

None.

Notes / Warnings

None.

CANopen_EmcyHistReset()

Description

Clears the EMCY history in the object dictionary.

Files

`canopen_emcy.h/canopen_emcy.c`

Prototype

```
void CANopen_EmcyHistReset (CANOPEN_NODE_HANDLE    node_handle)
```

Arguments

`node_handle`

Handle to CANopen node object.

Returned Value

None.

Notes / Warnings

None.

CANopen_NmtReset()

Description

Resets the CANopen device with the given type.

Files

`canopen_nmt.h/canopen_nmt.c`

Prototype

```
void CANopen_NmtReset (CANOPEN_NODE_HANDLE    node_handle,
                       CANOPEN_NMT_RESET      type,
                       RTOS_ERR              *p_err)
```

Arguments

`node_handle`

Handle to CANopen node object.

`type`

The requested NMT reset type:

- `CANOPEN_RESET_NODE`
- `CANOPEN_RESET_COMM`

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NULL_PTR`

Returned Value

None.

Notes / Warnings

None.

CANopen_NmtStateSet()

Description

Sets the requested CANopen NMT state machine state.

Files

`canopen_nmt.h/canopen_nmt.c`

Prototype

```
void CANopen_NmtStateSet (CANOPEN_NODE_HANDLE    node_handle,
                          CANOPEN_NODE_STATE     state,
                          RTOS_ERR              *p_err)
```

Arguments

`node_handle`

Handle to CANopen node object.

`state`

The requested NMT state:

- `CANOPEN_INIT`
- `CANOPEN_PREOP`
- `CANOPEN_OPERATIONAL`
- `CANOPEN_STOP`

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NULL_PTR`

Returned Value

None.

Notes / Warnings

None.

CANopen_NmtStateGet()

Description

Returns the current CANopen NMT state machine state.

Files

`canopen_nmt.h/canopen_nmt.c`

Prototype

```
CANOPEN_NODE_STATE CANopen_NmtStateGet (CANOPEN_NODE_HANDLE    node_handle)
```

Arguments

`node_handle`

Handle to CANopen node object.

Returned Value

NMT state of the parent node.

Notes / Warnings

None.

CANopen_NmtHbConsEventsGet()

Description

Gets the number of missed heartbeat events.

Files

`canopen_nmt.h/canopen_hbcons.c`

Prototype

```
CPU_INT16S CANopen_NmtHbConsEventsGet (CANOPEN_NODE_HANDLE    node_handle,
                                       CPU_INT08U             node_id,
                                       RTOS_ERR              *p_err)
```

Arguments

`node_handle`

Handle to CANopen node object.

`node_id`

node ID of monitored node (or 0 for master node).

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NULL_PTR`

Returned Value

- The number of missed heartbeat events for given node ID.
- < 0, if error detected (e.g., node ID is not monitored).

Notes / Warnings

None.

CANopen_NmtHbConsLastStateGet()

Description

Returns the last received heartbeat state of a given node.

Files

`canopen_nmt.h/canopen_hbcons.c`

Prototype

```
CANOPEN_NODE_STATE CANopen_NmtHbConsLastStateGet (CANOPEN_NODE_HANDLE    node_handle,
                                                  CPU_INT08U             node_id,
                                                  RTOS_ERR              *p_err)
```

Arguments

`node_handle`

Handle to CANopen node object.

`node_id`

Node ID of monitored node.

`p_err`

Pointer to the variable that will receive one of the following error code(s) from this function:

- `RTOS_ERR_NONE`
- `RTOS_ERR_NULL_PTR`

Returned Value

Last detected node state for given node ID:

- `CANOPEN_INVALID`
- `CANOPEN_INIT`
- `CANOPEN_PREOP`
- `CANOPEN_OPERATIONAL`
- `CANOPEN_STOP`

Notes / Warnings

None.

CAN API Structures And Data Types

CAN API: Structures and Data Types

Contents:

- CAN Bus API: Structures and Data Types
- CANopen API: Structures and Data Types

CAN Bus API: Structures and Data Types

CAN Bus API : Structures and Data Types

Contents:

- CAN_CTRLR_HW_INFO
- CAN_CTRLR_DRV_INFO
- CAN_CTRLR_BSP_API

CAN_CTRLR_HW_INFO

Files

`can_bus.h`

Structure Member Descriptions

Name	Definition	Description
.BaseAddr	CPU_ADDR	Controller's registers base address.
.InfoExtPtr	const void *	Extended (driver specific) hardware information.
.IF_Rx	CPU_INT08U	BSP RX interface.
.IF_Tx	CPU_INT08U	BSP TX interface.

CAN_CTRLR_DRV_INFO

Files

`can_bus.h`

Structure Member Descriptions

Name	Definition	Description
.HW_Info	CAN_CTRLR_HW_INFO	Hardware information structure.
.BSP_API_Ptr	const CAN_CTRLR_BSP_API *	Pointer to BSP API structure.

CAN_CTRLR_BSP_API

Files

`can_bus.h`

Structure Member Descriptions

Name	Definition	Description
.Open	void(* Open) (void)	BSP open function pointer.
.Close	void(* Close) (void)	BSP close function pointer.
.IntCtrl	void(* IntCtrl) (CAN_BUS_HANDLE bus_handle)	BSP interrupt control pointer.
.TmrCfg	void(* TmrCfg) (CPU_INT32U tmr_period)	BSP timer configuration pointer.

CANopen API: Structures and Data Types

CANopen API : Structures and Data Types

Contents:

- CANopen Core API: Structures and Data Types
- CANopen Communication Object API: Structures and Data Types

CANopen Core API: Structures and Data Types

CANopen Core API : Structures and Data Types

Contents:

- CANOPEN_NODE_STATE
- CANOPEN_NMT_RESET
- CANOPEN_EVENT_FNCTS
- canopen_tmr_action
- CANOPEN_TMR_MEM
- CANOPEN_PARAM
- CANOPEN_INIT_CFG
- CANOPEN_OBJ
- canopen_tmr_time
- CANOPEN_IF_FRM
- canopen_obj_type
- CANOPEN_NODE_SPEC
- CANOPEN_DOMAIN_STR

CANOPEN_NODE_STATE

Files

`canopen_types.h`

Enum Type Descriptions

Name	Description
CANOPEN_INVALID	Device in INVALID state.
CANOPEN_INIT	Device in INIT state.
CANOPEN_PREOP	Device in PRE-OPERATIONAL state.
CANOPEN_OPERATIONAL	Device in OPERATIONAL state.
CANOPEN_STOP	Device in STOP state.
CANOPEN_STATE_QTY	Number of device states.

CANOPEN_NMT_RESET

Files

canopen_types.h

Enum Type Descriptions

Name	Description
CANOPEN_RESET_NODE	Reset application (and communication).
CANOPEN_RESET_COMM	Reset communication.
CANOPEN_RESET_QTY	Number of reset types.

CANOPEN_EVENT_FNCTS

Files

canopen_core.h

Structure Member Descriptions

Name	Definition	Description
.RpdoOnRx	CPU_INT16S(* RpdoOnRx) (CANOPEN_NODE_HANDLE handle, CANOPEN_IF_FRM *p_frm)	This callback is used when a RPDO is received.
.TpdoOnTx	void(* TpdoOnTx) (CANOPEN_NODE_HANDLE handle, CANOPEN_IF_FRM *p_frm)	This callback is used before a TPDO is transmitted.
.StateOnChange	void(* StateOnChange) (CANOPEN_NODE_HANDLE handle, CANOPEN_NODE_STATE state)	This callback is used when the NMT state is changed.
.HbcOnEvent	void(* HbcOnEvent) (CANOPEN_NODE_HANDLE handle, CPU_INT08U node_id)	This callback is used when a heartbeat consumer monitor timer elapses, before receiving the corresponding heartbeat message.
.HbcOnChange	void(* HbcOnChange) (CANOPEN_NODE_HANDLE handle, CPU_INT08U node_id, CANOPEN_NODE_STATE state)	This callback is used when a heartbeat consumer monitor detects a state change, of a monitored node.
.ParamOnLoad	CPU_BOOLEAN(* ParamOnLoad) (CANOPEN_NODE_HANDLE handle, CANOPEN_PARAM *p_pg)	This callback is used when the NMT slave node is reseted by the NMT master node.
.ParamOnSave	CPU_BOOLEAN(* ParamOnSave) (CANOPEN_NODE_HANDLE handle, CANOPEN_PARAM *p_pg)	This callback is used when the standard object "Store" at index 0x1010 is written.
.ParamOnDflt	CPU_BOOLEAN(* ParamOnDflt) (CANOPEN_NODE_HANDLE handle, CANOPEN_PARAM *p_pg)	This callback is used when the standard object "Restore default parameters" at index 0x1011 is written.

canopen_tmr_action

Files

canopen_types.h

Structure Member Descriptions

Name	Definition	Description
.Id	CPU_INT16U	Unique action identifier.

Name	Definition	Description
.NextActionPtr	CANOPEN_TMR_ACTION *	Link to next action.
.Fnct	CANOPEN_TMR_FNCT	Pointer to callback function.
.ParamPtr	void *	Callback function parameter.
.CycleTime	CPU_INT32U	Action cycle time in ticks.

Notes / Warnings

This structure holds all data, which are needed for managing a timer timed action.

CANOPEN_TMR_MEM

Files

`canopen_types.h`

Structure Member Descriptions

Name	Definition	Description
.Action	canopen_tmr_action	Memory portion for action info.
.Tmr	canopen_tmr_time	Memory portion for timer info.

Notes / Warnings

This structure is intended to simplify the memory allocation in the application. The number of actions and timer structures are always the same, therefore we can reduce the configuration effort to the memory array, and the length of this memory array.

CANOPEN_PARAM

Files

`canopen_param.h`

Structure Member Descriptions

Name	Definition	Description
.MemBlkSize	CPU_INT32U	Size of parameter memory block.
.StartMemBlkPtr	CPU_INT08U *	Start of parameter memory block.
.DfltMemBlkPtr	CPU_INT08U *	Start of default memory block.
.ResetType	CANOPEN_NMT_RESET	Parameter reset type.
.IdPtr	void *	Pointer to User Identification-Code for this group.
.Val	CPU_INT32U	Value when reading parameter object.

CANOPEN_INIT_CFG

Files

`canopen_core.h`

Structure Member Descriptions

Name	Definition	Description
.SvcTaskStkSizeElements	CPU_INT32U	Service task's stack size, in quantity of elements.
.SvcTaskStkPtr	void *	Pointer to Service stack's stack base.
.MemSegPtr	MEM_SEG *	Pointer to memory segment used for internal data.
.EventQtyTot	CPU_SIZE_T	Number of events allocated from the pool.
.HwTmrPeriod	CPU_INT32U	Timer base time in microsecond.

CANOPEN_OBJ

Files

`canopen_obj.h`

Structure Member Descriptions

Name	Definition	Description
.Key	CPU_INT32U	Bitmap with information about object (see Note #2).
.TypePtr	canopen_obj_type *	Pointer to object access specialized functions. See Note #3.
.Data	CPU_INT32U	Address of value/data structure or data value for direct access.

Notes / Warnings

(1) This structure holds all data, needed for managing a single object entry.

(2) The key field is encoded as follows: 16-bit index + 8-bit sub-index + 8-bit flags. The flags are the following:

- bit 0: 1 = read access allowed
- bit 1: 1 = write access allowed
- bit 2: 1 = PDO mappable
- bit 3: 1 = signed value
- bit 4: |- valid bytes in 2^n; used if
- bit 5: | Type=0, or Type->Size ptr=0
- bit 6: 1 = +/- node-id on read/write
- bit 7: 1 = direct access, 0 = access via specialized functions

(3) This pointer is optional. If the pointer is null, the object value is accessed directly via the field 'Data'. If the pointer is defined, the object value is accessed by some specialized functions referenced by 'TypePtr'.

canopen_tmr_time

Files

`canopen_types.h`

Structure Member Descriptions

Name	Definition	Description
.NextPtr	CANOPEN_TMR_TIME *	Link to next timer.
.ActionPtr	canopen_tmr_action *	Root of linked action list.
.ActionEndPtr	canopen_tmr_action *	Last element in linked action list.
.Delta	CPU_INT32U	Delta ticks from previous timer event.

Notes / Warnings

This structure holds all data, which are needed for managing a timer event.

CANOPEN_IF_FRM

Files

`canopen_if.h`

Structure Member Descriptions

Name	Definition	Description
.MsgId	CPU_INT32U	CAN message identifier.
.Data	CPU_INT08U	CAN message Data (payload).
.DLC	CPU_INT08U	CAN message data length code (DLC).
.MsgNbr	CPU_INT08U	CAN message number.

Notes / Warnings

This type definition ensures the independence of the stack from interface driver definitions.

canopen_obj_type

Files

`canopen_obj .h`

Structure Member Descriptions

Name	Definition	Description
.SizeCb	CANOPEN_OBJ_SIZE_FNCT	Get size of type function.
.CtrlCb	CANOPEN_OBJ_CTRL_FNCT	Special type control function.
.RdCb	CANOPEN_OBJ_RD_FNCT	Read function.
.WrCb	CANOPEN_OBJ_WR_FNCT	Write function.

Notes / Warnings

This structure holds all data, needed for managing a special object entry type.

CANOPEN_NODE_SPEC

Files

`canopen_core.h`

Structure Member Descriptions

Name	Definition	Description
.NodeId	CPU_INT08U	Default Node ID.
.Baudrate	CPU_INT32U	Default baudrate for this node.
.DictPtr	CANOPEN_OBJ *	Pointer to object dictionary associated to node.
.DictLen	CPU_INT16U	Object dictionary (maximum) length.
.EmcyCodePtr	CANOPEN_EMCY_TBL *	Pointer to application EMCY information fields.
.TmrMemPtr	CANOPEN_TMR_MEM *	Pointer to timer memory blocks.
.TmrQty	CPU_INT16U	Number of timer memory blocks.
.SdoBufPtr	CPU_INT08U *	Pointer to SDO transfer buffer memory.

Notes / Warnings

This data structure holds all configurable components of a complete CANopen node.

CANOPEN_DOMAIN_STR

Files

`canopen_obj.h`

Structure Member Descriptions

Name	Definition	Description
.DataMemStartPtr	CPU_INT08U *	Pointer to Domain/String memory region.
.DataMemSize	CPU_INT32U	Domain/String memory region size, in bytes.
.DataMemOffset	CPU_INT32U	Offset within Domain/String memory region.

Notes / Warnings

This structure holds all data, which are needed for the domain and string objects management within the object directory.

CANopen Communication Object API: Structures and Data Types

CANopen Communication object API : Structures and Data Types

Contents:

- CANOPEN_EMCY_HIST
- CANOPEN_EMCY_TBL
- CANOPEN_EMCY_USR
- canopen_hbcons

CANOPEN_EMCY_HIST

Files

`canopen_emcy.h`

Structure Member Descriptions

Name	Definition	Description
.TotLen	CPU_INT08U	Total length of EMCY history.
.HistQty	CPU_INT08U	Number of EMCY in history.
.Offset	CPU_INT08U	Sub-index-Offset to newest EMCY entry.

CANOPEN_EMCY_TBL

Files

`canopen_emcy.h`

Structure Member Descriptions

Name	Definition	Description
.Reg	CPU_INT08U	Bit number (0..7) in error register.
.Code	CPU_INT16U	Error code (See Note #1).

Notes / Warnings

(1) Some pre-defined standard error code values are listed in the section EMERGENCY CODE

CANOPEN_EMCY_USR

Files

`canopen_emcy.h`

Structure Member Descriptions

Name	Definition	Description
.Hist	CPU_INT16U	Manufacturer specific field in History.
.Emcy	CPU_INT08U	Manufacturer specific field in EMCY message.

Notes / Warnings

(1) This structure holds the optional manufacturer specific fields for the EMCY message and for the EMCY history.

(2) To reduce memory consumption, the configuration values of CANOPEN_EMCY_HIST_MAN_EN and CANOPEN_EMCY_HIST_MAN_EN may remove some bytes. If no manufacturer specific field is enabled, the structure holds an unused dummy byte.

canopen_hbcons

Files

`canopen_nmt.h`

Structure Member Descriptions

Name	Definition	Description
.NextPtr	CANOPEN_HBCONS *	Link to next consumer in active chain.
.State	CANOPEN_NODE_STATE	Received Node-State.
.TmrId	CPU_INT16S	Timer Identifier.
.TimeMs	CPU_INT16U	Time (Bit 00-15 when read object).
.NodeId	CPU_INT08U	Node ID (Bit 16-23 when read object).
.MissedEventCnt	CPU_INT08U	Event Counter.

Notes / Warnings

(1) This structure holds all data, which are needed for the heartbeat consumer handling within the object dictionary.

Error Codes Description

Error Codes Description

Error Codes Description Table

Error Codes Description Table

Error code	String associated
RTOS_ERR_NONE	No error.
RTOS_ERR_FAIL	Generic failure for operation.
RTOS_ERR_NOT_READY	Module is not ready for requested operation.
RTOS_ERR_ASSERT_DBG_FAIL	Debug assertion failed.
RTOS_ERR_ASSERT_CRITICAL_FAIL	Critical assertion failed.
RTOS_ERR_ASSERT_ERR_PTR_NULL	Pointer to error variable to return error code is null.
RTOS_ERR_NOT_AVAIL	Feature not avail (due to cfg val(s).
RTOS_ERR_NOT_SUPPORTED	Feature not supported.
RTOS_ERR_INVALID_ARG	Invalid argument or consequence of invalid argument.
RTOS_ERR_INVALID_CFG	Invalid configuration provided.
RTOS_ERR_NULL_PTR	Invalid null pointer received as argument.
RTOS_ERR_INVALID_HANDLE	Invalid handle passed to function.
RTOS_ERR_INVALID_TYPE	Invalid type for operation.
RTOS_ERR_INVALID_CREDENTIALS	Credentials used are invalid.
RTOS_ERR_NOT_FOUND	Requested item could not be found.
RTOS_ERR_ALLOC	Generic allocation error.
RTOS_ERR_NO_MORE_RSRC	Resource not available to perform the operation.
RTOS_ERR_INIT	Initialization failed.
RTOS_ERR_NOT_INIT	Module has not been initialized.
RTOS_ERR_ALREADY_INIT	Module has already been initialized.
RTOS_ERR_ALREADY_EXISTS	Item already exists.
RTOS_ERR_SEG_OVF	Mem seg would overflow.
RTOS_ERR_POOL_FULL	Mem pool full; i.e. all mem blks avail in mem pool.
RTOS_ERR_POOL_EMPTY	Mem pool empty; i.e. NO mem blks avail in mem pool.
RTOS_ERR_POOL_UNLIMITED	Mem pool is unlimited, cannot obtain rem nbr of blks.
RTOS_ERR_BLK_ALLOC_CALLBACK	Block alloc callback failed.
RTOS_ERR_OWNERSHIP	Ownership error.
RTOS_ERR_PERMISSION	Operation not allowed.
RTOS_ERR_WOULD_BLOCK	Non-blocking operation would block.
RTOS_ERR_WOULD_OVF	Item would overflow.
RTOS_ERR_INVALID_STATE	Cannot execute requested operation while in current state.
RTOS_ERR_ISR	Illegal call from ISR.
RTOS_ERR_ABORT	Operation aborted.

Error code	String associated
RTOS_ERR_TIMEOUT	Operation timed out.
RTOS_ERR_IS_OWNER	Already/still owning resource.
RTOS_ERR_NONE_WAITING	No task waiting/pending for that action/event.
RTOS_ERR_IO	Generic I/O err.
RTOS_ERR_IO_FATAL	Generic I/O fatal err.
RTOS_ERR_TX	Generic TX err.
RTOS_ERR_RX	Generic RX err.
RTOS_ERR_RX_OVERRUN	Overrun error, in RX.
RTOS_ERR_OS_SCHED_LOCKED	Operation cannot be done when the scheduler is locked.
RTOS_ERR_OS_TASK_WAITING	Operation not allowed when tasks are waiting/pending on OS object.
RTOS_ERR_OS_TASK_SUSPENDED	Task is still suspended.
RTOS_ERR_OS_ILLEGAL_RUN_TIME	Operation not allowed after call to OSStart.
RTOS_ERR_OS_OBJ_DEL	Object has been deleted during pend.
RTOS_ERR_OS	Generic OS err.
RTOS_ERR_SHELL_CMD_EXEC	Error when shell executed command.
RTOS_ERR_CMD_EMPTY	Shell command is empty.
RTOS_ERR_TIME_INVALID	Time is invalid.
RTOS_ERR_SIZE_INVALID	Size is invalid.
RTOS_ERR_NAME_INVALID	Name contains illegal characters.
RTOS_ERR_ECC_CORR	Correctable ECC error.
RTOS_ERR_ECC_CRITICAL_CORR	Critical correctable ECC error.
RTOS_ERR_ECC_UNCORR	Uncorrectable ECC error.
RTOS_ERR_ENTRY_PARENT_NOT_DIR	Entry parent must be a directory.
RTOS_ERR_ENTRY_ROOT_DIR	Operation not allowed on root directory.
RTOS_ERR_ENTRY_MAX_DEPTH_EXCEEDED	Maximum directory tree depth exceeded.
RTOS_ERR_ENTRY_OPENED	Operation not allowed on opened entry.
RTOS_ERR_ENTRY_CLOSED	Operation not allowed on closed entry.
RTOS_ERR_FILE_ACCESS_MODE_INVALID	File access mode is invalid.
RTOS_ERR_FILE_ERR_STATE	Operation cannot be performed on a file in error state.
RTOS_ERR_DIR_FULL	Directory is full.
RTOS_ERR_DIR_NOT_EMPTY	Directory contains files or sub-directories.
RTOS_ERR_WRK_DIR_CLOSED	Working directory is closed.
RTOS_ERR_VOL_FMT_INVALID	Volume format is invalid.
RTOS_ERR_VOL_CORRUPTED	Volume metadata is corrupted.
RTOS_ERR_VOL_OPENED	Operation not allowed on opened volume.
RTOS_ERR_VOL_CLOSED	Operation not allowed on closed volume.
RTOS_ERR_VOL_FULL	Volume is full.
RTOS_ERR_PARTITION_INVALID	Partition is invalid.
RTOS_ERR_PARTITION_MAX_EXCEEDED	Maximum partition count exceeded.
RTOS_ERR_BLK_DEV_FMT_INCOMPATIBLE	Block device low-level format is incompatible with user configuration.
RTOS_ERR_BLK_DEV_FMT_INVALID	Block device low-level format is invalid.

Error code	String associated
RTOS_ERR_BLK_DEV_CORRUPTED	Block device metadata is corrupted.
RTOS_ERR_BLK_DEV_OPENED	Block device is opened.
RTOS_ERR_BLK_DEV_CLOSED	Block device is closed.
RTOS_ERR_DEV_ALLOC	Device allocation failed.
RTOS_ERR_CONFIG_ALLOC	Configuration allocation failed.
RTOS_ERR_IF_ALLOC	Interface allocation failed.
RTOS_ERR_IF_ALT_ALLOC	Interface allocation failed.
RTOS_ERR_IF_GRP_ALLOC	Interface group allocation failed.
RTOS_ERR_FNCT_ALLOC	No more class function in this config.
RTOS_ERR_EP_ALLOC	USB core unable to allocate endpoint.
RTOS_ERR_EP_NONE_AVAIL	Physical endpoint/pipe NOT available.
RTOS_ERR_URB_ALLOC	USB core unable to allocate URB.
RTOS_ERR_DRV_EP_ALLOC	USB driver/controller unable to allocate endpoint.
RTOS_ERR_DRV_URB_ALLOC	USB driver/controller unable to allocate URB.
RTOS_ERR_CLASS_INSTANCE_ALLOC	Unable to allocate class instance.
RTOS_ERR_SUBCLASS_INSTANCE_ALLOC	Unable to allocate subclass instance.
RTOS_ERR_AOAP_FNCT_ALLOC	Unable to allocate AOAP class function instance.
RTOS_ERR_CDC_FNCT_ALLOC	Unable to allocate CDC class function instance.
RTOS_ERR_CDC_ACM_FNCT_ALLOC	Unable to allocate CDC-ACM subclass function instance.
RTOS_ERR_USB2SER_FNCT_ALLOC	Unable to allocate USB2Ser class function instance.
RTOS_ERR_HID_FNCT_ALLOC	Unable to allocate HID class function instance.
RTOS_ERR_MSC_FNCT_ALLOC	Unable to allocate MSC class function instance.
RTOS_ERR_INVALID_DEV_STATE	Invalid device state.
RTOS_ERR_INVALID_CLASS_STATE	Invalid class state.
RTOS_ERR_INVALID_EP_STATE	Invalid endpoint state.
RTOS_ERR_DEV_HANDLE_HAS_CHANGED	Handle has changed since providing it to caller.
RTOS_ERR_INVALID_DESC	Descriptor content is invalid.
RTOS_ERR_OVERFLOW_DESC	Descriptor's size is bigger than buffer size.
RTOS_ERR_INVALID_DEV_SPD	Invalid device speed.
RTOS_ERR_DEV_CONN_DECLINED	Connection for device was declined by app.
RTOS_ERR_DEV_SUSPEND	Operation failed because device is suspended.
RTOS_ERR_EP_INVALID	Invalid endpoint (address, type or direction).
RTOS_ERR_EP_BW_NOT_AVAIL	Not enough bandwidth available to open endpoint.
RTOS_ERR_EP_QUEUING	Unable to queue transfer on endpoint.
RTOS_ERR_EP_STALL	Endpoint was/is stalled.
RTOS_ERR_CLASS_DRV_NOT_FOUND	No class driver found.
RTOS_ERR_SUBCLASS_DRV_NOT_FOUND	No subclass driver found.
RTOS_ERR_USB2SER_FLOW_CTRL_EN	Operation not allowed when flow control enabled.
RTOS_ERR_NET_INVALID_ADDR_SRC	Address source not found to send data.
RTOS_ERR_NET_IF_LINK_DOWN	The interface link is down.
RTOS_ERR_NET_RETRY_MAX	The maximum number of retry was reached.

Error code	String associated
RTOS_ERR_NET_ADDR_UNRESOLVED	The stack was unable to resolved the IP address.
RTOS_ERR_NET_ICMP_ECHO_REPLY_DATA_CMP	Data received in echo reply is not the same as the data sent.
RTOS_ERR_NET_NEXT_HOP	The stack was unable to found a valid next hop to send the packet to.
RTOS_ERR_NET_INVALID_CONN	The socket connection is invalid.
RTOS_ERR_NET_CONN_CLOSE_RX	The stack received a connection closed from other half.
RTOS_ERR_NET_CONN_CLOSED_FAULT	The connection was closed abruptly by other half.
RTOS_ERR_NET_OP_IN_PROGRESS	The current operation is still in progress.
RTOS_ERR_NET_SOCK_CLOSED	The operation cannot be done because the socket is closed.
RTOS_ERR_NET_STR_ADDR_INVALID	The string address is in an invalid format.
RTOS_ERR_NET_PHY_TIMEOUT_AUTO_NEG	The PHY auto-negotiation timed out.
RTOS_ERR_MQTT_MSG_FAIL	Message failed to complete correctly.
RTOS_ERR_MQTTc_QoS_LEVEL_NOT_GRANTED	Broker failed to grant QoS level requested.

Segger Systemview

SEGGER SystemView

Trace Tools: SEGGER SystemView

SystemView consists of target-resident code that implements the Trace Recorder and a host application that displays and analyzes the Trace.

The image below shows the host application displaying the events table and the timeline for a typical Micrium OS Kernel-based application. These are two of the most important views offered by the tool.

This section will guide you through the steps necessary to setup SEGGER SystemView as the Trace Recorder tool for Micrium OS Kernel.

In particular, this section will describe the following steps:

- Step 1: Including the Trace Recorder
- Step 2: Configuring the Trace Recorder
- Step 3: Initializing the Trace Recorder
- Step 4: Starting the Trace Recorder

For more information on SystemView, including how to analyze the trace, see the SystemView Documentation from SEGGER's website at https://www.segger.com/systemview.html

Including The Trace Recorder

Including the Trace Recorder

Including the SystemView Trace Recorder Files

Micrium OS Kernel includes the Trace Recorder code from SEGGER SystemView with support for ARM Cortex-M devices.

To include this Trace recorder in your project simply include the entire SystemView folder as illustrated in Figure - SystemView files in the *Including the Trace Recorder* page.

Figure SystemView files

The files in this folder can be described as follows:

Sample/MicriumOSKernel/Config

The configuration file templates to configure the RTT channel and SystemView settings.

Sample/MicriumOSKernel/Config

(4) The configuration file template to configure platform related settings that are usually set once.

(5) The SystemView port for Micrium OS Kernel.

(6) The SystemView code that implements the trace recorder and the communication channel via RTT.

Note: In case you want to get the latest recorder code with potentially support for more CPUs, you can download the code directly from SEGGER at https://www.segger.com/systemview.html *and extract the contents to the folder* Micrium/Tools/SystemView

Configuring the Compiler's Include Paths

Configure your compiler with the following include paths:

```
$/util/third_party/segger/systemview/Config

$/util/third_party/segger/systemview/SEGGER
```

Including Header Files

From the application level code where you intend to initialize the Trace Recorder insert the following line of code:

```
#include  <rtos/kernel/include/os_trace.h>
```

Configuring The Trace Recorder

Configuring the Trace Recorder

os_cfg.h

From this file, you can enable and disable the trace recorder.

Listing - os_cfg.h

```
#define OS_CFG_TRACE_EN              DEF_ENABLED
```

os_trace_cfg.h

From this file, you can enable and disable the recording of the beginning and end of each of the Micrium OS Kernel API function calls. If not enabled, the trace will only have interrupt events and the Micrium OS Kernel scheduler events.

You can also configure the maximum number of tasks and other kernel objects to include in the trace. The code below shows an example:

Listing - os_cfg_trace.h

```
#define OS_CFG_TRACE_API_ENTER_EN       DEF_ENABLED

#define OS_CFG_TRACE_API_EXIT_EN        DEF_ENABLED

#define OS_CFG_TRACE_MAX_TASK              32u

#define OS_CFG_TRACE_MAX_RESOURCES         256u
```

SEGGER_RTT_Conf.h

From this file, you can configure the maximum number of bytes for the ring buffer used to stream the trace data up to the host application running on the PC. Low numbers will result in less data footprint at the expense of possible dropped events due to overflows during the communication. The code below shows an example:

Listing - SEGGER_RTT_Conf.h

```
#define BUFFER_SIZE_UP               4096u
```

Note: The file SEGGER_RTT_Conf.h *contains other settings that are beyond the scope of this document. For more information on this configuration file refer to the* SEGGER SystemView Documentation.

SEGGER_SYSVIEW_Conf.h

From this file, you can configure the maximum number of bytes for the ring buffer used to stream the trace data up to the host application running on the PC. Low numbers will result in less data footprint at the expense of possible dropped events due to overflows during the communication. The code below shows an example:

Listing - SEGGER_SYSVIEW_Conf.h

```
#define SEGGER_SYSVIEW_RTT_BUFFER_SIZE      4096u
```

From this file you also get to configure the way the trace recorder will get a system timestamp and the currently active interrupt ID. This file already has that implemented for the ARM Cortex-M devices. If your device is not an ARM Cortex-M, then this is the place to configure those two things.

Note: The file SEGGER_SYSVIEW_Conf.h *contains other settings that are beyond the scope of this document. For more information on this configuration file refer to the* SEGGER SystemView Documentation.

SEGGER_SYSVIEW_Config_MicriumOSKernel.c

From this file, you can configure the BSP function that returns the system clock in Hertz. For example:

Listing - SEGGER_SYSVIEW_Config_MicriumOSKernel.c - System Clock

```
#define SYSVIEW_TIMESTAMP_FREQ (CPU_TS_TmrFreqGet(&local_err))

#define SYSVIEW_CPU_FREQ       (CPU_TS_TmrFreqGet(&local_err))
```

From this file you also get to configure the lowest RAM address where you intend to allocate the Trace buffer. For example:

Listing - SEGGER_SYSVIEW_Config_MicriumOSKernel.c - RAM Base Address

```
#define SYSVIEW_RAM_BASE            (0x20000000)
```

And finally, from this file you get to register each of the interrupts you are interested in. By default, the interrupts will be displayed in the Analyzer with their vector number only. So, if you want to specify a name for the interrupts, then you can do so from this file. For example:

Listing - SEGGER_SYSVIEW_Config_MicriumOSKernel.c - Interrupt Names

```
static void _cbSendSystemDesc(void) {

  SEGGER_SYSVIEW_SendSysDesc("N="SYSVIEW_APP_NAME",D="SYSVIEW_DEVICE_NAME",O=Micrium OS Kernel");

  SEGGER_SYSVIEW_SendSysDesc("I#15=SysTick ISR");

  SEGGER_SYSVIEW_SendSysDesc("I#77=Ethernet ISR");

  SEGGER_SYSVIEW_SendSysDesc("I#83=USB OTG FS ISR");

  SEGGER_SYSVIEW_SendSysDesc("I#104=LCD TFT ISR");

  //

  SYSVIEW_SendResourceList();

}
```

Note: The file `SEGGER_SYSVIEW_Config_MicriumOSKernel.h` *contains other settings that are beyond the scope of this document. For more information on this configuration file refer to the* SEGGER SystemView Documentation.

Initializing The Trace Recorder

Initializing the Trace Recorder

Initialize the SystemView Trace Recorder anywhere after the BSP has been initialized.

The code below shows an example where it gets initialized from the Startup task:

Listing - Initializing the SystemView Trace Recorder

```
#include  <rtos/kernel/include/os_trace.h>

.

.

.

static  void  Ex_MainStartTask (void  *p_arg)

{

    PP_UNUSED_PARAM(p_arg);                              /* Prevent compiler warning.              */

    BSP_TickInit();                                      /* Initialize Kernel tick source.         */

    .

    .

    .

#if (OS_CFG_TRACE_EN == DEF_ENABLED)

    OS_TRACE_INIT();                                     /* Initialize the Trace recorder.         */

#endif
```

Starting The Trace Recorder

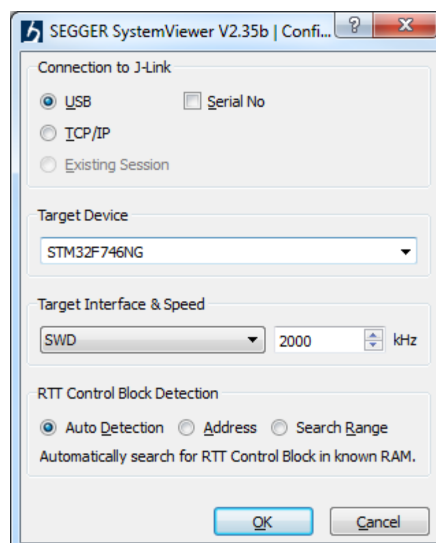
Starting the Trace Recorder

Starting the SystemView Trace Recorder

To start recording you need the host application, ***SystemViewer***, which can be downloaded from the following link: https://www.segger.com/systemview.html

The host application will connect to the embedded target via an on board or external J-Link and will stream all trace data up to the PC.

To start a recording open the SystemViewer host application and press the ***F5*** key or click ***Target -> Start Recording*** and you will be presented with the dialog window shown below.

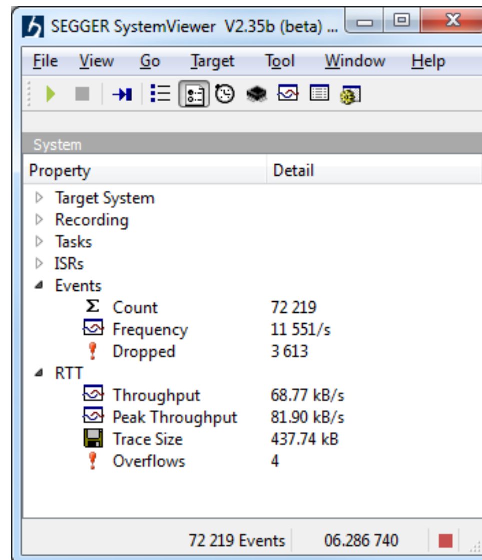


Before pressing the ***OK*** button, make sure that you have confirmed the following settings:

- The ***Target Device*** name matches your embedded target and the one configured by other tools using the same J-Link session (e.g., Debugger and/or µC/Probe).
- The ***Target Interface & Speed*** match the ones supported by your embedded target and the ones configured by other tools using the same J-Link session (e.g., Debugger and/or µC/Probe).
- The ***RTT Control Block Detection*** setting makes an attempt to locate the symbol `_SEGGER_RTT` where the trace data is stored using one of three methods: automatically or by manually specifying the address and range of_SEGGER_RTT.

Recording Status

During the streaming process, you will be presented with a window similar to the one shown below.



Two categories in that window, ***Events*** and ***RTT***, will display any potential ***Dropped Events*** due to ***Overflows*** in the communication.

Overflow events occur when the SystemView RTT buffer in the embedded target is full.

This can happen due to the following reasons:

- J-Link is kept busy by another application using the same J-Link session and cannot read the data fast enough.
- The embedded target interface speed is too low to read the data fast enough.
- The embedded application generates too many events to fit into the buffer.

To prevent this:

- Minimize the interactions of the debugger with J-Link while the embedded target is running. (i.e., disable live watches)
- Select a higher interface speed in all instances connected to J-Link. (e.g., The debugger, µC/Probe and SystemViewer)
- Choose a larger buffer size for SystemView. (1 - 4 kByte)
- Run SystemViewer stand-alone without a debugger and/or µC/Probe.

Further Reading

For additional and more detailed coverage of SystemView refer to SEGGER SystemView Documentation.

Appendix A Internal Tasks

Appendix A - Internal Tasks

Micrium OS Internal Tasks

Micrium OS Internal tasks

Table - Micrium OS Internal tasks in the Appendix A - Internal Tasks page lists all the internal tasks automatically created by the Micrium OS along with their default priority. It is always possible to change the task priority after the module initialization.

Changing tasks priority must be done with care. Some tasks may have special requirements. Refer to the module's section of this user manual for more information.

Note that the default stack sizes have been deliberately set to a high value. It is recommended to configure them to a more appropriate size once your application is completed.

Table - Micrium OS Internal tasks

Task	Module	Description	Default priority	Default stack size	Change priority with...	Change stack configuration with...
Timer	Kernel	Handles timer-based operations	5	256	OS_ConfigureTmrTask()	OS_ConfigureTmrTask()
Stat	Kernel	Maintain Kernel usage statistics	6	256	OS_ConfigureStatTask()	OS_ConfigureStatTask()
Service	CANopen	Handles CAN frames for different CANopen communication objects	9	512	CANopen_ConfigureSvcTaskStk()	CANopen_SvcTaskPrioSet()
Hub	USB Host Core	Handles the USB Host Hub events	10	768	USBH_HUB_TaskPrioSet()	USBH_ConfigureHubTaskStk()
Scheduler	USB Host PBHCI	Handles Pipe-Based Host Controller Interface events	11	512	USBH_PBHCI_SchedTaskPrioSet()	USBH_PBHCI_ConfigureSchedTaskStk
SD Core	IO-SD	Handles SD events (card connection, disconnection, interrupts) and data transfers	14	512	SD_CoreTaskPrioSet()	SD_ConfigureCoreTaskStk()

Task	Module	Description	Default priority	Default stack size	Change priority with...	Change stack configuration with...
Net Core	Net Core	Handles timers and packet reception & deallocation	16	1024	Net_CoreTaskPrioSet()	Net_ConfigureCoreTaskStk()
Net Core WiFi*	Net Core	Handles third party drivers that require it	17	-		
HTTP_Client	HTTP Client Application		18	768	HTTPc_TaskPrioSet()	HTTPc_ConfigureTaskStk()
MQTT Client	MQTT Client Application		19	512	MQTTc_TaskPrioSet()	MQTTc_ConfigureTaskStk()
IPerf	IPerf Application	Handles test runs	20	512	IPerf_TaskPrioSet()	Iperf_ConfigureTaskStk()
Telnet Server	Telnet Session	Handles client's session	21	512	TELNETs_InstanceInit()	TELNETs_InstanceInit()
Telnet Server	Telnet Service	Handles new client request	22	256	TELNETs_InstanceInit()	TELNETs_InstanceInit()
TFTP Server	TFTP Server	Handles TFTP requests	23	512	TFTPs_TaskPrioSet()	TFTPs_ConfigureTaskStk()
Asynchronous	USB Host Core	Handles asynchronous events	25	512	USBH_AsyncTaskPrioSet()	USBH_ConfigureAsyncTaskStk()
Asynchronous	IO-SD	Handles SD data transfers	26	512	SD_AsyncTaskPrioSet()	SD_ConfigureAsyncTaskStk()
Media Poll	File System Storage	Handles removable media status polling	27	512	FSStorage_PollTaskPrioSet()	FSStorage_ConfigureMediaPollTaskStk()
Net Services	Net Core	Handles network services such as DHCP	28	512	Net_CoreSvcTaskPrioSet()	Net_ConfigureCoreSvcTaskStk()
Timer	USB Device HID class	Handles timer-based HID events	29	512	USBD_HID_TmrTaskPrioSet()	USBD_HID_ConfigureTmrTaskStk()

* The Net Core WiFi Interface task may be required by some WiFi driver implementations, as is the case with the Qualcomm QCA400x. When applicable, the driver will take care of setting the stack size, and will use the task priority mentioned above. Such task configuration can be overridden by the user application and passed to the NetIF_WiFi_Add() function.

Appendix B Shell Commands Description

Appendix B - Shell Commands Description

This appendix describes the commands available through Micrium OS Common's Shell module.

- help
- lsusb
- File System Commands
  - fs_cat
  - fs_cd
  - fs_cp
  - fs_date
  - fs_df
  - fs_ls
  - fs_lsblk
  - fs_mkdir
  - fs_mkfs
  - fs_mount
  - fs_mv
  - fs_od
  - fs_pwd
  - fs_rm
  - fs_rmdir
  - fs_touch
  - fs_umount
  - fs_wc
- Network Commands
  - ifconfig
  - net_ping
  - net_if_start
  - net_if_stop
  - net_if_restart
  - net_if_reset
  - net_if_set_mtu
  - net_if_buf_rx
  - net_if_buf_tx_l
  - net_if_buf_tx_s
  - net_ip_setup
  - net_route_add
  - net_route_remove
  - net_sock_accept
  - net_sock_listen
  - net_sock_bind
  - net_sock_open
  - net_sock_close
  - net_sock_connect
  - net_sock_rx
  - net_sock_tx
  - net_sock_opt_set_child
  - net_sock_mcast_join
  - net_sock_mcast_leave
  - net_wifi_create

- net_wifi_scan
- net_wifi_join
- net_wifi_leave
- net_wifi_peer
- sntp_help
- sntp_get

Help

help

Description

Displays a list of the available commands.

Requirements

Shell

Synopsis

```
help
```

Arguments

None.

Notes / Warnings

None.

Lsusb

lsusb

Description

Lists all the devices currently connected to the USB host. Also displays some information about the devices.

Requirements

USB Host core

Synopsis

```
lsusb [OPTION]
```

Arguments

Argument	Detail	Example
-s	Displays the device's strings, when available.	
-d #a [#b]	Displays extended information on device at address #a on host #b .	lsusb -d 1 , lsusb -d 4 1
-h	Displays information on the USB host setup.	
--help	Displays a description of the command.	

Notes / Warnings

None.

File System Commands

File System Commands

- fs_cat
- fs_cd
- fs_cp
- fs_date
- fs_df
- fs_ls
- fs_lsblk
- fs_mkdir
- fs_mkfs
- fs_mount
- fs_mv
- fs_od
- fs_pwd
- fs_rm
- fs_rmdir
- fs_touch
- fs_umount
- fs_wc

fs_cat

Description

Print file contents to the terminal output. File contents, in the ASCII character set.

Non-printable/non-space characters are transmitted as full stops (“periods”, character code 46).

For a more convenient display of binary files use `fs_od` .

Requirements

File System Core

Synopsis

```
fs_cat [file]
```

Arguments

Argument	Detail	Example
`file`	Path of file to print to terminal output.	`fs_cat hello.txt`
`-h`	Displays a description of the command.	`fs_cat -h`

Notes / Warnings

None.

fs_cd

Description

Change the working directory.

Requirements

File System Core

Synopsis

```
fs_cd [dir]
```

Arguments

Argument	Detail	Example
`dir`	Absolute directory path **or** path relative to current working directory.	`fs_cd root_dir`
`-h`	Displays a description of the command.	`fs_cd -h`

Notes / Warnings

The new working directory is formed in three steps:

1. If the argument dir begins with the path separator character (slash, '\') or a volume name, it will be interpreted as an absolute directory path and will become the preliminary working directory. Otherwise the preliminary working directory path is formed by the concatenation of the current working directory, a path separator character and dir.
2. The preliminary working directory path is then refined, from the first to last path component:
   1. If the component is a 'dot' component, it is removed
   2. If the component is a 'dot dot' component, and the preliminary working directory path is not NULL, the previous path component is removed. In any case, the 'dot dot' component is removed.
   3. Trailing path separator characters are removed, and multiple path separator characters are replaced by a single path separator character.

The volume is examined to determine whether the preliminary working directory exists. If it does, it becomes the new working directory. Otherwise, an error is output, and the working directory is unchanged.

fs_cp

Description

Copy a file.

Requirements

File System Core

Synopsis

```
fs_cp [source_file] [dest_file]
```

Arguments

Argument	Detail	Example
`source_file`	Source file path.	
`dest_file`	Destination file path.	`fs_cp source_file.txt dest_file.txt`
`-h`	Displays a description of the command.	`fs_cp -h`

Notes / Warnings

1. In the first form of this command, neither argument may be an existing directory. The contents of `source_file` will be copied to a file named `dest_file` located in the same directory as `source_file` .
2.

In the second form of this command, the first argument must not be an existing directory and the second argument must be an existing directory. The contents of `source_file` will be copied to a file with name formed by concatenating `dest_dir`, a path separator character and the final component of `source_file`.

fs_date

Description

Write the date and time to terminal output, or set the system date and time.

Requirements

file System Core

Synopsis

```
fs_date

fs_date [time]
```

Arguments

`time`

If specified, time to set, in the form `mmddhhmmccyy`:

field	range
1st mm	the month (1-12)
dd	the day (1-29, 30 or 31)
hh	the hour (0-23)
2nd mm	the minute (0-59)
ccyy	the year (1900 or larger)

Example: `fs_date 092123591989`

`-h`

Displays a description of the command.

Example: `fs_date -h`

Notes / Warnings

None.

fs_df

Description

Report disk free space.

Requirements

File System Core

Synopsis

```
fs_df

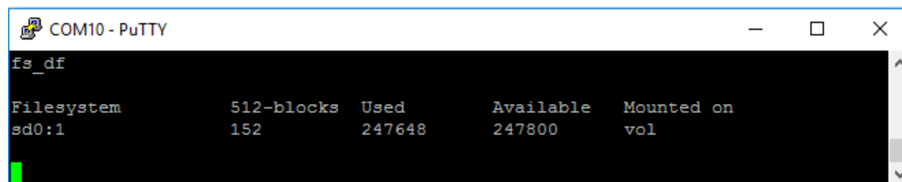
fs_df [vol]
```

Arguments

Argument	Detail	Example
vol	If specified, volume on which to report free space. Otherwise, information about all volumes will be output.	fs_df volum e1
-h	Displays a description of the command.	fs_df -h

Notes / Warnings

None.



fs_ls

Description

List directory contents.

Requirements

File System Core

Synopsis

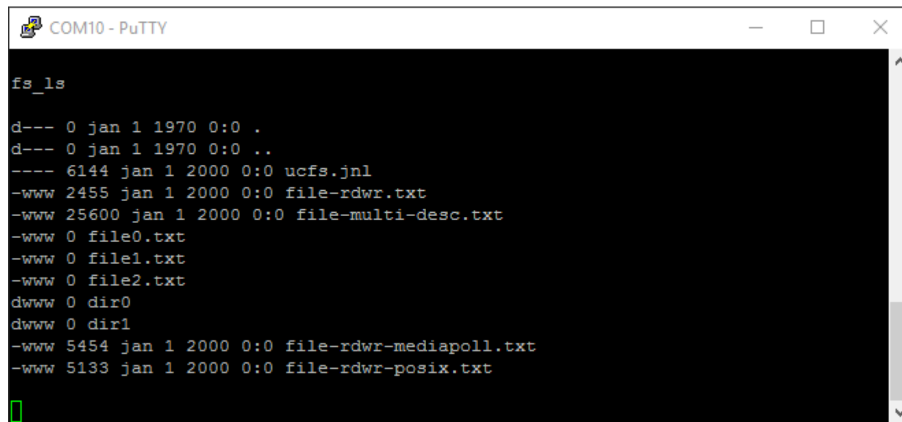
```
fs_ls
```

Arguments

None.

Notes / Warnings

The output resembles the output from the standard UNIX command ls -l. See the figure below.



fs_lsblk

Description

List all open block devices and display information about each partition composing the block device.

Requirements

File System Core

Synopsis

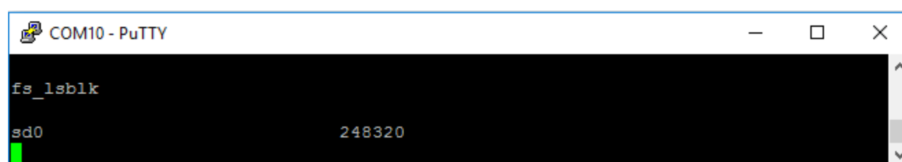
```
fs_lsblk
```

Arguments

None.

Notes / Warnings

The output resembles the output from the standard UNIX command lsblk -l. See the figure below.



fs_mkdir

Description

Make a directory.

Requirements

File System Core

Synopsis

```
fs_mkdir [dir]
```

Arguments

Argument	Detail	Example
dir	Directory path.	fs_mkdir source_dir
-h	Displays a description of the command.	fs_mkdir -h

Notes / Warnings

None.

fs_mkfs

Description

Format a volume.

Requirements

File System Core

Usages

```
fs_mkfs [vol]
```

Arguments

Argument	Detail	Example
vol	Volume name.	fs_mkfs volume1
-h	Displays a description of the command.	fs_mkfs -h

Notes / Warnings

None.

fs_mount

Description

Mount volume.

Requirements

File System Core

Usages

```
fs_mount [dev] [part] [vol]
```

Arguments

Argument	Detail	Example
dev	Device to mount.	
part	Partition number	
vol	Name which will be given to volume.	fs_mount sd0 1 volume1
-h	Displays a description of the command.	fs_mount -h

Notes / Warnings

None.

fs_mv

Description

Move files.

Requirements

File System Core

Synopsis

```
fs_mv [source_entry] [dest_entry]
```

Arguments

Argument	Detail	Example
source_entry	Source entry path.	
dest_entry	Destination entry path.	fs_mv src.txt dest.txt , fs_mv src_dir dest_dir
-h	Displays a description of the command.	fs_mv -h

Notes / Warnings

1. In the first form of this command, the second argument must not be an existing directory. The file source_entry will be renamed dest_entry .
2. In the second form of this command, the second argument must be an existing directory. source_entry will be renamed to an entry with name formed by concatenating dest_dir, a path separator character and the final component of source_entry .

In both forms, if source_entry is a directory, the entire directory tree rooted at source_entry will be copied and then deleted. Additionally, both source_entry and dest_entry or dest_dir must specify locations on the same volume.

fs_od

Description

Dump file contents to the terminal output.

Requirements

File System Core

Synopsis

```
fs_od [file]
```

Arguments

Argument	Detail	Example
file	Name of file to dump to terminal output.	fs_od file.txt
-h	Displays a description of the command.	fs_cat -h

Notes / Warnings

None.

fs_pwd

Description

Write to terminal output pathname of current working directory.

Requirements

File System Core

Synopsis

```
fs_pwd
```

Arguments

None.

Notes / Warnings

None.

```
COM10 - PuTTY

d--- 0 jan 1 1970 0:0 .
d--- 0 jan 1 1970 0:0 ..
---- 6144 jan 1 2000 0:0 ucfs.jnl
-www 2455 jan 1 2000 0:0 file-rdwr.txt
-www 25600 jan 1 2000 0:0 file-multi-desc.txt
-www 0 file0.txt
-www 0 file1.txt
-www 0 file2.txt
dwww 0 dir0
dwww 0 dir1
-www 5454 jan 1 2000 0:0 file-rdwr-mediapoll.txt
-www 1034 feb 23 2017 15:13 file-rdwr-posix.txt
dwww 0 feb 23 2017 15:12 System Volume Information

fs_od file-rdwr-posix.txt

00000000 03020120 07060504 0B0A0908 0F0E0D0C 13121110 17161514 1B1A1918 1F1E1D1C      ................................
00000020 23222120 27262524 2B2A2928 2F2E2D2C 33323130 37363534 3B3A3938 3F3E3D3C       !"#$%&'()*+,-./0123456789:;<=>?
00000040 43424140 47464544 4B4A4948 4F4E4D4C 53525150 57565554 5B5A5958 5F5E5D5C      @ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
00000060 63626160 67666564 6B6A6968 6F6E6D6C 73727170 77767574 7B7A7978 7F7E7D7C      `abcdefghijklmnopqrstuvwxyz{|}~.
00000080 83828180 87868584 8B8A8988 8F8E8D8C 93929190 97969594 9B9A9998 9F9E9D9C      ................................
000000A0 A3A2A1A0 A7A6A5A4 ABAAA9A8 AFAEADAC B3B2B1B0 B7B6B5B4 BBBAB9B8 BFBEBDBC      ................................
000000C0 C3C2C1C0 C7C6C5C4 CBCAC9C8 CFCECDCC D3D2D1D0 D7D6D5D4 DBDAD9D8 DFDEDDDC      ................................
000000E0 E3E2E1E0 E7E6E5E4 EBEAE9E8 EFEEEDEC F3F2F1F0 F7F6F5F4 FBFAF9F8 FFFEFDFC      ................................
00000100 03020120 07060504 0B0A0908 0F0E0D0C 13121110 17161514 1B1A1918 1F1E1D1C      ................................
00000120 23222120 27262524 2B2A2928 2F2E2D2C 33323130 37363534 3B3A3938 3F3E3D3C       !"#$%&'()*+,-./0123456789:;<=>?
00000140 43424140 47464544 4B4A4948 4F4E4D4C 53525150 57565554 5B5A5958 5F5E5D5C      @ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
00000160 63626160 67666564 6B6A6968 6F6E6D6C 73727170 77767574 7B7A7978 7F7E7D7C      `abcdefghijklmnopqrstuvwxyz{|}~.
00000180 83828180 87868584 8B8A8988 8F8E8D8C 93929190 97969594 9B9A9998 9F9E9D9C      ................................
000001A0 A3A2A1A0 A7A6A5A4 ABAAA9A8 AFAEADAC B3B2B1B0 B7B6B5B4 BBBAB9B8 BFBEBDBC      ................................
000001C0 C3C2C1C0 C7C6C5C4 CBCAC9C8 CFCECDCC D3D2D1D0 D7D6D5D4 DBDAD9D8 DFDEDDDC      ................................
000001E0 E3E2E1E0 E7E6E5E4 EBEAE9E8 EFEEEDEC F3F2F1F0 F7F6F5F4 FBFAF9F8 FFFEFDFC      ................................
00000200 03020120 07060504 0B0A0908 0F0E0D0C 13121110 17161514 1B1A1918 1F1E1D1C      ................................
00000220 23222120 27262524 2B2A2928 2F2E2D2C 33323130 37363534 3B3A3938 3F3E3D3C       !"#$%&'()*+,-./0123456789:;<=>?
00000240 43424140 47464544 4B4A4948 4F4E4D4C 53525150 57565554 5B5A5958 5F5E5D5C      @ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
00000260 63626160 67666564 6B6A6968 6F6E6D6C 73727170 77767574 7B7A7978 7F7E7D7C      `abcdefghijklmnopqrstuvwxyz{|}~.
00000280 83828180 87868584 8B8A8988 8F8E8D8C 93929190 97969594 9B9A9998 9F9E9D9C      ................................
000002A0 A3A2A1A0 A7A6A5A4 ABAAA9A8 AFAEADAC B3B2B1B0 B7B6B5B4 BBBAB9B8 BFBEBDBC      ................................
000002C0 C3C2C1C0 C7C6C5C4 CBCAC9C8 CFCECDCC D3D2D1D0 D7D6D5D4 DBDAD9D8 DFDEDDDC      ................................
000002E0 E3E2E1E0 E7E6E5E4 EBEAE9E8 EFEEEDEC F3F2F1F0 F7F6F5F4 FBFAF9F8 FFFEFDFC      ................................
00000300 03020120 07060504 0B0A0908 0F0E0D0C 13121110 17161514 1B1A1918 1F1E1D1C      ................................
00000320 23222120 27262524 2B2A2928 2F2E2D2C 33323130 37363534 3B3A3938 3F3E3D3C       !"#$%&'()*+,-./0123456789:;<=>?
00000340 43424140 47464544 4B4A4948 4F4E4D4C 53525150 57565554 5B5A5958 5F5E5D5C      @ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
00000360 63626160 67666564 6B6A6968 6F6E6D6C 73727170 77767574 7B7A7978 7F7E7D7C      `abcdefghijklmnopqrstuvwxyz{|}~.
00000380 83828180 87868584 8B8A8988 8F8E8D8C 93929190 97969594 9B9A9998 9F9E9D9C      ................................
000003A0 A3A2A1A0 A7A6A5A4 ABAAA9A8 AFAEADAC B3B2B1B0 B7B6B5B4 BBBAB9B8 BFBEBDBC      ................................
000003C0 C3C2C1C0 C7C6C5C4 CBCAC9C8 CFCECDCC D3D2D1D0 D7D6D5D4 DBDAD9D8 DFDEDDDC      ................................
000003E0 E3E2E1E0 E7E6E5E4 EBEAE9E8 EFEEEDEC F3F2F1F0 F7F6F5F4 FBFAF9F8 FFFEFDFC      ................................
00000400 20202020 20202020 00002020
```

fs_rm

Description

Remove a file.

Requirements

File System Core

Synopsis

```
fs_rm [file]
```

Arguments

Argument	Detail	Example
file	File name.	fs_rm file.txt
-h	Displays a description of the command.	fs_rm -h

Notes / Warnings

None.

fs_rm

Description

Remove a file.

Requirements

File System Core

Synopsis

```
fs_rm [file]
```

Arguments

Argument	Detail	Example
file	File name.	fs_rm file.txt
-h	Displays a description of the command.	fs_rm -h

Notes / Warnings

None.

fs_rmdir

Description

Remove a directory.

Requirements

File System Core

Synopsis

```
fs_rmdir [dir]
```

Arguments

Argument	Detail	Example
dir	Directory name.	fs_rmdir dir
-h	Displays a description of the command.	fs_rmdir -h

Notes / Warnings

None.

fs_touch

Description

Change file modification time.

Requirements

File System Core

Synopsis

```
fs_touch [file]
```

Arguments

Argument	Detail	Example
file	File name.	fs_touch file.txt
-h	Displays a description of the command.	fs_touch -h

Notes / Warnings

The file modification time is set to the current time.

fs_umount

Description

Unount volume.

Requirements

File System Core

Synopsis

```
fs_umount [vol]
```

Arguments

Argument	Detail	Example
vol	Volume to unmount.	fs_umount volume1
-h	Displays a description of the command.	fs_umount -h

Notes / Warnings

None.

fs_wc

Description

Determine the number of newlines, words and bytes in a file.

Requirements

File System Core

Synopsis

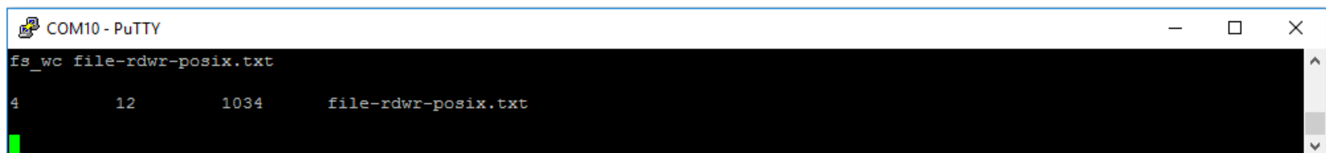
```
fs_wc [file]
```

Arguments

Argument	Detail	Example
file	File to examine.	fs_wc file.txt
-h	Displays a description of the command.	fs_wc -h

Notes/Warnings

None.



Network Commands

Network Commands

- ifconfig
- net_ping
- net_if_start
- net_if_stop
- net_if_restart
- net_if_reset
- net_if_set_mtu
- net_if_buf_rx
- net_if_buf_tx_l
- net_if_buf_tx_s
- net_ip_setup
- net_route_add
- net_route_remove
- net_sock_accept
- net_sock_listen
- net_sock_bind
- net_sock_open
- net_sock_close
- net_sock_connect
- net_sock_rx
- net_sock_tx
- net_sock_opt_set_child
- net_sock_mcast_join
- net_sock_mcast_leave
- net_wifi_create
- net_wifi_scan
- net_wifi_join
- net_wifi_leave
- net_wifi_peer
- sntp_help
- sntp_get

ifconfig

Description

Prints out interfaces information

Requirements

Network Module

Synopsis

```
ifconfig
```

Arguments

None.

Notes / Warnings

None.

net_ping

Description

Issue a ICMP Echo Request from the target. (IPv4 or IPv6)

Requirements

Network Module

Synopsis

```
net_ping ip_addr
```

Arguments

`ip_addr` Address of the remote host.

Notes / Warnings

None.

net_if_start

Description

Start an interface.

Requirements

Network Module

Synopsis

```
net_if_start -i if_nbr
```

Arguments

`-i if_nbr` Specifies the interface to start.

Notes / Warnings

None.

net_if_stop

Description

Stop an interface.

Requirements

Network Module

Synopsis

```
net_if_stop -i if_nbr
```

Arguments

`-i if_nbr` Specifies the which interface to stop.

Notes / Warnings

None.

net_if_restart

Description

Restart a network interface.

Requirements

Network Module

Synopsis

```
net_if_restart -i if_nbr [-t wait_time ]
```

Arguments

Argument	Detail
-i if_nbr	Interface number to restart.
-t wait_time	Time to wait after interface stop, before issuing the interface start command, in seconds. Default = 10s.

Notes / Warnings

None.

net_if_reset

Description

Reset interface(s); remove all configured addresses, IPv4 or IPv6 or both, on specified interface.

Requirements

Network Module

Synopsis

```
net_if_reset [-i if_nbr] [-4|-6]
```

Arguments

Argument	Detail
-i if_nbr	Specifies on which interface the reset will occur. If no interface is specified, all existing interfaces will be reset.
-4	Specifies to clear only the IPv4 addresses of the interface.
-6	Specifies to clear only the IPv6 addresses of the interface.

Notes / Warnings

None.

net_if_set_mtu

Description

Configure MTU of given Interface

Requirements

Network Module

Synopsis

```
net_if_set_mtu -i if_nbr -M mtu
```

Arguments

Argument	Detail
-i if_nbr	Specifies on which interface to set the MTU.
-M mtu	Specifies the new MTU value to configure.

Notes / Warnings

None.

net_if_buf_rx

Description

Get Rx large Buffer statistics

Requirements

Network Module

Synopsis

```
net_if_buf_rx [-i if_nbr]
```

Arguments

-i if_nbr Specifies the interface for which to return the number of large Rx buffers

Notes / Warnings

None.

net_if_buf_tx_l

Description

Get Tx large Buffer statistics

Requirements

Network Module

Synopsis

```
net_if_buf_tx_l [-i interface_nbr]
```

Arguments

-i [interface_nbr] Specifies the interface for which to return the number of large Tx buffers

Notes / Warnings

None.

net_if_buf_tx_s

Description

Get Tx small Buffer statistics

Requirements

Network Module

Synopsis

```
net_if_buf_tx_s [-i interface_nbr]
```

Arguments

`-i [interface_nbr]` , Specifies the interface for which to return the number of small Tx buffers

Notes / Warnings

None.

net_ip_setup

Description

Configure an IPv4 address.

Requirements

Network Module

Synopsis

```
net_ip_setup -i if_nbr -a ip_addr -m mask
```

Arguments

Argument	Detail
`-i if_nbr`	Interface number to setup
`- a ip_addr`	IP address to assign to the interface
`- m mask`	Subnet mask to assign to the interface

Notes / Warnings

None.

net_route_add

Description

Add IP address route

Requirements

Network Module

Synopsis

```
net_route_add -i if_nbr [-4 ip_addr mask gateway] [-6 ip_addr prefix_len]
```

Arguments

Argument	Detail
-i if_nbr	Specifies on which interface to add the route
-4 ip_add mask gateway	IPv4 addresses.
-6 ip_addr prefix_len	IPv6 addresses.

Notes / Warnings

None.

net_route_remove

Description

Remove a route (previously added using net route add).

Requirements

Network Module

Synopsis

```
net_route_remove {-i [if_nbr]} {-4 [IP Address] [Mask] [Gateway]} {-6 [IP Address] [Prefix Length]}
```

Arguments

Argument	Detail
-i [Interface number] ,	Specified on which interface the reset will occur. If no interface is specified, all existing interfaces will be reset.
-4 [IP_address Mask Gateway] ,	IPv4 addresses.
-6 [IP_address Prefix_len] ,	IPv6 addresses.

Notes/Warnings

None.

net_sock_accept

Description

Accept connection on a socket

Requirements

Network Module

Synopsis

```
net_sock_accept -s sock_id
```

Arguments

-s sock_id Specifies the Socket descriptor/handle identifier of listen socket

Notes / Warnings

None.

net_sock_listen

Description

Set socket to listen for connection requests.

Requirements

Network Module

Synopsis

```
net_sock_listen -s sock_id -q queue_size
```

Arguments

Argument	Detail
-s sock_id	Socket descriptor/handle identifier of socket to listen.
-q queue_size	Maximum number of connection requests to accept & queue on listen socket.

Notes / Warnings

None.

net_sock_bind

Description

Bind a network socket to a local address.

Requirements

Network Module

Synopsis

```
net_sock_bind -s sock_id -p port_nbr -f family
```

Arguments

Argument	Detail
-s sock_id	Specifies the Socket descriptor/handle identifier of listen socket
-p port_nbr	Port to bind on.
-f family	IP address family to use (4 or 6)

Notes / Warnings

None.

net_sock_open

Description

Open a network socket.

Requirements

Network Module

Synopsis

```
net_sock_open -f family -t protocol_type
```

Arguments

Argument	Detail
-f family	IP address family to use: IPv4 (4) or IPv6 (6)
-t protocol_type	Protocol type: stream (s) or datagram (d)

Notes / Warnings

None.

net_sock_close

Description

Close a network socket.

Requirements

Network Module

Synopsis

```
net_sock_close -s sock_id
```

Arguments

-s sock_id Specifies the Socket descriptor/handle identifier of the socket to close

Notes / Warnings

None.

net_sock_connect

Description

Connect a network socket to a remote host.

Requirements

Network Module

Synopsis

```
net_sock_conn -s sock_id -p port_nbr ip_addr
```

Arguments

Argument	Detail
-s sock_id	Specifies the Socket descriptor/handle identifier of listen socket
-p port_nbr	Port number to which to connect
ip_addr	IP Address to which to connect

Notes / Warnings

None.

net_sock_rx

Description

Receive data on a socket

Requirements

Network Module

Synopsis

```
net_sock_rx -s sock_id -l data_len -F outfmt [ -a address ] [-p port ] [-r retry ]
```

Arguments

Argument	Detail
-s sock_id	Socket ID.
-l data_len	Data length to receive
-F outfmt	Output format Hexadecimal (h) or Char/string (s)
-a address	IP address to receive from (optional)
-p port	Port number (optional; required if -a is specified)
-r retry	Number of retry attempts after timeout

Notes / Warnings

None.

net_sock_tx

Description

Receive data on a socket

Requirements

Network Module

Synopsis

```
net_sock_tx -s sock_id -l data_len -d data
```

Arguments

Argument	Detail
-s sock_id	Socket ID
-l data_len	Data length to send
-d data	Data to transmit

Notes / Warnings

None.

net_sock_opt_set_child

Description

Configure socket's child connection queue size

Requirements

Network Module

Synopsis

```
net_sock_opt_set_child -s sock_id -v queue_size
```

Arguments

Argument	Detail
-s sock_id	Socket ID.
-v queue_size	Desired child connection queue size

Notes / Warnings

None.

net_sock_mcast_join

Description

Join a multicast group

Requirements

Network Module

Synopsis

```
net_sock_mcast_join -i if_nbr -a ip_address
```

Arguments

Argument	Detail
-i if_nbr	Interface number onto which to join the multicast group.
-a ip_address	IP address of the multicast group

Notes / Warnings

None.

net_sock_mcast_leave

Description

Leave a multicast group

Requirements

Network Module

Synopsis

```
net_sock_mcast_leave -i if_nbr -a ip_address
```

Arguments

Argument	Detail
-i if_nbr	Interface number from which to leave the multicast group.
-a ip_address	IP address of the multicast group

Notes / Warnings

None.

net_wifi_create

Description

Create a WiFi Network.

Requirements

Network Module

Synopsis

```
net_wifi_create ssid -p pwd -c channel -i if_nbr -t net_type -s sec_type
```

Arguments

Argument	Detail
ssid	SSID of the Wi-Fi network to join
-p pwd	Password to use to access the WiFi network
-c channel	Wi-Fi channel
-i if_nbr	Interface number on which to create the Wi-Fi network
-t net_type	Network Type (infra or adhoc)
-s sec_type	Network security type (open, wep, wpa or wpa2)

Notes / Warnings

1. This command will failed if the interface has already joined or created an network. In this case, use net_wifi_leave command before.
2. The argument `[-p password]` is not applicable if the security_type specified is 'open'.

net_wifi_scan

Description

Scan for available WiFi SSID.

Requirements

Network Module

Synopsis

```
net_wifi_scan [ssid] -i if_nbr [-c channel]
```

Arguments

Argument	Detail
ssid	SSID to find (opt.)
-i if_nbr	Interface number to use
-c channel	WiFi Channel to scan (opt.)

Notes / Warnings

None.

net_wifi_join

Description

Join a Wi-Fi Network.

Requirements

Network Module

Synopsis

```
net_wifi_join ssid -p pwd -i if_nbr -t net_type -s sec_type
```

Arguments

Argument	Detail
ssid	Wi-Fi network to join
-p pwd	Password to use to access the Wi-Fi network
-i if_nbr	Interface number on which to join the Wi-Fi network
-t net_type	Network Type (infra or adhoc)
-s sec_type	Network security type (open, wep, wpa or wpa2).

Notes / Warnings

1. This command will failed if the interface has already joined or created an network. In this case, use 1 command before.
2. The argument `[-p password]` is not applicable if the security_type specified is `open`.

net_wifi_leave

Description

Leave a WiFi Network.

Requirements

Network Module

Synopsis

```
net_wifi_leave -i if_nbr
```

Arguments

`-i if_nbr` Interface number of the Wi-Fi network to leave.

Notes / Warnings

None.

net_wifi_peer

Description

Output the peer information when acting as an Wi-Fi Access point.

Requirements

Network Module

Synopsis

```
net_wifi_peer -i if_nbr
```

Arguments

`-i if_nbr` Specified on which interface to use.

Notes / Warnings

This command is applicable only if a Wi-Fi network is created by the interface.

sntp_help

Description

Displays information on how to use the SNTP client related commands.

Requirements

SNTP Client

Synopsis

```
sntp_help
```

Arguments

None.

Notes / Warnings

None.

sntp_get

Description

Retrieves and displays current time from a remote NTP server.

Requirements

SNTP Client

Synopsis

```
sntp_get [OPTION]
```

Arguments

Argument	Detail	Example
`-6`	Test SNTP client using IPv6.	

Argument	Detail	Example
-4	Test SNTP client using IPv4.	
-d [hostname]	Retrieves time from hostname 0.pool.ntp.org or the one provided.	sntp_get -d, sntp_get -d 1.pool.ntp.org.

Notes / Warnings

None.

Copyright © 2023 Silicon Laboratories. All rights reserved.