RAIL

RAIL_BLE_Supports1Mbps
RAIL_BLE_Supports2MbpsNonViterbi
RAIL_BLE_Supports2MbpsViterbi
RAIL_BLE_Supports2Mbps
RAIL_BLE_SupportsAntennaSwitching
RAIL_BLE_SupportsCodedPhy
RAIL_BLE_SupportsCte
RAIL_BLE_SupportsIQSampling
RAIL_BLE_SupportsPhySwitchToRx
RAIL_BLE_SupportsQuuppa
RAIL_BLE_SupportsSignalIdentifier
RAIL_BLE_SupportsSimulscanPhy
RAIL_SupportsProtocolIEEE802154
RAIL_IEEE802154_SupportsCoexPhy
RAIL_SupportsIEEE802154Band2P4
RAIL_SupportsThermalProtection
RAIL_IEEE802154_SupportsRxChannelSwitching
RAIL_IEEE802154_SupportsCustom1Phy
RAIL_IEEE802154_SupportsFemPhy
RAIL_IEEE802154_SupportsCancelFramePendingLookup
RAIL_IEEE802154_SupportsEarlyFramePendingLookup
RAIL_IEEE802154_SupportsDualPaConfig
RAIL_IEEE802154_SupportsEEnhancedAck
RAIL_IEEE802154_SupportsEMultipurposeFrames
RAIL_IEEE802154_SupportsESubsetGB868
RAIL_IEEE802154_SupportsG4ByteCrc
RAIL_IEEE802154_SupportsGDynFec
RAIL_SupportsProtocolWiSUN
RAIL_IEEE802154_SupportsGModeSwitch
RAIL_IEEE802154_SupportsGSubsetGB868
RAIL_IEEE802154_SupportsGUnwhitenedRx
RAIL_IEEE802154_SupportsGUnwhitenedTx
RAIL_WMBUS_SupportsSimultaneousTCRx
RAIL_SupportsProtocolZWave
RAIL_ZWAVE_SupportsConcPhy
RAIL_ZWAVE_SupportsEnergyDetectPhy
RAIL_ZWAVE_SupportsRegionPti
RAIL_IEEE802154_SupportsSignalIdentifier
RAIL_SupportsFastRx2Rx
RAIL_SupportsCollisionDetection
RAIL_SupportsProtocolSidewalk
RAIL_SUPPORTS_DUAL_BAND
RAIL_FEAT_DUAL_BAND_RADIO

RAIL_SUPPORTS_2P4GHZ_BAND
RAIL_FEAT_2G4_RADIO
RAIL_SUPPORTS_SUBGHZ_BAND
RAIL_FEAT_SUBGIG_RADIO
RAIL_SUPPORTS_OFDM_PA
RAIL_SUPPORTS_ADDR_FILTER_ADDRESS_BIT_MASK
RAIL_SUPPORTS_ADDR_FILTER_MASK
RAIL_SUPPORTS_ALTERNATE_TX_POWER
RAIL_FEAT_ALTERNATE_POWER_TX_SUPPORTED
RAIL_SUPPORTS_ANTENNA_DIVERSITY
RAIL_FEAT_ANTENNA_DIVERSITY
RAIL_SUPPORTS_PATH_DIVERSITY
RAIL_SUPPORTS_CHANNEL_HOPPING
RAIL_FEAT_CHANNEL_HOPPING
RAIL_SUPPORTS_DUAL_SYNC_WORDS
RAIL_SUPPORTS_TX_TO_TX
RAIL_SUPPORTS_TX_REPEAT_START_TO_START
RAIL_SUPPORTS_EXTERNAL_THERMISTOR
RAIL_FEAT_EXTERNAL_THERMISTOR
RAIL_SUPPORTS_HFXO_COMPENSATION
RAIL_SUPPORTS_AUXADC
RAIL_SUPPORTS_PRECISION_LFRCO
RAIL_SUPPORTS_RADIO_ENTROPY
RAIL_SUPPORTS_RFSENSE_ENERGY_DETECTION
RAIL_SUPPORTS_RFSENSE_SELECTIVE_OOK
RAIL_FEAT_RFSENSE_SELECTIVE_OOK_MODE_SUPPORTED
RAIL_SUPPORTS_EFF
RAIL_SUPPORTS_PROTOCOL_BLE
RAIL_BLE_SUPPORTS_1MBPS_NON_VITERBI
RAIL_BLE_SUPPORTS_1MBPS_VITERBI
RAIL_BLE_SUPPORTS_1MBPS
RAIL_BLE_SUPPORTS_2MBPS_NON_VITERBI
RAIL_BLE_SUPPORTS_2MBPS_VITERBI
RAIL_BLE_SUPPORTS_2MBPS
RAIL_BLE_SUPPORTS_ANTENNA_SWITCHING
RAIL_BLE_SUPPORTS_CODED_PHY
RAIL_FEAT_BLE_CODED
RAIL_BLE_SUPPORTS_SIMULSCAN_PHY
RAIL_BLE_SUPPORTS_CTE
RAIL_BLE_SUPPORTS_QUUPPA
RAIL_BLE_SUPPORTS_IQ_SAMPLING
RAIL_BLE_SUPPORTS_AOX
RAIL_FEAT_BLE_AOX_SUPPORTED

- RAIL_SetFixedLength
- RAIL_ConfigChannels
- RAIL_GetChannelMetadata
- RAIL_IsValidChannel
- RAIL_PrepareChannel
- RAIL_GetChannel
- RAIL_GetChannelAlt
- RAIL_GetSymbolRate
- RAIL_GetBitRate
- RAIL_SetPaCTune
- RAIL_GetSyncWords
- RAIL_ConfigSyncWords
- RAIL_SETFIXEDLENGTH_INVALID
- RADIO_CONFIG_ENABLE_CONC_PHY
- RADIO_CONFIG_ENABLE_STACK_INFO

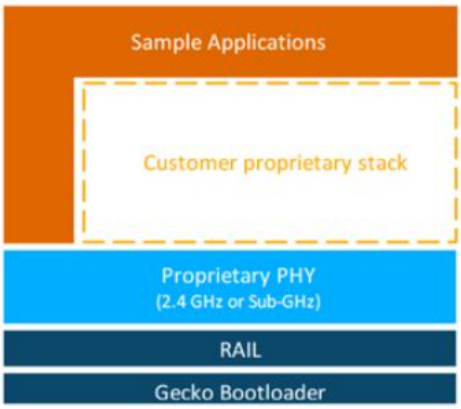
Receive

- RAIL_ScheduleRxConfig_t
  - start
  - startMode
  - end
  - endMode
  - rxTransitionEndSchedule
  - hardWindowEnd
- RAIL_RxPacketInfo_t
  - packetStatus
  - packetBytes
  - firstPortionBytes
  - firstPortionData
  - lastPortionData
  - filterMask
- RAIL_RxPacketDetails_t
  - timeReceived
  - crcPassed
  - isAck
  - rssi
  - lqi
  - syncWordId
  - subPhyId
  - antennaId
  - channelHoppingChannelIndex
  - channel
- Address Filtering
  - RAIL_AddrConfig_t
    - offsets

Developing with RAIL

Developing with Silicon Labs RAIL

Silicon Labs RAIL (Radio Abstraction Interface Layer) provides an intuitive and easily-customizable radio interface that is designed to support proprietary or standards-based wireless protocols. RAIL allows customers to adopt the latest RF technology without sacrificing their previous development investment. It also future-proofs the code migration to future EFR32 ICs. The unified radio software API abstracts the significant number of hardware registers and complexity of the lower-level radio block, allowing customers to focus on their proprietary wireless application development instead of mastering device-specific details. RAIL is delivered through the Proprietary Flex SDK component of the Gecko SDK Suite (GSDK).



Silicon Labs RAIL is the lowest layer for all networking stacks in Silicon Labs GSDK. RAIL supports a diverse set of radio configurations and functionality and is one of the key underlying technologies of Silicon Labs wireless products. The content on these pages is intended for developers who need to control their device's radio interface and performance through their embedded application.

For details about this release: Links to release notes are available on the silabs.com Gecko SDK page.

For Silicon Labs' proprietary product information: See the product pages on silabs.com.

For background about RAIL: RAIL Fundamentals is a good place to start.

To get started with development: See the Getting Started section to get started working with example applications.

If you are already in development: See the Developer's Guide or the PHY Development section for details or go directly to the API Reference.

Overview

Getting Started with RAIL - Overview

To get started, download the Simplicity Studio® 5 (SSv5) Development environment as described in the Simplicity Studio 5 User's Guide.

This will also prompt you to install the Gecko SDK (GSDK). The Radio Configurator tool and the RAIL library are installed with the GSDK as part of the Proprietary Flex package. The GSDK combines Silicon Labs wireless software development kits (SDKs) and Gecko Platform into a single, integrated package. The GSDK is your primary tool for developing in the Silicon Labs IoT Software ecosystem. All of Silicon Labs' stacks are written in-house to provide a seamless experience from silicon to tools, allowing you to unlock powerful features with ease, including:

- Abstraction of complex requirements like multiprotocol and pre-certification
- Industry-leading ability to support a large number of nodes
- Ultra-low power consumption
- Strong network reliability

Silicon Labs also helps future-proof your devices with over-the-air software and security updates, helping to minimize maintenance cost and improve your end user product experience.

Once you have downloaded Simplicity Studio and the GSDK, detailed instructions for using the RAIL examples and configuration tools are provided in the **Proprietary Flex Quick-Start Guide (QSG168)**.

Information about the distinguishing features of different EFR32 families that are most relevant to porting proprietary wireless applications between them is provided in **EFR32 Migration Guide for Proprietary Applications (AN1244)**. This is also helpful when selecting an initial target platform for proprietary wireless solutions.

Note: The recommended method to get started with the GSDK is to first install Simplicity Studio 5, which will set up your development environment and walk you through the GSDK installation. Alternatively, GSDK and other required tools may be installed manually from the GitHub GSDK site.

Development Tools

As well as Simplicity Studio, Silicon Labs provides the following development tools.

Network Analyzer: SSv5's Network Analyzer enables debugging of complex wireless systems. This tool captures a trace of wireless network activity that can be examined in detail live or at a later time. See the Network Analyzer section of the Simplicity Studio 5 User's Guide for more information.

Wireshark: Is this of interest to RAIL? Wireshark is the recommended network protocol analyzer for use with **for example, Wi-SUN**. Download instructions are provided for Windows/Mac users or Linux users. Simplicity Studio® 5 supports live interaction between the application running on a Silicon Labs device and Wireshark.

Energy Profiler: SSv5's Energy Profiler enables you to visualize the energy consumption of individual devices, multiple devices on one target system, or a network of interacting wireless devices to analyze and improve the power performance of these systems. Real-time information on current consumption is correlated with the program counter providing advanced energy software monitoring capabilities. It also provides a basic level of integration with the Network Analyzer network analysis tool. See the Energy Profiler section of the Simplicity Studio 5 User's Guide for more information.

Simplicity Commander: Simplicity Commander is a single, all-purpose tool to be used in a production environment. It is invoked using a simple Command Line Interface (CLI) that is also scriptable. Simplicity Commander enables customers to complete essential tasks such as configuring and building applications and bootloaders and flashing images to their devices.

Simplicity Commander is available through SSv5 or can be downloaded through system-specific installers. The **Simplicity Commander User's Guide** provides more information.

Silicon Labs Configurator (SLC): SLC offers command-line access to application configuration and generation functions. **Software Project Generation and Configuration with SLC-CLI** provides instructions on downloading and using the SLC-CLI tool.

Overview

RAIL Developers Guide

These pages provide details for developers who wish to include RAIL functionality in their applications. The focus of these pages is the firmware development, including basic usage, optimizing for timing and/or power consumption, and debugging techniques.

Content includes:

- Power Amplifier Power Conversion Functions in RAIL 2.x and Higher (AN1127): Outlines how to account for the variation in output characteristics across custom boards and applications for the Silicon Labs EFR32 family of chips.
- Detailed Timing Test Results for RAIL (AN1392): Provides information about various timing measurements that may be of interest when using RAIL to develop an application.

Overview

RAIL PHY Development

The content in this section is related to the working directly with the physical layer using tools and features provided with RAIL in the Proprietary SDK. This is configurable only when using RAIL or Connect. The content includes theory of and basic information on configuring the EFR32 radio, measurement results and recommendations to comply with various standards, as well as guides on how start from scratch or how to set up the radio to be compatible with an existing setup.

- EFR32 Radio Configurator Guide for Simplicity Studio 5 (AN1253): Describes the radio configurator GUI for RAIL framework applications in Simplicity Studio 5. With it, you can create standard or custom radio configurations on which to run your RAIL-based applications. The role of each GUI item is explained.
- EFR32 Series 1 Long Range Configuration Reference (UG460): Introduces the long-range radio profile, describes its development, and examines underlying details that enable it to realize extended range. Instructions for using example applications are included.
- EFR32 Series 2 Long Range Configuration Reference (UG460.2): Describes how to test long range performance on EFR32 Series 2 devices using Simplicity Studio 5 and Silicon Labs development hardware.
- EFR32 RF Evaluation Guide (AN972): Describes using RAILTest to evaluate radio functionality, as well as peripherals, deep sleep states, etc. With it you can fully evaluate the receiving and transmitting performance and test RF functionality of development kit hardware or custom hardware.

RAIL Library

RAIL Library

Silicon Labs Radio Abstraction Interface Layer (RAIL) is a library that can be used as a generic interface for all Silicon Labs radio parts. By programming against this API, you can write code that easily ports across different radio parts while having access to hardware acceleration wherever possible.

Introduction

The RAIL library is mostly standalone with a few external dependencies. Simplicity Studio application builder is not required but is recommended because it simplifies building applications using RAIL. Depending on the hardware platform that you're using, you may be required to provide certain HAL functionality for the library to work properly. These functions are described in hardware-specific documentation sections and valid implementations can be found as a part of the provided emlib/emdrv HAL layers. It is recommended that you use these versions for maximum support, but you're free to re-implement them if necessary.

Using RAIL

Features

At a high level, the functionality supported by RAIL is shown below. For more specifics on the APIs and how to use them, see the module documentation.

- General
  - Initialize the RAIL API layer.
  - Collect entropy from the radio (if available).
- Radio Configuration
  - Configure the radio frequency, packet format, channel configuration and other PHY parameters.
  - Query current PHY data rates and parameters like current channel.
- State Transitions
  - Configure automatic radio state transitions and transition timings.
- Auto-ACK
  - Configure the radio for automatic acknowledgments.
  - Load the auto ACK payload.
- System Timing
  - Get the current time in the RAIL timebase.
  - Configure a timer to trigger an event callback after an absolute or relative delay.
  - Specify where within a packet its timestamp is desired.
- Events
  - Configure which radio or RAIL events the application wants to be told about via callback.
- Data Management
  - Allows the application to choose the type of data and the method of data interaction through RAIL.
- Receive
  - Configure receive options like CRC checking.
  - Start or schedule when to receive.
  - Enable and configure Address Filtering for each packet.
  - Enable and configure RX Channel Hopping for receiving packets across several channels.
- Transmit
  - Configure the power amplifier (Power Amplifier (PA)) and set transmit power output.
  - Load and send packet data, either immediately, scheduled, or using CSMA or LBT.

  - Control per-transmit options like CRC generation, ACK waiting, etc.
- Antenna Control
  - Configuring multi-antenna selection.
- Multiprotocol
  - Manage time-sharing of the radio among different protocols.
- Calibration
  - APIs for handling various radio calibrations for optimal performance.
- RF Sense
  - Enable RF energy sensing of specified duration across the 2.4 GHz and/or Sub-GHz bands (EFR32 only).
- Packet Trace (PTI)
  - Configure Packet Trace pins and serial protocol.
  - Specify the stack protocol to aid network analyzer packet decoding.
- Energy Friendly Front End Module (EFF)
  - Configure RAIL to transmit/receive via an attached Energy Friendly Front End Module (EFF).
- Assertions
  - Detecting and handling fatal runtime errors detected by RAIL.
- Diagnostic
  - Output debug signals like an unmodulated tone and a continuously modulated stream of data.
  - Configure crystal tuning for your radio.
  - Fine-tune the radio tuner frequency.
- Features
  - Macros to indicate which features exist on which chip families.

Protocol-specific hardware acceleration:

- IEEE 802.15.4
  - Configure the IEEE802.15.4 2.4GHz PHY.
  - Configure node address and address filtering for IEEE 802.15.4.
  - Configure auto ACK for IEEE 802.15.4.
- BLE
  - Configure the Bluetooth Low Energy PHYs available for your device (1Mbps, 2Mbps, Coded PHY).
  - Preamble, sync word and whitening adjustment function for connections.
- Z-Wave
  - Configure the Z-Wave PHY appropriate for your region.
  - Configure your NodeId and HomeId for receive filtering and wakeup beam filtering.

Getting Started Example

Below is a simple example to show you how to initialize the RAIL library in your application. Notice that this requires the channel configurations, which can be generated from the radio calculator in Simplicity Studio. By default, RAIL interacts with data on a per-packet basis. For more information, see Data Management.

Note: Before the radio can transmit, ensure the PA is initialized for your platform, which is described in the Hardware-specific Configuration section below.

```
#include "rail.h"
#include "rail_config.h" // Generated by radio calculator

#define TX_FIFO_SIZE  (128)  // Any power of 2 from [64, 4096] on the EFR32

static RAIL_Handle_t gRailHandle = NULL;
static RAIL_TxPower_t txPower = 200; // Default to 20 dBm
static uint8_t txFifo[TX_FIFO_SIZE];

static const RAIL_TxPowerConfig_t railTxPowerConfig = { // May be const
  // ... desired PA settings
};

static void radioConfigChangedHandler(RAIL_Handle_t railHandle,
                                      const RAIL_ChannelConfigEntry_t *entry)
{
  bool isSubgig = (entry->baseFrequency < 1000000000UL);

  // ... handle radio configuration change, e.g., select the desired PA possibly
  // using isSubgig to handle multiple configurations
  RAIL_ConfigTxPower(railHandle, &railTxPowerConfig);

  // The TX power must be reapplied after changing the PA above.
  RAIL_SetTxPowerDbm(railHandle, txPower);
}

static void radioEventHandler(RAIL_Handle_t railHandle,
                              RAIL_Events_t events)
{
  // ... handle RAIL events, e.g., receive and transmit completion
}

#if MULTIPROTOCOL
static RAILSched_Config_t schedCfg; // Must never be const
static RAIL_Config_t railCfg = {  // Must never be const
  .eventsCallback = &radioEventHandler,
  .protocol = NULL, // For BLE, pointer to a RAIL_BLE_State_t
  .scheduler = &schedCfg, // For MultiProtocol, additional scheduler memory
};
#else
static RAIL_Config_t railCfg = {  // Must never be const
  .eventsCallback = &radioEventHandler,
  .protocol = NULL,
  .scheduler = NULL,
};
#endif

// Initializes the radio out of startup so that it's ready to receive.
void radioInitialize(void)
{
  // Initializes the RAIL library and any internal state it requires.
  gRailHandle = RAIL_Init(&railCfg, NULL);

  // Configures calibration settings.
  RAIL_ConfigCal(gRailHandle, RAIL_CAL_ALL);

  // Configures radio according to the generated radio settings.
  RAIL_ConfigChannels(gRailHandle, channelConfigs[0], &radioConfigChangedHandler);

  // Configures the most useful callbacks and catches a few errors.
  RAIL_ConfigEvents(gRailHandle,
                    RAIL_EVENTS_ALL,
                    RAIL_EVENT_TX_PACKET_SENT
                    | RAIL_EVENT_RX_PACKET_RECEIVED
                    | RAIL_EVENT_RX_FRAME_ERROR // invalid CRC
                    | RAIL_EVENT_RX_ADDRESS_FILTERED);
```

```
// Sets automatic transitions to always receive once started.
  RAIL_StateTransitions_t railStateTransitions ={.success = RAIL_RF_STATE_RX, .error   =
RAIL_RF_STATE_RX,};RAIL_SetRxTransitions(gRailHandle,&railStateTransitions);RAIL_SetTxTransitions(gRailHandle,&railStateTransitions);// Sets up the
transmit buffer.RAIL_SetTxFifo(gRailHandle, txFifo,0, TX_FIFO_SIZE);}
```

Hardware-Specific Configuration

For hardware-specific configuration and setup information, see subsequent sections.

EFR32

RAIL API

RAIL API

This is the primary API layer for the Radio Abstraction Interface Layer (RAIL)

Modules

Antenna Control

Antenna Control

Basic APIs to control the antenna functionality.

These enumerations and structures are used with RAIL Antenna Control API.

EFR32 supports up to two antennas with configurable pin locations.

Modules

RAIL_AntennaConfig_t

Enumerations

enum RAIL_AntennaSel_t {

RAIL_ANTENNA_0 = 0
RAIL_ANTENNA_1 = 1
RAIL_ANTENNA_AUTO = 255

}

Antenna path Selection enumeration.

Functions

RAIL_Status_t RAIL_ConfigAntenna(RAIL_Handle_t railHandle, const RAIL_AntennaConfig_t *config)

Configure antenna path and pin locations.

RAIL_Status_t RAIL_GetRfPath(RAIL_Handle_t railHandle, RAIL_AntennaSel_t *rfPath)

Get the default RF path.

Macros

#define defaultPath ant0Loc

Maps EFR32 Series 2 defaultPath onto Series 1 ant0Loc field.

Enumeration Documentation

RAIL_AntennaSel_t

> RAIL_AntennaSel_t

Antenna path Selection enumeration.

	Enumerator
RAIL_ANTENNA_0	Enum for antenna path 0.
RAIL_ANTENNA_1	Enum for antenna path 1.
RAIL_ANTENNA_AUTO	Enum for antenna path auto.

Definition at line 4353 of file common/rail_types.h

Function Documentation

RAIL_ConfigAntenna

> RAIL_Status_t RAIL_ConfigAntenna (RAIL_Handle_t railHandle, const RAIL_AntennaConfig_t *config)

Configure antenna path and pin locations.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	config	A configuration structure applied to the relevant Antenna Configuration registers. A NULL configuration will produce undefined behavior.

Warnings

- This API must be called before any TX or RX occurs. Otherwise, the antenna configurations for those functions will not take effect.

Returns

- Status code indicating success of the function call.

This function informs RAIL how to select each antenna, but not when. Antenna selection for receive is controlled by the RAIL_RxOptions_t::RAIL_RX_OPTION_ANTENNA0 and RAIL_RxOptions_t::RAIL_RX_OPTION_ANTENNA1 options (and the RAIL_RxOptions_t::RAIL_RX_OPTION_ANTENNA_AUTO combination). Antenna selection for transmit is controlled by the RAIL_TxOptions_t::RAIL_TX_OPTION_ANTENNA0 and RAIL_TxOptions_t::RAIL_TX_OPTION_ANTENNA1 options.

Although a RAIL handle is included for potential future expansion of this function, it is currently not used. That is, only one antenna configuration can be active on a chip, regardless of the number of protocols (unless the application updates the configuration upon a protocol switch), and the configuration is not saved in the RAIL instance. For optimal future compatibility, pass in a chip-specific handle, such as RAIL_EFR32_HANDLE.

Definition at line `504` of file `common/rail.h`

RAIL_GetRfPath

> RAIL_Status_t RAIL_GetRfPath (RAIL_Handle_t railHandle, RAIL_AntennaSel_t *rfPath)

Get the default RF path.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	rfPath	Pointer to RF path.

Returns

- A status code indicating success of the function call.

If multiple protocols are used, this function returns RAIL_STATUS_INVALID_STATE if it is called and the given railHandle is not active. In that case, the caller must attempt to re-call this function later, for example when RAIL_EVENT_CONFIG_SCHEDULED trigger.

Definition at line `519` of file `common/rail.h`

Macro Definition Documentation

defaultPath

```
#define defaultPath
```

Value:

```
ant0Loc
```

Maps EFR32 Series 2 defaultPath onto Series 1 ant0Loc field.

For EFR32 Series 2, defaultPath should be a RAIL_AntennaSel_t value specifying the internal default RF path. It is ignored on EFR32 Series 2 parts that have only one RF path bonded out and on EFR32xG28 dual-band OPNs where the appropriate RF path is automatically set by RAIL to 0 for 2.4GHZ band and 1 for SubGHz band PHYs. On EFR32xG23 and EFR32xG28 single-band OPNs where both RF paths are bonded out this can be set to RAIL_ANTENNA_AUTO to effect internal RF path diversity on PHYs supporting diversity. This avoids the need for an external RF switch and the associated GPIO(s) needed to control its antenna selection.

Definition at line `4411` of file `common/rail_types.h`

RAIL_AntennaConfig_t

A configuration for antenna selection.

Public Attributes

bool	ant0PinEn Antenna 0 Pin Enable.
bool	ant1PinEn Antenna 1 Pin Enable.
uint8_t	ant0Loc Antenna 0 location for ant0Port/Pin on EFR32 Series 1 and on EFR32 Series 2 this field is called defaultPath (see define and usage below).
uint8_t	ant0Port Antenna 0 output GPIO port.
uint8_t	ant0Pin Antenna 0 output GPIO pin.
uint8_t	ant1Loc Antenna 1 location for ant1Port/Pin.
uint8_t	ant1Port Antenna 1 output GPIO port.
uint8_t	ant1Pin Antenna 1 output GPIO pin.

Public Attribute Documentation

ant0PinEn

```
bool RAIL_AntennaConfig_t::ant0PinEn
```

Antenna 0 Pin Enable.

Definition at line 4375 of file common/rail_types.h

ant1PinEn

```
bool RAIL_AntennaConfig_t::ant1PinEn
```

Antenna 1 Pin Enable.

Definition at line 4377 of file common/rail_types.h

ant0Loc

uint8_t RAIL_AntennaConfig_t::ant0Loc

Antenna 0 location for ant0Port/Pin on EFR32 Series 1 and on EFR32 Series 2 this field is called defaultPath (see define and usage below).

Definition at line 4383 of file common/rail_types.h

ant0Port

uint8_t RAIL_AntennaConfig_t::ant0Port

Antenna 0 output GPIO port.

Definition at line 4385 of file common/rail_types.h

ant0Pin

uint8_t RAIL_AntennaConfig_t::ant0Pin

Antenna 0 output GPIO pin.

Definition at line 4387 of file common/rail_types.h

ant1Loc

uint8_t RAIL_AntennaConfig_t::ant1Loc

Antenna 1 location for ant1Port/Pin.

Only needed on EFR32 Series 1; ignored on other platforms.

Definition at line 4390 of file common/rail_types.h

ant1Port

uint8_t RAIL_AntennaConfig_t::ant1Port

Antenna 1 output GPIO port.

Definition at line 4392 of file common/rail_types.h

ant1Pin

uint8_t RAIL_AntennaConfig_t::ant1Pin

Antenna 1 output GPIO pin.

Definition at line 4394 of file common/rail_types.h

Assertions

Assertions

Callbacks called by assertions.

The assertion framework was implemented to not only assert that certain conditions are true in a block of code, but also to handle them more appropriately. In previous implementations, the behavior upon a failed assert was to hang in a while(1) loop. However, with the callback, each assert is given a unique error code so that they can be handled on a more case-by-case basis. For documentation on each of the errors, see the rail_assert_error_codes.h file. RAIL_ASSERT_ERROR_MESSAGES[errorCode] gives the explanation of the error. With asserts built into the library, users can choose how to handle each error inside the callback.

Enumerations

enum RAIL_AssertErrorCodes_t {

RAIL_ASSERT_FAILED_APPENDED_INFO_MISSING = 0
RAIL_ASSERT_FAILED_RX_FIFO_BYTES = 1
RAIL_ASSERT_UNUSED_2 = 2
RAIL_ASSERT_FAILED_ILLEGAL_RXLEN_ENTRY_STATUS = 3
RAIL_ASSERT_FAILED_BAD_PACKET_LENGTH = 4
RAIL_ASSERT_FAILED_SYNTH_DIVCTRL_ENUM_CONVERSION_ERROR = 5
RAIL_ASSERT_FAILED_UNEXPECTED_STATE_RX_FIFO = 6
RAIL_ASSERT_FAILED_UNEXPECTED_STATE_RXLEN_FIFO = 7
RAIL_ASSERT_FAILED_UNEXPECTED_STATE_TX_FIFO = 8
RAIL_ASSERT_FAILED_UNEXPECTED_STATE_TXACK_FIFO = 9
RAIL_ASSERT_UNUSED_10 = 10
RAIL_ASSERT_UNUSED_11 = 11
RAIL_ASSERT_UNUSED_12 = 12
RAIL_ASSERT_FAILED_RTCC_POST_WAKEUP = 13
RAIL_ASSERT_FAILED_SYNTH_VCO_FREQUENCY = 14
RAIL_ASSERT_FAILED_RAC_STATE = 15
RAIL_ASSERT_FAILED_SYNTH_INVALID_VCOCTRL = 16
RAIL_ASSERT_FAILED_NESTED_SEQUENCER_LOCK = 17
RAIL_ASSERT_FAILED_RSSI_AVERAGE_DONE = 18
RAIL_ASSERT_UNUSED_19 = 19
RAIL_ASSERT_FAILED_PROTIMER_RANDOM_SEED = 20
RAIL_ASSERT_FAILED_EFR32XG1_REGISTER_SIZE = 21
RAIL_ASSERT_FAILED_PROTIMER_CHANNEL = 22
RAIL_ASSERT_UNUSED_23 = 23
RAIL_ASSERT_FAILED_BASECNTTOP = 24
RAIL_ASSERT_UNUSED_25 = 25
RAIL_ASSERT_FAILED_RTCC_SYNC_MISSED = 26
RAIL_ASSERT_FAILED_CLOCK_SOURCE_NOT_READY = 27
RAIL_ASSERT_UNUSED_28 = 28
RAIL_ASSERT_NULL_HANDLE = 29
RAIL_ASSERT_UNUSED_30 = 30
RAIL_ASSERT_FAILED_NO_ACTIVE_CONFIG = 31
RAIL_ASSERT_UNUSED_32 = 32
RAIL_ASSERT_UNUSED_33 = 33
RAIL_ASSERT_UNUSED_34 = 34
RAIL_ASSERT_UNUSED_35 = 35
RAIL_ASSERT_UNUSED_36 = 36
RAIL_ASSERT_UNUSED_37 = 37
RAIL_ASSERT_UNUSED_38 = 38
RAIL_ASSERT_DEPRECATED_FUNCTION = 39
RAIL_ASSERT_MULTIPROTOCOL_NO_EVENT = 40
RAIL_ASSERT_FAILED_INVALID_INTERRUPT_ENABLED = 41
RAIL_ASSERT_UNUSED_42 = 42
RAIL_ASSERT_DIVISION_BY_ZERO = 43
RAIL_ASSERT_CANT_USE_HARDWARE = 44
RAIL_ASSERT_NULL_PARAMETER = 45
RAIL_ASSERT_UNUSED_46 = 46
RAIL_ASSERT_SMALL_SYNTH_RADIO_CONFIG_BUFFER = 47
RAIL_ASSERT_CHANNEL_HOPPING_BUFFER_TOO_SHORT = 48
RAIL_ASSERT_INVALID_MODULE_ACTION = 49
RAIL_ASSERT_CHANNEL_HOPPING_INVALID_RADIO_CONFIG = 50
RAIL_ASSERT_CHANNEL_CHANGE_FAILED = 51
RAIL_ASSERT_INVALID_REGISTER = 52
RAIL_ASSERT_UNUSED_53 = 53
RAIL_ASSERT_CACHE_CONFIG_FAILED = 54
RAIL_ASSERT_NULL_TRANSITIONS = 55
RAIL_ASSERT_BAD_LDMA_TRANSFER = 56
RAIL_ASSERT_INVALID_RTCC_SYNC_VALUES = 57
RAIL_ASSERT_SEQUENCER_FAULT = 58
RAIL_ASSERT_BUS_ERROR = 59
RAIL_ASSERT_INVALID_FILTERING_CONFIG = 60
RAIL_ASSERT_RETIMING_CONFIG = 61
RAIL_ASSERT_FAILED_TX_CRC_CONFIG = 62
RAIL_ASSERT_INVALID_PA_OPERATION = 63
RAIL_ASSERT_SEQ_INVALID_PA_SELECTED = 64
RAIL_ASSERT_FAILED_INVALID_CHANNEL_CONFIG = 65
RAIL_ASSERT_INVALID_XTAL_FREQUENCY = 66
RAIL_ASSERT_UNUSED_67 = 67
RAIL_ASSERT_UNSUPPORTED_SOFTWARE_MODEM_MODULATION = 68

```
    RAIL_ASSERT_FAILED_RTCC_SYNC_STOP = 69
    RAIL_ASSERT_FAILED_MULTITIMER_CORRUPT = 70
    RAIL_ASSERT_FAILED_TEMPCAL_ERROR = 71
    RAIL_ASSERT_INVALID_EFF_CONFIGURATION = 72
    RAIL_ASSERT_INVALID_RFFPLL_CONFIGURATION = 73
    RAIL_ASSERT_SECURE_ACCESS_FAULT = 74
    RAIL_ASSERT_FAILED_SYSRTC0_NOT_RUNNING = 75
    RAIL_ASSERT_RADIO_CONFIG_NOT_UP_TO_DATE = 76
    RAIL_ASSERT_FAILED_RSSI_THRESHOLD = 77
    RAIL_ASSERT_INCORRECT_ZWAVE_REGION = 78
    RAIL_ASSERT_FAILED_RTCC_SYNC_STALE_DATA = 79
    RAIL_ASSERT_INVALID_LOG2X4_CLEAR_CONDITION = 80
    RAIL_ASSERT_FAILED_DMA_WRITE_INCOMPLETE = 81
    RAIL_ASSERT_CALCULATOR_NOT_SUPPORTED = 82
    RAIL_ASSERT_INVALID_SEQUENCER_IMAGE = 83
    RAIL_ASSERT_MISSING_SEQUENCER_IMAGE = 84
    RAIL_ASSERT_INVALID_OR_MISSING_SOFTWARE_MODEM_IMAGE
    = 85
}
```

Enumeration of all possible error codes from RAIL_ASSERT.

Functions

void RAILCb_AssertFailed(RAIL_Handle_t railHandle, RAIL_AssertErrorCodes_t errorCode)

Callback called upon failed assertion.

Macros

#define RAIL_ASSERT_ERROR_MESSAGES undefined

Use this define to create an array of error messages that map to the codes in RAIL_AssertErrorCodes_t.

Enumeration Documentation

RAIL_AssertErrorCodes_t

> RAIL_AssertErrorCodes_t

Enumeration of all possible error codes from RAIL_ASSERT.

Enumerator	
RAIL_ASSERT_FAILED_APPENDED_INFO_MISSING	Appended info missing from RX packet.
RAIL_ASSERT_FAILED_RX_FIFO_BYTES	Receive FIFO too small for IR calibration.
RAIL_ASSERT_UNUSED_2	Invalid assert, no longer used.
RAIL_ASSERT_FAILED_ILLEGAL_RXLEN_ENTRY_STATUS	Receive FIFO entry has invalid status.
RAIL_ASSERT_FAILED_BAD_PACKET_LENGTH	Receive FIFO entry bad packet length.
RAIL_ASSERT_FAILED_SYNTH_DIVCTRL_ENUM_CONVERSION_ERROR	Unable to configure radio for IR calibration.
RAIL_ASSERT_FAILED_UNEXPECTED_STATE_RX_FIFO	Reached unexpected state while handling RX FIFO events.
RAIL_ASSERT_FAILED_UNEXPECTED_STATE_RXLEN_FIFO	Reached unexpected state while handling RXLEN FIFO events.
RAIL_ASSERT_FAILED_UNEXPECTED_STATE_TX_FIFO	Reached unexpected state while handling TX FIFO events.
RAIL_ASSERT_FAILED_UNEXPECTED_STATE_TXACK_FIFO	Reached unexpected state while handling TX ACK FIFO events.

RAIL_ASSERT_UNUSED_10	Invalid assert, no longer used.
RAIL_ASSERT_UNUSED_11	Invalid assert, no longer used.
RAIL_ASSERT_UNUSED_12	Invalid assert, no longer used.
RAIL_ASSERT_FAILED_RTCC_POST_WAKEUP	Error synchronizing the RAIL timebase after sleep.
RAIL_ASSERT_FAILED_SYNTH_VCO_FREQUENCY	VCO frequency outside supported range.
RAIL_ASSERT_FAILED_RAC_STATE	Radio active while changing channels.
RAIL_ASSERT_FAILED_SYNTH_INVALID_VCOCTRL	Invalid Synth VCOCTRL field calculation.
RAIL_ASSERT_FAILED_NESTED_SEQUENCER_LOCK	Nested attempt to lock the sequencer.
RAIL_ASSERT_FAILED_RSSI_AVERAGE_DONE	RSSI averaging enabled without a valid callback.
RAIL_ASSERT_UNUSED_19	Invalid assert, no longer used.
RAIL_ASSERT_FAILED_PROTIMER_RANDOM_SEED	Unable to seed radio pseudo random number generator.
RAIL_ASSERT_FAILED_EFR32XG1_REGISTER_SIZE	Timeout exceeds EFR32XG1 register size.
RAIL_ASSERT_FAILED_PROTIMER_CHANNEL	Invalid timer channel specified.
RAIL_ASSERT_UNUSED_23	Invalid assert, no longer used.
RAIL_ASSERT_FAILED_BASECNTTOP	LBT config exceeds EFR32XG1 register size.
RAIL_ASSERT_UNUSED_25	Invalid assert, no longer used.
RAIL_ASSERT_FAILED_RTCC_SYNC_MISSED	Could not synchronize RAIL timebase with the RTC.
RAIL_ASSERT_FAILED_CLOCK_SOURCE_NOT_READY	Clock source not ready.
RAIL_ASSERT_UNUSED_28	Invalid assert, no longer used.
RAIL_ASSERT_NULL_HANDLE	NULL was supplied as a RAIL_Handle_t argument.
RAIL_ASSERT_UNUSED_30	Invalid assert, no longer used.
RAIL_ASSERT_FAILED_NO_ACTIVE_CONFIG	API improperly called while protocol inactive.
RAIL_ASSERT_UNUSED_32	Invalid assert, no longer used.
RAIL_ASSERT_UNUSED_33	Invalid assert, no longer used.
RAIL_ASSERT_UNUSED_34	Invalid assert, no longer used.
RAIL_ASSERT_UNUSED_35	Invalid assert, no longer used.
RAIL_ASSERT_UNUSED_36	Invalid assert, no longer used.
RAIL_ASSERT_UNUSED_37	Invalid assert, no longer used.
RAIL_ASSERT_UNUSED_38	Invalid assert, no longer used.
RAIL_ASSERT_DEPRECATED_FUNCTION	This function is deprecated and must not be called.
RAIL_ASSERT_MULTIPROTOCOL_NO_EVENT	Multiprotocol task started with no event to run.
RAIL_ASSERT_FAILED_INVALID_INTERRUPT_ENABLED	Invalid interrupt enabled.
RAIL_ASSERT_UNUSED_42	Invalid assert, no longer used.
RAIL_ASSERT_DIVISION_BY_ZERO	Division by zero.
RAIL_ASSERT_CANT_USE_HARDWARE	Function cannot be called without access to the hardware.
RAIL_ASSERT_NULL_PARAMETER	Pointer parameter was passed as NULL.
RAIL_ASSERT_UNUSED_46	Invalid assert, no longer used.
RAIL_ASSERT_SMALL_SYNTH_RADIO_CONFIG_BUFFER	Synth radio config buffer for channel hopping too small.
RAIL_ASSERT_CHANNEL_HOPPING_BUFFER_TOO_SHORT	Buffer provided for RX Channel Hopping is too small.
RAIL_ASSERT_INVALID_MODULE_ACTION	Invalid action was attempted on a module.
RAIL_ASSERT_CHANNEL_HOPPING_INVALID_RADIO_CONFIG	The radio config for this channel is not compatible with channel hopping.
RAIL_ASSERT_CHANNEL_CHANGE_FAILED	Channel change failed.

RAIL_ASSERT_INVALID_REGISTER	Attempted to read invalid register.
RAIL_ASSERT_UNUSED_53	Invalid assert, no longer used.
RAIL_ASSERT_CACHE_CONFIG_FAILED	DMP radio config caching failed.
RAIL_ASSERT_NULL_TRANSITIONS	NULL was supplied as a RAIL_StateTransitions_t argument.
RAIL_ASSERT_BAD_LDMA_TRANSFER	LDMA transfer failed.
RAIL_ASSERT_INVALID_RTCC_SYNC_VALUES	Attempted to wake up with invalid RTCC sync data.
RAIL_ASSERT_SEQUENCER_FAULT	Radio sequencer hit a fault condition.
RAIL_ASSERT_BUS_ERROR	Bus fault.
RAIL_ASSERT_INVALID_FILTERING_CONFIG	The current radio config cannot be used with packet filtering.
RAIL_ASSERT_RETIMING_CONFIG	Retiming configuration error.
RAIL_ASSERT_FAILED_TX_CRC_CONFIG	TX CRC configuration is corrupt.
RAIL_ASSERT_INVALID_PA_OPERATION	The current PA config does not allow for this operation.
RAIL_ASSERT_SEQ_INVALID_PA_SELECTED	The sequencer selected an invalid PA.
RAIL_ASSERT_FAILED_INVALID_CHANNEL_CONFIG	Invalid/unsupported channel config.
RAIL_ASSERT_INVALID_XTAL_FREQUENCY	Radio Calculator configuration HFXO frequency mismatch with chip.
RAIL_ASSERT_UNUSED_67	Invalid assert, no longer used.
RAIL_ASSERT_UNSUPPORTED_SOFTWARE_MODEM_MODULATION	Software modem image does not support requested modulation
RAIL_ASSERT_FAILED_RTCC_SYNC_STOP	Failed to disable RTCC synchronization.
RAIL_ASSERT_FAILED_MULTITIMER_CORRUPT	Multitimer linked list corrupted.
RAIL_ASSERT_FAILED_TEMPCAL_ERROR	Unable to configure radio for temperature calibration.
RAIL_ASSERT_INVALID_EFF_CONFIGURATION	Invalid EFF configuration.
RAIL_ASSERT_INVALID_RFFPLL_CONFIGURATION	Invalid RFFPLL configuration.
RAIL_ASSERT_SECURE_ACCESS_FAULT	Secure access fault.
RAIL_ASSERT_FAILED_SYSRTC0_NOT_RUNNING	SYSRTC0 not running.
RAIL_ASSERT_RADIO_CONFIG_NOT_UP_TO_DATE	Radio Configurator not updated.
RAIL_ASSERT_FAILED_RSSI_THRESHOLD	Failed to set the event for configurable RSSI threshold.
RAIL_ASSERT_INCORRECT_ZWAVE_REGION	Intended and actual Z-Wave region configuration mismatch.
RAIL_ASSERT_FAILED_RTCC_SYNC_STALE_DATA	Attempted to sleep with stale RTCC synchronization data.
RAIL_ASSERT_INVALID_LOG2X4_CLEAR_CONDITION	Attempted to clear LOG2X4 with a DEC1 value not equal to 0.
RAIL_ASSERT_FAILED_DMA_WRITE_INCOMPLETE	Failed to complete DMA write.
RAIL_ASSERT_CALCULATOR_NOT_SUPPORTED	RAIL does not support this Radio Calculator configuration.
RAIL_ASSERT_INVALID_SEQUENCER_IMAGE	Invalid binary image was loaded onto the sequencer.
RAIL_ASSERT_MISSING_SEQUENCER_IMAGE	No common or protocol image selected to be loaded onto the sequencer.
RAIL_ASSERT_INVALID_OR_MISSING_SOFTWARE_MODEM_IMAGE	Software modem image invalid or missing.

Definition at line 50 of file common/rail_assert_error_codes.h

Function Documentation

RAILCb_AssertFailed

> void RAILCb_AssertFailed (RAIL_Handle_t railHandle, RAIL_AssertErrorCodes_t errorCode)

Callback called upon failed assertion.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	errorCode	Value passed in by the calling assertion API indicating the RAIL error that is indicated by the failing assertion.

Definition at line 6459 of file common/rail.h

Macro Definition Documentation

RAIL_ASSERT_ERROR_MESSAGES

> #define RAIL_ASSERT_ERROR_MESSAGES

Use this define to create an array of error messages that map to the codes in RAIL_AssertErrorCodes_t.

You can use these to print slightly more detailed error strings related to a particular assert error code if desired. For example, you could implement your assert failed callback as follows to make use of this.

```
void RAILCb_AssertFailed(RAIL_Handle_t railHandle, uint32_t errorCode)
{
 static const char* railErrorMessages[] = RAIL_ASSERT_ERROR_MESSAGES;
 const char *errorMessage = "Unknown";

 // If this error code is within the range of known error messages then use
 // the appropriate error message.
 if (errorCode < (sizeof(railErrorMessages) / sizeof(char*))) {
  errorMessage = railErrorMessages[errorCode];
 }
 printf(errorMessage);

 // Reset the chip since an assert is a fatal error
 NVIC_SystemReset();
}
```

Definition at line 340 of file common/rail_assert_error_codes.h

Auto-ACK

Auto-ACK

APIs for configuring auto-ACK functionality.

These APIs configure the radio for automatic acknowledgment features. Auto-ACK inherently changes how the underlying state machine behaves so users should not modify RAIL_SetRxTransitions() and RAIL_SetTxTransitions() while using auto-ACK features.

```
// Go to RX after ACK operation.
RAIL_AutoAckConfig_t autoAckConfig = {
  .enable = true,
  .ackTimeout = 1000,
  // "error" param ignored
  .rxTransitions = { RAIL_RF_STATE_RX, RAIL_RF_STATE_RX},
  // "error" param ignored
  .txTransitions = { RAIL_RF_STATE_RX, RAIL_RF_STATE_RX}
};

RAIL_Status_t status = RAIL_ConfigAutoAck(railHandle, &autoAckConfig);

uint8_t ackData[] = {0x05, 0x02, 0x10, 0x00};

RAIL_Status_t status = RAIL_WriteAutoAckFifo(railHandle,
                                             ackData,
                                             sizeof(ackData));
```

The acknowledgment transmits based on the frame format configured via the Radio Configurator. For example, if the frame format is using a variable length scheme, the ACK will be sent according to that scheme. If a 10-byte packet is loaded into the ACK, but the variable length field of the ACK payload specifies a length of 5, only 5 bytes will transmit for the ACK. The converse is also true, if the frame length is configured to be a fixed 10-byte packet but only 5 bytes are loaded into the ACK buffer, a TX underflow occurs during the ACK transmit.

Unlike in non-auto-ACK mode, auto-ACK mode will always return to a single state after all ACK sequences complete, regardless of whether the ACK was successfully received/sent or not. See the documentation of RAIL_ConfigAutoAck() for configuration information. To suspend automatic acknowledgment of a series of packets after transmit or receive call RAIL_PauseTxAutoAck() or RAIL_PauseRxAutoAck() respectively with the pause parameter set to true. When auto-ACKing is paused, after receiving or transmitting a packet (regardless of success), the radio transitions to the same state it would use while ACKing. To return to normal state transition logic outside of ACKing, call RAIL_ConfigAutoAck() with the RAIL_AutoAckConfig_t::enable field false and specify the desired transitions in the RAIL_AutoAckConfig_t::rxTransitions and RAIL_AutoAckConfig_t::txTransitions fields. To get out of a paused state and resume auto-ACKing, call RAIL_PauseTxAutoAck() and/or RAIL_PauseRxAutoAck() with the pause parameter set to false.

Applications can cancel the transmission of an ACK with RAIL_CancelAutoAck(). Conversely, applications can control if a transmit operation should wait for an ACK after transmitting by using the RAIL_TX_OPTION_WAIT_FOR_ACK option.

When Antenna Control is used for multiple antennas, ACKs are transmitted on the antenna that was selected to receive the packet being acknowledged. When receiving an ACK, the RAIL_RxOptions_t antenna options are used just like for any other receive.

If the ACK payload is dynamic, the application must call RAIL_WriteAutoAckFifo() with the appropriate ACK payload after the application processes the receive. RAIL can auto-ACK from the normal transmit buffer if RAIL_UseTxFifoForAutoAck() is called before the radio transmits the ACK. Ensure the transmit buffer contains data loaded by RAIL_WriteTxFifo().

Standard-based protocols that contain auto-ACK functionality are normally configured in the protocol-specific configuration function. For example, RAIL_IEEE802154_Init() provides auto-ACK configuration parameters in RAIL_IEEE802154_Config_t

and should only be configured through that function. It is not advisable to call both RAIL_IEEE802154_Init() and RAIL_ConfigAutoAck(). However, ACK modification functions are still valid to use with protocol-specific ACKs. To cancel an IEEE 802.15.4 ACK transmit, use RAIL_CancelAutoAck().

Modules

RAIL_AutoAckConfig_t

Functions

RAIL_Status_t	RAIL_ConfigAutoAck(RAIL_Handle_t railHandle, const RAIL_AutoAckConfig_t *config) Configure and enable automatic acknowledgment.
bool	RAIL_IsAutoAckEnabled(RAIL_Handle_t railHandle) Return the enable status of the auto-ACK feature.
RAIL_Status_t	RAIL_WriteAutoAckFifo(RAIL_Handle_t railHandle, const uint8_t *ackData, uint8_t ackDataLen) Load the auto-ACK buffer with ACK data.
RAIL_Status_t	RAIL_GetAutoAckFifo(RAIL_Handle_t railHandle, uint8_t **ackBuffer, uint16_t *ackBufferBytes) Get the address and size of the auto-ACK buffer for direct access.
void	RAIL_PauseRxAutoAck(RAIL_Handle_t railHandle, bool pause) Pause/resume RX auto-ACK functionality.
bool	RAIL_IsRxAutoAckPaused(RAIL_Handle_t railHandle) Return whether the RX auto-ACK is paused.
void	RAIL_PauseTxAutoAck(RAIL_Handle_t railHandle, bool pause) Pause/resume TX auto-ACK functionality.
bool	RAIL_IsTxAutoAckPaused(RAIL_Handle_t railHandle) Return whether the TX auto-ACK is paused.
RAIL_Status_t	RAIL_UseTxFifoForAutoAck(RAIL_Handle_t railHandle) Modify the upcoming ACK to use the Transmit FIFO.
RAIL_Status_t	RAIL_CancelAutoAck(RAIL_Handle_t railHandle) Cancel the upcoming ACK.
bool	RAIL_IsAutoAckWaitingForAck(RAIL_Handle_t railHandle) Return whether the radio is currently waiting for an ACK.

Macros

#define	RAIL_AUTOACK_MAX_LENGTH (64U) Acknowledgment packets cannot be longer than 64 bytes.

Function Documentation

RAIL_ConfigAutoAck

> RAIL_Status_t RAIL_ConfigAutoAck (RAIL_Handle_t railHandle, const RAIL_AutoAckConfig_t *config)

Configure and enable automatic acknowledgment.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	config	Auto-ACK configuration structure.

Returns

- Status code indicating success of the function call.

Configures the RAIL state machine to for hardware-accelerated automatic acknowledgment. ACK timing parameters are defined in the configuration structure.

While auto-ACKing is enabled, do not call the following RAIL functions:

- RAIL_SetRxTransitions()
- RAIL_SetTxTransitions()

Note that if you are enabling auto-ACK (i.e., "enable" field is true) the "error" fields of rxTransitions and txTransitions are ignored. After all ACK sequences, (success or fail) the state machine will return the radio to the "success" state, which can be either RAIL_RF_STATE_RX or RAIL_RF_STATE_IDLE (returning to RAIL_RF_STATE_TX is not supported). If you need information about the actual success of the ACK sequence, use RAIL events such as RAIL_EVENT_TXACK_PACKET_SENT to make sure an ACK was sent, or RAIL_EVENT_RX_ACK_TIMEOUT to make sure that an ACK was received within the specified timeout.

To set a certain turnaround time (i.e., txToRx and rxToTx in RAIL_StateTiming_t), make txToRx lower than desired to ensure you get to RX in time to receive the ACK. Silicon Labs recommends setting 10 us lower than desired:

```
void setAutoAckStateTimings()
{
  RAIL_StateTiming_t timings;

  // User is already in auto-ACK and wants a turnaround of 192 us.
  timings.rxToTx = 192;
  timings.txToRx = 192 - 10;

  // Set other fields of timings...
  timings.idleToRx = 100;
  timings.idleToTx = 100;
  timings.rxSearchTimeout = 0;
  timings.txToRxSearchTimeout = 0;

  RAIL_SetStateTiming(railHandle, &timings);
}
```

As opposed to an explicit "Disable" API, set the "enable" field of the RAIL_AutoAckConfig_t to false. Then, auto-ACK will be disabled and state transitions will be returned to the values set in RAIL_AutoAckConfig_t. When disabling, the "ackTimeout" field isn't used.

Note

- Auto-ACKing may not be enabled while RX Channel Hopping is enabled, or when BLE is enabled.

Definition at line `4698` of file `common/rail.h`

RAIL_IsAutoAckEnabled

```
bool RAIL_IsAutoAckEnabled (RAIL_Handle_t railHandle)
```

Return the enable status of the auto-ACK feature.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if auto-ACK is enabled, false if disabled.

Definition at line 4707 of file common/rail.h

RAIL_WriteAutoAckFifo

> RAIL_Status_t RAIL_WriteAutoAckFifo (RAIL_Handle_t railHandle, const uint8_t *ackData, uint8_t ackDataLen)

Load the auto-ACK buffer with ACK data.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	ackData	A pointer to ACK data to transmit. This may be NULL, in which case it's assumed the data has already been emplaced into the ACK buffer and RAIL just needs to be told how many bytes are there. Use RAIL_GetAutoAckFifo() to get the address of RAIL's AutoACK buffer in RAM and its size.
[in]	ackDataLen	The number of bytes in ACK data.

Returns

- Status code indicating success of the function call.

If the ACK buffer is available for updates, load the ACK buffer with data. If it is not available, RAIL_STATUS_INVALID_STATE is returned. If ackDataLen exceeds RAIL_AUTOACK_MAX_LENGTH then RAIL_STATUS_INVALID_PARAMETER will be returned and nothing is written to the ACK buffer (unless ackData is NULL in which case this indicates the application has already likely corrupted RAM).

Definition at line 4728 of file common/rail.h

RAIL_GetAutoAckFifo

> RAIL_Status_t RAIL_GetAutoAckFifo (RAIL_Handle_t railHandle, uint8_t **ackBuffer, uint16_t *ackBufferBytes)

Get the address and size of the auto-ACK buffer for direct access.

Parameters

[in]	railHandle	A RAIL instance handle.
[inout]	ackBuffer	A pointer to a uint8_t pointer that will be updated to the RAM base address of the TXACK buffer.
[inout]	ackBufferBytes	A pointer to a uint16_t that will be updated to the size of the TXACK buffer, in bytes, which is currently RAIL_AUTOACK_MAX_LENGTH.

Returns

- Status code indicating success of the function call.

Applications can use this to more flexibly write AutoAck data into the buffer directly and in pieces, passing NULL ackData parameter to RAIL_WriteAutoAckFifo() or RAIL_IEEE802154_WriteEnhAck() to inform RAIL of its final length.

Definition at line 4748 of file common/rail.h

RAIL_PauseRxAutoAck

> void RAIL_PauseRxAutoAck (RAIL_Handle_t railHandle, bool pause)

Pause/resume RX auto-ACK functionality.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	pause	Pause or resume RX auto-ACKing.

When RX auto-ACKing is paused, the radio transitions to default state after receiving a packet and does not transmit an ACK. When RX auto-ACK is resumed, the radio resumes automatically ACKing every successfully received packet.

Definition at line `4763` of file `common/rail.h`

RAIL_IsRxAutoAckPaused

```
bool RAIL_IsRxAutoAckPaused (RAIL_Handle_t railHandle)
```

Return whether the RX auto-ACK is paused.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if RX auto-ACK is paused, false if not paused.

Definition at line `4772` of file `common/rail.h`

RAIL_PauseTxAutoAck

```
void RAIL_PauseTxAutoAck (RAIL_Handle_t railHandle, bool pause)
```

Pause/resume TX auto-ACK functionality.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	pause	Pause or resume TX auto-ACKing.

When TX auto-ACKing is paused, the radio transitions to a default state after transmitting a packet and does not wait for an ACK. When TX auto-ACK is resumed, the radio resumes automatically waiting for an ACK after a successful transmit.

Definition at line `4785` of file `common/rail.h`

RAIL_IsTxAutoAckPaused

```
bool RAIL_IsTxAutoAckPaused (RAIL_Handle_t railHandle)
```

Return whether the TX auto-ACK is paused.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if TX auto-ACK is paused, false if not paused.

Definition at line `4793` of file `common/rail.h`

RAIL_UseTxFifoForAutoAck

```
RAIL_Status_t RAIL_UseTxFifoForAutoAck (RAIL_Handle_t railHandle)
```

Modify the upcoming ACK to use the Transmit FIFO.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Status code indicating success of the function call. The call will fail if it is too late to modify the outgoing ACK.

This function allows the application to use the normal Transmit FIFO as the data source for the upcoming ACK. The ACK modification to use the Transmit FIFO only applies to one ACK transmission.

This function only returns true if the following conditions are met:

- Radio has not already decided to use the ACK buffer AND
- Radio is either looking for sync, receiving the packet after sync, or in the Rx2Tx turnaround before the ACK is sent.

Note

- The Transmit FIFO must not be used for AutoACK when IEEE 802.15.4, Z-Wave, or BLE protocols are active.

Definition at line `4814` of file `common/rail.h`

RAIL_CancelAutoAck

```
RAIL_Status_t RAIL_CancelAutoAck (RAIL_Handle_t railHandle)
```

Cancel the upcoming ACK.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Status code indicating success of the function call. This call will fail if it is too late to modify the outgoing ACK.

This function allows the application to cancel the upcoming automatic acknowledgment.

This function only returns true if the following conditions are met:

- Radio has not already decided to transmit the ACK AND
- Radio is either looking for sync, receiving the packet after sync or in the Rx2Tx turnaround before the ACK is sent.

Definition at line `4831` of file `common/rail.h`

RAIL_IsAutoAckWaitingForAck

```
bool RAIL_IsAutoAckWaitingForAck (RAIL_Handle_t railHandle)
```

Return whether the radio is currently waiting for an ACK.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- True if radio is waiting for ACK, false if radio is not waiting for an ACK.

This function allows the application to query whether the radio is currently waiting for an ACK after a transmit operation.

Definition at line 4843 of file common/rail.h

Macro Definition Documentation

RAIL_AUTOACK_MAX_LENGTH

```
#define RAIL_AUTOACK_MAX_LENGTH
```

Value:

```
(64U)
```

Acknowledgment packets cannot be longer than 64 bytes.

Definition at line 4337 of file common/rail_types.h

RAIL_AutoAckConfig_t

Enable/disable the auto-ACK algorithm, based on "enable".

The structure provides a default state (the "success" of tx/rxTransitions when ACKing is enabled) for the radio to return to after an ACK operation occurs (transmitting or attempting to receive an ACK), or normal state transitions to return to in the case ACKing is disabled. Regardless whether the ACK operation was successful, the radio returns to the specified success state.

ackTimeout specifies how long to stay in receive and wait for an ACK to start (sync detected) before issuing a RAIL_EVENT_RX_ACK_TIMEOUT event and return to the default state.

Public Attributes

bool	enable Indicate whether auto-ACKing should be enabled or disabled.
uint16_t	ackTimeout Define the RX ACK timeout duration in microseconds up to 65535 microseconds maximum.
RAIL_StateTransitions_t	rxTransitions State transitions to do after receiving a packet.
RAIL_StateTransitions_t	txTransitions State transitions to do after transmitting a packet.

Public Attribute Documentation

enable

```
bool RAIL_AutoAckConfig_t::enable
```

Indicate whether auto-ACKing should be enabled or disabled.

Definition at line 4306 of file common/rail_types.h

ackTimeout

```
uint16_t RAIL_AutoAckConfig_t::ackTimeout
```

Define the RX ACK timeout duration in microseconds up to 65535 microseconds maximum.

Only applied when auto-ACKing is enabled. The ACK timeout timer starts at the completion of a RAIL_TX_OPTION_WAIT_FOR_ACK transmit and expires only while waiting for a packet (prior to SYNC detect), triggering RAIL_EVENT_RX_ACK_TIMEOUT. During packet reception that event is held off until packet completion and suppressed entirely if the received packet is the expected ACK.

Definition at line 4317 of file common/rail_types.h

rxTransitions

> RAIL_StateTransitions_t RAIL_AutoAckConfig_t::rxTransitions

State transitions to do after receiving a packet.

When auto-ACKing is enabled, the "error" transition is always ignored and the radio will return to the "success" state after any ACKing sequence (RAIL_RF_STATE_RX or RAIL_RF_STATE_IDLE). See RAIL_ConfigAutoAck for more details on this.

Definition at line 4325 of file common/rail_types.h

txTransitions

> RAIL_StateTransitions_t RAIL_AutoAckConfig_t::txTransitions

State transitions to do after transmitting a packet.

When auto-ACKing is enabled, the "error" transition is always ignored and the radio will return to the "success" state after any ACKing sequence (RAIL_RF_STATE_RX or RAIL_RF_STATE_IDLE). See RAIL_ConfigAutoAck for more details on this.

Definition at line 4333 of file common/rail_types.h

Calibration

Calibration

APIs for calibrating the radio.

IEEE802154 protocol-specific APIs for calibrating the radio.

Bluetooth protocol-specific APIs for calibrating the radio.

The EFR32 supports the Image Rejection (IR) calibration and a temperature-dependent calibration.

These APIs calibrate the radio. The RAIL library determines which calibrations are necessary. Calibrations can be enabled/disabled with the RAIL_CalMask_t parameter.

Some calibrations produce values that can be saved and reapplied to avoid repeating the calibration process.

Calibrations can either be run with RAIL_Calibrate, or with the individual chip-specific calibration routines. An example for running code with RAIL_Calibrate looks like the following:

```
static RAIL_CalValues_t calValues = RAIL_CALVALUES_UNINIT;

void RAILCb_Event(RAIL_Handle_t railHandle, RAIL_Events_t events) {
  // Omitting other event handlers
  if (events & RAIL_EVENT_CAL_NEEDED) {
    // Run all pending calibrations, and save the results
    RAIL_Calibrate(railHandle, &calValues, RAIL_CAL_ALL_PENDING);
  }
}
```

Alternatively, if the image rejection calibration for your chip can be determined ahead of time, such as by running the calibration on a separate firmware image on each chip, the following calibration process will result in smaller code.

```
static uint32_t imageRejection = IRCAL_VALUE;

void RAILCb_Event(RAIL_Handle_t railHandle, RAIL_Events_t events) {
  // Omitting other event handlers
  if (events & RAIL_EVENT_CAL_NEEDED) {
    RAIL_CalMask_t pendingCals = RAIL_GetPendingCal(railHandle);
    // Disable the radio if we have to do an offline calibration
    if (pendingCals & RAIL_CAL_TEMP_VCO) {
      RAIL_CalibrateTemp(railHandle);
    }
    if (pendingCals & RAIL_CAL_ONETIME_IRCAL) {
      RAIL_ApplyIrCalibration(railHandle, imageRejection);
    }
  }
}
```

The IR calibration can be computed once and stored off or computed each time at startup. Because it is PHY-specific and provides sensitivity improvements, it is highly recommended. The IR calibration should only be run when the radio is IDLE.

The temperature-dependent calibrations are used to recalibrate the synth if the temperature crosses 0C or the temperature delta since the last calibration exceeds 70C while in receive. RAIL will run the VCO calibration automatically upon entering receive or transmit states, so the application can omit this calibration if the stack re-enters receive or

transmit with enough frequency to avoid reaching the temperature delta. If the application does not calibrate for temperature, it's possible to miss receive packets due to a drift in the carrier frequency.

Modules

RAIL_TxIrCalValues_t

RAIL_IrCalValues_t

EFR32

Typedefs

typedef uint32_t **RAIL_CalMask_t**
A calibration mask type.

typedef uint32_t **RAIL_RxIrCalValues_t**[2]
RAIL_RxIrCalValues_t.

typedef RAIL_IrCalValues_t **RAIL_CalValues_t**
A calibration value structure.

Functions

RAIL_Status_t **RAIL_ConfigCal**(RAIL_Handle_t railHandle, RAIL_CalMask_t calEnable)
Initialize RAIL calibration.

RAIL_Status_t **RAIL_Calibrate**(RAIL_Handle_t railHandle, RAIL_CalValues_t *calValues, RAIL_CalMask_t calForce)
Start the calibration process.

RAIL_CalMask_t **RAIL_GetPendingCal**(RAIL_Handle_t railHandle)
Return the current set of pending calibrations.

RAIL_Status_t **RAIL_ApplyIrCalibration**(RAIL_Handle_t railHandle, uint32_t imageRejection)
Apply a given image rejection calibration value.

RAIL_Status_t **RAIL_ApplyIrCalibrationAlt**(RAIL_Handle_t railHandle, RAIL_IrCalValues_t *imageRejection, RAIL_AntennaSel_t rfPath)
Apply a given image rejection calibration value.

RAIL_Status_t **RAIL_CalibrateIr**(RAIL_Handle_t railHandle, uint32_t *imageRejection)
Run the image rejection calibration.

RAIL_Status_t **RAIL_CalibrateIrAlt**(RAIL_Handle_t railHandle, RAIL_IrCalValues_t *imageRejection, RAIL_AntennaSel_t rfPath)
Run the image rejection calibration.

RAIL_Status_t **RAIL_CalibrateTemp**(RAIL_Handle_t railHandle)
Run the temperature calibration.

RAIL_Status_t **RAIL_CalibrateHFXO**(RAIL_Handle_t railHandle, int8_t *crystalPPMError)
Performs HFXO compensation.

void **RAIL_EnablePaCal**(bool enable)
Enable/disable the PA calibration.

RAIL_Status_t **RAIL_BLE_CalibrateIr**(RAIL_Handle_t railHandle, uint32_t *imageRejection)
Calibrate image rejection for Bluetooth Low Energy.

RAIL_Status_t **RAIL_IEEE802154_CalibrateIr2p4Ghz**(RAIL_Handle_t railHandle, uint32_t *imageRejection)
Calibrate image rejection for IEEE 802.15.4 2.4 GHz.

RAIL_Status_t RAIL_IEEE802154_CalibrateIrSubGhz(RAIL_Handle_t railHandle, uint32_t *imageRejection)
Calibrate image rejection for IEEE 802.15.4 915 MHz and 868 MHz.

Macros

#define RAIL_CAL_TEMP_VCO (0x00000001U)
EFR32-specific temperature calibration bit.

#define RAIL_CAL_TEMP_HFXO (0x00000002U)
EFR32-specific HFXO temperature check bit.

#define RAIL_CAL_COMPENSATE_HFXO (0x00000004U)
EFR32-specific HFXO compensation bit.

#define RAIL_CAL_RX_IRCAL (0x00010000U)
EFR32-specific IR calibration bit.

#define RAIL_CAL_OFDM_TX_IRCAL (0x00100000U)
EFR32-specific Tx IR calibration bit.

#define RAIL_CAL_ONETIME_IRCAL (RAIL_CAL_RX_IRCAL | RAIL_CAL_OFDM_TX_IRCAL)
A mask to run EFR32-specific IR calibrations.

#define RAIL_CAL_TEMP (RAIL_CAL_TEMP_VCO | RAIL_CAL_TEMP_HFXO | RAIL_CAL_COMPENSATE_HFXO)
A mask to run temperature-dependent calibrations.

#define RAIL_CAL_ONETIME (RAIL_CAL_ONETIME_IRCAL)
A mask to run one-time calibrations.

#define RAIL_CAL_PERF (0)
A mask to run optional performance calibrations.

#define RAIL_CAL_OFFLINE (RAIL_CAL_ONETIME_IRCAL)
A mask for calibrations that require the radio to be off.

#define RAIL_CAL_ALL (RAIL_CAL_TEMP | RAIL_CAL_ONETIME)
A mask to run all possible calibrations for this chip.

#define RAIL_CAL_ALL_PENDING (0x00000000U)
A mask to run all pending calibrations.

#define RAIL_CAL_INVALID_VALUE (0xFFFFFFFFU)
An invalid calibration value.

#define RAIL_MAX_RF_PATHS 2
Indicates the maximum number of RF Paths supported across all platforms.

#define RAIL_IRCALVALUES_RX_UNINIT undefined
A define to set all RAIL_RxIrCalValues_t values to uninitialized.

#define RAIL_IRCALVALUES_TX_UNINIT undefined
A define to set all RAIL_TxIrCalValues_t values to uninitialized.

#define RAIL_IRCALVALUES_UNINIT undefined
A define to set all RAIL_IrCalValues_t values to uninitialized.

#define RAIL_IRCALVAL (irCalStruct, rfPath)
A define allowing Rx calibration value access compatibility between non-OFDM and OFDM platforms.

#define RAIL_CALVALUES_UNINIT RAIL_IRCALVALUES_UNINIT
A define to set all RAIL_CalValues_t values to uninitialized.

#define RAIL_PACTUNE_IGNORE (255U)

Use this value with either TX or RX values in RAIL_SetPaCTune to use whatever value is already set and do no update.

Typedef Documentation

RAIL_CalMask_t

```
RAIL_CalMask_t
```

A calibration mask type.

This type is a bitmask of different RAIL calibration values. The exact meaning of these bits depends on what a particular chip supports.

Definition at line 4502 of file common/rail_types.h

RAIL_RxIrCalValues_t

```
typedef uint32_t RAIL_RxIrCalValues_t[2] [2]
```

RAIL_RxIrCalValues_t.

RX IR calibration values.

Platforms with fewer RAIL_RF_PATHS than RAIL_MAX_RF_PATHS will only respect and update RAIL_RF_PATHS indices and ignore the rest.

Definition at line 4550 of file common/rail_types.h

RAIL_CalValues_t

```
RAIL_CalValues_t
```

A calibration value structure.

This structure contains the set of persistent calibration values for EFR32. You can set these beforehand and apply them at startup to save the time required to compute them. Any of these values may be set to RAIL_CAL_INVALID_VALUE to force the code to compute that calibration value.

Definition at line 4631 of file common/rail_types.h

Function Documentation

RAIL_ConfigCal

```
RAIL_Status_t RAIL_ConfigCal (RAIL_Handle_t railHandle, RAIL_CalMask_t calEnable)
```

Initialize RAIL calibration.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	calEnable	A bitmask that indicates which calibrations to enable for a callback notification. The exact meaning of these bits is chip-specific.

Returns

- Status code indicating success of the function call.

Calibration initialization provides the calibration settings that correspond to the current radio configuration.

Definition at line 4912 of file common/rail.h

RAIL_Calibrate

> RAIL_Status_t RAIL_Calibrate (RAIL_Handle_t railHandle, RAIL_CalValues_t *calValues, RAIL_CalMask_t calForce)

Start the calibration process.

Parameters

[in]	railHandle	A RAIL instance handle.
[inout]	calValues	A structure of calibration values to apply. If a valid calibration structure is provided and the structure contains valid calibration values, those values will be applied to the hardware and the RAIL library will cache those values for use again later. If a valid calibration structure is provided and the structure contains a calibration value of RAIL_CAL_INVALID_VALUE for the desired calibration, the desired calibration will run, the calibration values structure will be updated with a valid calibration value, and the RAIL library will cache that value for use again later. If a NULL pointer is provided, the desired calibration will run and the RAIL library will cache that value for use again later. However, the valid calibration value will not be returned to the application.
[in]	calForce	A mask to force specific calibration(s) to execute. To run all pending calibrations, use the value RAIL_CAL_ALL_PENDING. Only the calibrations specified will run, even if not enabled during initialization.

Returns

- Status code indicating success of the function call.

If calibrations were performed previously and the application saves the calibration values (i.e., call this function with a calibration values structure containing calibration values of RAIL_CAL_INVALID_VALUE before a reset), the application can later bypass the time it would normally take to recalibrate hardware by reusing previous calibration values (i.e., call this function with a calibration values structure containing valid calibration values after a reset).

If multiple protocols are used, this function will make the given railHandle active, if not already, and perform calibration. If called during a protocol switch, to perform an IR calibration for the first time, it will return RAIL_STATUS_INVALID_STATE, in which case the application must defer calibration until after the protocol switch is complete. Silicon Labs recommends calling this function from the application main loop.

Note

- Instead of this function, consider using the individual chip-specific functions. Using the individual functions will allow for better dead-stripping if not all calibrations are run.
- Some calibrations should only be executed when the radio is IDLE. See chip-specific documentation for more details.

Definition at line 4958 of file common/rail.h

RAIL_GetPendingCal

> RAIL_CalMask_t RAIL_GetPendingCal (RAIL_Handle_t railHandle)

Return the current set of pending calibrations.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- A mask of all pending calibrations that the user has been asked to perform.

This function returns a full set of pending calibrations. The only way to clear pending calibrations is to perform them using the RAIL_Calibrate() API with the appropriate list of calibrations.

Definition at line 4973 of file common/rail.h

RAIL_ApplyIrCalibration

```
RAIL_Status_t RAIL_ApplyIrCalibration (RAIL_Handle_t railHandle, uint32_t imageRejection)
```

Apply a given image rejection calibration value.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	imageRejection	The image rejection value to apply.

Returns

- A status code indicating success of the function call.

Take an image rejection calibration value and apply it. This value should be determined from a previous run of RAIL_CalibrateIr on the same physical device with the same radio configuration. The imageRejection value will also be stored to the RAIL_ChannelConfigEntry_t::attr, if possible.

If multiple protocols are used, this function will return RAIL_STATUS_INVALID_STATE if it is called and the given railHandle is not active. In that case, the caller must attempt to re-call this function later.

DeprecatedPlease use RAIL_ApplyIrCalibrationAlt instead.

Definition at line 4993 of file common/rail.h

RAIL_ApplyIrCalibrationAlt

```
RAIL_Status_t RAIL_ApplyIrCalibrationAlt (RAIL_Handle_t railHandle, RAIL_IrCalValues_t *imageRejection, RAIL_AntennaSel_t rfPath)
```

Apply a given image rejection calibration value.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	imageRejection	Pointer to the image rejection value to apply.
[in]	rfPath	RF path to calibrate.

Returns

- A status code indicating success of the function call.

Take an image rejection calibration value and apply it. This value should be determined from a previous run of RAIL_CalibrateIrAlt on the same physical device with the same radio configuration. The imageRejection value will also be stored to the RAIL_ChannelConfigEntry_t::attr, if possible. **Note**

- : To make sure the imageRejection value is stored/configured correctly, RAIL_ConfigAntenna should be called before calling this API.

If multiple protocols are used, this function will return RAIL_STATUS_INVALID_STATE if it is called and the given railHandle is not active. In that case, the caller must attempt to re-call this function later.

Definition at line 5015 of file common/rail.h

RAIL_CalibrateIr

```
RAIL_Status_t RAIL_CalibrateIr (RAIL_Handle_t railHandle, uint32_t *imageRejection)
```

Run the image rejection calibration.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	imageRejection	The result of the image rejection calibration.

Returns

- A status code indicating success of the function call.

Run the image rejection calibration and apply the resulting value. If the imageRejection parameter is not NULL, store the value at that location. The imageRejection value will also be stored to the RAIL_ChannelConfigEntry_t::attr, if possible. This is a long-running calibration that adds significant code space when run and can be run with a separate firmware image on each device to save code space in the final image.

If multiple protocols are used, this function will make the given railHandle active, if not already, and perform calibration. If called during a protocol switch, it will return RAIL_STATUS_INVALID_STATE. In this case, RAIL_ApplyIrCalibration may be called to apply a previously determined IR calibration value, or the app must defer calibration until the protocol switch is complete. Silicon Labs recommends calling this function from the application main loop.

DeprecatedPlease use RAIL_CalibrateIrAlt instead.

Definition at line 5044 of file common/rail.h

RAIL_CalibrateIrAlt

```
RAIL_Status_t RAIL_CalibrateIrAlt (RAIL_Handle_t railHandle, RAIL_IrCalValues_t *imageRejection, RAIL_AntennaSel_t rfPath)
```

Run the image rejection calibration.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	imageRejection	Pointer to the image rejection result.
[in]	rfPath	RF path to calibrate.

Returns

- A status code indicating success of the function call.

Run the image rejection calibration and apply the resulting value. If the imageRejection parameter is not NULL, store the value at that location. The imageRejection value will also be stored to the RAIL_ChannelConfigEntry_t::attr, if possible. This is a long-running calibration that adds significant code space when run and can be run with a separate firmware image on each device to save code space in the final image. **Note**

- : To make sure the imageRejection value is stored/configured correctly, RAIL_ConfigAntenna should be called before calling this API.

If multiple protocols are used, this function will return RAIL_STATUS_INVALID_STATE if it is called and the given railHandle is not active. In that case, the caller must attempt to re-call this function later.

Definition at line 5069 of file common/rail.h

RAIL_CalibrateTemp

```
RAIL_Status_t RAIL_CalibrateTemp (RAIL_Handle_t railHandle)
```

Run the temperature calibration.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- A status code indicating success of the function call.

Run the temperature calibration, which needs to recalibrate the synth if the temperature crosses 0C or the temperature delta since the last calibration exceeds 70C while in receive. RAIL will run the VCO calibration automatically upon entering receive or transmit states, so the application can omit this calibration if the stack re-enters receive or transmit with enough frequency to avoid reaching the temperature delta. If the application does not calibrate for temperature, it's possible to miss receive packets due to a drift in the carrier frequency.

If multiple protocols are used, this function will return RAIL_STATUS_INVALID_STATE if it is called and the given railHandle is not active. In that case, the calibration will be automatically performed next time the radio enters receive.

Definition at line 5093 of file common/rail.h

RAIL_CalibrateHFXO

```
RAIL_Status_t RAIL_CalibrateHFXO (RAIL_Handle_t railHandle, int8_t *crystalPPMError)
```

Performs HFXO compensation.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	crystalPPMError	Current deviation that has been corrected, measured in PPM. May be NULL.

Returns

- A status code indicating the result of the function call.

Compute the PPM correction using the thermistor value available when RAIL_EVENT_THERMISTOR_DONE occurs, after RAIL_StartThermistorMeasurement() call. Then correct the RF frequency as well as TX and RX sampling.

This function calls the following RAIL functions in sequence saving having to call them individually:

- RAIL_ConvertThermistorImpedance()
- RAIL_ComputeHFXOPPMError()
- RAIL_CompensateHFXO()

Note

- This function makes the radio idle.

Definition at line 5116 of file common/rail.h

RAIL_EnablePaCal

> void RAIL_EnablePaCal (bool enable)

Enable/disable the PA calibration.

Parameters

[in]	enable	Enables/disables the PA calibration.

Enabling will ensure that the PA power remains constant chip-to-chip. By default, this feature is disabled after reset.

Note

- Call this function before RAIL_ConfigTxPower() if this feature is desired.

Definition at line `5129` of file `common/rail.h`

RAIL_BLE_CalibrateIr

> RAIL_Status_t RAIL_BLE_CalibrateIr (RAIL_Handle_t railHandle, uint32_t *imageRejection)

Calibrate image rejection for Bluetooth Low Energy.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	imageRejection	The result of the image rejection calibration.

Returns

- A status code indicating success of the function call.

Some chips have protocol-specific image rejection calibrations programmed into their flash. This function will either get the value from flash and apply it, or run the image rejection algorithm to find the value.

Definition at line `1671` of file `protocol/ble/rail_ble.h`

RAIL_IEEE802154_CalibrateIr2p4Ghz

> RAIL_Status_t RAIL_IEEE802154_CalibrateIr2p4Ghz (RAIL_Handle_t railHandle, uint32_t *imageRejection)

Calibrate image rejection for IEEE 802.15.4 2.4 GHz.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	imageRejection	The result of the image rejection calibration.

Returns

- A status code indicating success of the function call.

Some chips have protocol-specific image rejection calibrations programmed into their flash. This function will either get the value from flash and apply it, or run the image rejection algorithm to find the value.

Definition at line `1711` of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_CalibrateIrSubGhz

```
RAIL_Status_t RAIL_IEEE802154_CalibrateIrSubGhz (RAIL_Handle_t railHandle, uint32_t *imageRejection)
```

Calibrate image rejection for IEEE 802.15.4 915 MHz and 868 MHz.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	imageRejection	The result of the image rejection calibration.

Returns

- A status code indicating success of the function call.

Some chips have protocol-specific image rejection calibrations programmed into their flash. This function will either get the value from flash and apply it, or run the image rejection algorithm to find the value.

DeprecatedPlease use RAIL_CalibrateIrAlt instead.

Definition at line `1727` of file `protocol/ieee802154/rail_ieee802154.h`

Macro Definition Documentation

RAIL_CAL_TEMP_VCO

```
#define RAIL_CAL_TEMP_VCO
```

Value:

```
(0x00000001U)
```

EFR32-specific temperature calibration bit.

Definition at line `4505` of file `common/rail_types.h`

RAIL_CAL_TEMP_HFXO

```
#define RAIL_CAL_TEMP_HFXO
```

Value:

```
(0x00000002U)
```

EFR32-specific HFXO temperature check bit.

(Ignored if platform lacks RAIL_SUPPORTS_HFXO_COMPENSATION.)

Definition at line `4508` of file `common/rail_types.h`

RAIL_CAL_COMPENSATE_HFXO

```
#define RAIL_CAL_COMPENSATE_HFXO
```

Value:

```
(0x00000004U)
```

EFR32-specific HFXO compensation bit.

(Ignored if platform lacks RAIL_SUPPORTS_HFXO_COMPENSATION.)

Definition at line 4511 of file common/rail_types.h

RAIL_CAL_RX_IRCAL

```
#define RAIL_CAL_RX_IRCAL
```

Value:

```
(0x00010000U)
```

EFR32-specific IR calibration bit.

Definition at line 4513 of file common/rail_types.h

RAIL_CAL_OFDM_TX_IRCAL

```
#define RAIL_CAL_OFDM_TX_IRCAL
```

Value:

```
(0x00100000U)
```

EFR32-specific Tx IR calibration bit.

(Ignored if platform lacks RAIL_SUPPORTS_OFDM_PA.)

Definition at line 4516 of file common/rail_types.h

RAIL_CAL_ONETIME_IRCAL

```
#define RAIL_CAL_ONETIME_IRCAL
```

Value:

```
(RAIL_CAL_RX_IRCAL | RAIL_CAL_OFDM_TX_IRCAL)
```

A mask to run EFR32-specific IR calibrations.

Definition at line 4519 of file common/rail_types.h

RAIL_CAL_TEMP

```
#define RAIL_CAL_TEMP
```

Value:

```
(RAIL_CAL_TEMP_VCO | RAIL_CAL_TEMP_HFXO | RAIL_CAL_COMPENSATE_HFXO)
```

A mask to run temperature-dependent calibrations.

Definition at line 4521 of file common/rail_types.h

RAIL_CAL_ONETIME

```
#define RAIL_CAL_ONETIME
```

Value:

```
(RAIL_CAL_ONETIME_IRCAL)
```

A mask to run one-time calibrations.

Definition at line 4523 of file common/rail_types.h

RAIL_CAL_PERF

```
#define RAIL_CAL_PERF
```

Value:

```
(0)
```

A mask to run optional performance calibrations.

Definition at line 4525 of file common/rail_types.h

RAIL_CAL_OFFLINE

```
#define RAIL_CAL_OFFLINE
```

Value:

```
(RAIL_CAL_ONETIME_IRCAL)
```

A mask for calibrations that require the radio to be off.

Definition at line 4527 of file common/rail_types.h

RAIL_CAL_ALL

```
#define RAIL_CAL_ALL
```

Value:

```
(RAIL_CAL_TEMP | RAIL_CAL_ONETIME)
```

A mask to run all possible calibrations for this chip.

Definition at line 4529 of file common/rail_types.h

RAIL_CAL_ALL_PENDING

```
#define RAIL_CAL_ALL_PENDING
```

Value:

```
(0x00000000U)
```

A mask to run all pending calibrations.

Definition at line 4531 of file common/rail_types.h

RAIL_CAL_INVALID_VALUE

```
#define RAIL_CAL_INVALID_VALUE
```

Value:

```
(0xFFFFFFFFU)
```

An invalid calibration value.

Definition at line 4533 of file common/rail_types.h

RAIL_MAX_RF_PATHS

```
#define RAIL_MAX_RF_PATHS
```

Value:

```
2
```

Indicates the maximum number of RF Paths supported across all platforms.

Definition at line 4540 of file common/rail_types.h

RAIL_IRCALVALUES_RX_UNINIT

```
#define RAIL_IRCALVALUES_RX_UNINIT
```

Value:

```
{                \
  [0 ... RAIL_MAX_RF_PATHS - 1] = RAIL_CAL_INVALID_VALUE, \
}
```

A define to set all RAIL_RxIrCalValues_t values to uninitialized.

This define can be used when you have no data to pass to the calibration routines but wish to compute and save all possible calibrations.

Definition at line 4558 of file common/rail_types.h

RAIL_IRCALVALUES_TX_UNINIT

```
#define RAIL_IRCALVALUES_TX_UNINIT
```

Value:

```
0   {\
0     RAIL_CAL_INVALID_VALUE,       \
0     RAIL_CAL_INVALID_VALUE,       \
0   }
```

A define to set all RAIL_TxIrCalValues_t values to uninitialized.

This define can be used when you have no data to pass to the calibration routines but wish to compute and save all possible calibrations.

Definition at line 4585 of file common/rail_types.h

RAIL_IRCALVALUES_UNINIT

```
#define RAIL_IRCALVALUES_UNINIT
```

Value:

```
0   {\
0     RAIL_IRCALVALUES_RX_UNINIT,  \
0     RAIL_IRCALVALUES_TX_UNINIT,  \
0   }
```

A define to set all RAIL_IrCalValues_t values to uninitialized.

This define can be used when you have no data to pass to the calibration routines but wish to compute and save all possible calibrations.

Definition at line 4610 of file common/rail_types.h

RAIL_IRCALVAL

```
#define RAIL_IRCALVAL
```

Value:

```
(irCalStruct, rfPath)
```

A define allowing Rx calibration value access compatibility between non-OFDM and OFDM platforms.

Definition at line 4619 of file common/rail_types.h

RAIL_CALVALUES_UNINIT

```
#define RAIL_CALVALUES_UNINIT
```

Value:

```
RAIL_IRCALVALUES_UNINIT
```

A define to set all RAIL_CalValues_t values to uninitialized.

This define can be used when you have no data to pass to the calibration routines but wish to compute and save all possible calibrations.

Definition at line 4639 of file common/rail_types.h

RAIL_PACTUNE_IGNORE

```
#define RAIL_PACTUNE_IGNORE
```

Value:

```
(255U)
```

Use this value with either TX or RX values in RAIL_SetPaCTune to use whatever value is already set and do no update.

This value is provided to provide consistency across EFR32 chips, but technically speaking, all PA capacitance tuning values are invalid on EFR32XG21 parts, as RAIL_SetPaCTune is not supported on those parts.

Definition at line 4647 of file common/rail_types.h

RAIL_TxIrCalValues_t

A Tx IR calibration value structure.

This definition contains the set of persistent calibration values for OFDM on EFR32. You can set these beforehand and apply them at startup to save the time required to compute them. Any of these values may be set to RAIL_IRCAL_INVALID_VALUE to force the code to compute that calibration value.

Only supported on platforms with RAIL_SUPPORTS_OFDM_PA enabled.

Public Attributes

uint32_t	dcOffsetIQ TXIRCAL result.
uint32_t	phiEpsilon TXIRCAL result.

Public Attribute Documentation

dcOffsetIQ

```
uint32_t RAIL_TxIrCalValues_t::dcOffsetIQ
```

TXIRCAL result.

Definition at line 4575 of file common/rail_types.h

phiEpsilon

```
uint32_t RAIL_TxIrCalValues_t::phiEpsilon
```

TXIRCAL result.

Definition at line 4576 of file common/rail_types.h

RAIL_IrCalValues_t

An IR calibration value structure.

This definition contains the set of persistent calibration values for EFR32. You can set these beforehand and apply them at startup to save the time required to compute them. Any of these values may be set to RAIL_IRCAL_INVALID_VALUE to force the code to compute that calibration value.

Public Attributes

RAIL_RxIrCalValues_t	rxIrCalValues RX Image Rejection (IR) calibration value.
RAIL_TxIrCalValues_t	txIrCalValues TX Image Rejection (IR) calibration value for OFDM.

Public Attribute Documentation

rxIrCalValues

```
RAIL_RxIrCalValues_t RAIL_IrCalValues_t::rxIrCalValues
```

RX Image Rejection (IR) calibration value.

Definition at line 4600 of file common/rail_types.h

txIrCalValues

```
RAIL_TxIrCalValues_t RAIL_IrCalValues_t::txIrCalValues
```

TX Image Rejection (IR) calibration value for OFDM.

Definition at line 4601 of file common/rail_types.h

EFR32

EFR32

EFR32-specific Calibrations.

Modules

RAIL_ChannelConfigEntryAttr

Macros

#define RAIL_RF_PATHS 1
Indicates the number of RF Paths supported.

Macro Definition Documentation

RAIL_RF_PATHS

```
#define RAIL_RF_PATHS
```

Value:

```
1
```

Indicates the number of RF Paths supported.

Definition at line 209 of file chip/efr32/efr32xg1x/rail_chip_specific.h

RAIL_ChannelConfigEntryAttr

A channel configuration entry attribute structure.

Items listed are designed to be altered and updated during run-time.

Public Attributes

RAIL_RxIrCalValues_t	calValues IR calibration attributes specific to each channel configuration entry.

Public Attribute Documentation

calValues

```
RAIL_RxIrCalValues_t RAIL_ChannelConfigEntryAttr_t::calValues
```

IR calibration attributes specific to each channel configuration entry.

Definition at line 222 of file chip/efr32/efr32xg1x/rail_chip_specific.h

Chip-Specific

Chip-Specific

Chip-Specific RAIL APIs, types, and information.

Modules

EFR32xG1x_Interrupts

EFR32xG2x_Interrupts

EFR32xG1x_Interrupts

EFR32xG1x-specific interrupt sources.

Below are the interrupt handlers implemented within the RAIL library for EFR32 Series 1. See EFR32 for more details about interrupt initialization and priorities. For more information about the interrupt vectors and where they are on your specific chip, see the startup_efr32xg1xx.c file for your chip. These can be found under the Device/SiliconLabs folder in your release.

Warnings

- RAC_RSM_IRQHandler() and RAC_SEQ_IRQHandler() must be set to the same priority level.

Public Functions

void FRC_PRI_IRQHandler(void)
RAIL interrupt handler for the high-priority FRC_PRI interrupt source.

void FRC_IRQHandler(void)
RAIL interrupt handler for the normal-priority FRC interrupt source.

void MODEM_IRQHandler(void)
RAIL interrupt handler for the MODEM interrupt source.

void RAC_SEQ_IRQHandler(void)
RAIL interrupt handler for the SEQ interrupt source.

void RAC_RSM_IRQHandler(void)
RAIL interrupt handler for the RAC_RSM interrupt source.

void BUFC_IRQHandler(void)
RAIL interrupt handler for the BUFC interrupt source.

void AGC_IRQHandler(void)
RAIL interrupt handler for the AGC interrupt source.

void PROTIMER_IRQHandler(void)
RAIL interrupt handler for the PROTIMER interrupt source.

void SYNTH_IRQHandler(void)
RAIL interrupt handler for the SYNTH interrupt source.

void RFSENSE_IRQHandler(void)
RAIL interrupt handler for the RFSENSE interrupt source.

void PRORTC_IRQHandler(void)
RAIL interrupt handler for the PRORTC interrupt source.

Public Function Documentation

FRC_PRI_IRQHandler

> void EFR32xG1x_Interrupts::FRC_PRI_IRQHandler (void)

RAIL interrupt handler for the high-priority FRC_PRI interrupt source.

Parameters

N/A		

This source handles very time-sensitive high-priority interrupt events for the FRC. It is enabled only on EFR32xG12 for IEEE 802.15.4 Dynamic FEC (see RAIL_IEEE802154_SUPPORTS_G_DYNFEC and RAIL_IEEE802154_G_OPTION_DYNFEC). When enabled, RAIL sets its priority to the highest possible, above ATOMIC. There are no corresponding RAIL_Events_t events for these interrupts.

Note

- The interrupt vector should point to the RAIL interrupt handler for compatibility with future RAIL releases.

Definition at line `58` of file `chip/efr32/efr32xg1x/rail_efr32xg1x_interrupts.h`

FRC_IRQHandler

> void EFR32xG1x_Interrupts::FRC_IRQHandler (void)

RAIL interrupt handler for the normal-priority FRC interrupt source.

Parameters

N/A		

This source handles packet completion events for the transmit side. The receive side events are queued and processed in the BUFC_IRQHandler. The following events are currently in use:

- TX Completed
  - This event cannot be disabled and is used to clean up after a transmit event. If enabled, one of the following events will be generated after transmit completion: RAIL_EVENT_TX_PACKET_SENT, RAIL_EVENT_TX_ABORTED, RAIL_EVENT_TX_BLOCKED, or RAIL_EVENT_TX_UNDERFLOW.
- TX ACK Completed
  - This event is only turned on if the user calls RAIL_ConfigEvents() with one of RAIL_EVENT_TXACK_PACKET_SENT, RAIL_EVENT_TXACK_ABORTED, RAIL_EVENT_TXACK_BLOCKED, or RAIL_EVENT_TXACK_UNDERFLOW. If enabled, the corresponding event will be generated when an ACK completes.

Definition at line `80` of file `chip/efr32/efr32xg1x/rail_efr32xg1x_interrupts.h`

MODEM_IRQHandler

> void EFR32xG1x_Interrupts::MODEM_IRQHandler (void)

RAIL interrupt handler for the MODEM interrupt source.

Parameters

N/A		

This source handles demodulator and modulator status and error events. The following events are currently in use:

- Preamble Detection
  - This event is enabled by RAIL_ConfigEvents() when given RAIL_EVENT_RX_PREAMBLE_DETECT. This event will call the RAIL eventsCallback when a preamble is detected by the demodulator if enabled.
- Preamble Lost
  - This event is enabled by RAIL_ConfigEvents() when given RAIL_EVENT_RX_PREAMBLE_LOST. This event will call the RAIL eventsCallback when a preamble is lost by the demodulator if enabled.
- Timing Detection

This event is enabled by RAIL_ConfigEvents() when given RAIL_EVENT_RX_TIMING_DETECT. This event will call the RAIL eventsCallback when timing is detected by the demodulator if enabled.

- Timing Lost
  - This event is enabled by RAIL_ConfigEvents() when given RAIL_EVENT_RX_TIMING_LOST. This event will call the RAIL eventsCallback when timing is lost by the demodulator if enabled.
- Sync 1 Detection
  - This event is enabled by RAIL_ConfigEvents() when given RAIL_EVENT_RX_SYNC1_DETECT. This event will call the RAIL eventsCallback when sync word 1 is detected by the demodulator if enabled.
- Sync 2 Detection
  - This event is enabled by RAIL_ConfigEvents() when given RAIL_EVENT_RX_SYNC2_DETECT. This event will call the RAIL eventsCallback when sync word 2 is detected by the demodulator if enabled.

Definition at line 117 of file chip/efr32/efr32xg1x/rail_efr32xg1x_interrupts.h

RAC_SEQ_IRQHandler

> void EFR32xG1x_Interrupts::RAC_SEQ_IRQHandler (void)

RAIL interrupt handler for the SEQ interrupt source.

Parameters

N/A		

This source handles interrupts generated from the radio sequencer. The following events are currently in use:

- RX timeout
  - This event cannot be disabled and occurs when a receive timeout, as configured via RAIL_SetStateTiming()'s RAIL_StateTiming_t::rxSearchTimeout or RAIL_StateTiming_t::txToRxSearchTimeout, expires and the radio exits receive. This event will call the generic callback if RAIL_EVENT_RX_TIMEOUT is enabled.
- ACK Timeout
  - This event is enabled by RAIL_ConfigEvents() when given RAIL_EVENT_RX_ACK_TIMEOUT. This event is only available when using Auto-ACK or IEEE 802.15.4 and will call the eventsCallback whenever an ACK packet is not received within the ACK timeout window.
- Address Filtering Passed
  - This event is enabled by RAIL_ConfigEvents() when given RAIL_EVENT_RX_FILTER_PASSED. This event is only enabled when using Address Filtering and will call the generic callback whenever a packet passes the filtering criteria.
- IEEE802154 Frame Pending
  - This event is enabled by RAIL_ConfigEvents() when given RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND. This event is only available when in IEEE 802.15.4 mode and will call the eventsCallback whenever a data request packet is received so you can set a frame-pending indicator in the outgoing ACK.
- Temperature Calibration
  - This event is enabled by RAIL_ConfigEvents() when given RAIL_EVENT_CAL_NEEDED. This event will fire whenever the temperature changes enough to require recalibration during the RX state.
- PA Voltage High
  - This event cannot be disabled and occurs once per transmit while the voltage on V_{PAVDD} is at or above the maximum rating. When this interrupt occurs, the output power of the PA for the current transmit operation is reduced by 31 RAIL_TxPowerLevel_t codes if the configured power level is between 222 and RAIL_TX_POWER_LEVEL_HP_MAX. Configured output power is restored after the transmit operation is complete.
- PA Output High
  - This event cannot be disabled and occurs while the voltage swing on the PA output pin is over the maximum rating. This interrupt can occur multiple times per transmit operation and when this interrupt occurs the power level of the PA is reduced by 31 RAIL_TxPowerLevel_t codes. Configured output power is restored after the transmit operation is complete.

Definition at line 139 of file chip/efr32/efr32xg1x/rail_efr32xg1x_interrupts.h

RAC_RSM_IRQHandler

> void EFR32xG1x_Interrupts::RAC_RSM_IRQHandler (void)

RAIL interrupt handler for the RAC_RSM interrupt source.

Parameters

N/A		

This source handles interrupts generated from the radio state machine. The following events are currently in use:

- Radio State Changed
  - Enabled in some debug modes.

Definition at line `150` of file `chip/efr32/efr32xg1x/rail_efr32xg1x_interrupts.h`

BUFC_IRQHandler

> void EFR32xG1x_Interrupts::BUFC_IRQHandler (void)

RAIL interrupt handler for the BUFC interrupt source.

Parameters

N/A		

This source handles interrupts generated from the radio buffer controller. The following events are currently in use:

- RX Buffer Threshold
  - This event is enabled by RAIL_ConfigEvents() when given RAIL_EVENT_RX_FIFO_ALMOST_FULL. This event will trigger an eventsCallback when the contents of the receive buffer exceed the receive buffer threshold set by RAIL_SetRxFifoThreshold().
- RX Buffer Underflow
  - This event cannot be disabled and indicates that too much data was read from the RX buffer. It should not occur in normal operation as all receive APIs protect against it.
- RX Buffer Corruption
  - This event cannot be disabled and indicates the pointers in the RX buffer have become corrupt. This event should not occur in normal operation.
- TX Buffer Threshold
  - This event is enabled by RAIL_ConfigEvents() when given RAIL_EVENT_TX_FIFO_ALMOST_EMPTY. This event will trigger an eventsCallback when the contents of the transmit buffer fall beneath the transmit threshold set by RAIL_SetTxFifoThreshold().
- TX Buffer Overflow
  - This event cannot be disabled and indicates too much data was placed into the TX buffer. This event should not occur in normal operation as all transmit APIs protect against it.
- TX Buffer Underflow
  - This event cannot be disabled and occurs when the transmit FIFO runs out of bytes before the end of the packet. This event will always perform any necessary cleanup and will call the eventsCallback if RAIL_EVENT_TX_UNDERFLOW is enabled.
- TX Buffer Corruption
  - This event cannot be disabled and indicates the pointers in the TX buffer have become corrupt. This event should not occur in normal operation.
- RX Entry Buffer Threshold
  - This event cannot be disabled when in receive mode and is used to indicate that a new packet is available in the receive buffer. This event will call the eventsCallback with any of the following if they are enabled: RAIL_EVENT_RX_PACKET_RECEIVED, RAIL_EVENT_RX_ADDRESS_FILTERED, RAIL_EVENT_RX_PACKET_ABORTED, RAIL_EVENT_RX_FRAME_ERROR, or RAIL_EVENT_RX_FIFO_OVERFLOW. The RAIL_EVENT_RX_ADDRESS_FILTERED event is only used with Address Filtering to indicate termination of a packet that does not meet the filtering criteria.
- RX Entry Buffer Underflow
  -

This event cannot be disabled and indicates that too much data was read from the RX entry buffer. This event should not occur in normal operation.
- RX Entry Buffer Overflow
  - This event cannot be disabled and will fire one time if the RX entry buffer overflows. It will cause the RAIL_EVENT_RX_FIFO_OVERFLOW event to happen later after all valid entries are processed in the RX entry buffer.
- RX Entry Buffer Underflow
  - This event cannot be disabled and indicates that too much data was read from the RX entry buffer. This event should not occur in normal operation.
- RX Entry Buffer Corruption
  - This event cannot be disabled and indicates the pointers in the RX entry buffer have become corrupt. This event should not occur in normal operation.

Definition at line 229 of file chip/efr32/efr32xg1x/rail_efr32xg1x_interrupts.h

AGC_IRQHandler

> void EFR32xG1x_Interrupts::AGC_IRQHandler (void)

RAIL interrupt handler for the AGC interrupt source.

Parameters

N/A		

This source handles interrupts generated from the automatic gain control block. The following events are currently in use:

- RSSI Valid
  - This event is enabled by RAIL_StartAverageRssi() and occurs when the first RSSI sample is ready after going into RX state. It will cause the eventsCallback to be called with RAIL_EVENT_RSSI_AVERAGE_DONE set.

Definition at line 243 of file chip/efr32/efr32xg1x/rail_efr32xg1x_interrupts.h

PROTIMER_IRQHandler

> void EFR32xG1x_Interrupts::PROTIMER_IRQHandler (void)

RAIL interrupt handler for the PROTIMER interrupt source.

Parameters

N/A		

This source handles interrupts generated from the RAIL timebase (PROTIMER). The following events are currently in use:

- Scheduled TX/RX
  - This event cannot be disabled and occurs when a scheduled transmit or receive begins.
- RX scheduled window end timeout
  - This event cannot be disabled and occurs when the scheduled receive window ends and the radio exits receive. This event will call the eventsCallback if RAIL_EVENT_RX_SCHEDULED_RX_END is enabled.
- Transmit with CCA Failed
  - This event cannot be disabled and occurs when CCA indicates the channel is busy. This event will call the eventsCallback if RAIL_EVENT_TX_CHANNEL_BUSY is enabled.
- Transmit with CCA Success
  - This event cannot be disabled and occurs when CCA indicates the channel is clear and the packet will be transmitted. This event will call the eventsCallback if RAIL_EVENT_TX_CHANNEL_CLEAR is enabled.
- Starting a CCA check
  - This event is enabled by RAIL_ConfigEvents() when given RAIL_EVENT_TX_START_CCA. It will occur when the radio begins warming up for a CCA check.
- CCA Retry

This event is enabled by RAIL_ConfigEvents() when given RAIL_EVENT_TX_CCA_RETRY. It will occur whenever CSMA or LBT have a failed channel check and are going to begin retrying. Note that on the EFR32xG1 platform this is enabled by default to handle issues with certain CCA threshold values.

- RAIL Timer
  - This event is enabled by RAIL_SetTimer() and will call the provided callback when the timer expires.

Definition at line 287 of file chip/efr32/efr32xg1x/rail_efr32xg1x_interrupts.h

SYNTH_IRQHandler

> void EFR32xG1x_Interrupts::SYNTH_IRQHandler (void)

RAIL interrupt handler for the SYNTH interrupt source.

Parameters

N/A		

Currently, there are no interrupt sources enabled within this group.

Note

- The interrupt vector should still point to the RAIL interrupt handler for compatibility with future RAIL releases.

Definition at line 297 of file chip/efr32/efr32xg1x/rail_efr32xg1x_interrupts.h

RFSENSE_IRQHandler

> void EFR32xG1x_Interrupts::RFSENSE_IRQHandler (void)

RAIL interrupt handler for the RFSENSE interrupt source.

Parameters

N/A		

This source handles interrupts generated from the RF Sense module. The following events are currently in use:

- RF Sensed
  - This event is enabled by when the RFSENSE block is initialized using RAIL_StartRfSense() and can be disabled by turning off the RFSENSE block using RAIL_StartRfSense() with the band set to RAIL_RFSENSE_OFF. This event will possibly generate a callback provided to RAIL_StartRfSense whenever the RFSENSE block detects RF energy.

Definition at line 312 of file chip/efr32/efr32xg1x/rail_efr32xg1x_interrupts.h

PRORTC_IRQHandler

> void EFR32xG1x_Interrupts::PRORTC_IRQHandler (void)

RAIL interrupt handler for the PRORTC interrupt source.

Parameters

N/A		

This source handles interrupts from the internal RTC source on chips that support it (EFR32XG13 and EFR32XG14). It is not used at this time.

Note

- The interrupt vector should still point to the RAIL interrupt handler for compatibility with future RAIL releases.

Definition at line `323` of file `chip/efr32/efr32xg1x/rail_efr32xg1x_interrupts.h`

EFR32xG2x_Interrupts

EFR32xG2x-specific interrupt sources.

Below are the interrupt handlers implemented within the RAIL library for EFR32 Series 2. See EFR32 for more details about interrupt initialization and priorities. For more information about the interrupt vectors and where they are on your specific chip, see the startup_efr32xg2xx.c file for your chip. These can be found under the Device/SiliconLabs folder in your release.

Warnings

- RAC_RSM_IRQHandler() and RAC_SEQ_IRQHandler() must be set to the same priority level.

Public Functions

void FRC_PRI_IRQHandler(void)
RAIL interrupt handler for the high-priority FRC_PRI interrupt source.

void FRC_IRQHandler(void)
RAIL interrupt handler for the normal-priority FRC interrupt source.

void MODEM_IRQHandler(void)
RAIL interrupt handler for the MODEM interrupt source.

void RAC_SEQ_IRQHandler(void)
RAIL interrupt handler for the SEQ interrupt source.

void RAC_RSM_IRQHandler(void)
RAIL interrupt handler for the RAC_RSM interrupt source.

void BUFC_IRQHandler(void)
RAIL interrupt handler for the BUFC interrupt source.

void AGC_IRQHandler(void)
RAIL interrupt handler for the AGC interrupt source.

void PROTIMER_IRQHandler(void)
RAIL interrupt handler for the PROTIMER interrupt source.

void SYNTH_IRQHandler(void)
RAIL interrupt handler for the SYNTH interrupt source.

void RFSENSE_IRQHandler(void)
RAIL interrupt handler for the RFSENSE interrupt source.

void PRORTC_IRQHandler(void)
RAIL interrupt handler for the PRORTC interrupt source.

void HOSTMAILBOX_IRQHandler(void)
RAIL interrupt handler for the HOSTMAILBOX interrupt source.

void RDMAILBOX_IRQHandler(void)
RAIL interrupt handler for the RDMAILBOX interrupt source.

void RFECA0_IRQHandler(void)
RAIL interrupt handler for the RFECA0 interrupt source.

void RFECA1_IRQHandler(void)
RAIL interrupt handler for the RFECA1 interrupt source.

void RFTIMER_IRQHandler(void)
RAIL interrupt handler for the RFTIMER interrupt source.

void SOFTM_IRQHandler(void)
RAIL interrupt handler for the SOFTM interrupt source.

void RFLDMA_IRQHandler(void)
RAIL interrupt handler for the RFDMA interrupt source.

void SYSRTC_SEQ_IRQHandler(void)
RAIL interrupt handler for the SYSRTC SEQ interrupt source.

void EMUDG_IRQHandler(void)
RAIL interrupt handler for the EMUDG interrupt source.

Public Function Documentation

FRC_PRI_IRQHandler

> void EFR32xG2x_Interrupts::FRC_PRI_IRQHandler (void)

RAIL interrupt handler for the high-priority FRC_PRI interrupt source.

Parameters

N/A		

This source handles interrupts from the FRC High Priority. It is not used at this time.

Note

- The interrupt vector should still point to the RAIL interrupt handler for compatibility with future RAIL releases.

Definition at line `53` of file `chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h`

FRC_IRQHandler

> void EFR32xG2x_Interrupts::FRC_IRQHandler (void)

RAIL interrupt handler for the normal-priority FRC interrupt source.

Parameters

N/A		

This source handles packet completion events for the transmit side. The receive side events are queued and processed in the BUFC_IRQHandler. The following events are currently in use:

- TX Completed
  - This event cannot be disabled and is used to clean up after a transmit event. If enabled, one of the following events will be generated after transmit completion: RAIL_EVENT_TX_PACKET_SENT, RAIL_EVENT_TX_ABORTED, RAIL_EVENT_TX_BLOCKED, or RAIL_EVENT_TX_UNDERFLOW.
- TX ACK Completed
  - This event is only turned on if the user calls RAIL_ConfigEvents() with one of RAIL_EVENT_TXACK_PACKET_SENT, RAIL_EVENT_TXACK_ABORTED, RAIL_EVENT_TXACK_BLOCKED, or RAIL_EVENT_TXACK_UNDERFLOW. If enabled, the corresponding event will be generated when an ACK completes.

Definition at line 58 of file chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h

MODEM_IRQHandler

> void EFR32xG2x_Interrupts::MODEM_IRQHandler (void)

RAIL interrupt handler for the MODEM interrupt source.

Parameters

N/A		

This source handles demodulator and modulator status and error events. The following events are currently in use:

- Preamble Detection
  - This event is enabled by RAIL_ConfigEvents() when given RAIL_EVENT_RX_PREAMBLE_DETECT. This event will call the RAIL eventsCallback when a preamble is detected by the demodulator if enabled.
- Preamble Lost
  - This event is enabled by RAIL_ConfigEvents() when given RAIL_EVENT_RX_PREAMBLE_LOST. This event will call the RAIL eventsCallback when a preamble is lost by the demodulator if enabled.
- Timing Detection
  - This event is enabled by RAIL_ConfigEvents() when given RAIL_EVENT_RX_TIMING_DETECT. This event will call the RAIL eventsCallback when timing is detected by the demodulator if enabled.
- Timing Lost
  - This event is enabled by RAIL_ConfigEvents() when given RAIL_EVENT_RX_TIMING_LOST. This event will call the RAIL eventsCallback when timing is lost by the demodulator if enabled.
- Sync 1 Detection
  - This event is enabled by RAIL_ConfigEvents() when given RAIL_EVENT_RX_SYNC1_DETECT. This event will call the RAIL eventsCallback when sync word 1 is detected by the demodulator if enabled.
- Sync 2 Detection
  - This event is enabled by RAIL_ConfigEvents() when given RAIL_EVENT_RX_SYNC2_DETECT. This event will call the RAIL eventsCallback when sync word 2 is detected by the demodulator if enabled.

Definition at line 63 of file chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h

RAC_SEQ_IRQHandler

> void EFR32xG2x_Interrupts::RAC_SEQ_IRQHandler (void)

RAIL interrupt handler for the SEQ interrupt source.

Parameters

N/A		

This source handles interrupts generated from the radio sequencer. The following events are currently in use:

- RX timeout
  - This event cannot be disabled and occurs when a receive timeout, as configured via RAIL_SetStateTiming()'s RAIL_StateTiming_t::rxSearchTimeout or RAIL_StateTiming_t::txToRxSearchTimeout, expires and the radio exits receive. This event will call the generic callback if RAIL_EVENT_RX_TIMEOUT is enabled.
- ACK Timeout
  - This event is enabled by RAIL_ConfigEvents() when given RAIL_EVENT_RX_ACK_TIMEOUT. This event is only available when using Auto-ACK or IEEE 802.15.4 and will call the eventsCallback whenever an ACK packet is not received within the ACK timeout window.
- Address Filtering Passed
  - This event is enabled by RAIL_ConfigEvents() when given RAIL_EVENT_RX_FILTER_PASSED. This event is only enabled when using Address Filtering and will call the generic callback whenever a packet passes the filtering criteria.
- IEEE802154 Frame Pending

This event is enabled by RAIL_ConfigEvents() when given RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND. This event is only available when in IEEE 802.15.4 mode and will call the eventsCallback whenever a data request packet is received so you can set a frame-pending indicator in the outgoing ACK.

- Temperature Calibration
  - This event is enabled by RAIL_ConfigEvents() when given RAIL_EVENT_CAL_NEEDED. This event will fire whenever the temperature changes enough to require recalibration during the RX state.

Definition at line 103 of file chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h

RAC_RSM_IRQHandler

> void EFR32xG2x_Interrupts::RAC_RSM_IRQHandler (void)

RAIL interrupt handler for the RAC_RSM interrupt source.

Parameters

N/A		

This source handles interrupts generated from the radio state machine. The following events are currently in use:

- Radio State Changed
  - Enabled in some debug modes.

Definition at line 108 of file chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h

BUFC_IRQHandler

> void EFR32xG2x_Interrupts::BUFC_IRQHandler (void)

RAIL interrupt handler for the BUFC interrupt source.

Parameters

N/A		

This source handles interrupts generated from the radio buffer controller. The following events are currently in use:

- RX Buffer Threshold
  - This event is enabled by RAIL_ConfigEvents() when given RAIL_EVENT_RX_FIFO_ALMOST_FULL. This event will trigger an eventsCallback when the contents of the receive buffer exceed the receive buffer threshold set by RAIL_SetRxFifoThreshold().
- RX Buffer Underflow
  - This event cannot be disabled and indicates that too much data was read from the RX buffer. It should not occur in normal operation as all receive APIs protect against it.
- RX Buffer Corruption
  - This event cannot be disabled and indicates the pointers in the RX buffer have become corrupt. This event should not occur in normal operation.
- TX Buffer Threshold
  - This event is enabled by RAIL_ConfigEvents() when given RAIL_EVENT_TX_FIFO_ALMOST_EMPTY. This event will trigger an eventsCallback when the contents of the transmit buffer fall beneath the transmit threshold set by RAIL_SetTxFifoThreshold().
- TX Buffer Overflow
  - This event cannot be disabled and indicates too much data was placed into the TX buffer. This event should not occur in normal operation as all transmit APIs protect against it.
- TX Buffer Underflow
  - This event cannot be disabled and occurs when the transmit FIFO runs out of bytes before the end of the packet. This event will always perform any necessary cleanup and will call the eventsCallback if RAIL_EVENT_TX_UNDERFLOW is enabled.

TX Buffer Corruption
  - This event cannot be disabled and indicates the pointers in the TX buffer have become corrupt. This event should not occur in normal operation.
- RX Entry Buffer Threshold
  - This event cannot be disabled when in receive mode and is used to indicate that a new packet is available in the receive buffer. This event will call the eventsCallback with any of the following if they are enabled: RAIL_EVENT_RX_PACKET_RECEIVED, RAIL_EVENT_RX_ADDRESS_FILTERED, RAIL_EVENT_RX_PACKET_ABORTED, RAIL_EVENT_RX_FRAME_ERROR, or RAIL_EVENT_RX_FIFO_OVERFLOW. The RAIL_EVENT_RX_ADDRESS_FILTERED event is only used with Address Filtering to indicate termination of a packet that does not meet the filtering criteria.
- RX Entry Buffer Underflow
  - This event cannot be disabled and indicates that too much data was read from the RX entry buffer. This event should not occur in normal operation.
- RX Entry Buffer Overflow
  - This event cannot be disabled and will fire one time if the RX entry buffer overflows. It will cause the RAIL_EVENT_RX_FIFO_OVERFLOW event to happen later after all valid entries are processed in the RX entry buffer.
- RX Entry Buffer Underflow
  - This event cannot be disabled and indicates that too much data was read from the RX entry buffer. This event should not occur in normal operation.
- RX Entry Buffer Corruption
  - This event cannot be disabled and indicates the pointers in the RX entry buffer have become corrupt. This event should not occur in normal operation.

Definition at line `113` of file `chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h`

AGC_IRQHandler

> void EFR32xG2x_Interrupts::AGC_IRQHandler (void)

RAIL interrupt handler for the AGC interrupt source.

Parameters

N/A		

This source handles interrupts generated from the automatic gain control block. The following events are currently in use:

- RSSI Valid
  - This event is enabled by RAIL_StartAverageRssi() and occurs when the first RSSI sample is ready after going into RX state. It will cause the eventsCallback to be called with RAIL_EVENT_RSSI_AVERAGE_DONE set.

Definition at line `118` of file `chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h`

PROTIMER_IRQHandler

> void EFR32xG2x_Interrupts::PROTIMER_IRQHandler (void)

RAIL interrupt handler for the PROTIMER interrupt source.

Parameters

N/A		

This source handles interrupts generated from the RAIL timebase (PROTIMER). The following events are currently in use:

- Scheduled TX/RX
  - This event cannot be disabled and occurs when a scheduled transmit or receive begins.
- RX scheduled window end timeout
  - This event cannot be disabled and occurs when the scheduled receive window ends and the radio exits receive. This event will call the eventsCallback if RAIL_EVENT_RX_SCHEDULED_RX_END is enabled.

Transmit with CCA Failed
  - This event cannot be disabled and occurs when CCA indicates the channel is busy. This event will call the eventsCallback if RAIL_EVENT_TX_CHANNEL_BUSY is enabled.
- Transmit with CCA Success
  - This event cannot be disabled and occurs when CCA indicates the channel is clear and the packet will be transmitted. This event will call the eventsCallback if RAIL_EVENT_TX_CHANNEL_CLEAR is enabled.
- Starting a CCA check
  - This event is enabled by RAIL_ConfigEvents() when given RAIL_EVENT_TX_START_CCA. It will occur when the radio begins warming up for a CCA check.
- CCA Retry
  - This event is enabled by RAIL_ConfigEvents() when given RAIL_EVENT_TX_CCA_RETRY. It will occur whenever CSMA or LBT have a failed channel check and are going to begin retrying. Note that on the EFR32xG1 platform this is enabled by default to handle issues with certain CCA threshold values.
- RAIL Timer
  - This event is enabled by RAIL_SetTimer() and will call the provided callback when the timer expires.

Definition at line 123 of file chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h

SYNTH_IRQHandler

> void EFR32xG2x_Interrupts::SYNTH_IRQHandler (void)

RAIL interrupt handler for the SYNTH interrupt source.

Parameters

N/A		

This source handles interrupts generated from the synth module. The following events are currently in use:

- Radio Sequencer Handshake
  - This event is used in to keep the main core awake during some radio operations.

Definition at line 135 of file chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h

RFSENSE_IRQHandler

> void EFR32xG2x_Interrupts::RFSENSE_IRQHandler (void)

RAIL interrupt handler for the RFSENSE interrupt source.

Parameters

N/A		

This source handles interrupts generated from the RF Sense module. This source is not available on all series-2 devices. The following events are currently in use:

- RF Sensed
  - This event is enabled by when the RFSENSE block is initialized using RAIL_StartRfSense() and can be disabled by turning off the RFSENSE block using RAIL_StartRfSense() with the band set to RAIL_RFSENSE_OFF. This event will possibly generate a callback provided to RAIL_StartRfSense whenever the RFSENSE block detects RF energy.

Definition at line 151 of file chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h

PRORTC_IRQHandler

> void EFR32xG2x_Interrupts::PRORTC_IRQHandler (void)

RAIL interrupt handler for the PRORTC interrupt source.

Parameters

N/A		

This source handles interrupts from the internal RTC source on chips that support it (EFR32XG21 and EFR32XG22). It is not used at this time.

Note

- The interrupt vector should still point to the RAIL interrupt handler for compatibility with future RAIL releases.

Definition at line `162` of file `chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h`

HOSTMAILBOX_IRQHandler

> void EFR32xG2x_Interrupts::HOSTMAILBOX_IRQHandler (void)

RAIL interrupt handler for the HOSTMAILBOX interrupt source.

Parameters

N/A		

This source handles interrupts generated from the mailbox between the radio sequencer and main core. This source is not available on all series-2 devices. The following events are currently in use:

- Mailbox Event
  - Mailbox message from the radio sequencer to the main core.

Definition at line `175` of file `chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h`

RDMAILBOX_IRQHandler

> void EFR32xG2x_Interrupts::RDMAILBOX_IRQHandler (void)

RAIL interrupt handler for the RDMAILBOX interrupt source.

Parameters

N/A		

This interrupt source was renamed. See /ref HOSTMAILBOX_IRQHandler for more details.

Definition at line `183` of file `chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h`

RFECA0_IRQHandler

> void EFR32xG2x_Interrupts::RFECA0_IRQHandler (void)

RAIL interrupt handler for the RFECA0 interrupt source.

Parameters

N/A		

Currently, there are no interrupt sources enabled within this group. This source is not available on all series-2 devices.

Note

- The interrupt vector should still point to the RAIL interrupt handler for compatibility with future RAIL releases.

Definition at line 194 of file chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h

RFECA1_IRQHandler

> void EFR32xG2x_Interrupts::RFECA1_IRQHandler (void)

RAIL interrupt handler for the RFECA1 interrupt source.

Parameters

N/A		

Currently, there are no interrupt sources enabled within this group. This source is not available on all series-2 devices.

Note

- The interrupt vector should still point to the RAIL interrupt handler for compatibility with future RAIL releases.

Definition at line 205 of file chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h

RFTIMER_IRQHandler

> void EFR32xG2x_Interrupts::RFTIMER_IRQHandler (void)

RAIL interrupt handler for the RFTIMER interrupt source.

Parameters

N/A		

Currently, there are no interrupt sources enabled within this group. This source is not available on all series-2 devices.

Note

- The interrupt vector should still point to the RAIL interrupt handler for compatibility with future RAIL releases.

Definition at line 216 of file chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h

SOFTM_IRQHandler

> void EFR32xG2x_Interrupts::SOFTM_IRQHandler (void)

RAIL interrupt handler for the SOFTM interrupt source.

Parameters

N/A		

This source handles demodulator status and error events on some phys configurations. This source is not available on all series-2 devices.

The following events are currently in use:

- Preamble Detection
  -

This event is enabled by RAIL_ConfigEvents() when given RAIL_EVENT_RX_PREAMBLE_DETECT. This event will call the RAIL eventsCallback when a preamble is detected by the demodulator if enabled.

- Preamble Lost
  - This event is enabled by RAIL_ConfigEvents() when given RAIL_EVENT_RX_PREAMBLE_LOST. This event will call the RAIL eventsCallback when a preamble is lost by the demodulator if enabled.
- Timing Detection
  - This event is enabled by RAIL_ConfigEvents() when given RAIL_EVENT_RX_TIMING_DETECT. This event will call the RAIL eventsCallback when timing is detected by the demodulator if enabled.
- Timing Lost
  - This event is enabled by RAIL_ConfigEvents() when given RAIL_EVENT_RX_TIMING_LOST. This event will call the RAIL eventsCallback when timing is lost by the demodulator if enabled.
- Sync 1 Detection
  - This event is enabled by RAIL_ConfigEvents() when given RAIL_EVENT_RX_SYNC1_DETECT. This event will call the RAIL eventsCallback when sync word 1 is detected by the demodulator if enabled.
- Sync 2 Detection
  - This event is enabled by RAIL_ConfigEvents() when given RAIL_EVENT_RX_SYNC2_DETECT. This event will call the RAIL eventsCallback when sync word 2 is detected by the demodulator if enabled.

Definition at line `256` of file `chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h`

RFLDMA_IRQHandler

> void EFR32xG2x_Interrupts::RFLDMA_IRQHandler (void)

RAIL interrupt handler for the RFDMA interrupt source.

Parameters

N/A		

Currently, there are no interrupt sources enabled within this group. This source is not available on all series-2 devices.

Note

- The interrupt vector should still point to the RAIL interrupt handler for compatibility with future RAIL releases.

Definition at line `267` of file `chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h`

SYSRTC_SEQ_IRQHandler

> void EFR32xG2x_Interrupts::SYSRTC_SEQ_IRQHandler (void)

RAIL interrupt handler for the SYSRTC SEQ interrupt source.

Parameters

N/A		

This source handles system rtc interrupts for the radio subsystem. This source is not available on all series-2 devices.

The following events are currently in use:

- SYSRTC compare channel
  - This event is enabled by the rail_power_manager and is only used during rail timebase syncronization after sleep.

Definition at line `280` of file `chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h`

EMUDG_IRQHandler

void EFR32xG2x_Interrupts::EMUDG_IRQHandler (void)

RAIL interrupt handler for the EMUDG interrupt source.

Parameters

N/A		

This source handles multiple features based on temperature such as:

- HFXO compensation defined by RAIL_SUPPORTS_HFXO_COMPENSATION.
  - This feature is enabled by RAIL_ConfigHFXOCompensation().
- Chip thermal protection defined by RAIL_SUPPORTS_THERMAL_PROTECTION.
  - This feature is enabled by RAIL_ConfigThermalProtection().
- OFDM PA compensation over temperature defined by RAIL_SUPPORTS_OFDM_PA
  - This feature is automatically enabled when an OFDM PHY is loaded.
- VCO calibration
  - This feature is automatically enabled.

This source is not available on all series-2 devices.

Definition at line `300` of file `chip/efr32/efr32xg2x/rail_efr32xg2x_interrupts.h`

Data Management

Data Management

Data management functions.

These functions allow the application to choose how data is presented to the application. RAIL provides data in a packet-based method or in a FIFO-based method. As originally conceived, RAIL_DataMethod_t::PACKET_MODE was designed for handling packets that fit within RAIL's FIFOs while RAIL_DataMethod_t::FIFO_MODE was designed for handling packets larger than RAIL's FIFOs could hold. Conceptually it is still useful to think of these modes this way, but functionally their distinction has become blurred by improvements in RAIL's flexibility – applications now have much more control over both receive and transmit FIFO sizes, and the FIFO-management and threshold APIs and related events are no longer restricted to RAIL_DataMethod_t::FIFO_MODE operation but can be used in RAIL_DataMethod_t::PACKET_MODE too.

The application can configure RAIL data management through RAIL_ConfigData(). This function allows the application to specify the type of radio data (RAIL_TxDataSource_t and RAIL_RxDataSource_t) and the method of interacting with data (RAIL_DataMethod_t). By default, RAIL configures TX and RX both with packet data source and RAIL_DataMethod_t::PACKET_MODE.

For transmit, RAIL_DataMethod_t::PACKET_MODE and RAIL_DataMethod_t::FIFO_MODE are functionally the same:

- When not actively transmitting, load a packet's initial transmit data using RAIL_WriteTxFifo() with reset set to true. Alternatively this data copying can be avoided by changing the transmit FIFO to an already-loaded section of memory with RAIL_SetTxFifo().
- When actively transmitting, load remaining transmit data with RAIL_WriteTxFifo() with reset set to false.
- If transmit packets exceed the FIFO size, set the transmit FIFO threshold through RAIL_SetTxFifoThreshold(). The RAIL_Config_t::eventsCallback with RAIL_EVENT_TX_FIFO_ALMOST_EMPTY will occur telling the application to load more TX packet data, if needed, to prevent a RAIL_EVENT_TX_UNDERFLOW event from occurring. One can get how much space is available in the transmit FIFO for more transmit data through RAIL_GetTxFifoSpaceAvailable().
- After transmit completes, the transmit FIFO can be manually reset with RAIL_ResetFifo(), but this should rarely be necessary.

The transmit FIFO is specified by the application and its size is the value returned from the most recent call to RAIL_SetTxFifo(). The transmit FIFO is edge-based in that it only provides the RAIL_EVENT_TX_FIFO_ALMOST_EMPTY event once when the threshold is crossed in the emptying direction.

For receive, the distinction between RAIL_DataMethod_t::PACKET_MODE and RAIL_DataMethod_t::FIFO_MODE basically boils down to how unsuccessfully-received packets are handled. In RAIL_DataMethod_t::PACKET_MODE, data from such packets is automatically rolled back as if the packet was never received, while in RAIL_DataMethod_t::FIFO_MODE, rollback does not occur putting more onus on the application to deal with that data.

In receive RAIL_DataMethod_t::PACKET_MODE data management:

- Packet lengths are determined from the Radio Configurator configuration and can be read out at the end using RAIL_GetRxPacketInfo().
- Received packet data is made available on successful packet completion via RAIL_Config_t::eventsCallback with RAIL_EVENT_RX_PACKET_RECEIVED which can then use RAIL_GetRxPacketInfo() and RAIL_GetRxPacketDetailsAlt() to access packet information and RAIL_PeekRxPacket() to access packet data.
- Filtered, Aborted, or FrameError received packet data is automatically rolled back (dropped) without the application needing to worry about consuming it. The application can choose to not even be bothered with the events related to such packets: RAIL_EVENT_RX_ADDRESS_FILTERED, RAIL_EVENT_RX_PACKET_ABORTED, or RAIL_EVENT_RX_FRAME_ERROR.

In receive RAIL_DataMethod_t::FIFO_MODE data management:

- Packet Lengths are determined from the Radio Configurator configuration or by application knowledge of packet payload structure.
- Received data can be retrieved prior to packet completion through RAIL_ReadRxFifo() and is never rolled back on Filtered, Aborted, or FrameError packets. The application should enable and handle these events so it can flush any packet data it's already retrieved.
- After packet completion, remaining packet data for Filtered, Aborted, or FrameError packets remains in the FIFO and the appropriate event is triggered to the user. This data may be consumed in the callback unlike in packet mode where it is automatically rolled back. At the end of the callback all remaining data in the FIFO will be cleaned up as usual. Keep in mind that RAIL_GetRxPacketDetailsAlt() provides packet detailed information only for successfully received packets.

Common receive data management features:

- Set the receive FIFO threshold through RAIL_SetRxFifoThreshold(). The RAIL_Config_t::eventsCallback with RAIL_EVENT_RX_FIFO_ALMOST_FULL will occur telling the application to consume some RX packet data to prevent a RAIL_EVENT_RX_FIFO_OVERFLOW event from occurring.
- Get receive FIFO count information through RAIL_GetRxPacketInfo(RAIL_RX_PACKET_HANDLE_NEWEST) (or RAIL_GetRxFifoBytesAvailable()).
- After receive completes and all its data has been consumed, the receive FIFO can be manually reset with RAIL_ResetFifo(), though this should rarely be necessary and should only be done with the radio idle.

When trying to determine an appropriate threshold, the application needs to know the size of each FIFO. The default receive FIFO is internal to RAIL with a size of 512 bytes. This can be changed, however, using RAIL_SetRxFifo() and the default may be removed entirely by calling this from the RAILCb_SetupRxFifo() callback. The receive FIFO event is level-based in that the RAIL_EVENT_RX_FIFO_ALMOST_FULL event will constantly pend if the threshold is exceeded. This normally means that inside this event's callback, the application should empty enough of the FIFO to go under the threshold. To defer reading the FIFO to main context, the application can disable or re-enable the receive FIFO threshold event using RAIL_ConfigEvents() with the mask RAIL_EVENT_RX_FIFO_ALMOST_FULL.

The receive FIFO can store multiple packets and processing of a packet can be deferred from the RAIL event callback to main-loop processing by using RAIL_HoldRxPacket() in the event callback and RAIL_ReleaseRxPacket() in the main-loop. On some platforms, the receive FIFO is supplemented by an internal fixed-size packet metadata FIFO that limits the number of packets RAIL and applications can hold onto for deferred processing. See chip-specific documentation, such as EFR32, for more information. Note that when using multiprotocol the receive FIFO is reset prior to a protocol switch so held packets will be lost if not processed before then.

While RAIL_EVENT_RX_FIFO_ALMOST_FULL occurs solely based on the state of the receive FIFO used for packet data, both RAIL_EVENT_RX_FIFO_FULL and RAIL_EVENT_RX_FIFO_OVERFLOW can occur coincident with packet completion when either that or the internal packet metadata FIFO fills or overflows. RAIL_EVENT_RX_FIFO_FULL informs the application it should immediately process and free up the oldest packets/data to make room for new packets/data, reducing the possibility of packet/data loss and RAIL_EVENT_RX_FIFO_OVERFLOW.

Before a packet is fully received you can always use RAIL_PeekRxPacket() to look at the contents. In FIFO mode, you may also consume its data with RAIL_ReadRxFifo(). Remember that none of these APIs will read across a packet boundary (even in FIFO mode) so you will need to handle each received packet individually.

While RAIL defaults to RAIL_DataMethod_t::PACKET_MODE, the application can explicitly initialize RAIL for RAIL_DataMethod_t::PACKET_MODE in the following manner:

```
extern RAIL_Handle_t railHandle;
static const RAIL_DataConfig_t railDataConfig = {
  .txSource = TX_PACKET_DATA,
  .rxSource = RX_PACKET_DATA,
  .txMethod = PACKET_MODE,
  .rxMethod = PACKET_MODE,
};

status = RAIL_ConfigData(railHandle, &railDataConfig);

// Events that can occur in Packet Mode:
//    RAIL_EVENT_TX_PACKET_SENT
//    RAIL_EVENT_RX_PACKET_RECEIVED
// and optionally (packet data automatically dropped):
//    RAIL_EVENT_RX_ADDRESS_FILTERED
//    RAIL_EVENT_RX_PACKET_ABORTED
//    RAIL_EVENT_RX_FRAME_ERROR
// and if enabled:
//   RAIL_EVENT_TX_UNDERFLOW
//   RAIL_EVENT_TXACK_UNDERFLOW
//   RAIL_EVENT_TX_FIFO_ALMOST_EMPTY
//   RAIL_EVENT_RX_FIFO_ALMOST_FULL
```

Initializing RAIL for RAIL_DataMethod_t::FIFO_MODE requires a few more function calls:

```
extern RAIL_Handle_t railHandle;
static const RAIL_DataConfig_t railDataConfig = {
  .txSource = TX_PACKET_DATA,
  .rxSource = RX_PACKET_DATA,
  .txMethod = FIFO_MODE,
  .rxMethod = FIFO_MODE,
};

status = RAIL_ConfigData(railHandle, &railDataConfig);

// Gets the size of the FIFOs.
// Assume that the transmit and receive FIFOs are the same size
uint16_t fifoSize = RAIL_GetTxFifoSpaceAvailable(railHandle);

// Sets the transmit and receive FIFO thresholds.
// For this example, set the threshold in the middle of each FIFO.
RAIL_SetRxFifoThreshold(railHandle, fifoSize / 2);
RAIL_SetTxFifoThreshold(railHandle, fifoSize / 2);

// Events that can occur in FIFO mode:
//   RAIL_EVENT_TX_FIFO_ALMOST_EMPTY
//   RAIL_EVENT_TX_UNDERFLOW
//   RAIL_EVENT_TXACK_UNDERFLOW
//   RAIL_EVENT_TX_PACKET_SENT
//   RAIL_EVENT_RX_FIFO_ALMOST_FULL
//   RAIL_EVENT_RX_FIFO_OVERFLOW
//   RAIL_EVENT_RX_ADDRESS_FILTERED
//   RAIL_EVENT_RX_PACKET_ABORTED
//   RAIL_EVENT_RX_FRAME_ERROR
//   RAIL_EVENT_RX_PACKET_RECEIVED
```

On receive, an application can use a different RAIL_RxDataSource_t that is only compatible with RAIL_DataMethod_t::FIFO_MODE. All that differs from the FIFO mode example above is the RAIL_DataConfig_t::rxSource setting. IQ data samples are taken at the hardware's oversample rate and the amount of data can easily overwhelm the CPU processing time. The sample rate depends on the chosen PHY, as determined by the data rate and the decimation chain. It is **not** recommended to use the IQ data source with sample rates above 300 k samples/second because the CPU might not be able to keep up with the data stream. Depending on the application and the needed CPU bandwidth, slower data rates may be required. On EFR32xG22 and later platforms, it is recommended to reset the RX buffer before initiating a receive for all modes except RAIL_RxDataSource_t::RX_PACKET_DATA since the RX buffer has to be 32-bit aligned. If the

buffer is **not** reset but is 32-bit aligned, capture is performed on the remaining space available. If the buffer is **not** reset and is **not** 32-bit aligned, then RAIL_ConfigData() returns RAIL_STATUS_INVALID_STATE.

```
// Reset RX buffer (EFR32xG22 and later platforms)
RAIL_ResetFifo(railHandle, false, true);

// IQ data is provided into the receive FIFO.
static const RAIL_DataConfig_t railDataConfig = {
  .txSource = TX_PACKET_DATA,
  .rxSource = RX_IQDATA_FILTLSB,
  .txMethod = FIFO_MODE,
  .rxMethod = FIFO_MODE,
};

// IQ data comes in the following format when reading out of the FIFO:
//-------------------------------------
// I[LSB] | I[MSB] | Q[LSB] | Q[MSB] |
//-------------------------------------
```

Modules

RAIL_DataConfig_t

Enumerations

enum RAIL_TxDataSource_t {

TX_PACKET_DATA
TX_MFM_DATA
RAIL_TX_DATA_SOURCE_COUNT

}

Transmit data sources supported by RAIL.

enum RAIL_RxDataSource_t {

RX_PACKET_DATA
RX_DEMOD_DATA
RX_IQDATA_FILTLSB
RX_IQDATA_FILTMSB
RX_DIRECT_MODE_DATA
RX_DIRECT_SYNCHRONOUS_MODE_DATA
RAIL_RX_DATA_SOURCE_COUNT

}

Receive data sources supported by RAIL.

enum RAIL_DataMethod_t {

PACKET_MODE
FIFO_MODE
RAIL_DATA_METHOD_COUNT

}

Methods for the application to provide and retrieve data from RAIL.

Functions

RAIL_Status_t RAIL_ConfigData(RAIL_Handle_t railHandle, const RAIL_DataConfig_t *dataConfig)

RAIL data management configuration.

uint16_t RAIL_WriteTxFifo(RAIL_Handle_t railHandle, const uint8_t *dataPtr, uint16_t writeLength, bool reset)

Write data to the transmit FIFO previously established by RAIL_SetTxFifo().

uint16_t	RAIL_SetTxFifo(RAIL_Handle_t railHandle, uint8_t *addr, uint16_t initLength, uint16_t size) Set the address of the transmit FIFO, a circular buffer used for TX data.
uint16_t	RAIL_SetTxFifoAlt(RAIL_Handle_t railHandle, uint8_t *addr, uint16_t startOffset, uint16_t initLength, uint16_t size) Set the address of the transmit FIFO, a circular buffer used for TX data which can start at offset distance from the FIFO base address.
RAIL_Status_t	RAIL_SetRxFifo(RAIL_Handle_t railHandle, uint8_t *addr, uint16_t *size) Set the address of the receive FIFO, a circular buffer used for RX data.
RAIL_Status_t	RAILCb_SetupRxFifo(RAIL_Handle_t railHandle) Set up the receive FIFO to use.
uint16_t	RAIL_ReadRxFifo(RAIL_Handle_t railHandle, uint8_t *dataPtr, uint16_t readLength) Read packet data from RAIL's internal receive FIFO.
uint16_t	RAIL_SetTxFifoThreshold(RAIL_Handle_t railHandle, uint16_t txThreshold) Configure the RAIL transmit FIFO almost empty threshold.
uint16_t	RAIL_SetRxFifoThreshold(RAIL_Handle_t railHandle, uint16_t rxThreshold) Configure the RAIL receive FIFO almost full threshold.
uint16_t	RAIL_GetTxFifoThreshold(RAIL_Handle_t railHandle) Get the RAIL transmit FIFO almost empty threshold value.
uint16_t	RAIL_GetRxFifoThreshold(RAIL_Handle_t railHandle) Get the RAIL receive FIFO almost full threshold value.
void	RAIL_ResetFifo(RAIL_Handle_t railHandle, bool txFifo, bool rxFifo) Reset the RAIL transmit and/or receive FIFOs.
uint16_t	RAIL_GetRxFifoBytesAvailable(RAIL_Handle_t railHandle) Get the number of bytes used in the receive FIFO.
uint16_t	RAIL_GetTxFifoSpaceAvailable(RAIL_Handle_t railHandle) Get the number of bytes unused in the transmit FIFO.

Macros

#define	RAIL_FIFO_ALIGNMENT_TYPE uint32_t Fixed-width type indicating the needed alignment for RX and TX FIFOs.
#define	RAIL_FIFO_ALIGNMENT (sizeof(RAIL_FIFO_ALIGNMENT_TYPE)) Alignment that is needed for the RX and TX FIFOs.
#define	RAIL_FIFO_THRESHOLD_DISABLED 0xFFFFU A FIFO threshold value that disables the threshold.

Enumeration Documentation

RAIL_TxDataSource_t

> RAIL_TxDataSource_t

Transmit data sources supported by RAIL.

Enumerator	
TX_PACKET_DATA	Uses the frame hardware to packetize data.
TX_MFM_DATA	Uses the multi-level frequency modulation data.

RAIL_TX_DATA_SOURCE_COUNT	A count of the choices in this enumeration.

Definition at line 2470 of file common/rail_types.h

RAIL_RxDataSource_t

> RAIL_RxDataSource_t

Receive data sources supported by RAIL.

Note

- Data sources other than RX_PACKET_DATA require use of RAIL_DataMethod_t::FIFO_MODE.

Enumerator	
RX_PACKET_DATA	Uses the frame hardware to packetize data.
RX_DEMOD_DATA	Gets 8-bit data output from the demodulator.
RX_IQDATA_FILTLSB	Gets lower 16 bits of I/Q data provided to the demodulator.
RX_IQDATA_FILTMSB	Gets highest 16 bits of I/Q data provided to the demodulator.
RX_DIRECT_MODE_DATA	Gets RX direct mode data output from the demodulator.
RX_DIRECT_SYNCHRONOUS_MODE_DATA	Gets synchronous RX direct mode data output from the demodulator.
RAIL_RX_DATA_SOURCE_COUNT	A count of the choices in this enumeration.

Definition at line 2497 of file common/rail_types.h

RAIL_DataMethod_t

> RAIL_DataMethod_t

Methods for the application to provide and retrieve data from RAIL.

For Transmit the distinction between RAIL_DataMethod_t::PACKET_MODE and RAIL_DataMethod_t::FIFO_MODE has become more cosmetic than functional, as the RAIL_WriteTxFifo() and RAIL_SetTxFifoThreshold() APIs and related RAIL_EVENT_TX_FIFO_ALMOST_EMPTY event can be used in either mode. For Receive the distinction is functionally important because in RAIL_DataMethod_t::PACKET_MODE rollback occurs automatically for unsuccessfully-received packets (RAIL_RxPacketStatus_t ABORT statuses), flushing their data. In RAIL_DataMethod_t::FIFO_MODE rollback is prevented, leaving the data from unsuccessfully-received packets in the receive FIFO for the application to deal with. This allows for packets larger than the receive FIFO size where automatic rollback would corrupt the receive FIFO.

Enumerator	
PACKET_MODE	Packet-based data method.
FIFO_MODE	FIFO-based data method.
RAIL_DATA_METHOD_COUNT	A count of the choices in this enumeration.

Definition at line 2553 of file common/rail_types.h

Function Documentation

RAIL_ConfigData

> RAIL_Status_t RAIL_ConfigData (RAIL_Handle_t railHandle, const RAIL_DataConfig_t *dataConfig)

RAIL data management configuration.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	dataConfig	RAIL data configuration structure.

Returns

- Status code indicating success of the function call.

This function configures how RAIL manages data. The application can configure RAIL to receive data in a packet-based or FIFO-based manner. RAIL_DataMethod_t::FIFO_MODE is necessary to receive packets larger than the radio's receive FIFO. It is also required for receive data sources other than RAIL_RxDataSource_t::RX_PACKET_DATA.

Generally with RAIL_DataMethod_t::FIFO_MODE, the application sets appropriate FIFO thresholds via RAIL_SetTxFifoThreshold() and RAIL_SetRxFifoThreshold() and then enables and handles the RAIL_EVENT_TX_FIFO_ALMOST_EMPTY event callback (to feed more packet data via RAIL_WriteTxFifo() before the FIFO underflows) and the RAIL_EVENT_RX_FIFO_ALMOST_FULL event callback (to consume packet data via RAIL_ReadRxFifo() before the receive FIFO overflows).

When configuring TX for RAIL_DataMethod_t::FIFO_MODE, this function resets the transmit FIFO. When configuring TX or RX for RAIL_DataMethod_t::PACKET_MODE, this function will reset the corresponding FIFO thresholds such that they won't trigger the RAIL_EVENT_RX_FIFO_ALMOST_FULL or RAIL_EVENT_TX_FIFO_ALMOST_EMPTY events.

When RAIL_DataConfig_t::rxMethod is set to RAIL_DataMethod_t::FIFO_MODE, the radio won't drop packet data of aborted or CRC error packets, but will present it to the application to deal with accordingly. On completion of erroneous packets, the RAIL_Config_t::eventsCallback with RAIL_EVENT_RX_PACKET_ABORTED, RAIL_EVENT_RX_FRAME_ERROR, or RAIL_EVENT_RX_ADDRESS_FILTERED will tell the application it can drop any data it read via RAIL_ReadRxFifo() during reception. For CRC error packets when the RAIL_RX_OPTION_IGNORE_CRC_ERRORS RX option is in effect, the application should check for that from the RAIL_RxPacketStatus_t obtained by calling RAIL_GetRxPacketInfo(). RAIL will automatically flush any remaining packet data after reporting one of these packet completion events or the application can explicitly flush it by calling RAIL_ReleaseRxPacket().

When RAIL_DataConfig_t::rxMethod is set to RAIL_DataMethod_t::PACKET_MODE, the radio will roll back (drop) all packet data associated with aborted packets including those with CRC errors (unless configured to ignore CRC errors via the RAIL_RX_OPTION_IGNORE_CRC_ERRORS RX option). The application will never have to deal with packet data from these packets. In either mode, the application can set RX options as needed.

When RAIL_DataConfig_t::rxSource is set to a value other than RX_PACKET_DATA and RAIL_Config_t::eventsCallbackRAIL_EVENT_RX_FIFO_OVERFLOW is enabled RX will be terminated if a RX FIFO overflow occurs. If RAIL_EVENT_RX_FIFO_OVERFLOW is not enabled, data will be discarded until the overflow condition is resolved. To continue capturing data RX must be restarted using RAIL_StartRx().

Definition at line `1814` of file `common/rail.h`

RAIL_WriteTxFifo

```
uint16_t RAIL_WriteTxFifo (RAIL_Handle_t railHandle, const uint8_t *dataPtr, uint16_t writeLength, bool reset)
```

Write data to the transmit FIFO previously established by RAIL_SetTxFifo().

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	dataPtr	An application-provided pointer to transmit data.
[in]	writeLength	A number of bytes to write to the transmit FIFO.
[in]	reset	If true, resets transmit FIFO before writing the data.

Returns

- The number of bytes written to the transmit FIFO.

This function reads data from the provided dataPtr and writes it to the transmit FIFO that was previously established by RAIL_SetTxFifo(). If the requested writeLength exceeds the current number of bytes open in the transmit FIFO, the function only writes until the transmit FIFO is full. The function returns the number of bytes written to the transmit FIFO or returns zero if railHandle is NULL or if the transmit FIFO is full.

Note

- The protocol's packet configuration, as set up by the radio configurator or via RAIL_SetFixedLength(), determines how many bytes of data are consumed from the transmit FIFO for a successful transmit operation, not the writeLength value passed in. If not enough data has been put into the transmit FIFO, a RAIL_EVENT_TX_UNDERFLOW event will occur. If too much data is put into the transmit FIFO, the extra data will either become the first bytes sent in a subsequent packet, or will be thrown away if the FIFO gets reset prior to the next transmit. In general, the proper number of packet bytes to put into the transmit FIFO are all payload bytes except for any CRC bytes, which the packet configuration causes to be sent automatically.
- This function does not create a critical section but, depending on the application, a critical section could be appropriate.

Definition at line 1849 of file common/rail.h

RAIL_SetTxFifo

```
uint16_t RAIL_SetTxFifo (RAIL_Handle_t railHandle, uint8_t *addr, uint16_t initLength, uint16_t size)
```

Set the address of the transmit FIFO, a circular buffer used for TX data.

Parameters

[in]	railHandle	A RAIL instance handle.
[inout]	addr	An appropriately-aligned (see below) pointer to a read-write memory location in RAM used as the transmit FIFO. This memory must persist until the next call to this function or RAIL_SetTxFifoAlt.
[in]	initLength	A number of initial bytes already in the transmit FIFO.
[in]	size	A desired size of the transmit FIFO in bytes.

Returns

- Returns the FIFO size in bytes, 0 if an error occurs.

This function sets the memory location for the transmit FIFO. RAIL_SetTxFifo or RAIL_SetTxFifoAlt must be called at least once before any transmit operations occur.

FIFO size can be determined by the return value of this function. The chosen size is determined based on the available FIFO sizes supported by the hardware. Similarly, some hardware has stricter FIFO alignment requirements; 32-bit alignment provides the maximum portability across all RAIL platforms. For more on supported FIFO sizes and alignments, see chip-specific documentation, such as EFR32. The returned FIFO size will be the closest allowed size less than or equal to the passed in size parameter, unless the size parameter is smaller than the minimum FIFO size, in that case 0 is returned. If the initLength parameter is larger than the returned size, the FIFO will be filled up to its size.

A user may write to the custom memory location directly before calling this function, or use RAIL_WriteTxFifo to write to the memory location after calling this function. Users must specify the initLength for previously-written memory to be set in the transmit FIFO.

This function reserves the block of RAM starting at addr with a length of the returned FIFO size, which is used internally as a circular buffer for the transmit FIFO. It must be able to hold the entire FIFO size. The caller must guarantee that the custom FIFO remains intact and unchanged (except via calls to RAIL_WriteTxFifo) until the next call to this function.

Note

-

The protocol's packet configuration, as set up by the radio configurator or via RAIL_SetFixedLength(), determines how many bytes of data are consumed from the transmit FIFO for a successful transmit operation, not the initLength value passed in. If not enough data has been put into the transmit FIFO, a RAIL_EVENT_TX_UNDERFLOW event will occur. If too much data is put into the transmit FIFO, the extra data will either become the first bytes sent in a subsequent packet, or will be thrown away if the FIFO gets reset prior to the next transmit. In general, the proper number of packet bytes to put into the transmit FIFO are all payload bytes except for any CRC bytes which the packet configuration causes to be sent automatically.

Definition at line 1903 of file common/rail.h

RAIL_SetTxFifoAlt

```
uint16_t RAIL_SetTxFifoAlt (RAIL_Handle_t railHandle, uint8_t *addr, uint16_t startOffset, uint16_t initLength, uint16_t size)
```

Set the address of the transmit FIFO, a circular buffer used for TX data which can start at offset distance from the FIFO base address.

Parameters

[in]	railHandle	A RAIL instance handle.
[inout]	addr	An appropriately-aligned (see RAIL_SetTxFifo description) pointer to a read-write memory location in RAM used as the transmit FIFO. This memory must persist until the next call to this function or RAIL_SetTxFifo.
[in]	startOffset	A number of bytes defining the start position of the TX data from the transmit FIFO base address, only valid if initLength is not 0.
[in]	initLength	The number of valid bytes already in the transmit FIFO after startOffset.
[in]	size	A desired size of the transmit FIFO in bytes.

Returns

- Returns the FIFO size in bytes, 0 if an error occurs.

This function is similar to RAIL_SetTxFifo except a startOffset can be specified to indicate where the transmit packet data starts. This allows an application to place unaligned initial packet data within the aligned transmit FIFO (initLength > 0). Specifying a startOffset will not reduce the FIFO threshold or affect RAIL_GetTxFifoSpaceAvailable(). RAIL_SetTxFifo or RAIL_SetTxFifoAlt must be called at least once before any transmit operations occur. FIFO size handling is quite same as RAIL_SetTxFifo. Only difference is that if the initLength plus startOffset parameters are larger than the returned size, the FIFO will be filled up to its size from startOffset. Note that the startOffset is essentially forgotten after the next transmit – i.e. it applies onto to the next transmit operation, and is not re-established when the transmit FIFO is reset.

Definition at line 1936 of file common/rail.h

RAIL_SetRxFifo

```
RAIL_Status_t RAIL_SetRxFifo (RAIL_Handle_t railHandle, uint8_t *addr, uint16_t *size)
```

Set the address of the receive FIFO, a circular buffer used for RX data.

Parameters

[in]	railHandle	A RAIL instance handle.
[inout]	addr	A pointer to a read-write memory location in RAM used as the receive FIFO. This memory must persist until the next call to this function.
[inout]	size	A desired size of the receive FIFO in bytes. This will be populated with the actual size during the function call.

Returns

- Status code indicating success of the function call.

This function sets the memory location for the receive FIFO. It must be called at least once before any receive operations occur.

Note

- After it is called, any prior receive FIFO is orphaned. To avoid orphaning the default internal 512-byte receive FIFO so it does not unnecessarily consume RAM resources in your application, implement RAILCb_SetupRxFifo() to call this function.

FIFO size can be determined by the return value of this function. The chosen size is determined based on the available FIFO sizes supported by the hardware. Similarly, some hardware has stricter FIFO alignment requirements; 32-bit alignment provides the maximum portability across all RAIL platforms. For more on supported FIFO sizes and alignments, see chip-specific documentation, such as EFR32. The returned FIFO size will be the closest allowed size less than or equal to the passed in size parameter, unless the size parameter is smaller than the minimum FIFO size.

This function reserves the block of RAM starting at addr with a length of size, which is used internally as a circular buffer for the receive FIFO. It must be able to hold the entire FIFO size. The caller must guarantee that the custom FIFO remains intact and unchanged (except via incoming packet data being written) until the next call to this function.

In multiprotocol, RAIL currently shares one receive FIFO across all protocols. This function will return RAIL_STATUS_INVALID_STATE if the requested RAIL_Handle_t is not active.

Definition at line 1979 of file common/rail.h

RAILCb_SetupRxFifo

> RAIL_Status_t RAILCb_SetupRxFifo (RAIL_Handle_t railHandle)

Set up the receive FIFO to use.

Parameters

[in]	railHandle	A RAIL instance handle.

This function is optional to implement.

Returns

- Status code indicating success of the function call.

This function is called during the RAIL_Init process to set up the FIFO to use for received packets. If not implemented by the application, a default implementation from within the RAIL library will be used to initialize an internal default 512-byte receive FIFO.

If this function returns an error, the RAIL_Init process will fail.

During this function, the application should generally call RAIL_SetRxFifo. If that does not happen, the application needs to set up the receive FIFO via a call to RAIL_SetRxFifo before attempting to receive any packets. An example implementation may look like the following:

```
#define RX_FIFO_SIZE 1024
static uint8_t rxFifo[RX_FIFO_SIZE];

RAIL_Status_t RAILCb_SetupRxFifo(RAIL_Handle_t railHandle)
{
  uint16_t rxFifoSize = RX_FIFO_SIZE;
  RAIL_Status_t status = RAIL_SetRxFifo(railHandle, &rxFifo[0], &rxFifoSize);
  if (rxFifoSize != RX_FIFO_SIZE) {
    // We set up an incorrect FIFO size
    return RAIL_STATUS_INVALID_PARAMETER;
  }
```

```
// Allow failures due to multiprotocolreturn RAIL_STATUS_NO_ERROR;}return status;}
```

Definition at line 2018 of file common/rail.h

RAIL_ReadRxFifo

> uint16_t RAIL_ReadRxFifo (RAIL_Handle_t railHandle, uint8_t *dataPtr, uint16_t readLength)

Read packet data from RAIL's internal receive FIFO.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	dataPtr	An application-provided pointer to store data. If NULL, the data is thrown away rather than copied out.
[in]	readLength	A number of packet bytes to read from the FIFO.

Returns

- The number of packet bytes read from the receive FIFO.

This function reads packet data from the head of receive FIFO and writes it to the provided dataPtr. It does not permit reading more data than is available in the FIFO, nor does it permit reading more data than remains in the oldest unreleased packet.

Because this function does not have a critical section, use it only in one context or make sure function calls are protected to prevent buffer corruption.

Warnings

- This function is intended for use only with RAIL_DataMethod_t::FIFO_MODE and should never be called in RAIL_DataMethod_t::PACKET_MODE where it could lead to receive FIFO corruption.

Note

- When reading data from an arriving packet that is not yet complete, keep in mind its data is highly suspect because it has not yet passed any CRC integrity checking. Also note that the packet could be aborted, canceled, or fail momentarily, invalidating its data in Packet mode. Furthermore, there is a small chance towards the end of packet reception that the receive FIFO could include not only packet data received so far, but also some raw radio-appended info detail bytes that RAIL's packet-completion processing will subsequently deal with. It's up to the application to know its packet format well enough to avoid reading this info because it will corrupt the packet's details and possibly corrupt the receive FIFO.

Definition at line 2055 of file common/rail.h

RAIL_SetTxFifoThreshold

> uint16_t RAIL_SetTxFifoThreshold (RAIL_Handle_t railHandle, uint16_t txThreshold)

Configure the RAIL transmit FIFO almost empty threshold.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	txThreshold	The threshold below which the RAIL_EVENT_TX_FIFO_ALMOST_EMPTY event will fire.

Returns

- Configured transmit FIFO threshold value.

This function configures the threshold for the transmit FIFO. When the number of bytes in the transmit FIFO falls below the configured threshold, RAIL_Config_t::eventsCallback will fire with RAIL_EVENT_TX_FIFO_ALMOST_EMPTY set. The txThreshold value should be smaller than or equal to the transmit FIFO size; higher values will be pegged to the FIFO size. A value of 0 or RAIL_FIFO_THRESHOLD_DISABLED will disable the threshold, returning RAIL_FIFO_THRESHOLD_DISABLED.

Definition at line 2076 of file common/rail.h

RAIL_SetRxFifoThreshold

```
uint16_t RAIL_SetRxFifoThreshold (RAIL_Handle_t railHandle, uint16_t rxThreshold)
```

Configure the RAIL receive FIFO almost full threshold.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	rxThreshold	The threshold above which the RAIL_EVENT_RX_FIFO_ALMOST_FULL event will fire.

Returns

- Configured receive FIFO threshold value.

This function configures the threshold for the receive FIFO. When the number of bytes of packet data in the receive FIFO exceeds the configured threshold, RAIL_Config_t::eventsCallback will keep firing with RAIL_EVENT_RX_FIFO_ALMOST_FULL set as long as the number of bytes in the receive FIFO exceeds the configured threshold value. The rxThreshold value should be smaller than the receive FIFO size; anything else, including a value of RAIL_FIFO_THRESHOLD_DISABLED, will disable the threshold, returning RAIL_FIFO_THRESHOLD_DISABLED.

Note

- To avoid sticking in the event handler (even in idle state):
  1. Disable the event (via the config events API or the RAIL_FIFO_THRESHOLD_DISABLED parameter)
  2. Increase FIFO threshold
  3. Read the FIFO (that's not an option in RAIL_DataMethod_t::PACKET_MODE) in the event handler

Definition at line 2104 of file common/rail.h

RAIL_GetTxFifoThreshold

```
uint16_t RAIL_GetTxFifoThreshold (RAIL_Handle_t railHandle)
```

Get the RAIL transmit FIFO almost empty threshold value.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Configured TX Threshold value.

Retrieves the configured TX threshold value.

Definition at line 2115 of file common/rail.h

RAIL_GetRxFifoThreshold

```
uint16_t RAIL_GetRxFifoThreshold (RAIL_Handle_t railHandle)
```

Get the RAIL receive FIFO almost full threshold value.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Configured RX Threshold value.

Retrieves the configured RX threshold value.

Definition at line `2125` of file `common/rail.h`

RAIL_ResetFifo

```
void RAIL_ResetFifo (RAIL_Handle_t railHandle, bool txFifo, bool rxFifo)
```

Reset the RAIL transmit and/or receive FIFOs.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	txFifo	If true, reset the transmit FIFO.
[in]	rxFifo	If true, reset the receive FIFO.

This function can reset each FIFO independently. The application should not reset the receive FIFO while receiving a frame, nor should it reset the transmit FIFO while transmitting a frame.

Definition at line `2138` of file `common/rail.h`

RAIL_GetRxFifoBytesAvailable

```
uint16_t RAIL_GetRxFifoBytesAvailable (RAIL_Handle_t railHandle)
```

Get the number of bytes used in the receive FIFO.

Parameters

[in]	railHandle	A RAIL instance handle.

Only use this function in RX RAIL_DataMethod_t::FIFO_MODE. Apps should use RAIL_GetRxPacketInfo() instead.

Returns

- Number of bytes used in the receive FIFO.

This function indicates how much packet-related data exists in the receive FIFO that could be read.

Note

- The number of bytes returned may not just reflect the current packet's data but could also include raw appended info bytes added after successful packet reception and bytes from subsequently received packets. It is up to the app to never try to consume more than the packet's actual data when using the value returned here in a subsequent call to RAIL_ReadRxFifo(), otherwise the receive FIFO will be corrupted.

Definition at line `2158` of file `common/rail.h`

RAIL_GetTxFifoSpaceAvailable

```
uint16_t RAIL_GetTxFifoSpaceAvailable (RAIL_Handle_t railHandle)
```

Get the number of bytes unused in the transmit FIFO.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Number of bytes unused in the transmit FIFO.

This function indicates how much space is available in the transmit FIFO for writing additional packet data.

Definition at line `2169` of file `common/rail.h`

Macro Definition Documentation

RAIL_FIFO_ALIGNMENT_TYPE

```
#define RAIL_FIFO_ALIGNMENT_TYPE
```

Value:

```
uint32_t
```

Fixed-width type indicating the needed alignment for RX and TX FIFOs.

Definition at line `2461` of file `common/rail_types.h`

RAIL_FIFO_ALIGNMENT

```
#define RAIL_FIFO_ALIGNMENT
```

Value:

```
(sizeof(RAIL_FIFO_ALIGNMENT_TYPE))
```

Alignment that is needed for the RX and TX FIFOs.

Definition at line `2464` of file `common/rail_types.h`

RAIL_FIFO_THRESHOLD_DISABLED

```
#define RAIL_FIFO_THRESHOLD_DISABLED
```

Value:

```
0xFFFFU
```

A FIFO threshold value that disables the threshold.

Definition at line `2571` of file `common/rail_types.h`

RAIL_DataConfig_t

RAIL data configuration structure.

Select the transmit/receive data sources and the method the application uses to provide/retrieve data from RAIL.

Public Attributes

RAIL_TxDataSource_t	txSource Source of TX Data.
RAIL_RxDataSource_t	rxSource Source of RX Data.
RAIL_DataMethod_t	txMethod Method of providing transmit data.
RAIL_DataMethod_t	rxMethod Method of retrieving receive data.

Public Attribute Documentation

txSource

> RAIL_TxDataSource_t RAIL_DataConfig_t::txSource

Source of TX Data.

Definition at line 2581 of file common/rail_types.h

rxSource

> RAIL_RxDataSource_t RAIL_DataConfig_t::rxSource

Source of RX Data.

Definition at line 2582 of file common/rail_types.h

txMethod

> RAIL_DataMethod_t RAIL_DataConfig_t::txMethod

Method of providing transmit data.

Definition at line 2583 of file common/rail_types.h

rxMethod

RAIL_DataMethod_t RAIL_DataConfig_t::rxMethod

Method of retrieving receive data.

Definition at line 2584 of file common/rail_types.h

Diagnostic

Diagnostic

APIs for diagnostic and test chip modes.

Modules

RAIL_DirectModeConfig_t

RAIL_VerifyConfig_t

Enumerations

enum RAIL_StreamMode_t {

RAIL_STREAM_CARRIER_WAVE = 0
RAIL_STREAM_PN9_STREAM = 1
RAIL_STREAM_10_STREAM = 2
RAIL_STREAM_CARRIER_WAVE_PHASENOISE = 3
RAIL_STREAM_RAMP_STREAM = 4
RAIL_STREAM_CARRIER_WAVE_SHIFTED = 5
RAIL_STREAM_MODES_COUNT

}

Possible stream output modes.

Typedefs

typedef int16_t RAIL_FrequencyOffset_t

Type that represents the number of Frequency Offset units.

typedef bool(* RAIL_VerifyCallbackPtr_t)(uint32_t address, uint32_t expectedValue, uint32_t actualValue)

A pointer to a verification callback function.

Functions

RAIL_Status_t RAIL_ConfigDirectMode(RAIL_Handle_t railHandle, const RAIL_DirectModeConfig_t *directModeConfig)

Configure direct mode for RAIL.

RAIL_Status_t RAIL_EnableDirectMode(RAIL_Handle_t railHandle, bool enable)

Enable or disable direct mode for RAIL.

RAIL_Status_t RAIL_EnableDirectModeAlt(RAIL_Handle_t railHandle, bool enableDirectTx, bool enableDirectRx)

Enable or disable direct mode for RAIL.

uint32_t RAIL_GetRadioClockFreqHz(RAIL_Handle_t railHandle)

Get the radio subsystem clock frequency in Hz.

RAIL_Status_t RAIL_SetTune(RAIL_Handle_t railHandle, uint32_t tune)

Set the crystal tuning.

uint32_t	RAIL_GetTune(RAIL_Handle_t railHandle) Get the crystal tuning.
RAIL_Status_t	RAIL_SetTuneDelta(RAIL_Handle_t railHandle, int32_t delta) Set the crystal tuning delta.
int32_t	RAIL_GetTuneDelta(RAIL_Handle_t railHandle) Get the crystal tuning delta on EFR32xG2 series devices.
RAIL_FrequencyOffset_t	RAIL_GetRxFreqOffset(RAIL_Handle_t railHandle) Get the frequency offset.
RAIL_Status_t	RAIL_SetFreqOffset(RAIL_Handle_t railHandle, RAIL_FrequencyOffset_t freqOffset) Set the nominal radio frequency offset.
RAIL_Status_t	RAIL_StartTxStream(RAIL_Handle_t railHandle, uint16_t channel, RAIL_StreamMode_t mode) Start transmitting a stream on a certain channel.
RAIL_Status_t	RAIL_StartTxStreamAlt(RAIL_Handle_t railHandle, uint16_t channel, RAIL_StreamMode_t mode, RAIL_TxOptions_t options) Start transmitting a stream on a certain channel with the ability to select an antenna.
RAIL_Status_t	RAIL_StopTxStream(RAIL_Handle_t railHandle) Stop stream transmission.
RAIL_Status_t	RAIL_StopInfinitePreambleTx(RAIL_Handle_t railHandle) Stop infinite preamble transmission started and start transmitting the rest of the packet.
RAIL_Status_t	RAIL_ConfigVerification(RAIL_Handle_t railHandle, RAIL_VerifyConfig_t *configVerify, RAIL_RadioConfig_t radioConfig, RAIL_VerifyCallbackPtr_t cb) Configure the verification of radio memory contents.
RAIL_Status_t	RAIL_Verify(RAIL_VerifyConfig_t *configVerify, uint32_t durationUs, bool restart) Verify radio memory contents.

Macros

#define	RAIL_FREQUENCY_OFFSET_MAX ((RAIL_FrequencyOffset_t) 0x3FFF) The maximum frequency offset value supported.
#define	RAIL_FREQUENCY_OFFSET_MIN ((RAIL_FrequencyOffset_t) -RAIL_FREQUENCY_OFFSET_MAX) The minimum frequency offset value supported.
#define	RAIL_FREQUENCY_OFFSET_INVALID ((RAIL_FrequencyOffset_t) 0x8000) Specify an invalid frequency offset value.
#define	RAIL_VERIFY_DURATION_MAX 0xFFFFFFFFUL This radio state verification duration indicates to RAIL that all memory contents should be verified by RAIL before returning to the application.

Enumeration Documentation

RAIL_StreamMode_t

> RAIL_StreamMode_t

Possible stream output modes.

Enumerator	
RAIL_STREAM_CARRIER_WAVE	An unmodulated carrier wave.

RAIL_STREAM_PN9_STREAM	PN9 byte sequence.
RAIL_STREAM_10_STREAM	101010 sequence.
RAIL_STREAM_CARRIER_WAVE_PHASENOISE	An unmodulated carrier wave with no change to PLL BW.
RAIL_STREAM_RAMP_STREAM	ramp sequence starting at a different offset for consecutive packets.
RAIL_STREAM_CARRIER_WAVE_SHIFTED	An unmodulated carrier wave not centered on DC but shifted roughly by channel_bandwidth/6 allowing an easy check of the residual DC.
RAIL_STREAM_MODES_COUNT	A count of the choices in this enumeration.

Definition at line `5333` of file `common/rail_types.h`

Typedef Documentation

RAIL_FrequencyOffset_t

> RAIL_FrequencyOffset_t

Type that represents the number of Frequency Offset units.

It is used with RAIL_GetRxFreqOffset() and RAIL_SetFreqOffset().

The units are chip-specific. For EFR32 they are radio synthesizer resolution steps (synthTicks) and is limited to 15 bits. A value of RAIL_FREQUENCY_OFFSET_INVALID means that this value is invalid.

Definition at line `5274` of file `common/rail_types.h`

RAIL_VerifyCallbackPtr_t

> typedef bool(* RAIL_VerifyCallbackPtr_t) (uint32_t address, uint32_t expectedValue, uint32_t actualValue))(uint32_t address, uint32_t expectedValue, uint32_t actualValue)

A pointer to a verification callback function.

Parameters

[in]	address	The address of the data in question.
[in]	expectedValue	The expected value of the data in question.
[in]	actualValue	The actual value of the data in question.

This will be called by the radio state verification feature built into RAIL when a verified memory value is different from its reference value.

Returns

- bool True indicates a data value difference is acceptable. False indicates a data value difference in unacceptable.

Note

- This callback will be issued when an address' value is different from its reference value and either of the following conditions are met: 1) The default radio configuration provided by the radio configurator is used for verification purposes (i.e., a custom radio configuration is not supplied as an input to RAIL_ConfigVerification()), and the radio configurator has flagged the address under question as being verifiable. 2) A custom radio configuration is provided to the verification API (i.e., a custom radio configuration is supplied as an input to RAIL_ConfigVerification()). When providing a custom radio configuration for verification purposes, all addresses in that configuration will be verified, regardless of whether or not the addresses are flagged as verifiable.

Definition at line 5385 of file common/rail_types.h

Function Documentation

RAIL_ConfigDirectMode

> RAIL_Status_t RAIL_ConfigDirectMode (RAIL_Handle_t railHandle, const RAIL_DirectModeConfig_t *directModeConfig)

Configure direct mode for RAIL.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	directModeConfig	Configuration structure to specify direct mode parameters. Default configuration will be used if NULL is passed.

Returns

- RAIL_STATUS_NO_ERROR on success and an error code on failure.

This API configures direct mode and should be called before calling RAIL_EnableDirectMode(). If this function is not called, the following default configuration will be used: **EFR32xG1x** Sync Rx : false Sync Tx : false TX data in (DIN) : EFR32_PC10 RX data out (DOUT) : EFR32_PC11 TX/RX clk out (DCLK) : EFR32_PC9 **EFR32xG2x:** Sync Rx : false Sync Tx : false TX data in (DIN) : EFR32_PA7 RX data out (DOUT) : EFR32_PA5 TX/RX clk out (DCLK) : EFR32_PA6

Warnings

- This API is not safe to use in a multiprotocol app.

Definition at line 5706 of file common/rail.h

RAIL_EnableDirectMode

> RAIL_Status_t RAIL_EnableDirectMode (RAIL_Handle_t railHandle, bool enable)

Enable or disable direct mode for RAIL.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	enable	Whether or not to enable direct mode for TX and RX.

Returns

- RAIL_STATUS_NO_ERROR on success and an error code on failure.
  - See RAIL_EnableDirectModeAlt() for more detailed function description.

Warnings

- New applications should consider using RAIL_EnableDirectModeAlt() for this functionality.

Note

- This feature is only available on certain devices. RAIL_SupportsDirectMode() can be used to check if a particular device supports this feature or not.

Warnings

- This API is not safe to use in a true multiprotocol app.

Definition at line 5728 of file common/rail.h

RAIL_EnableDirectModeAlt

> RAIL_Status_t RAIL_EnableDirectModeAlt (RAIL_Handle_t railHandle, bool enableDirectTx, bool enableDirectRx)

Enable or disable direct mode for RAIL.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	enableDirectTx	Enable direct mode for data being transmitted out of the radio.
[in]	enableDirectRx	Enable direct mode for data being received from the radio.

Returns

- RAIL_STATUS_NO_ERROR on success and an error code on failure.

This API enables or disables the modem and GPIOs for direct mode operation. see RAIL_ConfigDirectMode for information on selecting the correct hardware configuration. If direct mode is enabled, packets are output and input directly to the radio via GPIO and RAIL packet handling is ignored.

Note

- This feature is only available on certain chips. RAIL_SupportsDirectMode() can be used to check if a particular chip supports this feature or not.

Warnings

- this API is not safe to use in a true multiprotocol app.

Definition at line 5753 of file common/rail.h

RAIL_GetRadioClockFreqHz

> uint32_t RAIL_GetRadioClockFreqHz (RAIL_Handle_t railHandle)

Get the radio subsystem clock frequency in Hz.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Radio subsystem clock frequency in Hz.

Definition at line 5763 of file common/rail.h

RAIL_SetTune

> RAIL_Status_t RAIL_SetTune (RAIL_Handle_t railHandle, uint32_t tune)

Set the crystal tuning.

Parameters

[in]	railHandle	A RAIL instance handle.

[in]	tune	A chip-dependent crystal capacitor bank tuning parameter.

Returns

- Status code indicating success of the function call.

Tunes the crystal that the radio depends on to change the location of the center frequency for transmitting and receiving. This function will only succeed if the radio is idle at the time of the call.

Note

- This function proportionally affects the entire chip's timing across all its peripherals, including radio tuning and channel spacing. It is recommended to call this function only when HFXO is not being used, as it can cause disturbance on the HFXO frequency. A separate function, RAIL_SetFreqOffset(), can be used to adjust just the radio tuner without disturbing channel spacing or other chip peripheral timing.
- On EFR32xG2 series devices, this API sets CTUNEXIANA and internally CTUNEXOANA = CTUNEXIANA + delta where delta is set or changed by RAIL_SetTuneDelta. The default delta may not be 0 on some devices.

Definition at line `5787` of file `common/rail.h`

RAIL_GetTune

```
uint32_t RAIL_GetTune (RAIL_Handle_t railHandle)
```

Get the crystal tuning.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- A chip-dependent crystal capacitor bank tuning parameter.

Retrieves the current tuning value used by the crystal that the radio depends on. **Note**

- On EFR32xG2 series devices, this is the CTUNEXIANA value.

Definition at line `5799` of file `common/rail.h`

RAIL_SetTuneDelta

```
RAIL_Status_t RAIL_SetTuneDelta (RAIL_Handle_t railHandle, int32_t delta)
```

Set the crystal tuning delta.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	delta	A chip-dependent crystal capacitor bank tuning delta.

Returns

- Status code indicating success of the function call.

Set the CTUNEXOANA delta for RAIL_SetTune to use on EFR32xG2 series devices: CTUNEXOANA = CTUNEXIANA + delta. This function does not change CTUNE values; call RAIL_SetTune to put a new delta into effect.

Definition at line `5813` of file `common/rail.h`

RAIL_GetTuneDelta

```
int32_t RAIL_GetTuneDelta (RAIL_Handle_t railHandle)
```

Get the crystal tuning delta on EFR32xG2 series devices.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- A chip-dependent crystal capacitor bank tuning delta.

Retrieves the current tuning delta used by RAIL_SetTune. **Note**

- The default delta if RAIL_SetTuneDelta has never been called is device-dependent and may not be 0.

Definition at line `5825` of file `common/rail.h`

RAIL_GetRxFreqOffset

```
RAIL_FrequencyOffset_t RAIL_GetRxFreqOffset (RAIL_Handle_t railHandle)
```

Get the frequency offset.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Returns the measured frequency offset on a received packet. The units are described in the RAIL_FrequencyOffset_t documentation. If this returns RAIL_FREQUENCY_OFFSET_INVALID, it was called while the radio wasn't active and there is no way to get the frequency offset.

Retrieves the measured frequency offset used during the previous received packet, which includes the current radio frequency offset (see RAIL_SetFreqOffset()). If the chip has not been in RX, it returns the nominal radio frequency offset.

Note

- Changing to any non-idle radio state after reception can cause this value to be overwritten so it is safest to capture during packet reception.

Definition at line `5845` of file `common/rail.h`

RAIL_SetFreqOffset

```
RAIL_Status_t RAIL_SetFreqOffset (RAIL_Handle_t railHandle, RAIL_FrequencyOffset_t freqOffset)
```

Set the nominal radio frequency offset.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	freqOffset	RAIL_FrequencyOffset_t parameter (signed, 2's complement).

Returns

-

Status code indicating success of the function call.

This function is used to adjust the radio's tuning frequency slightly up or down. It might be used in conjunction with RAIL_GetRxFreqOffset() after receiving a packet from a peer to adjust the tuner to better match the peer's tuned frequency.

Note

- Unlike RAIL_SetTune(), which affects the entire chip's timing including radio tuning and channel spacing, this function only affects radio tuning without disturbing channel spacing or other chip peripheral timing.

Definition at line `5865` of file `common/rail.h`

RAIL_StartTxStream

> RAIL_Status_t RAIL_StartTxStream (RAIL_Handle_t railHandle, uint16_t channel, RAIL_StreamMode_t mode)

Start transmitting a stream on a certain channel.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	channel	A channel on which to emit a stream.
[in]	mode	Choose the stream mode (PN9, and so on).

Returns

- Status code indicating success of the function call.

Begins streaming onto the given channel. The sources can either be an unmodulated carrier wave or an encoded stream of bits from a PN9 source. All ongoing radio operations will be stopped before transmission begins.

Definition at line `5880` of file `common/rail.h`

RAIL_StartTxStreamAlt

> RAIL_Status_t RAIL_StartTxStreamAlt (RAIL_Handle_t railHandle, uint16_t channel, RAIL_StreamMode_t mode, RAIL_TxOptions_t options)

Start transmitting a stream on a certain channel with the ability to select an antenna.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	channel	A channel on which to emit a stream.
[in]	mode	Choose the stream mode (PN9, and so on).
[in]	options	Choose the TX Antenna option. Takes only RAIL_TX_OPTION_ANTENNA0, RAIL_TX_OPTION_ANTENNA1, RAIL_TX_OPTIONS_DEFAULT or RAIL_TX_OPTIONS_NONE from the RAIL_TxOptions_t. If some other value is used then, transmission is possible on any antenna.

Returns

- Status code indicating success of the function call.

Begins streaming onto the given channel. The sources can either be an unmodulated carrier wave or an encoded stream of bits from a PN9 source. All ongoing radio operations will be stopped before transmission begins.

Definition at line `5902` of file `common/rail.h`

RAIL_StopTxStream

> RAIL_Status_t RAIL_StopTxStream (RAIL_Handle_t railHandle)

Stop stream transmission.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Status code indicating success of the function call.

Halts the transmission started by RAIL_StartTxStream().

Definition at line `5915` of file `common/rail.h`

RAIL_StopInfinitePreambleTx

> RAIL_Status_t RAIL_StopInfinitePreambleTx (RAIL_Handle_t railHandle)

Stop infinite preamble transmission started and start transmitting the rest of the packet.

Parameters

[in]	railHandle	A RAIL instance handle.

This function is only useful for radio configurations that specify an infinite preamble. Call this API only after RAIL_EVENT_TX_STARTED has occurred and the radio is transmitting.

Returns

- Status code indicating success of the function call: RAIL_STATUS_NO_ERROR if infinite preamble was stopped; RAIL_STATUS_INVALID_CALL if the radio isn't configured for infinite preamble; RAIL_STATUS_INVALID_STATE if the radio isn't transmitting.

Definition at line `5932` of file `common/rail.h`

RAIL_ConfigVerification

> RAIL_Status_t RAIL_ConfigVerification (RAIL_Handle_t railHandle, RAIL_VerifyConfig_t *configVerify, RAIL_RadioConfig_t radioConfig, RAIL_VerifyCallbackPtr_t cb)

Configure the verification of radio memory contents.

Parameters

[in]	railHandle	A RAIL instance handle.
[inout]	configVerify	A configuration structure made available to RAIL to perform radio state verification. This structure must be allocated in application global read-write memory. RAIL may modify fields within or referenced by this structure during its operation.
[in]	radioConfig	A pointer to a radioConfig that is to be used as a white list for verifying memory contents.
[in]	cb	A callback that notifies the application of a mismatch in expected vs actual memory contents. A NULL parameter may be passed in if a callback is not provided by the application.

Returns

RAIL_STATUS_NO_ERROR if setup of the verification feature successfully occurred. RAIL_STATUS_INVALID_PARAMETER is returned if the provided railHandle or configVerify structures are invalid.

Definition at line 5952 of file common/rail.h

RAIL_Verify

```
RAIL_Status_t RAIL_Verify (RAIL_VerifyConfig_t *configVerify, uint32_t durationUs, bool restart)
```

Verify radio memory contents.

Parameters

[inout]	configVerify	A configuration structure made available to RAIL to perform radio state verification. This structure must be allocated in application global read-write memory. RAIL may modify fields within or referenced by this structure during its operation.
[in]	durationUs	The duration (in microseconds) for how long memory verification should occur before returning to the application. A value of RAIL_VERIFY_DURATION_MAX indicates that all memory contents should be verified before returning to the application.
[in]	restart	This flag only has meaning if a previous call of this function returned RAIL_STATUS_SUSPENDED. By restarting (true), the verification process starts over from the beginning, or by resuming where verification left off after being suspended (false), verification can proceed towards completion.

Returns

- RAIL_STATUS_NO_ERROR if the contents of all applicable memory locations have been verified. RAIL_STATUS_SUSPENDED is returned if the provided test duration expired but the time was not sufficient to verify all memory contents. By calling RAIL_Verify again, further verification will commence. RAIL_STATUS_INVALID_PARAMETER is returned if the provided verifyConfig structure pointer is not configured for use by the active RAIL handle. RAIL_STATUS_INVALID_STATE is returned if any of the verified memory contents are different from their reference values.

Definition at line 5984 of file common/rail.h

Macro Definition Documentation

RAIL_FREQUENCY_OFFSET_MAX

```
#define RAIL_FREQUENCY_OFFSET_MAX
```

Value:

```
((RAIL_FrequencyOffset_t) 0x3FFF)
```

The maximum frequency offset value supported.

Definition at line 5279 of file common/rail_types.h

RAIL_FREQUENCY_OFFSET_MIN

```
#define RAIL_FREQUENCY_OFFSET_MIN
```

Value:

```
((RAIL_FrequencyOffset_t) -RAIL_FREQUENCY_OFFSET_MAX)
```

The minimum frequency offset value supported.

Definition at line 5284 of file common/rail_types.h

RAIL_FREQUENCY_OFFSET_INVALID

```
#define RAIL_FREQUENCY_OFFSET_INVALID
```

Value:

```
((RAIL_FrequencyOffset_t) 0x8000)
```

Specify an invalid frequency offset value.

This will be returned if you call RAIL_GetRxFreqOffset() at an invalid time.

Definition at line 5290 of file common/rail_types.h

RAIL_VERIFY_DURATION_MAX

```
#define RAIL_VERIFY_DURATION_MAX
```

Value:

```
0xFFFFFFFFUL
```

This radio state verification duration indicates to RAIL that all memory contents should be verified by RAIL before returning to the application.

Definition at line 5360 of file common/rail_types.h

RAIL_DirectModeConfig_t

Allows the user to specify direct mode parameters using RAIL_ConfigDirectMode().

Public Attributes

bool	syncRx Enable synchronous RX DOUT using DCLK vs.
bool	syncTx Enable synchronous TX DIN using DCLK vs.
uint8_t	doutPort RX Data output (DOUT) GPIO port.
uint8_t	doutPin RX Data output (DOUT) GPIO pin.
uint8_t	dclkPort Data clock (DCLK) GPIO port.
uint8_t	dclkPin Data clock (DCLK) GPIO pin.
uint8_t	dinPort TX Data input (DIN) GPIO port.
uint8_t	dinPin TX Data input (DIN) GPIO pin.
uint8_t	doutLoc RX Data output (DOUT) location for doutPort/Pin.
uint8_t	dclkLoc Data clock (DCLK) location for dclkPort/Pin.
uint8_t	dinLoc TX Data input (DIN) location for tdinPort/Pin.

Public Attribute Documentation

syncRx

> bool RAIL_DirectModeConfig_t::syncRx

Enable synchronous RX DOUT using DCLK vs.

asynchronous RX DOUT.

Definition at line 5299 of file common/rail_types.h

syncTx

bool RAIL_DirectModeConfig_t::syncTx

Enable synchronous TX DIN using DCLK vs.

asynchronous TX DIN.

Definition at line 5301 of file common/rail_types.h

doutPort

uint8_t RAIL_DirectModeConfig_t::doutPort

RX Data output (DOUT) GPIO port.

Definition at line 5304 of file common/rail_types.h

doutPin

uint8_t RAIL_DirectModeConfig_t::doutPin

RX Data output (DOUT) GPIO pin.

Definition at line 5306 of file common/rail_types.h

dclkPort

uint8_t RAIL_DirectModeConfig_t::dclkPort

Data clock (DCLK) GPIO port.

Only used in synchronous mode

Definition at line 5309 of file common/rail_types.h

dclkPin

uint8_t RAIL_DirectModeConfig_t::dclkPin

Data clock (DCLK) GPIO pin.

Only used in synchronous mode

Definition at line 5311 of file common/rail_types.h

dinPort

uint8_t RAIL_DirectModeConfig_t::dinPort

TX Data input (DIN) GPIO port.

Definition at line 5314 of file common/rail_types.h

dinPin

```
uint8_t RAIL_DirectModeConfig_t::dinPin
```

TX Data input (DIN) GPIO pin.

Definition at line 5316 of file common/rail_types.h

doutLoc

```
uint8_t RAIL_DirectModeConfig_t::doutLoc
```

RX Data output (DOUT) location for doutPort/Pin.

Only needed on EFR32 Series 1; ignored on other platforms.

Definition at line 5320 of file common/rail_types.h

dclkLoc

```
uint8_t RAIL_DirectModeConfig_t::dclkLoc
```

Data clock (DCLK) location for dclkPort/Pin.

Only needed on EFR32 Series 1; ignored on other platforms.

Definition at line 5323 of file common/rail_types.h

dinLoc

```
uint8_t RAIL_DirectModeConfig_t::dinLoc
```

TX Data input (DIN) location for tdinPort/Pin.

Only needed on EFR32 Series 1; ignored on other platforms.

Definition at line 5326 of file common/rail_types.h

RAIL_VerifyConfig_t

The configuration array provided to RAIL for use by the radio state verification feature.

This structure will be populated with appropriate values by calling RAIL_ConfigVerification(). The application should not set or alter any of these structure elements.

Public Attributes

RAIL_Handle_t	correspondingHandle Internal verification tracking information.
uint32_t	nextIndexToVerify Internal verification tracking information.
RAIL_RadioConfig_t	override Internal verification tracking information.
RAIL_VerifyCallbackPtr_t	cb Internal verification tracking information.

Public Attribute Documentation

correspondingHandle

```
RAIL_Handle_t RAIL_VerifyConfig_t::correspondingHandle
```

Internal verification tracking information.

Definition at line 5398 of file common/rail_types.h

nextIndexToVerify

```
uint32_t RAIL_VerifyConfig_t::nextIndexToVerify
```

Internal verification tracking information.

Definition at line 5400 of file common/rail_types.h

override

```
RAIL_RadioConfig_t RAIL_VerifyConfig_t::override
```

Internal verification tracking information.

Definition at line 5402 of file common/rail_types.h

cb

RAIL_VerifyCallbackPtr_t RAIL_VerifyConfig_t::cb

Internal verification tracking information.

Definition at line 5404 of file common/rail_types.h

Energy Friendly Front End Module (EFF)

Energy Friendly Front End Module (EFF)

APIs for configuring and controlling an attached Energy Friendly Front End Module (EFF).

The EFF is a high-performance, transmit/receive (T/R) front end module (FEM) for sub-GHz EFR32 devices. RAIL includes built-in functionality to transmit and receive via an attached EFF. This functionality optimizes RF performance while ensuring that the EFF stays within safe operating temperature limits.

Configuration and control of the EFF is performed by the Energy Friendly Front End Module (EFF) Utility.

Note

- The EFF is only supported with EFR32XG25 devices.

Modules

RAIL_EffCalConfig_t

RAIL_EffClpcSensorConfig_t

RAIL_EffClpcConfig_t

RAIL_EffClpcResults_t

RAIL_EffConfig_t

Enumerations

enum RAIL_EffDevice_t {

RAIL_EFF_DEVICE_NONE = 0
RAIL_EFF_DEVICE_EFF01A11NMFA0
RAIL_EFF_DEVICE_EFF01B11NMFA0
RAIL_EFF_DEVICE_EFF01A11IMFB0
RAIL_EFF_DEVICE_EFF01B11IMFB0
RAIL_EFF_DEVICE_COUNT

}

EFF part numbers.

enum RAIL_EffLnaMode_t {

RAIL_EFF_LNA_MODE_RURAL = (0x01U << 0)
RAIL_EFF_LNA_MODE_URBAN = (0x01U << 1)
RAIL_EFF_LNA_MODE_BYPASS = (0x01U << 2)
RAIL_EFF_LNA_MODE_COUNT = (0x01U << 3)

}

EFF LNA Modes.

enum RAIL_ClpcEnable_t {

RAIL_EFF_CLPC_DISABLED = 0
RAIL_EFF_CLPC_MODE_CHANGE = 1
RAIL_EFF_CLPC_POWER_SLOW = 2
RAIL_EFF_CLPC_POWER_FAST = 3
RAIL_EFF_CLPC_POWER_BOTH = 4
RAIL_EFF_CLPC_POWER_SLOW_STOPPED = 5
RAIL_EFF_CLPC_POWER_FAST_STOPPED = 6
RAIL_EFF_CLPC_POWER_BOTH_STOPPED = 7
RAIL_EFF_CLPC_COUNT = 8

}

EFF Closed Loop Power Control (CLPC) Enable states.

enum RAIL_EffModeSensor_t {

RAIL_EFF_MODE_SENSOR_FSK_ANTV = 0
RAIL_EFF_MODE_SENSOR_FSK_SAW2 = 1
RAIL_EFF_MODE_SENSOR_OFDM_ANTV = 2
RAIL_EFF_MODE_SENSOR_OFDM_SAW2 = 3
RAIL_EFF_MODE_SENSOR_COUNT

}

EFF Closed Loop Power Control (CLPC) Mode Sensor Indices.

Functions

RAIL_Status_t RAIL_ConfigEff(RAIL_Handle_t genericRailHandle, const RAIL_EffConfig_t *config)

Configure the attached EFF device.

RAIL_Status_t RAIL_GetTemperature(RAIL_Handle_t railHandle, int16_t tempBuffer[((3U)+(6U)+(2U)+(1U))], bool reset)

Get the different temperature measurements in Kelvin done by sequencer or host.

RAIL_Status_t RAIL_GetSetEffClpcControl(RAIL_Handle_t railHandle, uint16_t tempBuffer[(52U)/sizeof(uint16_t)], bool reset)

Get the different EFF Control measurements.

RAIL_Status_t RAIL_GetSetEffClpcFemdata(RAIL_Handle_t railHandle, uint8_t *newMode, bool changeMode)

Copy the current FEM_DATA pin values into newMode.

RAIL_Status_t RAIL_GetSetEffLnaRuralUrbanMv(RAIL_Handle_t railHandle, uint16_t *newTrip, bool changeTrip)

Copy the current Rural to Urban trip voltage into newTrip.

RAIL_Status_t RAIL_GetSetEffLnaUrbanBypassMv(RAIL_Handle_t railHandle, uint16_t *newTrip, bool changeTrip)

Copy the current Urban to Bypass trip voltage into newTrip.

RAIL_Status_t RAIL_GetSetEffLnaUrbanDwellTimeMs(RAIL_Handle_t railHandle, uint32_t *newDwellTime, bool changeDwellTime)

Copy the current Urban dwell time into newDwellTime.

RAIL_Status_t RAIL_GetSetEffLnaBypassDwellTimeMs(RAIL_Handle_t railHandle, uint32_t *newDwellTime, bool changeDwellTime)

Copy the current Bypass dwell time into newDwellTime.

RAIL_Status_t RAIL_GetSetEffClpcFastLoopCal(RAIL_Handle_t railHandle, RAIL_EffModeSensor_t modeSensorIndex, RAIL_EffCalConfig_t *calibrationEntry, bool changeValues)

If changeValues is true, update current CLPC Fast Loop calibration values using the new variables.

RAIL_Status_t RAIL_GetSetEffClpcFastLoopCalSlp(RAIL_Handle_t railHandle, RAIL_EffModeSensor_t modeSensorIndex, int16_t *newSlope1e1MvPerDdbm, int16_t *newoffset290Ddbm, bool changeValues)

If changeValues is true, update current CLPC Fast Loop calibration equations using the new variables.

RAIL_Status_t RAIL_GetSetEffClpcFastLoop(RAIL_Handle_t railHandle, RAIL_EffModeSensor_t modeSensorIndex, uint16_t *newTargetMv, uint16_t *newSlopeMvPerPaLevel, bool changeValues)

If changeValues is true, update current CLPC Fast Loop Target and Slope.

RAIL_Status_t RAIL_GetSetEffClpcEnable(RAIL_Handle_t railHandle, uint8_t *newClpcEnable, bool changeClpcEnable)
Copy the current CLPC Enable in to newClpcEnable.

RAIL_Status_t RAIL_GetSetEffTempThreshold(RAIL_Handle_t railHandle, uint16_t *newThresholdK, bool changeThreshold)
Get and set the EFF temperature threshold.

Macros

#define RAIL_CLPC_MINIMUM_POWER 180
Minimum power for CLPC usage in deci-dBm.

#define RAIL_EFF_TEMP_MEASURE_COUNT (6U)
Number of temperature values provided for the EFF thermal protection.

#define RAIL_EFF_TEMP_MEASURE_DEPRECATED_COUNT (2U)
Number of deprecated temperature values in EFF thermal protection.

#define RAIL_HFXO_TEMP_MEASURE_COUNT (1U)
Number of temperature values provided for HFXO metrics.

#define RAIL_TEMP_MEASURE_COUNT undefined
Total number of temperature values provided by RAIL_GetTemperature().

#define RAIL_EFF_CONTROL_SIZE (52U)
Number of bytes provided by RAIL_GetSetEffClpcControl().

#define RAIL_EFF_SUPPORTS_TRANSMIT (x)
A macro that checks for EFFxx devices that support high power transmit.

#define RAIL_EFF_SUPPORTS_RECEIVE (x)
A macro that checks for EFFxx devices that support receive.

#define RAIL_EFF_TEMP_THRESHOLD_MAX (383U)
Maximum EFF internal temperature in Kelvin, allowing transmissions when RAIL_SUPPORTS_EFF is enabled.

#define RAIL_EFF_MODE_SENSOR_ENUM_NAMES undefined
A macro that is string versions of the calibration enums.

#define RAIL_EFF_CLPC_ENABLE_ENUM_NAMES undefined
A macro that is string versions of the calibration enums.

Enumeration Documentation

RAIL_EffDevice_t

> RAIL_EffDevice_t

EFF part numbers.

The part number of the attached EFF device is passed to RAIL_ConfigEff() as RAIL_EffConfig_t::device. The Energy Friendly Front End Module (EFF) Utility configures and controls the EFF based on the capabilities of the attached EFF.

	Enumerator
RAIL_EFF_DEVICE_NONE	No EFF device attached.
RAIL_EFF_DEVICE_EFF01A11NMFA0	+30 dBm, LNA, QFN24, +105C max ambient
RAIL_EFF_DEVICE_EFF01B11NMFA0	PA Bypass, LNA, QFN24, +105C max ambient.
RAIL_EFF_DEVICE_EFF01A11IMFB0	+30 dBm, LNA, QFN24, +125C max ambient
RAIL_EFF_DEVICE_EFF01B11IMFB0	PA Bypass, LNA, QFN24, +125C max ambient.
RAIL_EFF_DEVICE_COUNT	A count of the choices in this enumeration.

Definition at line 5426 of file common/rail_types.h

RAIL_EffLnaMode_t

> RAIL_EffLnaMode_t

EFF LNA Modes.

The enabled EFF LNA modes are passed toRAIL_ConfigEff() as the RAIL_EffConfig_t::enabledLnaModes. The Energy Friendly Front End Module (EFF) Utility dynamically transitions between enabled LNA modes to maximize receive performance.

Enumerator	
RAIL_EFF_LNA_MODE_RURAL	Rural LNA Mode.
RAIL_EFF_LNA_MODE_URBAN	Urban LNA Mode.
RAIL_EFF_LNA_MODE_BYPASS	Bypass LNA Mode.
RAIL_EFF_LNA_MODE_COUNT	A count of the choices in this enumeration.

Definition at line 5476 of file common/rail_types.h

RAIL_ClpcEnable_t

> RAIL_ClpcEnable_t

EFF Closed Loop Power Control (CLPC) Enable states.

The EFF CLPC Enable state is passed to RAIL_ConfigEff() as the RAIL_EffConfig_t::clpcEnable. The Energy Friendly Front End Module (EFF) Utility uses advanced power controls to tune EFF output to match Surface Acoustic Wave (SAW) filter losses and antenna performance.

Enumerator	
RAIL_EFF_CLPC_DISABLED	CLPC actions are completely disabled.
RAIL_EFF_CLPC_MODE_CHANGE	CLPC actions are completely disabled.
RAIL_EFF_CLPC_POWER_SLOW	CLPC actions allows Slow Loop.
RAIL_EFF_CLPC_POWER_FAST	CLPC actions allows Fast Loop.
RAIL_EFF_CLPC_POWER_BOTH	CLPC actions are completely enabled.
RAIL_EFF_CLPC_POWER_SLOW_STOPPED	CLPC actions allows Slow Loop.
RAIL_EFF_CLPC_POWER_FAST_STOPPED	CLPC actions allows Fast Loop.
RAIL_EFF_CLPC_POWER_BOTH_STOPPED	CLPC actions are completely enabled.
RAIL_EFF_CLPC_COUNT	A count of the choices in this enumeration.

Definition at line 5500 of file common/rail_types.h

RAIL_EffModeSensor_t

> RAIL_EffModeSensor_t

EFF Closed Loop Power Control (CLPC) Mode Sensor Indices.

The mode sensor indices are used to access specific settings with CLPC.

Enumerator

RAIL_EFF_MODE_SENSOR_FSK_ANTV	CLPC FSK ANTV Sensor.
RAIL_EFF_MODE_SENSOR_FSK_SAW2	CLPC FSK SAW2 Sensor.
RAIL_EFF_MODE_SENSOR_OFDM_ANTV	CLPC OFDM ANTV power calibration entry 1.
RAIL_EFF_MODE_SENSOR_OFDM_SAW2	CLPC OFDM SAW2 power calibration entry 1.
RAIL_EFF_MODE_SENSOR_COUNT	A count of the choices in this enumeration.

Definition at line `5531` of file `common/rail_types.h`

Function Documentation

RAIL_ConfigEff

> RAIL_Status_t RAIL_ConfigEff (RAIL_Handle_t genericRailHandle, const RAIL_EffConfig_t *config)

Configure the attached EFF device.

Parameters

[in]	genericRailHandle	A generic RAIL instance handle.
[in]	config	A pointer to a RAIL_EffConfig_t struct that contains configuration data for the EFF.

Returns

- Status code indicating success of the function call.

Definition at line `6095` of file `common/rail.h`

RAIL_GetTemperature

> RAIL_Status_t RAIL_GetTemperature (RAIL_Handle_t railHandle, int16_t tempBuffer[((3U)+(6U)+(2U)+(1U))], bool reset)

Get the different temperature measurements in Kelvin done by sequencer or host.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	tempBuffer	The address of the array that will contain temperatures. tempBuffer array must be at least RAIL_TEMP_MEASURE_COUNT int16_t.
[in]	reset	Reset min, max and average temperature values.

Values that are not populated yet or incorrect are set to 0.

Temperatures, in Kelvin, are stored in tempBuffer such as: tempBuffer[0] is the chip temperature tempBuffer[1] is the minimal chip temperature tempBuffer[2] is the maximal chip temperature

If RAIL_SUPPORTS_EFF is defined tempBuffer[3] is the EFF temperature before Tx tempBuffer[4] is the EFF temperature after Tx tempBuffer[5] is the minimal EFF temperature value before Tx tempBuffer[6] is the minimal EFF temperature value after Tx tempBuffer[7] is the maximal EFF temperature value before Tx tempBuffer[8] is the maximal EFF temperature value after Tx tempBuffer[9] is not used tempBuffer[10] is not used

If RAIL_SUPPORTS_HFXO_COMPENSATION tempBuffer[11] is the HFXO temperature

Returns

- Status code indicating success of the function call.

Definition at line 6139 of file common/rail.h

RAIL_GetSetEffClpcControl

> RAIL_Status_t RAIL_GetSetEffClpcControl (RAIL_Handle_t railHandle, uint16_t tempBuffer[(52U)/sizeof(uint16_t)], bool reset)

Get the different EFF Control measurements.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	tempBuffer	The address of the target array. tempBuffer array must be at least RAIL_EFF_CONTROL_SIZE bytes.
[in]	reset	Reset the EFF Control measurements.

Returns

- Status code indicating success of the function call.

Definition at line 6155 of file common/rail.h

RAIL_GetSetEffClpcFemdata

> RAIL_Status_t RAIL_GetSetEffClpcFemdata (RAIL_Handle_t railHandle, uint8_t *newMode, bool changeMode)

Copy the current FEM_DATA pin values into newMode.

Parameters

[in]	railHandle	A RAIL instance handle
[inout]	newMode	A pointer to a uint8_t that will contain the FEM_DATA pin values
[in]	changeMode	If true, use newMode to update FEM_DATA pin values

If changeMode is true, update FEM_DATA pin values with newMode first.

Returns

- Status code indicating success of the function call.

Definition at line 6168 of file common/rail.h

RAIL_GetSetEffLnaRuralUrbanMv

> RAIL_Status_t RAIL_GetSetEffLnaRuralUrbanMv (RAIL_Handle_t railHandle, uint16_t *newTrip, bool changeTrip)

Copy the current Rural to Urban trip voltage into newTrip.

Parameters

[in]	railHandle	A RAIL instance handle
[inout]	newTrip	A pointer to a uint16_t that will contain the Rural to Urban trip voltage, in millivolts
[in]	changeTrip	If true, use newTrip to update current Rural to Urban trip voltage

If changeTrip is true, update current Rural to Urban trip voltage with newTrip first.

Returns

Status code indicating success of the function call.

Definition at line 6181 of file common/rail.h

RAIL_GetSetEffLnaUrbanBypassMv

> RAIL_Status_t RAIL_GetSetEffLnaUrbanBypassMv (RAIL_Handle_t railHandle, uint16_t *newTrip, bool changeTrip)

Copy the current Urban to Bypass trip voltage into newTrip.

Parameters

[in]	railHandle	A RAIL instance handle
[inout]	newTrip	A pointer to a uint16_t that will contain the Urban to Bypass trip voltage, in millivolts
[in]	changeTrip	If true, use newTrip to update current Urban to Bypass trip voltage

If changeTrip is true, update current Urban to Bypass trip voltage with newTrip first.

Returns

- Status code indicating success of the function call.

Definition at line 6194 of file common/rail.h

RAIL_GetSetEffLnaUrbanDwellTimeMs

> RAIL_Status_t RAIL_GetSetEffLnaUrbanDwellTimeMs (RAIL_Handle_t railHandle, uint32_t *newDwellTime, bool changeDwellTime)

Copy the current Urban dwell time into newDwellTime.

Parameters

[in]	railHandle	A RAIL instance handle
[inout]	newDwellTime	A pointer to a uint16_t that will contain the Urban dwell dwell time, in milliseconds
[in]	changeDwellTime	If true, use newDwellTime to update current Urban dwell time

If changeDwellTime is true, update current Urban dwell time with newDwellTime first.

Returns

- Status code indicating success of the function call.

Definition at line 6208 of file common/rail.h

RAIL_GetSetEffLnaBypassDwellTimeMs

> RAIL_Status_t RAIL_GetSetEffLnaBypassDwellTimeMs (RAIL_Handle_t railHandle, uint32_t *newDwellTime, bool changeDwellTime)

Copy the current Bypass dwell time into newDwellTime.

Parameters

[in]	railHandle	A RAIL instance handle
[inout]	newDwellTime	A pointer to a uint16_t that will contain the Bypass dwell dwell time, in milliseconds

[in]	changeDwellTime	If true, use newDwellTime to update current Bypass dwell time

If changeDwellTime is true, update current Bypass dwell time with newDwellTime first.

Returns

- Status code indicating success of the function call.

Definition at line 6222 of file common/rail.h

RAIL_GetSetEffClpcFastLoopCal

```
RAIL_Status_t RAIL_GetSetEffClpcFastLoopCal (RAIL_Handle_t railHandle, RAIL_EffModeSensor_t modeSensorIndex,
RAIL_EffCalConfig_t *calibrationEntry, bool changeValues)
```

If changeValues is true, update current CLPC Fast Loop calibration values using the new variables.

Parameters

[in]	railHandle	A RAIL instance handle
[in]	modeSensorIndex	The mode sensor to use for this operation.
[inout]	calibrationEntry	The calibration entry to retrieve or update
[in]	changeValues	If true, use new values to update the CLPC fast loop calibration

If false, copy the current CLPC Fast Loop calibration values into new variables.

Returns

- Status code indicating success of the function call.

Definition at line 6237 of file common/rail.h

RAIL_GetSetEffClpcFastLoopCalSlp

```
RAIL_Status_t RAIL_GetSetEffClpcFastLoopCalSlp (RAIL_Handle_t railHandle, RAIL_EffModeSensor_t modeSensorIndex,
int16_t *newSlope1e1MvPerDdbm, int16_t *newoffset290Ddbm, bool changeValues)
```

If changeValues is true, update current CLPC Fast Loop calibration equations using the new variables.

Parameters

[in]	railHandle	A RAIL instance handle
[in]	modeSensorIndex	The mode sensor to use for this operation.
[inout]	newSlope1e1MvPerDdbm	A pointer to a uint16_t that will contain the CLPC Cal slope, in mV/ddBm * 10
[inout]	newoffset290Ddbm	A pointer to a uint16_t that will contain the CLPC Cal offset from 29 dB
[in]	changeValues	If true, use new values to update the CLPC fast loop calibration equations

If false, copy the current CLPC Fast Loop calibration equations into new variables.

Returns

- Status code indicating success of the function call.

Definition at line 6254 of file common/rail.h

RAIL_GetSetEffClpcFastLoop

RAIL_Status_t RAIL_GetSetEffClpcFastLoop (RAIL_Handle_t railHandle, RAIL_EffModeSensor_t modeSensorIndex, uint16_t *newTargetMv, uint16_t *newSlopeMvPerPaLevel, bool changeValues)

If changeValues is true, update current CLPC Fast Loop Target and Slope.

Parameters

[in]	railHandle	A RAIL instance handle
[in]	modeSensorIndex	The mode sensor to use for this operation.
[inout]	newTargetMv	A pointer to a uint16_t that will contain the CLPC Fast Loop Target in mV
[inout]	newSlopeMvPerPaLevel	A pointer to a uint16_t that will contain the CLPC Fast Loop Slope in mV/(PA power level)
[in]	changeValues	If true, use newTargetMv and newSlopeMvPerPaLevel to update the CLPC Fast Loop values

If false, copy the current CLPC Fast Loop values into newTarget and newSlope.

Returns

- Status code indicating success of the function call.

Definition at line `6272` of file `common/rail.h`

RAIL_GetSetEffClpcEnable

RAIL_Status_t RAIL_GetSetEffClpcEnable (RAIL_Handle_t railHandle, uint8_t *newClpcEnable, bool changeClpcEnable)

Copy the current CLPC Enable in to newClpcEnable.

Parameters

[in]	railHandle	A RAIL instance handle
[inout]	newClpcEnable	A pointer to a uint8_t that will contain the CLPC Enable
[in]	changeClpcEnable	If true, use newClpcEnable to update the current CLPC Enable

If changeClpcEnable is true, update current CLPC Enable with newClpcEnable first.

Returns

- Status code indicating success of the function call.

Definition at line `6287` of file `common/rail.h`

RAIL_GetSetEffTempThreshold

RAIL_Status_t RAIL_GetSetEffTempThreshold (RAIL_Handle_t railHandle, uint16_t *newThresholdK, bool changeThreshold)

Get and set the EFF temperature threshold.

Parameters

[in]	railHandle	A RAIL instance handle.
[inout]	newThresholdK	A pointer to a uint16_t that will contain the current EFF temperature threshold, in Kelvin.
[in]	changeThreshold	If true, use newThresholdK to update current EFF temperature threshold.

The EFR32 device periodically takes temperature measurements of the attached EFF device. If the EFF temperature ever exceeds the EFF temperature threshold, any active transmit operation is aborted and future transmit operations are blocked until the EFF temperature falls below the threshold.

Returns

- Status code indicating the result of the function call.

Definition at line 6321 of file common/rail.h

Macro Definition Documentation

RAIL_CLPC_MINIMUM_POWER

```
#define RAIL_CLPC_MINIMUM_POWER
```

Value:

```
180
```

Minimum power for CLPC usage in deci-dBm.

Below this power CLPC will not activate. Recommend staying above 19 dBm for best performance. Signed unit, do not add U.

Definition at line 6084 of file common/rail.h

RAIL_EFF_TEMP_MEASURE_COUNT

```
#define RAIL_EFF_TEMP_MEASURE_COUNT
```

Value:

```
(6U)
```

Number of temperature values provided for the EFF thermal protection.

Definition at line 6099 of file common/rail.h

RAIL_EFF_TEMP_MEASURE_DEPRECATED_COUNT

```
#define RAIL_EFF_TEMP_MEASURE_DEPRECATED_COUNT
```

Value:

```
(2U)
```

Number of deprecated temperature values in EFF thermal protection.

Definition at line 6101 of file common/rail.h

RAIL_HFXO_TEMP_MEASURE_COUNT

```
#define RAIL_HFXO_TEMP_MEASURE_COUNT
```

Value:

```
(1U)
```

Number of temperature values provided for HFXO metrics.

Definition at line 6103 of file common/rail.h

RAIL_TEMP_MEASURE_COUNT

```
#define RAIL_TEMP_MEASURE_COUNT
```

Value:

```
                    (RAIL_CHIP_TEMP_MEASURE_COUNT         \
                    + RAIL_EFF_TEMP_MEASURE_COUNT         \
                    + RAIL_EFF_TEMP_MEASURE_DEPRECATED_COUNT \
                    + RAIL_HFXO_TEMP_MEASURE_COUNT)
```

Total number of temperature values provided by RAIL_GetTemperature().

Definition at line 6106 of file common/rail.h

RAIL_EFF_CONTROL_SIZE

```
#define RAIL_EFF_CONTROL_SIZE
```

Value:

```
(52U)
```

Number of bytes provided by RAIL_GetSetEffClpcControl().

Definition at line 6144 of file common/rail.h

RAIL_EFF_SUPPORTS_TRANSMIT

```
#define RAIL_EFF_SUPPORTS_TRANSMIT
```

Value:

```
                    ( ((x) == RAIL_EFF_DEVICE_EFF01A11NMFA0)   \
                    || ((x) == RAIL_EFF_DEVICE_EFF01A11IMFB0) \
                    )
```

A macro that checks for EFFxx devices that support high power transmit.

Definition at line 5449 of file common/rail_types.h

RAIL_EFF_SUPPORTS_RECEIVE

```
#define RAIL_EFF_SUPPORTS_RECEIVE
```

Value:

```
                    ( ((x) == RAIL_EFF_DEVICE_EFF01A11NMFA0)   \
                    || ((x) == RAIL_EFF_DEVICE_EFF01B11NMFA0) \
                    || ((x) == RAIL_EFF_DEVICE_EFF01A11IMFB0) \
                    || ((x) == RAIL_EFF_DEVICE_EFF01B11IMFB0) \
                    )
```

A macro that checks for EFFxx devices that support receive.

Definition at line 5456 of file common/rail_types.h

RAIL_EFF_TEMP_THRESHOLD_MAX

```
#define RAIL_EFF_TEMP_THRESHOLD_MAX
```

Value:

```
(383U)
```

Maximum EFF internal temperature in Kelvin, allowing transmissions when RAIL_SUPPORTS_EFF is enabled.

Definition at line 5465 of file common/rail_types.h

RAIL_EFF_MODE_SENSOR_ENUM_NAMES

```
#define RAIL_EFF_MODE_SENSOR_ENUM_NAMES
```

Value:

```
  { \
  "RAIL_EFF_MODE_SENSOR_FSK_ANTV",    \
  "RAIL_EFF_MODE_SENSOR_FSK_SAW2",    \
  "RAIL_EFF_MODE_SENSOR_OFDM_ANTV",   \
  "RAIL_EFF_MODE_SENSOR_OFDM_SAW2",   \
}
```

A macro that is string versions of the calibration enums.

Definition at line 5552 of file common/rail_types.h

RAIL_EFF_CLPC_ENABLE_ENUM_NAMES

```
#define RAIL_EFF_CLPC_ENABLE_ENUM_NAMES
```

Value:

```
  { \
  "RAIL_EFF_CLPC_DISABLED",        \
  "RAIL_EFF_CLPC_MODE_CHANGE",     \
  "RAIL_EFF_CLPC_POWER_SLOW",      \
  "RAIL_EFF_CLPC_POWER_FAST",      \
  "RAIL_EFF_CLPC_POWER_BOTH",      \
  "RAIL_EFF_CLPC_POWER_SLOW_STOPPED", \
  "RAIL_EFF_CLPC_POWER_FAST_STOPPED", \
```

```
    "RAIL_EFF_CLPC_POWER_BOTH_STOPPED", \
    "RAIL_EFF_CLPC_COUNT",              \
}
```

A macro that is string versions of the calibration enums.

Definition at line 5563 of file common/rail_types.h

RAIL_EffCalConfig_t

Calibration data for CLPC in a specific mode.

Public Attributes

RAIL_TxPower_t	cal1Ddbm Measured Output Power for CAL1 (nominally 270 ddBm)
uint16_t	cal1Mv Measured Output Voltage using sensor at CAL1 ddBm.
RAIL_TxPower_t	cal2Ddbm Measured Output Power for CAL2 (nominally at 290 ddBm)
uint16_t	cal2Mv Measured Output Voltage using sensor at CAL2 ddBm.

Public Attribute Documentation

cal1Ddbm

```
RAIL_TxPower_t RAIL_EffCalConfig_t::cal1Ddbm
```

Measured Output Power for CAL1 (nominally 270 ddBm)

Definition at line `5580` of file `common/rail_types.h`

cal1Mv

```
uint16_t RAIL_EffCalConfig_t::cal1Mv
```

Measured Output Voltage using sensor at CAL1 ddBm.

Definition at line `5581` of file `common/rail_types.h`

cal2Ddbm

```
RAIL_TxPower_t RAIL_EffCalConfig_t::cal2Ddbm
```

Measured Output Power for CAL2 (nominally at 290 ddBm)

Definition at line `5582` of file `common/rail_types.h`

cal2Mv

uint16_t RAIL_EffCalConfig_t::cal2Mv

Measured Output Voltage using sensor at CAL2 ddBm.

Definition at line 5583 of file common/rail_types.h

RAIL_EffClpcSensorConfig_t

Configuration data for a CLPC sensor.

A structure of type RAIL_EffClpcSensorConfig_t stores curve and calibration information for a CLPC sensor.

Public Attributes

int64_t	coefA Coefficient A for Sensor Voltage curve.
int64_t	coefB Coefficient B for Sensor Voltage curve.
int64_t	coefC Coefficient C for Sensor Voltage curve.
int64_t	coefD Coefficient D for Sensor Voltage curve.
RAIL_EffCalConfig_t	calData Calibration data for Sensor for this mode.
int16_t	slope1e1MvPerDdbm Calculated slope * 10 for Sensor calibration, measured in mV/ddBm.
int16_t	offset290Ddbm Calculated effective offset at 290 ddBm for Sensor calibration.

Public Attribute Documentation

coefA

```
int64_t RAIL_EffClpcSensorConfig_t::coefA
```

Coefficient A for Sensor Voltage curve.

Multiplied by 1e7.

Definition at line 5593 of file common/rail_types.h

coefB

```
int64_t RAIL_EffClpcSensorConfig_t::coefB
```

Coefficient B for Sensor Voltage curve.

Multiplied by 1e7.

Definition at line 5594 of file common/rail_types.h

coefC

> int64_t RAIL_EffClpcSensorConfig_t::coefC

Coefficient C for Sensor Voltage curve.

Multiplied by 1e7.

Definition at line 5595 of file common/rail_types.h

coefD

> int64_t RAIL_EffClpcSensorConfig_t::coefD

Coefficient D for Sensor Voltage curve.

Multiplied by 1e7.

Definition at line 5596 of file common/rail_types.h

calData

> RAIL_EffCalConfig_t RAIL_EffClpcSensorConfig_t::calData

Calibration data for Sensor for this mode.

Definition at line 5597 of file common/rail_types.h

slope1e1MvPerDdbm

> int16_t RAIL_EffClpcSensorConfig_t::slope1e1MvPerDdbm

Calculated slope * 10 for Sensor calibration, measured in mV/ddBm.

Definition at line 5598 of file common/rail_types.h

offset290Ddbm

> int16_t RAIL_EffClpcSensorConfig_t::offset290Ddbm

Calculated effective offset at 290 ddBm for Sensor calibration.

Definition at line 5599 of file common/rail_types.h

RAIL_EffClpcConfig_t

Configuration data for CLPC in a specific mode.

A structure of type RAIL_EffClpcConfig_t stores calibration information for each sensor in a specific mode.

Public Attributes

RAIL_EffClpcSensorConfig_t	antv ANTV sensor configuration.
RAIL_EffClpcSensorConfig_t	saw2 SAW2 sensor configuration.

Public Attribute Documentation

antv

```
RAIL_EffClpcSensorConfig_t RAIL_EffClpcConfig_t::antv
```

ANTV sensor configuration.

Definition at line 5609 of file common/rail_types.h

saw2

```
RAIL_EffClpcSensorConfig_t RAIL_EffClpcConfig_t::saw2
```

SAW2 sensor configuration.

Definition at line 5610 of file common/rail_types.h

RAIL_EffClpcResults_t

Structure for passing information from the CLPC.

A structure of type RAIL_EffClpcResults_t returns the measurements and decisions from the Closed Loop Power Control back to the application side.

Public Attributes

int8_t	rawShift CLPC shift directly from formula.
int8_t	clampedShift Shift after clamping to maximum allowed for this pass.
uint8_t	currIndex Powersetting table index before shifting.
uint8_t	newIndex Power table index after shifting.

Public Attribute Documentation

rawShift

```
int8_t RAIL_EffClpcResults_t::rawShift
```

CLPC shift directly from formula.

Definition at line 5621 of file common/rail_types.h

clampedShift

```
int8_t RAIL_EffClpcResults_t::clampedShift
```

Shift after clamping to maximum allowed for this pass.

Definition at line 5622 of file common/rail_types.h

currIndex

```
uint8_t RAIL_EffClpcResults_t::currIndex
```

Powersetting table index before shifting.

Definition at line 5623 of file common/rail_types.h

newIndex

uint8_t RAIL_EffClpcResults_t::newIndex

Power table index after shifting.

Definition at line 5624 of file common/rail_types.h

RAIL_EffConfig_t

Configuration data for the attached EFF device.

A structure of type RAIL_EffConfig_t is passed to RAIL_ConfigEff().

Public Attributes

RAIL_EffDevice_t	device EFF Device Type.
uint8_t	testPort TEST output GPIO port.
uint8_t	testPin TEST output GPIO pin.
RAIL_EffLnaMode_t	enabledLnaModes LNA modes enable bitmask.
uint16_t	ruralUrbanMv Trip point from rural to urban mode, in millivolts.
uint16_t	urbanBypassMv Trip point from urban to bypass mode, in millivolts.
uint16_t	lnaReserved Reserved for future use.
uint32_t	urbanDwellTimeMs Time to stay in urban mode before transitioning to rural mode, in milliseconds.
uint32_t	bypassDwellTimeMs Time to stay in bypass mode before transitioning to urban or rural mode, in milliseconds.
RAIL_EffClpcConfig_t	fskClpcConfig Config Structure for FSK CLPC settings.
RAIL_EffClpcConfig_t	ofdmClpcConfig Config Structure for OFDM CLPC settings.
uint8_t	clpcReserved Reserved for future use.
RAIL_ClpcEnable_t	clpcEnable Select CLPC mode.
bool	advProtectionEnable Indicates whether the advanced thermal protection is enabled.
uint8_t	reservedByte Word alignment.
uint16_t	tempThresholdK Temperature of EFF above which transmit is not allowed, in degrees Kelvin.

Public Attribute Documentation

device

```
RAIL_EffDevice_t RAIL_EffConfig_t::device
```

EFF Device Type.

Definition at line `5635` of file `common/rail_types.h`

testPort

```
uint8_t RAIL_EffConfig_t::testPort
```

TEST output GPIO port.

Definition at line `5636` of file `common/rail_types.h`

testPin

```
uint8_t RAIL_EffConfig_t::testPin
```

TEST output GPIO pin.

Definition at line `5637` of file `common/rail_types.h`

enabledLnaModes

```
RAIL_EffLnaMode_t RAIL_EffConfig_t::enabledLnaModes
```

LNA modes enable bitmask.

Definition at line `5638` of file `common/rail_types.h`

ruralUrbanMv

```
uint16_t RAIL_EffConfig_t::ruralUrbanMv
```

Trip point from rural to urban mode, in millivolts.

Definition at line `5639` of file `common/rail_types.h`

urbanBypassMv

```
uint16_t RAIL_EffConfig_t::urbanBypassMv
```

Trip point from urban to bypass mode, in millivolts.

Definition at line 5640 of file common/rail_types.h

lnaReserved

```
uint16_t RAIL_EffConfig_t::lnaReserved
```

Reserved for future use.

Definition at line 5641 of file common/rail_types.h

urbanDwellTimeMs

```
uint32_t RAIL_EffConfig_t::urbanDwellTimeMs
```

Time to stay in urban mode before transitioning to rural mode, in milliseconds.

Definition at line 5642 of file common/rail_types.h

bypassDwellTimeMs

```
uint32_t RAIL_EffConfig_t::bypassDwellTimeMs
```

Time to stay in bypass mode before transitioning to urban or rural mode, in milliseconds.

Definition at line 5643 of file common/rail_types.h

fskClpcConfig

```
RAIL_EffClpcConfig_t RAIL_EffConfig_t::fskClpcConfig
```

Config Structure for FSK CLPC settings.

Definition at line 5644 of file common/rail_types.h

ofdmClpcConfig

```
RAIL_EffClpcConfig_t RAIL_EffConfig_t::ofdmClpcConfig
```

Config Structure for OFDM CLPC settings.

Definition at line 5645 of file common/rail_types.h

clpcReserved

```
uint8_t RAIL_EffConfig_t::clpcReserved
```

Reserved for future use.

Definition at line 5646 of file common/rail_types.h

clpcEnable

```
RAIL_ClpcEnable_t RAIL_EffConfig_t::clpcEnable
```

Select CLPC mode.

Definition at line 5647 of file common/rail_types.h

advProtectionEnable

```
bool RAIL_EffConfig_t::advProtectionEnable
```

Indicates whether the advanced thermal protection is enabled.

Definition at line 5648 of file common/rail_types.h

reservedByte

```
uint8_t RAIL_EffConfig_t::reservedByte
```

Word alignment.

Definition at line 5649 of file common/rail_types.h

tempThresholdK

```
uint16_t RAIL_EffConfig_t::tempThresholdK
```

Temperature of EFF above which transmit is not allowed, in degrees Kelvin.

Definition at line 5650 of file common/rail_types.h

Events

Events

APIs related to events.

Enumerations

enum RAIL_Events_t {

RAIL_EVENT_RSSI_AVERAGE_DONE_SHIFT = 0
RAIL_EVENT_RX_ACK_TIMEOUT_SHIFT
RAIL_EVENT_RX_FIFO_ALMOST_FULL_SHIFT
RAIL_EVENT_RX_PACKET_RECEIVED_SHIFT
RAIL_EVENT_RX_PREAMBLE_LOST_SHIFT
RAIL_EVENT_RX_PREAMBLE_DETECT_SHIFT
RAIL_EVENT_RX_SYNC1_DETECT_SHIFT
RAIL_EVENT_RX_SYNC2_DETECT_SHIFT
RAIL_EVENT_RX_FRAME_ERROR_SHIFT
RAIL_EVENT_RX_FIFO_FULL_SHIFT
RAIL_EVENT_RX_FIFO_OVERFLOW_SHIFT
RAIL_EVENT_RX_ADDRESS_FILTERED_SHIFT
RAIL_EVENT_RX_TIMEOUT_SHIFT
RAIL_EVENT_SCHEDULED_RX_STARTED_SHIFT
RAIL_EVENT_RX_SCHEDULED_RX_END_SHIFT
RAIL_EVENT_RX_SCHEDULED_RX_MISSED_SHIFT
RAIL_EVENT_RX_PACKET_ABORTED_SHIFT
RAIL_EVENT_RX_FILTER_PASSED_SHIFT
RAIL_EVENT_RX_TIMING_LOST_SHIFT
RAIL_EVENT_RX_TIMING_DETECT_SHIFT
RAIL_EVENT_RX_CHANNEL_HOPPING_COMPLETE_SHIFT
RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND_SHIFT
RAIL_EVENT_ZWAVE_BEAM_SHIFT
RAIL_EVENT_TX_FIFO_ALMOST_EMPTY_SHIFT
RAIL_EVENT_TX_PACKET_SENT_SHIFT
RAIL_EVENT_TXACK_PACKET_SENT_SHIFT
RAIL_EVENT_TX_ABORTED_SHIFT
RAIL_EVENT_TXACK_ABORTED_SHIFT
RAIL_EVENT_TX_BLOCKED_SHIFT
RAIL_EVENT_TXACK_BLOCKED_SHIFT
RAIL_EVENT_TX_UNDERFLOW_SHIFT
RAIL_EVENT_TXACK_UNDERFLOW_SHIFT
RAIL_EVENT_TX_CHANNEL_CLEAR_SHIFT
RAIL_EVENT_TX_CHANNEL_BUSY_SHIFT
RAIL_EVENT_TX_CCA_RETRY_SHIFT
RAIL_EVENT_TX_START_CCA_SHIFT
RAIL_EVENT_TX_STARTED_SHIFT
RAIL_EVENT_TX_SCHEDULED_TX_MISSED_SHIFT
RAIL_EVENT_CONFIG_UNSCHEDULED_SHIFT
RAIL_EVENT_CONFIG_SCHEDULED_SHIFT
RAIL_EVENT_SCHEDULER_STATUS_SHIFT
RAIL_EVENT_CAL_NEEDED_SHIFT
RAIL_EVENT_RF_SENSED_SHIFT
RAIL_EVENT_PA_PROTECTION_SHIFT
RAIL_EVENT_SIGNAL_DETECTED_SHIFT
RAIL_EVENT_IEEE802154_MODESWITCH_START_SHIFT
RAIL_EVENT_IEEE802154_MODESWITCH_END_SHIFT
RAIL_EVENT_DETECT_RSSI_THRESHOLD_SHIFT
RAIL_EVENT_THERMISTOR_DONE_SHIFT
RAIL_EVENT_TX_BLOCKED_TOO_HOT_SHIFT
RAIL_EVENT_TEMPERATURE_TOO_HOT_SHIFT
RAIL_EVENT_TEMPERATURE_COOL_DOWN_SHIFT

}

RAIL events passed to the event callback.

Functions

RAIL_Status_t RAIL_ConfigEvents(RAIL_Handle_t railHandle, RAIL_Events_t mask, RAIL_Events_t events)

Configure radio events.

Macros

#define RAIL_EVENT_SCHEDULED_TX_STARTED_SHIFT RAIL_EVENT_SCHEDULED_RX_STARTED_SHIFT
Shift position of RAIL_EVENT_SCHEDULED_TX_STARTED bit.

#define RAIL_EVENT_RX_DUTY_CYCLE_RX_END_SHIFT RAIL_EVENT_RX_CHANNEL_HOPPING_COMPLETE_SHIFT
Shift position of RAIL_EVENT_RX_DUTY_CYCLE_RX_END bit.

#define RAIL_EVENT_ZWAVE_LR_ACK_REQUEST_COMMAND_SHIFT
RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND_SHIFT
Shift position of RAIL_EVENT_ZWAVE_LR_ACK_REQUEST_COMMAND_SHIFT bit.

#define RAIL_EVENT_MFM_TX_BUFFER_DONE_SHIFT RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND_SHIFT
Shift position of RAIL_EVENT_MFM_TX_BUFFER_DONE bit.

#define RAIL_EVENTS_NONE 0ULL
A value representing no events.

#define RAIL_EVENT_RSSI_AVERAGE_DONE (1ULL << RAIL_EVENT_RSSI_AVERAGE_DONE_SHIFT)
Occurs when the hardware-averaged RSSI is done in response to RAIL_StartAverageRssi() to indicate that the hardware has completed averaging.

#define RAIL_EVENT_RX_ACK_TIMEOUT (1ULL << RAIL_EVENT_RX_ACK_TIMEOUT_SHIFT)
Occurs when the ACK timeout expires while waiting to receive the sync word of an expected ACK.

#define RAIL_EVENT_RX_FIFO_ALMOST_FULL (1ULL << RAIL_EVENT_RX_FIFO_ALMOST_FULL_SHIFT)
Keeps occurring as long as the number of bytes in the receive FIFO exceeds the configured threshold value.

#define RAIL_EVENT_RX_PACKET_RECEIVED (1ULL << RAIL_EVENT_RX_PACKET_RECEIVED_SHIFT)
Occurs whenever a packet is received with RAIL_RX_PACKET_READY_SUCCESS or RAIL_RX_PACKET_READY_CRC_ERROR.

#define RAIL_EVENT_RX_PREAMBLE_LOST (1ULL << RAIL_EVENT_RX_PREAMBLE_LOST_SHIFT)
Occurs when the radio has lost a preamble.

#define RAIL_EVENT_RX_PREAMBLE_DETECT (1ULL << RAIL_EVENT_RX_PREAMBLE_DETECT_SHIFT)
Occurs when the radio has detected a preamble.

#define RAIL_EVENT_RX_SYNC1_DETECT (1ULL << RAIL_EVENT_RX_SYNC1_DETECT_SHIFT)
Occurs when the first sync word is detected.

#define RAIL_EVENT_RX_SYNC2_DETECT (1ULL << RAIL_EVENT_RX_SYNC2_DETECT_SHIFT)
Occurs when the second sync word is detected.

#define RAIL_EVENT_RX_FRAME_ERROR (1ULL << RAIL_EVENT_RX_FRAME_ERROR_SHIFT)
Occurs when a receive is aborted with RAIL_RX_PACKET_ABORT_CRC_ERROR which only happens after any filtering has passed.

#define RAIL_EVENT_RX_FIFO_FULL (1ULL << RAIL_EVENT_RX_FIFO_FULL_SHIFT)
When using RAIL_RxDataSource_t::RX_PACKET_DATA this event occurs coincident to a receive packet completion event in which the receive FIFO or any supplemental packet metadata FIFO (see Data Management) are full and further packet reception is jeopardized.

#define RAIL_EVENT_RX_FIFO_OVERFLOW (1ULL << RAIL_EVENT_RX_FIFO_OVERFLOW_SHIFT)
When using RAIL_RxDataSource_t::RX_PACKET_DATA this event occurs when a receive is aborted with RAIL_RX_PACKET_ABORT_OVERFLOW due to overflowing the receive FIFO or any supplemental packet metadata FIFO (see Data Management).

#define RAIL_EVENT_RX_ADDRESS_FILTERED (1ULL << RAIL_EVENT_RX_ADDRESS_FILTERED_SHIFT)
Occurs when a receive is aborted with RAIL_RX_PACKET_ABORT_FILTERED because its address does not match the filtering settings.

#define RAIL_EVENT_RX_TIMEOUT (1ULL << RAIL_EVENT_RX_TIMEOUT_SHIFT)
Occurs when an RX event times out.

#define RAIL_EVENT_SCHEDULED_RX_STARTED (1ULL << RAIL_EVENT_SCHEDULED_RX_STARTED_SHIFT)
Occurs when a scheduled RX begins turning on the receiver.

#define RAIL_EVENT_SCHEDULED_TX_STARTED (1ULL << RAIL_EVENT_SCHEDULED_TX_STARTED_SHIFT)
Occurs when a scheduled TX begins turning on the transmitter.

#define RAIL_EVENT_RX_SCHEDULED_RX_END (1ULL << RAIL_EVENT_RX_SCHEDULED_RX_END_SHIFT)
Occurs when the scheduled RX window ends.

#define RAIL_EVENT_RX_SCHEDULED_RX_MISSED (1ULL << RAIL_EVENT_RX_SCHEDULED_RX_MISSED_SHIFT)
Occurs when start of a scheduled receive is missed.

#define RAIL_EVENT_RX_PACKET_ABORTED (1ULL << RAIL_EVENT_RX_PACKET_ABORTED_SHIFT)
Occurs when a receive is aborted during filtering with RAIL_RX_PACKET_ABORT_FORMAT or after filtering with RAIL_RX_PACKET_ABORT_ABORTED for reasons other than address filtering mismatch (which triggers RAIL_EVENT_RX_ADDRESS_FILTERED instead).

#define RAIL_EVENT_RX_FILTER_PASSED (1ULL << RAIL_EVENT_RX_FILTER_PASSED_SHIFT)
Occurs when the packet has passed any configured address and frame filtering options.

#define RAIL_EVENT_RX_TIMING_LOST (1ULL << RAIL_EVENT_RX_TIMING_LOST_SHIFT)
Occurs when the modem timing is lost.

#define RAIL_EVENT_RX_TIMING_DETECT (1ULL << RAIL_EVENT_RX_TIMING_DETECT_SHIFT)
Occurs when the modem timing is detected.

#define RAIL_EVENT_RX_CHANNEL_HOPPING_COMPLETE (1ULL << RAIL_EVENT_RX_CHANNEL_HOPPING_COMPLETE_SHIFT)
Occurs when RX Channel Hopping is enabled and channel hopping finishes receiving on the last channel in its sequence.

#define RAIL_EVENT_RX_DUTY_CYCLE_RX_END (1ULL << RAIL_EVENT_RX_DUTY_CYCLE_RX_END_SHIFT)
Occurs during RX duty cycle mode when the radio finishes its time in receive mode.

#define RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND (1ULL << RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND_SHIFT)
Indicate a Data Request is received when using IEEE 802.15.4 functionality.

#define RAIL_EVENT_ZWAVE_BEAM (1ULL << RAIL_EVENT_ZWAVE_BEAM_SHIFT)
Indicate a Z-Wave Beam Request relevant to the node was received.

#define RAIL_EVENT_MFM_TX_BUFFER_DONE (1ULL << RAIL_EVENT_MFM_TX_BUFFER_DONE_SHIFT)
Indicate a MFM buffer has completely transmitted.

#define RAIL_EVENT_ZWAVE_LR_ACK_REQUEST_COMMAND (1ULL << RAIL_EVENT_ZWAVE_LR_ACK_REQUEST_COMMAND_SHIFT)
Indicate a request for populating Z-Wave LR ACK packet.

#define RAIL_EVENTS_RX_COMPLETION undefined
The mask representing all events that determine the end of a received packet.

#define RAIL_EVENT_TX_FIFO_ALMOST_EMPTY (1ULL << RAIL_EVENT_TX_FIFO_ALMOST_EMPTY_SHIFT)
Occurs when the number of bytes in the transmit FIFO falls below the configured threshold value.

#define RAIL_EVENT_TX_PACKET_SENT (1ULL << RAIL_EVENT_TX_PACKET_SENT_SHIFT)
Occurs after a packet has been transmitted.

#define RAIL_EVENT_TXACK_PACKET_SENT (1ULL << RAIL_EVENT_TXACK_PACKET_SENT_SHIFT)
Occurs after an ACK packet has been transmitted.

#define RAIL_EVENT_TX_ABORTED (1ULL << RAIL_EVENT_TX_ABORTED_SHIFT)
Occurs when a transmit is aborted by the user.

#define RAIL_EVENT_TXACK_ABORTED (1ULL << RAIL_EVENT_TXACK_ABORTED_SHIFT)
Occurs when an ACK transmit is aborted by the user.

#define RAIL_EVENT_TX_BLOCKED (1ULL << RAIL_EVENT_TX_BLOCKED_SHIFT)
Occurs when a transmit is blocked from occurring because RAIL_EnableTxHoldOff() was called.

#define RAIL_EVENT_TXACK_BLOCKED (1ULL << RAIL_EVENT_TXACK_BLOCKED_SHIFT)
Occurs when an ACK transmit is blocked from occurring because RAIL_EnableTxHoldOff() was called.

#define RAIL_EVENT_TX_UNDERFLOW (1ULL << RAIL_EVENT_TX_UNDERFLOW_SHIFT)
Occurs when the transmit buffer underflows.

#define RAIL_EVENT_TXACK_UNDERFLOW (1ULL << RAIL_EVENT_TXACK_UNDERFLOW_SHIFT)
Occurs when the ACK transmit buffer underflows.

#define RAIL_EVENT_TX_CHANNEL_CLEAR (1ULL << RAIL_EVENT_TX_CHANNEL_CLEAR_SHIFT)
Occurs when Carrier Sense Multiple Access (CSMA) or Listen Before Talk (LBT) succeeds.

#define RAIL_EVENT_TX_CHANNEL_BUSY (1ULL << RAIL_EVENT_TX_CHANNEL_BUSY_SHIFT)
Occurs when Carrier Sense Multiple Access (CSMA) or Listen Before Talk (LBT) fails.

#define RAIL_EVENT_TX_CCA_RETRY (1ULL << RAIL_EVENT_TX_CCA_RETRY_SHIFT)
Occurs during CSMA or LBT when an individual Clear Channel Assessment (CCA) check fails, but there are more tries needed before the overall operation completes.

#define RAIL_EVENT_TX_START_CCA (1ULL << RAIL_EVENT_TX_START_CCA_SHIFT)
Occurs when the receiver is activated to perform a Clear Channel Assessment (CCA) check.

#define RAIL_EVENT_TX_STARTED (1ULL << RAIL_EVENT_TX_STARTED_SHIFT)
Occurs when the radio starts transmitting a normal packet on the air.

#define RAIL_TX_STARTED_BYTES 0U
A value to pass as RAIL_GetTxTimePreambleStart() totalPacketBytes parameter to retrieve the RAIL_EVENT_TX_STARTED timestamp.

#define RAIL_EVENT_TX_SCHEDULED_TX_MISSED (1ULL << RAIL_EVENT_TX_SCHEDULED_TX_MISSED_SHIFT)
Occurs when the start of a scheduled transmit is missed.

#define RAIL_EVENTS_TX_COMPLETION undefined
A mask representing all events that determine the end of a transmitted packet.

#define RAIL_EVENTS_TXACK_COMPLETION undefined
A mask representing all events that determine the end of a transmitted ACK packet.

#define RAIL_EVENT_CONFIG_UNSCHEDULED (1ULL << RAIL_EVENT_CONFIG_UNSCHEDULED_SHIFT)
Occurs when the scheduler switches away from this configuration.

#define RAIL_EVENT_CONFIG_SCHEDULED (1ULL << RAIL_EVENT_CONFIG_SCHEDULED_SHIFT)
Occurs when the scheduler switches to this configuration.

#define RAIL_EVENT_SCHEDULER_STATUS (1ULL << RAIL_EVENT_SCHEDULER_STATUS_SHIFT)
Occurs when the scheduler has a status to report.

#define RAIL_EVENT_CAL_NEEDED (1ULL << RAIL_EVENT_CAL_NEEDED_SHIFT)
Occurs when the application needs to run a calibration, as determined by the RAIL library.

#define RAIL_EVENT_RF_SENSED (1ULL << RAIL_EVENT_RF_SENSED_SHIFT)
Occurs when RF energy is sensed from the radio.

#define RAIL_EVENT_PA_PROTECTION (1ULL << RAIL_EVENT_PA_PROTECTION_SHIFT)
Occurs when PA protection circuit kicks in.

#define RAIL_EVENT_SIGNAL_DETECTED (1ULL << RAIL_EVENT_SIGNAL_DETECTED_SHIFT)
Occurs after enabling the signal detection using RAIL_BLE_EnableSignalDetection or RAIL_IEEE802154_EnableSignalDetection when a signal is detected.

#define RAIL_EVENT_IEEE802154_MODESWITCH_START (1ULL << RAIL_EVENT_IEEE802154_MODESWITCH_START_SHIFT)
Occurs when a Wi-SUN mode switch packet has been received, after switching to the new PHY.

#define RAIL_EVENT_IEEE802154_MODESWITCH_END (1ULL << RAIL_EVENT_IEEE802154_MODESWITCH_END_SHIFT)
Occurs when switching back to the original base PHY in effect prior to the Wi-SUN mode switch reception.

#define RAIL_EVENT_DETECT_RSSI_THRESHOLD (1ULL << RAIL_EVENT_DETECT_RSSI_THRESHOLD_SHIFT)
Occurs when the sampled RSSI is above the threshold set by RAIL_SetRssiDetectThreshold().

#define RAIL_EVENT_THERMISTOR_DONE (1ULL << RAIL_EVENT_THERMISTOR_DONE_SHIFT)
Occurs when the thermistor has finished its measurement in response to RAIL_StartThermistorMeasurement().

#define RAIL_EVENT_TX_BLOCKED_TOO_HOT (1ULL << RAIL_EVENT_TX_BLOCKED_TOO_HOT_SHIFT)
Occurs when a Tx has been blocked because of temperature exceeding the safety threshold.

#define RAIL_EVENT_TEMPERATURE_TOO_HOT (1ULL << RAIL_EVENT_TEMPERATURE_TOO_HOT_SHIFT)
Occurs when die internal temperature exceeds the temperature threshold subtracted by the cool down parameter from RAIL_ChipTempConfig_t.

#define RAIL_EVENT_TEMPERATURE_COOL_DOWN (1ULL << RAIL_EVENT_TEMPERATURE_COOL_DOWN_SHIFT)
Occurs when die internal temperature falls below the temperature threshold subtracted by the cool down parameter from RAIL_ChipTempConfig_t.

#define RAIL_EVENTS_ALL 0xFFFFFFFFFFFFFFFFULL
A value representing all possible events.

Enumeration Documentation

RAIL_Events_t

> RAIL_Events_t

RAIL events passed to the event callback.

More than one event may be indicated due to interrupt latency.

Enumerator	
RAIL_EVENT_RSSI_AVERAGE_DONE_SHIFT	Shift position of RAIL_EVENT_RSSI_AVERAGE_DONE bit.
RAIL_EVENT_RX_ACK_TIMEOUT_SHIFT	Shift position of RAIL_EVENT_RX_ACK_TIMEOUT bit.
RAIL_EVENT_RX_FIFO_ALMOST_FULL_SHIFT	Shift position of RAIL_EVENT_RX_FIFO_ALMOST_FULL bit.
RAIL_EVENT_RX_PACKET_RECEIVED_SHIFT	Shift position of RAIL_EVENT_RX_PACKET_RECEIVED bit.
RAIL_EVENT_RX_PREAMBLE_LOST_SHIFT	Shift position of RAIL_EVENT_RX_PREAMBLE_LOST bit.
RAIL_EVENT_RX_PREAMBLE_DETECT_SHIFT	Shift position of RAIL_EVENT_RX_PREAMBLE_DETECT bit.
RAIL_EVENT_RX_SYNC1_DETECT_SHIFT	Shift position of RAIL_EVENT_RX_SYNC1_DETECT bit.
RAIL_EVENT_RX_SYNC2_DETECT_SHIFT	Shift position of RAIL_EVENT_RX_SYNC2_DETECT bit.
RAIL_EVENT_RX_FRAME_ERROR_SHIFT	Shift position of RAIL_EVENT_RX_FRAME_ERROR bit.
RAIL_EVENT_RX_FIFO_FULL_SHIFT	Shift position of RAIL_EVENT_RX_FIFO_FULL bit.
RAIL_EVENT_RX_FIFO_OVERFLOW_SHIFT	Shift position of RAIL_EVENT_RX_FIFO_OVERFLOW bit.
RAIL_EVENT_RX_ADDRESS_FILTERED_SHIFT	Shift position of RAIL_EVENT_RX_ADDRESS_FILTERED bit.
RAIL_EVENT_RX_TIMEOUT_SHIFT	Shift position of RAIL_EVENT_RX_TIMEOUT bit.

RAIL_EVENT_SCHEDULED_RX_STARTED_SHIFT	Shift position of RAIL_EVENT_SCHEDULED_RX_STARTED bit.
RAIL_EVENT_RX_SCHEDULED_RX_END_SHIFT	Shift position of RAIL_EVENT_RX_SCHEDULED_RX_END bit.
RAIL_EVENT_RX_SCHEDULED_RX_MISSED_SHIFT	Shift position of RAIL_EVENT_RX_SCHEDULED_RX_MISSED bit.
RAIL_EVENT_RX_PACKET_ABORTED_SHIFT	Shift position of RAIL_EVENT_RX_PACKET_ABORTED bit.
RAIL_EVENT_RX_FILTER_PASSED_SHIFT	Shift position of RAIL_EVENT_RX_FILTER_PASSED bit.
RAIL_EVENT_RX_TIMING_LOST_SHIFT	Shift position of RAIL_EVENT_RX_TIMING_LOST bit.
RAIL_EVENT_RX_TIMING_DETECT_SHIFT	Shift position of RAIL_EVENT_RX_TIMING_DETECT bit.
RAIL_EVENT_RX_CHANNEL_HOPPING_COMPLETE_SHIFT	Shift position of RAIL_EVENT_RX_CHANNEL_HOPPING_COMPLETE bit.
RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND_SHIFT	Shift position of RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND bit.
RAIL_EVENT_ZWAVE_BEAM_SHIFT	Shift position of RAIL_EVENT_ZWAVE_BEAM bit.
RAIL_EVENT_TX_FIFO_ALMOST_EMPTY_SHIFT	Shift position of RAIL_EVENT_TX_FIFO_ALMOST_EMPTY bit.
RAIL_EVENT_TX_PACKET_SENT_SHIFT	Shift position of RAIL_EVENT_TX_PACKET_SENT bit.
RAIL_EVENT_TXACK_PACKET_SENT_SHIFT	Shift position of RAIL_EVENT_TXACK_PACKET_SENT bit.
RAIL_EVENT_TX_ABORTED_SHIFT	Shift position of RAIL_EVENT_TX_ABORTED bit.
RAIL_EVENT_TXACK_ABORTED_SHIFT	Shift position of RAIL_EVENT_TXACK_ABORTED bit.
RAIL_EVENT_TX_BLOCKED_SHIFT	Shift position of RAIL_EVENT_TX_BLOCKED bit.
RAIL_EVENT_TXACK_BLOCKED_SHIFT	Shift position of RAIL_EVENT_TXACK_BLOCKED bit.
RAIL_EVENT_TX_UNDERFLOW_SHIFT	Shift position of RAIL_EVENT_TX_UNDERFLOW bit.
RAIL_EVENT_TXACK_UNDERFLOW_SHIFT	Shift position of RAIL_EVENT_TXACK_UNDERFLOW bit.
RAIL_EVENT_TX_CHANNEL_CLEAR_SHIFT	Shift position of RAIL_EVENT_TX_CHANNEL_CLEAR bit.
RAIL_EVENT_TX_CHANNEL_BUSY_SHIFT	Shift position of RAIL_EVENT_TX_CHANNEL_BUSY bit.
RAIL_EVENT_TX_CCA_RETRY_SHIFT	Shift position of RAIL_EVENT_TX_CCA_RETRY bit.
RAIL_EVENT_TX_START_CCA_SHIFT	Shift position of RAIL_EVENT_TX_START_CCA bit.
RAIL_EVENT_TX_STARTED_SHIFT	Shift position of RAIL_EVENT_TX_STARTED bit.
RAIL_EVENT_TX_SCHEDULED_TX_MISSED_SHIFT	Shift position of RAIL_EVENT_TX_SCHEDULED_TX_MISSED bit.
RAIL_EVENT_CONFIG_UNSCHEDULED_SHIFT	Shift position of RAIL_EVENT_CONFIG_UNSCHEDULED bit.
RAIL_EVENT_CONFIG_SCHEDULED_SHIFT	Shift position of RAIL_EVENT_CONFIG_SCHEDULED bit.
RAIL_EVENT_SCHEDULER_STATUS_SHIFT	Shift position of RAIL_EVENT_SCHEDULER_STATUS bit.
RAIL_EVENT_CAL_NEEDED_SHIFT	Shift position of RAIL_EVENT_CAL_NEEDED bit.
RAIL_EVENT_RF_SENSED_SHIFT	Shift position of RAIL_EVENT_RF_SENSED bit.
RAIL_EVENT_PA_PROTECTION_SHIFT	Shift position of RAIL_EVENT_PA_PROTECTION bit.
RAIL_EVENT_SIGNAL_DETECTED_SHIFT	Shift position of RAIL_EVENT_SIGNAL_DETECTED bit.
RAIL_EVENT_IEEE802154_MODESWITCH_START_SHIFT	Shift position of RAIL_EVENT_IEEE802154_MODESWITCH_START bit.
RAIL_EVENT_IEEE802154_MODESWITCH_END_SHIFT	Shift position of RAIL_EVENT_IEEE802154_MODESWITCH_END bit.
RAIL_EVENT_DETECT_RSSI_THRESHOLD_SHIFT	Shift position of RAIL_EVENT_DETECT_RSSI_THRESHOLD bit.
RAIL_EVENT_THERMISTOR_DONE_SHIFT	Shift position of RAIL_EVENT_THERMISTOR_DONE bit.
RAIL_EVENT_TX_BLOCKED_TOO_HOT_SHIFT	Shift position of RAIL_EVENT_TX_BLOCKED_TOO_HOT bit.

RAIL_EVENT_TEMPERATURE_TOO_HOT_SHIFT	Shift position of RAIL_EVENT_TEMPERATURE_TOO_HOT bit.
RAIL_EVENT_TEMPERATURE_COOL_DOWN_SHIFT	Shift position of RAIL_EVENT_TEMPERATURE_COOL_DOWN bit.

Definition at line 777 of file common/rail_types.h

Function Documentation

RAIL_ConfigEvents

```
RAIL_Status_t RAIL_ConfigEvents (RAIL_Handle_t railHandle, RAIL_Events_t mask, RAIL_Events_t events)
```

Configure radio events.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	mask	A bitmask of events to configure.
[in]	events	A bitmask of events to trigger RAIL_Config_t::eventsCallback For a full list of available callbacks, see RAIL_EVENT_* set of defines.

Returns

- Status code indicating success of the function call.

Sets up which radio interrupts generate a RAIL event. The full list of options is in RAIL_Events_t.

Definition at line 1509 of file common/rail.h

Macro Definition Documentation

RAIL_EVENT_SCHEDULED_TX_STARTED_SHIFT

```
#define RAIL_EVENT_SCHEDULED_TX_STARTED_SHIFT
```

Value:

```
RAIL_EVENT_SCHEDULED_RX_STARTED_SHIFT
```

Shift position of RAIL_EVENT_SCHEDULED_TX_STARTED bit.

Definition at line 896 of file common/rail_types.h

RAIL_EVENT_RX_DUTY_CYCLE_RX_END_SHIFT

```
#define RAIL_EVENT_RX_DUTY_CYCLE_RX_END_SHIFT
```

Value:

```
RAIL_EVENT_RX_CHANNEL_HOPPING_COMPLETE_SHIFT
```

Shift position of RAIL_EVENT_RX_DUTY_CYCLE_RX_END bit.

Definition at line 898 of file common/rail_types.h

RAIL_EVENT_ZWAVE_LR_ACK_REQUEST_COMMAND_SHIFT

```
#define RAIL_EVENT_ZWAVE_LR_ACK_REQUEST_COMMAND_SHIFT
```

Value:

```
RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND_SHIFT
```

Shift position of RAIL_EVENT_ZWAVE_LR_ACK_REQUEST_COMMAND_SHIFT bit.

Definition at line 900 of file common/rail_types.h

RAIL_EVENT_MFM_TX_BUFFER_DONE_SHIFT

```
#define RAIL_EVENT_MFM_TX_BUFFER_DONE_SHIFT
```

Value:

```
RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND_SHIFT
```

Shift position of RAIL_EVENT_MFM_TX_BUFFER_DONE bit.

Definition at line 902 of file common/rail_types.h

RAIL_EVENTS_NONE

```
#define RAIL_EVENTS_NONE
```

Value:

```
0ULL
```

A value representing no events.

Definition at line 907 of file common/rail_types.h

RAIL_EVENT_RSSI_AVERAGE_DONE

```
#define RAIL_EVENT_RSSI_AVERAGE_DONE
```

Value:

```
(1ULL << RAIL_EVENT_RSSI_AVERAGE_DONE_SHIFT)
```

Occurs when the hardware-averaged RSSI is done in response to RAIL_StartAverageRssi() to indicate that the hardware has completed averaging.

Call RAIL_GetAverageRssi() to get the result.

Definition at line 916 of file common/rail_types.h

RAIL_EVENT_RX_ACK_TIMEOUT

```
#define RAIL_EVENT_RX_ACK_TIMEOUT
```

Value:

```
(1ULL << RAIL_EVENT_RX_ACK_TIMEOUT_SHIFT)
```

Occurs when the ACK timeout expires while waiting to receive the sync word of an expected ACK.

If the timeout occurs within packet reception, this event won't be signaled until after packet completion has determined the packet wasn't the expected ACK. See RAIL_RxPacketDetails_t::isAck for the definition of an expected ACK.

This event only occurs after calling RAIL_ConfigAutoAck() and after transmitting a packet with RAIL_TX_OPTION_WAIT_FOR_ACK set.

Definition at line `929` of file `common/rail_types.h`

RAIL_EVENT_RX_FIFO_ALMOST_FULL

```
#define RAIL_EVENT_RX_FIFO_ALMOST_FULL
```

Value:

```
(1ULL << RAIL_EVENT_RX_FIFO_ALMOST_FULL_SHIFT)
```

Keeps occurring as long as the number of bytes in the receive FIFO exceeds the configured threshold value.

Call RAIL_GetRxFifoBytesAvailable() to get the number of bytes available. When using this event, the threshold should be set via RAIL_SetRxFifoThreshold().

How to avoid sticking in the event handler (even in idle state):

1. Disable the event (via the config events API or the RAIL_FIFO_THRESHOLD_DISABLED parameter)
2. Increase FIFO threshold
3. Read the FIFO (that's not an option in RAIL_DataMethod_t::PACKET_MODE) in the event handler

Definition at line `946` of file `common/rail_types.h`

RAIL_EVENT_RX_PACKET_RECEIVED

```
#define RAIL_EVENT_RX_PACKET_RECEIVED
```

Value:

```
(1ULL << RAIL_EVENT_RX_PACKET_RECEIVED_SHIFT)
```

Occurs whenever a packet is received with RAIL_RX_PACKET_READY_SUCCESS or RAIL_RX_PACKET_READY_CRC_ERROR.

Call RAIL_GetRxPacketInfo() to get basic information about the packet along with a handle to this packet for subsequent use with RAIL_PeekRxPacket(), RAIL_GetRxPacketDetails(), RAIL_HoldRxPacket(), and RAIL_ReleaseRxPacket() as needed.

Definition at line `957` of file `common/rail_types.h`

RAIL_EVENT_RX_PREAMBLE_LOST

```
#define RAIL_EVENT_RX_PREAMBLE_LOST
```

Value:

```
(1ULL << RAIL_EVENT_RX_PREAMBLE_LOST_SHIFT)
```

Occurs when the radio has lost a preamble.

This event can occur multiple times while searching for a packet and is generally used for diagnostic purposes. It can only occur after a RAIL_EVENT_RX_PREAMBLE_DETECT event has already occurred.

Note

- See warning for RAIL_EVENT_RX_PREAMBLE_DETECT.

Definition at line `969` of file `common/rail_types.h`

RAIL_EVENT_RX_PREAMBLE_DETECT

```
#define RAIL_EVENT_RX_PREAMBLE_DETECT
```

Value:

```
(1ULL << RAIL_EVENT_RX_PREAMBLE_DETECT_SHIFT)
```

Occurs when the radio has detected a preamble.

This event can occur multiple times while searching for a packet and is generally used for diagnostic purposes. It can only occur after a RAIL_EVENT_RX_TIMING_DETECT event has already occurred.

Warnings

- This event, along with RAIL_EVENT_RX_PREAMBLE_LOST, may not work on some demodulators. Some demodulators usurped the signals on which these events are based for another purpose. These demodulators in particular are available on the EFR32xG23, EFR32xG25, and the EFR32xG28 platforms. Enabling these events on these platforms may cause the events to fire infinitely and possibly freeze the application.

Definition at line `986` of file `common/rail_types.h`

RAIL_EVENT_RX_SYNC1_DETECT

```
#define RAIL_EVENT_RX_SYNC1_DETECT
```

Value:

```
(1ULL << RAIL_EVENT_RX_SYNC1_DETECT_SHIFT)
```

Occurs when the first sync word is detected.

After this event occurs, one of the events in the RAIL_EVENTS_RX_COMPLETION mask will occur.

Definition at line `994` of file `common/rail_types.h`

RAIL_EVENT_RX_SYNC2_DETECT

```
#define RAIL_EVENT_RX_SYNC2_DETECT
```

Value:

```
(1ULL << RAIL_EVENT_RX_SYNC2_DETECT_SHIFT)
```

Occurs when the second sync word is detected.

After this event occurs, one of the events in the RAIL_EVENTS_RX_COMPLETION mask will occur.

Definition at line 1002 of file common/rail_types.h

RAIL_EVENT_RX_FRAME_ERROR

```
#define RAIL_EVENT_RX_FRAME_ERROR
```

Value:

```
(1ULL << RAIL_EVENT_RX_FRAME_ERROR_SHIFT)
```

Occurs when a receive is aborted with RAIL_RX_PACKET_ABORT_CRC_ERROR which only happens after any filtering has passed.

For EFR32 parts, this event includes CRC errors, block decoding errors, and illegal frame length – when detected after filtering. (When such errors are detected during filtering, they're signaled as RAIL_EVENT_RX_PACKET_ABORTED instead.)

If RAIL_RX_OPTION_IGNORE_CRC_ERRORS is set, this event will not occur for CRC errors, but could still occur for the other errors.

Definition at line 1016 of file common/rail_types.h

RAIL_EVENT_RX_FIFO_FULL

```
#define RAIL_EVENT_RX_FIFO_FULL
```

Value:

```
(1ULL << RAIL_EVENT_RX_FIFO_FULL_SHIFT)
```

When using RAIL_RxDataSource_t::RX_PACKET_DATA this event occurs coincident to a receive packet completion event in which the receive FIFO or any supplemental packet metadata FIFO (see Data Management) are full and further packet reception is jeopardized.

It signals that an overflow is imminent (and may already have occurred) telling the application it should release the oldest packet(s) as soon as possible. This event may may be posted multiple times with subsequent receive completion events if the FIFO(s) remain full, and should also occur coincident with RAIL_EVENT_RX_FIFO_OVERFLOW.

When not using RAIL_RxDataSource_t::RX_PACKET_DATA this event is not tied to packet completion and will occur coincident with RAIL_EVENT_RX_FIFO_OVERFLOW when the receive FIFO has filled and overflowed. The application should consume receive FIFO data via RAIL_ReadRxFifo() as soon as possible to minimize lost raw data.

Definition at line 1036 of file common/rail_types.h

RAIL_EVENT_RX_FIFO_OVERFLOW

```
#define RAIL_EVENT_RX_FIFO_OVERFLOW
```

Value:

```
(1ULL << RAIL_EVENT_RX_FIFO_OVERFLOW_SHIFT)
```

When using RAIL_RxDataSource_t::RX_PACKET_DATA this event occurs when a receive is aborted with RAIL_RX_PACKET_ABORT_OVERFLOW due to overflowing the receive FIFO or any supplemental packet metadata FIFO (see Data Management).

The radio suspends receiving packets until this event is posted and the receive FIFO(s) have been fully processed (drained and released or reset). It is not guaranteed that a RAIL_EVENT_RX_FIFO_FULL will precede this event, but both events should be coincident.

When not using RAIL_RxDataSource_t::RX_PACKET_DATA this event is not tied to packet completion and will occur coincident with RAIL_EVENT_RX_FIFO_FULL when the receive FIFO has filled and overflowed. The application should consume receive FIFO data via RAIL_ReadRxFifo() as soon as possible to minimize lost raw data.

Definition at line 1055 of file common/rail_types.h

RAIL_EVENT_RX_ADDRESS_FILTERED

```
#define RAIL_EVENT_RX_ADDRESS_FILTERED
```

Value:

```
(1ULL << RAIL_EVENT_RX_ADDRESS_FILTERED_SHIFT)
```

Occurs when a receive is aborted with RAIL_RX_PACKET_ABORT_FILTERED because its address does not match the filtering settings.

This event can only occur after calling RAIL_EnableAddressFilter().

Definition at line 1063 of file common/rail_types.h

RAIL_EVENT_RX_TIMEOUT

```
#define RAIL_EVENT_RX_TIMEOUT
```

Value:

```
(1ULL << RAIL_EVENT_RX_TIMEOUT_SHIFT)
```

Occurs when an RX event times out.

This event can only occur if the RAIL_StateTiming_t::rxSearchTimeout passed to RAIL_SetStateTiming() is not zero.

Definition at line 1072 of file common/rail_types.h

RAIL_EVENT_SCHEDULED_RX_STARTED

```
#define RAIL_EVENT_SCHEDULED_RX_STARTED
```

Value:

```
(1ULL << RAIL_EVENT_SCHEDULED_RX_STARTED_SHIFT)
```

Occurs when a scheduled RX begins turning on the receiver.

This event has the same numerical value as RAIL_EVENT_SCHEDULED_TX_STARTED because one cannot schedule both RX and TX simultaneously.

Definition at line 1079 of file common/rail_types.h

RAIL_EVENT_SCHEDULED_TX_STARTED

```
#define RAIL_EVENT_SCHEDULED_TX_STARTED
```

Value:

```
(1ULL << RAIL_EVENT_SCHEDULED_TX_STARTED_SHIFT)
```

Occurs when a scheduled TX begins turning on the transmitter.

This event has the same numerical value as RAIL_EVENT_SCHEDULED_RX_STARTED because one cannot schedule both RX and TX simultaneously.

Definition at line 1086 of file common/rail_types.h

RAIL_EVENT_RX_SCHEDULED_RX_END

```
#define RAIL_EVENT_RX_SCHEDULED_RX_END
```

Value:

```
(1ULL << RAIL_EVENT_RX_SCHEDULED_RX_END_SHIFT)
```

Occurs when the scheduled RX window ends.

This event only occurs in response to a scheduled receive timeout after calling RAIL_ScheduleRx(). If RAIL_ScheduleRxConfig_t::rxTransitionEndSchedule was passed as false, this event will occur unless the receive is aborted (due to a call to RAIL_Idle() or a scheduler preemption, for instance). If RAIL_ScheduleRxConfig_t::rxTransitionEndSchedule was passed as true, any of the RAIL_EVENTS_RX_COMPLETION events occurring will also cause this event not to occur, since the scheduled receive will end with the transition at the end of the packet. However, if the application has not enabled the specific RAIL_EVENTS_RX_COMPLETION event which implicitly ended the scheduled receive, this event will be posted instead.

Definition at line 1103 of file common/rail_types.h

RAIL_EVENT_RX_SCHEDULED_RX_MISSED

```
#define RAIL_EVENT_RX_SCHEDULED_RX_MISSED
```

Value:

```
(1ULL << RAIL_EVENT_RX_SCHEDULED_RX_MISSED_SHIFT)
```

Occurs when start of a scheduled receive is missed.

This can occur if the radio is put to sleep and not woken up with enough time to configure the scheduled receive event.

Definition at line 1111 of file common/rail_types.h

RAIL_EVENT_RX_PACKET_ABORTED

```
#define RAIL_EVENT_RX_PACKET_ABORTED
```

Value:

```
(1ULL << RAIL_EVENT_RX_PACKET_ABORTED_SHIFT)
```

Occurs when a receive is aborted during filtering with RAIL_RX_PACKET_ABORT_FORMAT or after filtering with RAIL_RX_PACKET_ABORT_ABORTED for reasons other than address filtering mismatch (which triggers RAIL_EVENT_RX_ADDRESS_FILTERED instead).

For EFR32 parts, this event includes CRC errors, block decoding errors, illegal frame length, and other RAIL built-in protocol-specific packet content errors – when detected during filtering. (When such errors are detected after filtering, they're signaled as RAIL_EVENT_RX_FRAME_ERROR instead.) It also includes application or multiprotocol scheduler aborting a receive after filtering has passed.

Definition at line 1127 of file common/rail_types.h

RAIL_EVENT_RX_FILTER_PASSED

```
#define RAIL_EVENT_RX_FILTER_PASSED
```

Value:

```
(1ULL << RAIL_EVENT_RX_FILTER_PASSED_SHIFT)
```

Occurs when the packet has passed any configured address and frame filtering options.

This event will only occur between the start of the packet, indicated by RAIL_EVENT_RX_SYNC1_DETECT or RAIL_EVENT_RX_SYNC2_DETECT and one of the events in the RAIL_EVENTS_RX_COMPLETION mask. It will always occur before or concurrently with RAIL_EVENT_RX_PACKET_RECEIVED. If IEEE 802.15.4 frame and address filtering are enabled, this event will occur immediately after destination address filtering.

Definition at line 1141 of file common/rail_types.h

RAIL_EVENT_RX_TIMING_LOST

```
#define RAIL_EVENT_RX_TIMING_LOST
```

Value:

```
(1ULL << RAIL_EVENT_RX_TIMING_LOST_SHIFT)
```

Occurs when the modem timing is lost.

This event can occur multiple times while searching for a packet and is generally used for diagnostic purposes. It can only occur after a RAIL_EVENT_RX_TIMING_DETECT event has already occurred.

Note

- See warning for RAIL_EVENT_RX_TIMING_DETECT.

Definition at line 1153 of file common/rail_types.h

RAIL_EVENT_RX_TIMING_DETECT

```
#define RAIL_EVENT_RX_TIMING_DETECT
```

Value:

```
(1ULL << RAIL_EVENT_RX_TIMING_DETECT_SHIFT)
```

Occurs when the modem timing is detected.

This event can occur multiple times while searching for a packet and is generally used for diagnostic purposes.

Warnings

- This event, along with RAIL_EVENT_RX_TIMING_LOST, may not work on some demodulators. Some demodulators usurped the signals on which these events are based for another purpose. These demodulators in particular are available on the EFR32xG23, EFR32xG25, and the EFR32xG28 platforms. Enabling these events on these platforms may cause the events to fire infinitely and possibly freeze the application.

Definition at line 1168 of file common/rail_types.h

RAIL_EVENT_RX_CHANNEL_HOPPING_COMPLETE

```
#define RAIL_EVENT_RX_CHANNEL_HOPPING_COMPLETE
```

Value:

```
(1ULL << RAIL_EVENT_RX_CHANNEL_HOPPING_COMPLETE_SHIFT)
```

Occurs when RX Channel Hopping is enabled and channel hopping finishes receiving on the last channel in its sequence.

The intent behind this event is to allow the user to keep the radio on for as short a time as possible. That is, once the channel sequence is complete, the application will receive this event and can trigger a sleep/idle until it is necessary to cycle through the channels again. If this event is left on indefinitely and not handled it will likely be a fairly noisy event, as it continues to fire each time the hopping algorithm cycles through the channel sequence.

Warnings

- This event currently does not occur when using RAIL_RxChannelHoppingMode_t::RAIL_RX_CHANNEL_HOPPING_MODE_MANUAL. As a workaround, an application can monitor the current hop channel with RAIL_GetChannelAlt().

Definition at line 1187 of file common/rail_types.h

RAIL_EVENT_RX_DUTY_CYCLE_RX_END

```
#define RAIL_EVENT_RX_DUTY_CYCLE_RX_END
```

Value:

```
(1ULL << RAIL_EVENT_RX_DUTY_CYCLE_RX_END_SHIFT)
```

Occurs during RX duty cycle mode when the radio finishes its time in receive mode.

The application can then trigger a sleep/idle until it needs to listen again.

Definition at line 1196 of file common/rail_types.h

RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND

```
#define RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND
```

Value:

```
(1ULL << RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND_SHIFT)
```

Indicate a Data Request is received when using IEEE 802.15.4 functionality.

It occurs when the command byte of an incoming ACK-requesting MAC Control frame is for a data request. This callback is called before the packet is fully received to allow the node to have more time to decide whether to indicate a frame is pending in the outgoing ACK. This event only occurs if the RAIL IEEE 802.15.4 functionality is enabled, but will never occur if promiscuous mode is enabled via RAIL_IEEE802154_SetPromiscuousMode().

Call RAIL_IEEE802154_GetAddress() to get the source address of the packet.

Definition at line 1212 of file common/rail_types.h

RAIL_EVENT_ZWAVE_BEAM

```
#define RAIL_EVENT_ZWAVE_BEAM
```

Value:

```
(1ULL << RAIL_EVENT_ZWAVE_BEAM_SHIFT)
```

Indicate a Z-Wave Beam Request relevant to the node was received.

This event only occurs if the RAIL Z-Wave functionality is enabled and its RAIL_ZWAVE_OPTION_DETECT_BEAM_FRAMES is enabled. This event is used in lieu of RAIL_EVENT_RX_PACKET_RECEIVED, which is reserved for Z-Wave packets other than Beams.

Call RAIL_ZWAVE_GetBeamNodeId() to get the NodeId to which the Beam was targeted, which would be either the broadcast id 0xFF or the node's own single-cast id.

Note

- All Z-Wave Beam requests are generally discarded, triggering RAIL_EVENT_RX_PACKET_ABORTED.

Definition at line 1229 of file common/rail_types.h

RAIL_EVENT_MFM_TX_BUFFER_DONE

```
#define RAIL_EVENT_MFM_TX_BUFFER_DONE
```

Value:

```
(1ULL << RAIL_EVENT_MFM_TX_BUFFER_DONE_SHIFT)
```

Indicate a MFM buffer has completely transmitted.

This event only occurs if the RAIL MFM functionality is enabled and a MFM buffer has completely transmitted.

Following this event, the application can update the MFM buffer that has transmitted to be used for the next transmission.

Definition at line 1240 of file common/rail_types.h

RAIL_EVENT_ZWAVE_LR_ACK_REQUEST_COMMAND

```
#define RAIL_EVENT_ZWAVE_LR_ACK_REQUEST_COMMAND
```

Value:

```
(1ULL << RAIL_EVENT_ZWAVE_LR_ACK_REQUEST_COMMAND_SHIFT)
```

Indicate a request for populating Z-Wave LR ACK packet.

This event only occurs if the RAIL Z-Wave functionality is enabled.

Following this event, the application must call RAIL_ZWAVE_SetLrAckData() to populate noise floor, TX power and receive RSSI fields of the Z-Wave Long Range ACK packet.

Definition at line 1250 of file common/rail_types.h

RAIL_EVENTS_RX_COMPLETION

```
#define RAIL_EVENTS_RX_COMPLETION
```

Value:

```
0                   (RAIL_EVENT_RX_PACKET_RECEIVED   \
0                   | RAIL_EVENT_RX_PACKET_ABORTED   \
0                   | RAIL_EVENT_RX_FRAME_ERROR     \
0                   | RAIL_EVENT_RX_FIFO_OVERFLOW    \
0                   | RAIL_EVENT_RX_ADDRESS_FILTERED \
0                   | RAIL_EVENT_RX_SCHEDULED_RX_MISSED)
```

The mask representing all events that determine the end of a received packet.

After a RAIL_EVENT_RX_SYNC1_DETECT or a RAIL_EVENT_RX_SYNC2_DETECT, exactly one of the following events will occur. When one of these events occurs, a state transition will take place based on the parameter passed to RAIL_SetRxTransitions(). The RAIL_StateTransitions_t::success transition will be followed only if the RAIL_EVENT_RX_PACKET_RECEIVED event occurs. Any of the other events will trigger the RAIL_StateTransitions_t::error transition.

Definition at line 1265 of file common/rail_types.h

RAIL_EVENT_TX_FIFO_ALMOST_EMPTY

```
#define RAIL_EVENT_TX_FIFO_ALMOST_EMPTY
```

Value:

```
(1ULL << RAIL_EVENT_TX_FIFO_ALMOST_EMPTY_SHIFT)
```

Occurs when the number of bytes in the transmit FIFO falls below the configured threshold value.

This event does not occur on initialization or after resetting the transmit FIFO with RAIL_ResetFifo().

Call RAIL_GetTxFifoSpaceAvailable() to get the number of bytes available in the transmit FIFO at the time of the callback dispatch. When using this event, the threshold should be set via RAIL_SetTxFifoThreshold().

Definition at line 1286 of file common/rail_types.h

RAIL_EVENT_TX_PACKET_SENT

```
#define RAIL_EVENT_TX_PACKET_SENT
```

Value:

```
(1ULL << RAIL_EVENT_TX_PACKET_SENT_SHIFT)
```

Occurs after a packet has been transmitted.

Call RAIL_GetTxPacketDetails() to get information about the packet that was transmitted. **Note**

- RAIL_GetTxPacketDetails() is only valid to call during the time frame of the RAIL_Config_t::eventsCallback.

Definition at line 1296 of file common/rail_types.h

RAIL_EVENT_TXACK_PACKET_SENT

```
#define RAIL_EVENT_TXACK_PACKET_SENT
```

Value:

```
(1ULL << RAIL_EVENT_TXACK_PACKET_SENT_SHIFT)
```

Occurs after an ACK packet has been transmitted.

Call RAIL_GetTxPacketDetails() to get information about the packet that was transmitted. This event can only occur after calling RAIL_ConfigAutoAck(). **Note**

- RAIL_GetTxPacketDetails() is only valid to call during the time frame of the RAIL_Config_t::eventsCallback.

Definition at line 1307 of file common/rail_types.h

RAIL_EVENT_TX_ABORTED

```
#define RAIL_EVENT_TX_ABORTED
```

Value:

```
(1ULL << RAIL_EVENT_TX_ABORTED_SHIFT)
```

Occurs when a transmit is aborted by the user.

This can happen due to calling RAIL_Idle() or due to a scheduler preemption.

Note

- The Transmit FIFO is left in an indeterminate state and should be reset prior to reuse for sending a new packet. Contrast this with RAIL_EVENT_TX_BLOCKED.

Definition at line 1319 of file common/rail_types.h

RAIL_EVENT_TXACK_ABORTED

```
#define RAIL_EVENT_TXACK_ABORTED
```

Value:

```
(1ULL << RAIL_EVENT_TXACK_ABORTED_SHIFT)
```

Occurs when an ACK transmit is aborted by the user.

This event can only occur after calling RAIL_ConfigAutoAck(), which can happen due to calling RAIL_Idle() or due to a scheduler preemption.

Definition at line 1328 of file common/rail_types.h

RAIL_EVENT_TX_BLOCKED

```
#define RAIL_EVENT_TX_BLOCKED
```

Value:

```
(1ULL << RAIL_EVENT_TX_BLOCKED_SHIFT)
```

Occurs when a transmit is blocked from occurring because RAIL_EnableTxHoldOff() was called.

Note

- Since the transmit never started, the Transmit FIFO remains intact after this event – no packet data was consumed from it. Contrast this with RAIL_EVENT_TX_ABORTED.

Definition at line 1338 of file common/rail_types.h

RAIL_EVENT_TXACK_BLOCKED

```
#define RAIL_EVENT_TXACK_BLOCKED
```

Value:

```
(1ULL << RAIL_EVENT_TXACK_BLOCKED_SHIFT)
```

Occurs when an ACK transmit is blocked from occurring because RAIL_EnableTxHoldOff() was called.

This event can only occur after calling RAIL_ConfigAutoAck().

Definition at line 1346 of file common/rail_types.h

RAIL_EVENT_TX_UNDERFLOW

```
#define RAIL_EVENT_TX_UNDERFLOW
```

Value:

```
(1ULL << RAIL_EVENT_TX_UNDERFLOW_SHIFT)
```

Occurs when the transmit buffer underflows.

This can happen due to the transmitted packet specifying an unintended length based on the current radio configuration or due to RAIL_WriteTxFifo() calls not keeping up with the transmit rate if the entire packet isn't loaded at once.

Note

- The Transmit FIFO is left in an indeterminate state and should be reset prior to reuse for sending a new packet. Contrast this with RAIL_EVENT_TX_BLOCKED.

Definition at line 1360 of file common/rail_types.h

RAIL_EVENT_TXACK_UNDERFLOW

```
#define RAIL_EVENT_TXACK_UNDERFLOW
```

Value:

```
(1ULL << RAIL_EVENT_TXACK_UNDERFLOW_SHIFT)
```

Occurs when the ACK transmit buffer underflows.

This can happen due to the transmitted packet specifying an unintended length based on the current radio configuration or due to RAIL_WriteAutoAckFifo() not being called at all before an ACK transmit.

This event can only occur after calling RAIL_ConfigAutoAck().

Definition at line 1372 of file common/rail_types.h

RAIL_EVENT_TX_CHANNEL_CLEAR

```
#define RAIL_EVENT_TX_CHANNEL_CLEAR
```

Value:

```
(1ULL << RAIL_EVENT_TX_CHANNEL_CLEAR_SHIFT)
```

Occurs when Carrier Sense Multiple Access (CSMA) or Listen Before Talk (LBT) succeeds.

This event can only happen after calling RAIL_StartCcaCsmaTx() or RAIL_StartCcaLbtTx().

Definition at line 1381 of file common/rail_types.h

RAIL_EVENT_TX_CHANNEL_BUSY

```
#define RAIL_EVENT_TX_CHANNEL_BUSY
```

Value:

```
(1ULL << RAIL_EVENT_TX_CHANNEL_BUSY_SHIFT)
```

Occurs when Carrier Sense Multiple Access (CSMA) or Listen Before Talk (LBT) fails.

This event can only happen after calling RAIL_StartCcaCsmaTx() or RAIL_StartCcaLbtTx().

Note

- Since the transmit never started, the Transmit FIFO remains intact after this event – no packet data was consumed from it.

Definition at line 1393 of file common/rail_types.h

RAIL_EVENT_TX_CCA_RETRY

```
#define RAIL_EVENT_TX_CCA_RETRY
```

Value:

```
(1ULL << RAIL_EVENT_TX_CCA_RETRY_SHIFT)
```

Occurs during CSMA or LBT when an individual Clear Channel Assessment (CCA) check fails, but there are more tries needed before the overall operation completes.

This event can occur multiple times based on the configuration of the ongoing CSMA or LBT transmission. It can only happen after calling RAIL_StartCcaCsmaTx() or RAIL_StartCcaLbtTx().

Definition at line 1404 of file common/rail_types.h

RAIL_EVENT_TX_START_CCA

```
#define RAIL_EVENT_TX_START_CCA
```

Value:

```
(1ULL << RAIL_EVENT_TX_START_CCA_SHIFT)
```

Occurs when the receiver is activated to perform a Clear Channel Assessment (CCA) check.

This event generally precedes the actual start of a CCA check by roughly the RAIL_StateTiming_t::idleToRx time (subject to RAIL_MINIMUM_TRANSITION_US). It can occur multiple times based on the configuration of the ongoing CSMA or LBT transmission. It can only happen after calling RAIL_StartCcaCsmaTx() or RAIL_StartCcaLbtTx().

Definition at line 1417 of file common/rail_types.h

RAIL_EVENT_TX_STARTED

```
#define RAIL_EVENT_TX_STARTED
```

Value:

```
(1ULL << RAIL_EVENT_TX_STARTED_SHIFT)
```

Occurs when the radio starts transmitting a normal packet on the air.

A start-of-transmit timestamp is captured for this event. It can be retrieved by calling RAIL_GetTxTimePreambleStart() passing RAIL_TX_STARTED_BYTES for its totalPacketBytes parameter.

Note

- This event does not apply to ACK transmits. Currently there is no equivalent event or timestamp captured for the start of an ACK transmit.

Definition at line 1430 of file common/rail_types.h

RAIL_TX_STARTED_BYTES

```
#define RAIL_TX_STARTED_BYTES
```

Value:

```
0U
```

A value to pass as RAIL_GetTxTimePreambleStart() totalPacketBytes parameter to retrieve the RAIL_EVENT_TX_STARTED timestamp.

Definition at line 1436 of file common/rail_types.h

RAIL_EVENT_TX_SCHEDULED_TX_MISSED

```
#define RAIL_EVENT_TX_SCHEDULED_TX_MISSED
```

Value:

```
(1ULL << RAIL_EVENT_TX_SCHEDULED_TX_MISSED_SHIFT)
```

Occurs when the start of a scheduled transmit is missed.

This can occur if the radio is put to sleep and not woken up with enough time to configure the scheduled transmit event.

Note

- Since the transmit never started, the Transmit FIFO remains intact after this event – no packet data was consumed from it.

Definition at line 1447 of file common/rail_types.h

RAIL_EVENTS_TX_COMPLETION

```
#define RAIL_EVENTS_TX_COMPLETION
```

Value:

```
0                   (RAIL_EVENT_TX_PACKET_SENT   \
0                   | RAIL_EVENT_TX_ABORTED     \
0                   | RAIL_EVENT_TX_BLOCKED     \
0                   | RAIL_EVENT_TX_UNDERFLOW   \
0                   | RAIL_EVENT_TX_CHANNEL_BUSY \
0                   | RAIL_EVENT_TX_SCHEDULED_TX_MISSED)
```

A mask representing all events that determine the end of a transmitted packet.

After a RAIL_STATUS_NO_ERROR return value from one of the transmit functions, exactly one of the following events will occur. When one of these events occurs, a state transition takes place based on the parameter passed to RAIL_SetTxTransitions(). The RAIL_StateTransitions_t::success transition will be followed only if the RAIL_EVENT_TX_PACKET_SENT event occurs. Any of the other events will trigger the RAIL_StateTransitions_t::error transition.

Definition at line 1459 of file common/rail_types.h

RAIL_EVENTS_TXACK_COMPLETION

```
#define RAIL_EVENTS_TXACK_COMPLETION
```

Value:

```
                    (RAIL_EVENT_TXACK_PACKET_SENT \
                    | RAIL_EVENT_TXACK_ABORTED  \
                    | RAIL_EVENT_TXACK_BLOCKED  \
                    | RAIL_EVENT_TXACK_UNDERFLOW)
```

A mask representing all events that determine the end of a transmitted ACK packet.

After an ACK-requesting receive, exactly one of the following events will occur. When one of these events occurs, a state transition takes place based on the RAIL_AutoAckConfig_t::rxTransitions passed to RAIL_ConfigAutoAck(). The receive transitions are used because the transmitted ACK packet is considered a part of the ACK-requesting received packet. The RAIL_StateTransitions_t::success transition will be followed only if the RAIL_EVENT_TXACK_PACKET_SENT event occurs. Any of the other events will trigger the RAIL_StateTransitions_t::error transition.

Definition at line `1477` of file `common/rail_types.h`

RAIL_EVENT_CONFIG_UNSCHEDULED

```
#define RAIL_EVENT_CONFIG_UNSCHEDULED
```

Value:

```
(1ULL << RAIL_EVENT_CONFIG_UNSCHEDULED_SHIFT)
```

Occurs when the scheduler switches away from this configuration.

This event will occur in dynamic multiprotocol scenarios each time a protocol is shutting down. When it does occur, it will be the only event passed to RAIL_Config_t::eventsCallback. Therefore, to optimize protocol switch time, this event should be handled among the first in that callback, and then the application can return immediately.

Note

- : To minimize protocol switch time, Silicon Labs recommends this event event being turned off unless it is used.

Definition at line `1497` of file `common/rail_types.h`

RAIL_EVENT_CONFIG_SCHEDULED

```
#define RAIL_EVENT_CONFIG_SCHEDULED
```

Value:

```
(1ULL << RAIL_EVENT_CONFIG_SCHEDULED_SHIFT)
```

Occurs when the scheduler switches to this configuration.

This event will occur in dynamic multiprotocol scenarios each time a protocol is starting up. When it does occur, it will be the only event passed to RAIL_Config_t::eventsCallback. Therefore, in order to optimize protocol switch time, this event should be handled among the first in that callback, and then the application can return immediately.

Note

- : To minimize protocol switch time, Silicon Labs recommends this event event being turned off unless it is used.

Definition at line 1511 of file common/rail_types.h

RAIL_EVENT_SCHEDULER_STATUS

```
#define RAIL_EVENT_SCHEDULER_STATUS
```

Value:

```
(1ULL << RAIL_EVENT_SCHEDULER_STATUS_SHIFT)
```

Occurs when the scheduler has a status to report.

The exact status can be found with RAIL_GetSchedulerStatus(). See RAIL_SchedulerStatus_t for more details. When this event does occur, it will be the only event passed to RAIL_Config_t::eventsCallback. Therefore, to optimize protocol switch time, this event should be handled among the first in that callback, and then the application can return immediately.

Note

- RAIL_GetSchedulerStatus() is only valid to call during the time frame of the RAIL_Config_t::eventsCallback.
- : To minimize protocol switch time, Silicon Labs recommends this event event being turned off unless it is used.

Definition at line 1529 of file common/rail_types.h

RAIL_EVENT_CAL_NEEDED

```
#define RAIL_EVENT_CAL_NEEDED
```

Value:

```
(1ULL << RAIL_EVENT_CAL_NEEDED_SHIFT)
```

Occurs when the application needs to run a calibration, as determined by the RAIL library.

The application determines the opportune time to call RAIL_Calibrate().

Definition at line 1539 of file common/rail_types.h

RAIL_EVENT_RF_SENSED

```
#define RAIL_EVENT_RF_SENSED
```

Value:

```
(1ULL << RAIL_EVENT_RF_SENSED_SHIFT)
```

Occurs when RF energy is sensed from the radio.

This event can be used as an alternative to the callback passed as RAIL_RfSense_CallbackPtr_t.

Alternatively, the application can poll using RAIL_IsRfSensed().

Note

- This event will not occur when waking up from EM4. Prefer RAIL_IsRfSensed() when waking from EM4.

Definition at line 1550 of file common/rail_types.h

RAIL_EVENT_PA_PROTECTION

```
#define RAIL_EVENT_PA_PROTECTION
```

Value:

```
(1ULL << RAIL_EVENT_PA_PROTECTION_SHIFT)
```

Occurs when PA protection circuit kicks in.

Definition at line 1555 of file common/rail_types.h

RAIL_EVENT_SIGNAL_DETECTED

```
#define RAIL_EVENT_SIGNAL_DETECTED
```

Value:

```
(1ULL << RAIL_EVENT_SIGNAL_DETECTED_SHIFT)
```

Occurs after enabling the signal detection using RAIL_BLE_EnableSignalDetection or RAIL_IEEE802154_EnableSignalDetection when a signal is detected.

This is only used on platforms that support signal identifier, where RAIL_BLE_SUPPORTS_SIGNAL_IDENTIFIER or RAIL_IEEE802154_SUPPORTS_SIGNAL_IDENTIFIER is true.

Definition at line 1564 of file common/rail_types.h

RAIL_EVENT_IEEE802154_MODESWITCH_START

```
#define RAIL_EVENT_IEEE802154_MODESWITCH_START
```

Value:

```
(1ULL << RAIL_EVENT_IEEE802154_MODESWITCH_START_SHIFT)
```

Occurs when a Wi-SUN mode switch packet has been received, after switching to the new PHY.

It doesn't occur when a mode switch packet is transmitted.

IEEE 802.15.4 option RAIL_IEEE802154_G_OPTION_WISUN_MODESWITCH must be enabled for this event to occur.

Only available on platforms where RAIL_IEEE802154_SUPPORTS_G_MODESWITCH is true.

Definition at line 1575 of file common/rail_types.h

RAIL_EVENT_IEEE802154_MODESWITCH_END

```
#define RAIL_EVENT_IEEE802154_MODESWITCH_END
```

Value:

```
(1ULL << RAIL_EVENT_IEEE802154_MODESWITCH_END_SHIFT)
```

Occurs when switching back to the original base PHY in effect prior to the Wi-SUN mode switch reception.

This typically occurs if no packet is seen within some timeframe after the mode switch packet was received or if the first packet received in the new PHY is aborted, filtered, or fails CRC.

IEEE 802.15.4 option RAIL_IEEE802154_G_OPTION_WISUN_MODESWITCH must be enabled for this event to occur.

Only available on platforms where RAIL_IEEE802154_SUPPORTS_G_MODESWITCH is true.

Definition at line 1587 of file common/rail_types.h

RAIL_EVENT_DETECT_RSSI_THRESHOLD

```
#define RAIL_EVENT_DETECT_RSSI_THRESHOLD
```

Value:

```
(1ULL << RAIL_EVENT_DETECT_RSSI_THRESHOLD_SHIFT)
```

Occurs when the sampled RSSI is above the threshold set by RAIL_SetRssiDetectThreshold().

Definition at line 1593 of file common/rail_types.h

RAIL_EVENT_THERMISTOR_DONE

```
#define RAIL_EVENT_THERMISTOR_DONE
```

Value:

```
(1ULL << RAIL_EVENT_THERMISTOR_DONE_SHIFT)
```

Occurs when the thermistor has finished its measurement in response to RAIL_StartThermistorMeasurement().

Definition at line 1599 of file common/rail_types.h

RAIL_EVENT_TX_BLOCKED_TOO_HOT

```
#define RAIL_EVENT_TX_BLOCKED_TOO_HOT
```

Value:

```
(1ULL << RAIL_EVENT_TX_BLOCKED_TOO_HOT_SHIFT)
```

Occurs when a Tx has been blocked because of temperature exceeding the safety threshold.

Only occurs on platforms where RAIL_SUPPORTS_EFF is true, and only when also reporting RAIL_EVENT_TX_BLOCKED.

Definition at line 1608 of file common/rail_types.h

RAIL_EVENT_TEMPERATURE_TOO_HOT

```
#define RAIL_EVENT_TEMPERATURE_TOO_HOT
```

Value:

```
(1ULL << RAIL_EVENT_TEMPERATURE_TOO_HOT_SHIFT)
```

Occurs when die internal temperature exceeds the temperature threshold subtracted by the cool down parameter from RAIL_ChipTempConfig_t.

Transmits are blocked until temperature has cooled enough, indicated by RAIL_EVENT_TEMPERATURE_COOL_DOWN.

Only occurs on platforms where RAIL_SUPPORTS_THERMAL_PROTECTION is true.

Definition at line 1618 of file common/rail_types.h

RAIL_EVENT_TEMPERATURE_COOL_DOWN

```
#define RAIL_EVENT_TEMPERATURE_COOL_DOWN
```

Value:

```
(1ULL << RAIL_EVENT_TEMPERATURE_COOL_DOWN_SHIFT)
```

Occurs when die internal temperature falls below the temperature threshold subtracted by the cool down parameter from RAIL_ChipTempConfig_t.

Transmits are no longer blocked by temperature limitation, indicated by RAIL_EVENT_TEMPERATURE_TOO_HOT.

Only occurs on platforms where RAIL_SUPPORTS_THERMAL_PROTECTION is true.

Definition at line 1628 of file common/rail_types.h

RAIL_EVENTS_ALL

```
#define RAIL_EVENTS_ALL
```

Value:

```
0xFFFFFFFFFFFFFFFFULL
```

A value representing all possible events.

Definition at line 1631 of file common/rail_types.h

External Thermistor

External Thermistor

APIs to measure temperature using an external thermistor.

This feature allows reading the temperature via an external thermistor on chips that support it. This will require connecting the necessary components and configuring the pins as required.

Modules

RAIL_HFXOThermistorConfig_t

RAIL_HFXOCompensationConfig_t

Functions

RAIL_Status_t RAIL_StartThermistorMeasurement(RAIL_Handle_t railHandle)
Start a thermistor measurement.

RAIL_Status_t RAIL_GetThermistorImpedance(RAIL_Handle_t railHandle, uint32_t *thermistorImpedance)
Get the thermistor impedance measurement and return RAIL_INVALID_THERMISTOR_VALUE if the thermistor is not properly configured or the thermistor measurement is not ready.

RAIL_Status_t RAIL_ConvertThermistorImpedance(RAIL_Handle_t railHandle, uint32_t thermistorImpedance, int16_t *thermistorTemperatureC)
Convert the thermistor impedance into temperature, in Celsius.

RAIL_Status_t RAIL_ComputeHFXOPPMError(RAIL_Handle_t railHandle, int16_t crystalTemperatureC, int8_t *crystalPPMError)
Compute the crystal PPM deviation from the thermistor temperature.

RAIL_Status_t RAIL_ConfigHFXOThermistor(RAIL_Handle_t railHandle, const RAIL_HFXOThermistorConfig_t *pHfxoThermistorConfig)
Configure the GPIO for thermistor usage.

RAIL_Status_t RAIL_ConfigHFXOCompensation(RAIL_Handle_t railHandle, const RAIL_HFXOCompensationConfig_t *pHfxoCompensationConfig)
Configure the temperature parameters for HFXO compensation.

RAIL_Status_t RAIL_GetHFXOCompensationConfig(RAIL_Handle_t railHandle, RAIL_HFXOCompensationConfig_t *pHfxoCompensationConfig)
Get the temperature parameters for HFXO compensation.

RAIL_Status_t RAIL_CompensateHFXO(RAIL_Handle_t railHandle, int8_t crystalPPMError)
Compute a frequency offset and compensate HFXO accordingly.

Macros

#define RAIL_INVALID_THERMISTOR_VALUE (0xFFFFFFFFU)
A sentinel value to indicate an invalid thermistor measurement value.

#define RAIL_INVALID_PPM_VALUE (-128)
A sentinel value to indicate an invalid PPM calculation value.

Function Documentation

RAIL_StartThermistorMeasurement

```
RAIL_Status_t RAIL_StartThermistorMeasurement (RAIL_Handle_t railHandle)
```

Start a thermistor measurement.

Parameters

[in]	railHandle	A RAIL instance handle.

To get the thermistor impedance, call the function RAIL_GetThermistorImpedance. On platforms having RAIL_SUPPORTS_EXTERNAL_THERMISTOR, this function reconfigures GPIO_THMSW_EN_PIN located in GPIO_THMSW_EN_PORT. To locate this pin, refer to the data sheet or appropriate header files of the device. For proper operation, RAIL_Init must be called before using this function.

Note

- This function is not designed for safe usage in multiprotocol applications.
- When an EFF is attached, this function must not be called during transmit.

Returns

- Status code indicating success of the function call. Returns RAIL_STATUS_INVALID_STATE if the thermistor is started while the radio is transmitting.

Definition at line `6497` of file `common/rail.h`

RAIL_GetThermistorImpedance

```
RAIL_Status_t RAIL_GetThermistorImpedance (RAIL_Handle_t railHandle, uint32_t *thermistorImpedance)
```

Get the thermistor impedance measurement and return RAIL_INVALID_THERMISTOR_VALUE if the thermistor is not properly configured or the thermistor measurement is not ready.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	thermistorImpedance	Current thermistor impedance measurement in Ohms.

Returns

- Status code indicating success of the function call.

Note

- This function is already called in RAIL_CalibrateHFXO(). It does not need to be manually called during the compensation sequence.

Definition at line `6512` of file `common/rail.h`

RAIL_ConvertThermistorImpedance

```
RAIL_Status_t RAIL_ConvertThermistorImpedance (RAIL_Handle_t railHandle, uint32_t thermistorImpedance, int16_t *thermistorTemperatureC)
```

Convert the thermistor impedance into temperature, in Celsius.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	thermistorImpedance	Current thermistor impedance measurement in Ohms.
[out]	thermistorTemperatureC	Pointer to current thermistor temperature in eighth of Celsius degree

Returns

- Status code indicating success of the function call.

A version of this function is provided in the Thermistor Utility plugin for Silicon Labs radio boards. For custom boards this function can be modified and re-implemented as needed.

The variable railHandle is only used to indicate to the user from where the function was called, so it is okay to use either a real protocol handle, or one of the chip-specific ones, such as RAIL_EFR32_HANDLE.

Note

- This plugin is mandatory on EFR32xG25 platform.

Definition at line 6535 of file common/rail.h

RAIL_ComputeHFXOPPMError

```
RAIL_Status_t RAIL_ComputeHFXOPPMError (RAIL_Handle_t railHandle, int16_t crystalTemperatureC, int8_t
*crystalPPMError)
```

Compute the crystal PPM deviation from the thermistor temperature.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	crystalTemperatureC	Current crystal temperature.
[out]	crystalPPMError	Pointer to current ppm error.

Returns

- Status code indicating success of the function call.

This function is provided in the rail_util_thermistor plugin to get accurate values from our boards thermistor. For a custom board, this function could be modified and re-implemented for other needs. The variable railHandle is only used to indicate to the user from where the function was called, so it is okay to use either a real protocol handle, or one of the chip-specific ones, such as RAIL_EFR32_HANDLE.

Note

- This plugin is mandatory on EFR32xG25 platform.

Definition at line 6556 of file common/rail.h

RAIL_ConfigHFXOThermistor

```
RAIL_Status_t RAIL_ConfigHFXOThermistor (RAIL_Handle_t railHandle, const RAIL_HFXOThermistorConfig_t
*pHfxoThermistorConfig)
```

Configure the GPIO for thermistor usage.

Parameters

N/A	railHandle	A RAIL instance handle.

[in]	pHfxoThermistorConfig	The thermistor configuration pointing to the GPIO port and pin to access.

Returns

- Status code indicating the result of the function call.

Note

- The port and pin that must be passed in RAIL_HFXOThermistorConfig_t are GPIO_THMSW_EN_PORT and GPIO_THMSW_EN_PIN respectively.

Definition at line 6571 of file common/rail.h

RAIL_ConfigHFXOCompensation

> RAIL_Status_t RAIL_ConfigHFXOCompensation (RAIL_Handle_t railHandle, const RAIL_HFXOCompensationConfig_t *pHfxoCompensationConfig)

Configure the temperature parameters for HFXO compensation.

Parameters

N/A	railHandle	A RAIL instance handle.
[in]	pHfxoCompensationConfig	HFXO compensation parameters pointing to the temperature variations used to trigger a compensation.

Returns

- Status code indicating the result of the function call.

Note

- This function must be called after RAIL_ConfigHFXOThermistor to succeed.

In RAIL_HFXOCompensationConfig_t, deltaNominal and deltaCritical define the temperature variation triggering a new compensation. The field zoneTemperatureC defines the temperature separating the nominal case (below) from the critical one (above).

When enabled and either deltaNominal or deltaCritical are exceeded, RAIL raises event RAIL_EVENT_CAL_NEEDED with RAIL_CAL_TEMP_HFXO bit set. The API RAIL_StartThermistorMeasurement() must be called afterwards. The latter will raise RAIL_EVENT_THERMISTOR_DONE with calibration bit RAIL_CAL_COMPENSATE_HFXO set and RAIL_CalibrateHFXO() must follow.

Note

- Set deltaNominal and deltaCritical to 0 to perform compensation after each transmit.

Definition at line 6598 of file common/rail.h

RAIL_GetHFXOCompensationConfig

> RAIL_Status_t RAIL_GetHFXOCompensationConfig (RAIL_Handle_t railHandle, RAIL_HFXOCompensationConfig_t *pHfxoCompensationConfig)

Get the temperature parameters for HFXO compensation.

Parameters

N/A	railHandle	A RAIL instance handle.

N/A	pHfxoCompensationConfig	HFXO compensation parameters pointing to the temperature variations used to trigger a compensation.

Returns

- Status code indicating the result of the function call.

Definition at line 6609 of file common/rail.h

RAIL_CompensateHFXO

```
RAIL_Status_t RAIL_CompensateHFXO (RAIL_Handle_t railHandle, int8_t crystalPPMError)
```

Compute a frequency offset and compensate HFXO accordingly.

Parameters

N/A	railHandle	A RAIL instance handle.
N/A	crystalPPMError	The current ppm error. Positive values indicate the HFXO frequency is too high; negative values indicate it's too low.

Returns

- Status code indicating success of the function call.

Note

- This function only works for platforms having RAIL_SUPPORTS_EXTERNAL_THERMISTOR alongside RAIL_SUPPORTS_HFXO_COMPENSATION.

Definition at line 6623 of file common/rail.h

Macro Definition Documentation

RAIL_INVALID_THERMISTOR_VALUE

```
#define RAIL_INVALID_THERMISTOR_VALUE
```

Value:

```
(0xFFFFFFFFU)
```

A sentinel value to indicate an invalid thermistor measurement value.

Definition at line 4424 of file common/rail_types.h

RAIL_INVALID_PPM_VALUE

```
#define RAIL_INVALID_PPM_VALUE
```

Value:

```
(-128)
```

A sentinel value to indicate an invalid PPM calculation value.

Definition at line 4426 of file common/rail_types.h

RAIL_HFXOThermistorConfig_t

Configure the port and pin of the thermistor.

Note

- This configuration is OPN dependent.

Public Attributes

uint8_t	port The GPIO port to access the thermistor.
uint8_t	pin The GPIO pin to set the thermistor.

Public Attribute Documentation

port

```
uint8_t RAIL_HFXOThermistorConfig_t::port
```

The GPIO port to access the thermistor.

Definition at line 4437 of file common/rail_types.h

pin

```
uint8_t RAIL_HFXOThermistorConfig_t::pin
```

The GPIO pin to set the thermistor.

Definition at line 4441 of file common/rail_types.h

RAIL_HFXOCompensationConfig_t

Set compensation specific parameters.

Public Attributes

bool	enableCompensation Indicates whether the HFXO compensation in temperature is activated.
int8_t	zoneTemperatureC The temperature reference delimiting the nominal zone from the critical one.
uint8_t	deltaNominal The temperature shift used to start a new compensation, in the nominal zone.
uint8_t	deltaCritical The temperature shift used to start a new compensation, in the critical zone.

Public Attribute Documentation

enableCompensation

> bool RAIL_HFXOCompensationConfig_t::enableCompensation

Indicates whether the HFXO compensation in temperature is activated.

Definition at line 4452 of file common/rail_types.h

zoneTemperatureC

> int8_t RAIL_HFXOCompensationConfig_t::zoneTemperatureC

The temperature reference delimiting the nominal zone from the critical one.

This field is relevant if enableCompensation is set to true.

Definition at line 4457 of file common/rail_types.h

deltaNominal

> uint8_t RAIL_HFXOCompensationConfig_t::deltaNominal

The temperature shift used to start a new compensation, in the nominal zone.

This field is relevant if enableCompensation is set to true.

Definition at line 4462 of file common/rail_types.h

deltaCritical

```
uint8_t RAIL_HFXOCompensationConfig_t::deltaCritical
```

The temperature shift used to start a new compensation, in the critical zone.

This field is relevant if enableCompensation is set to true.

Definition at line `4467` of file `common/rail_types.h`

Features

Features

Overview of support for various features across hardware platforms.

These defines can be used at compile time to determine which features are available on your platform. However, keep in mind that these defines hold true for chip families. Your specific part may have further restrictions (band limitations, power amplifier restrictions, and so on) on top of those listed below, for which runtime RAIL_Supports*() APIs can be used to check availability on a particular chip (after RAIL_Init() has been called). In general, an attempt to call an API that is not supported on your chip family as listed below will result in a RAIL_STATUS_INVALID_CALL.

Functions

bool RAIL_Supports2p4GHzBand(RAIL_Handle_t railHandle)
Indicate whether RAIL supports 2.4 GHz band operation on this chip.

bool RAIL_SupportsSubGHzBand(RAIL_Handle_t railHandle)
Indicate whether RAIL supports SubGHz band operation on this chip.

bool RAIL_SupportsDualBand(RAIL_Handle_t railHandle)
Indicate whether this chip supports dual 2.4 GHz and SubGHz band operation.

bool RAIL_SupportsAddrFilterAddressBitMask(RAIL_Handle_t railHandle)
Indicate whether this chip supports bit masked address filtering.

bool RAIL_SupportsAddrFilterMask(RAIL_Handle_t railHandle)
Indicate whether this chip supports address filter mask information for incoming packets in RAIL_RxPacketInfo_t::filterMask and RAIL_IEEE802154_Address_t::filterMask.

bool RAIL_SupportsAlternateTxPower(RAIL_Handle_t railHandle)
Indicate whether this chip supports alternate TX power settings.

bool RAIL_SupportsAntennaDiversity(RAIL_Handle_t railHandle)
Indicate whether this chip supports antenna diversity.

bool RAIL_SupportsAuxAdc(RAIL_Handle_t railHandle)
Indicate whether RAIL supports AUXADC measurements on this chip.

bool RAIL_SupportsChannelHopping(RAIL_Handle_t railHandle)
Indicate whether RAIL supports channel hopping on this chip.

bool RAIL_SupportsDirectMode(RAIL_Handle_t railHandle)
Indicate whether this chip supports direct mode.

bool RAIL_SupportsDualSyncWords(RAIL_Handle_t railHandle)
Indicate whether this chip supports dual sync words.

bool RAIL_SupportsTxRepeatStartToStart(RAIL_Handle_t railHandle)
Indicate whether this chip supports start to start TX repeats.

bool RAIL_SupportsEff(RAIL_Handle_t railHandle)
Indicate whether this chip supports EFF.

bool RAIL_SupportsExternalThermistor(RAIL_Handle_t railHandle)
Indicate whether RAIL supports thermistor measurements on this chip.

bool RAIL_SupportsHFXOCompensation(RAIL_Handle_t railHandle)
Indicate whether RAIL supports HFXO compensation on this chip.

bool RAIL_SupportsMfm(RAIL_Handle_t railHandle)
Indicate whether this chip supports MFM protocol.

bool RAIL_SupportsOFDMPA(RAIL_Handle_t railHandle)
Indicate whether RAIL supports OFDM band operation on this chip.

bool RAIL_SupportsPrecisionLFRCO(RAIL_Handle_t railHandle)
Indicate whether this chip supports a high-precision LFRCO.

bool RAIL_SupportsRadioEntropy(RAIL_Handle_t railHandle)
Indicate whether this chip supports radio entropy.

bool RAIL_SupportsRfSenseEnergyDetection(RAIL_Handle_t railHandle)
Indicate whether RAIL supports RFSENSE Energy Detection Mode on this chip.

bool RAIL_SupportsRfSenseSelectiveOok(RAIL_Handle_t railHandle)
Indicate whether RAIL supports RFSENSE Selective(OOK) Mode on this chip.

bool RAIL_SupportsRssiDetectThreshold(RAIL_Handle_t railHandle)
Indicate whether this chip supports configurable RSSI threshold set by RAIL_SetRssiDetectThreshold().

bool RAIL_SupportsRxDirectModeDataToFifo(RAIL_Handle_t railHandle)
Indicate whether this chip supports RX direct mode data to FIFO.

bool RAIL_SupportsRxRawData(RAIL_Handle_t railHandle)
Indicate whether this chip supports raw RX data sources other than RAIL_RxDataSource_t::RX_PACKET_DATA.

bool RAIL_SupportsSQPhy(RAIL_Handle_t railHandle)
Indicate whether this chip supports SQ-based PHY.

bool RAIL_SupportsTxPowerMode(RAIL_Handle_t railHandle, RAIL_TxPowerMode_t powerMode, RAIL_TxPowerLevel_t *pMaxPowerLevel)
Indicate whether this chip supports a particular power mode (PA).

bool RAIL_SupportsTxPowerModeAlt(RAIL_Handle_t railHandle, RAIL_TxPowerMode_t *powerMode, RAIL_TxPowerLevel_t *maxPowerLevel, RAIL_TxPowerLevel_t *minPowerLevel)
Indicate whether this chip supports a particular power mode (PA) and provides the maximum and minimum power level for that power mode if supported by the chip.

bool RAIL_SupportsTxToTx(RAIL_Handle_t railHandle)
Indicate whether this chip supports automatic TX to TX transitions.

bool RAIL_SupportsProtocolBLE(RAIL_Handle_t railHandle)
Indicate whether RAIL supports the BLE protocol on this chip.

bool RAIL_BLE_Supports1MbpsNonViterbi(RAIL_Handle_t railHandle)
Indicate whether this chip supports BLE 1Mbps Non-Viterbi PHY.

bool RAIL_BLE_Supports1MbpsViterbi(RAIL_Handle_t railHandle)
Indicate whether this chip supports BLE 1Mbps Viterbi PHY.

bool RAIL_BLE_Supports1Mbps(RAIL_Handle_t railHandle)
Indicate whether this chip supports BLE 1Mbps operation.

bool RAIL_BLE_Supports2MbpsNonViterbi(RAIL_Handle_t railHandle)
Indicate whether this chip supports BLE 2Mbps Non-Viterbi PHY.

bool RAIL_BLE_Supports2MbpsViterbi(RAIL_Handle_t railHandle)
Indicate whether this chip supports BLE 2Mbps Viterbi PHY.

bool RAIL_BLE_Supports2Mbps(RAIL_Handle_t railHandle)
Indicate whether this chip supports BLE 2Mbps operation.

bool RAIL_BLE_SupportsAntennaSwitching(RAIL_Handle_t railHandle)
Indicate whether this chip supports BLE Antenna Switching needed for Angle-of-Arrival receives or Angle-of-Departure transmits.

bool RAIL_BLE_SupportsCodedPhy(RAIL_Handle_t railHandle)
Indicate whether this chip supports BLE Coded PHY used for Long-Range.

bool RAIL_BLE_SupportsCte(RAIL_Handle_t railHandle)
Indicate whether this chip supports BLE CTE (Constant Tone Extension) needed for Angle-of-Arrival/Departure transmits.

bool RAIL_BLE_SupportsIQSampling(RAIL_Handle_t railHandle)
Indicate whether this chip supports BLE IQ Sampling needed for Angle-of-Arrival/Departure receives.

bool RAIL_BLE_SupportsPhySwitchToRx(RAIL_Handle_t railHandle)
Indicate whether this chip supports BLE PHY switch to RX functionality, which is used to switch BLE PHYs at a specific time to receive auxiliary packets.

bool RAIL_BLE_SupportsQuuppa(RAIL_Handle_t railHandle)
Indicate whether this chip supports the Quuppa PHY.

bool RAIL_BLE_SupportsSignalIdentifier(RAIL_Handle_t railHandle)
Indicate whether this chip supports BLE signal identifier.

bool RAIL_BLE_SupportsSimulscanPhy(RAIL_Handle_t railHandle)
Indicate whether this chip supports BLE Simulscan PHY used for simultaneous BLE 1Mbps and Coded PHY reception.

bool RAIL_SupportsProtocolIEEE802154(RAIL_Handle_t railHandle)
Indicate whether this chip supports the IEEE 802.15.4 protocol.

bool RAIL_IEEE802154_SupportsCoexPhy(RAIL_Handle_t railHandle)
Indicate whether this chip supports the IEEE 802.15.4 Wi-Fi Coexistence PHY.

bool RAIL_SupportsIEEE802154Band2P4(RAIL_Handle_t railHandle)
Indicate whether this chip supports the IEEE 802.15.4 2.4 GHz band variant.

bool RAIL_SupportsThermalProtection(RAIL_Handle_t railHandle)
Indicate whether this chip supports the thermal protection.

bool RAIL_IEEE802154_SupportsRxChannelSwitching(RAIL_Handle_t railHandle)
Indicate whether this chip supports the IEEE 802.15.4 2.4 RX channel switching.

bool RAIL_IEEE802154_SupportsCustom1Phy(RAIL_Handle_t railHandle)
Indicate whether this chip supports the IEEE 802.15.4 PHY with custom settings.

bool RAIL_IEEE802154_SupportsFemPhy(RAIL_Handle_t railHandle)
Indicate whether this chip supports the IEEE 802.15.4 front end module optimized PHY.

bool RAIL_IEEE802154_SupportsCancelFramePendingLookup(RAIL_Handle_t railHandle)
Indicate whether this chip supports canceling the frame-pending lookup event RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND when the radio transitions to a state that renders the the reporting of this event moot (i.e., too late for the stack to influence the outgoing ACK).

bool RAIL_IEEE802154_SupportsEarlyFramePendingLookup(RAIL_Handle_t railHandle)
Indicate whether this chip supports early triggering of the frame-pending lookup event RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND just after MAC address fields have been received.

bool RAIL_IEEE802154_SupportsDualPaConfig(RAIL_Handle_t railHandle)
Indicate whether RAIL supports dual PA mode on this chip.

bool RAIL_IEEE802154_SupportsEEnhancedAck(RAIL_Handle_t railHandle)
Indicate whether this chip supports IEEE 802.15.4E-2012 Enhanced ACKing.

bool RAIL_IEEE802154_SupportsEMultipurposeFrames(RAIL_Handle_t railHandle)
Indicate whether this chip supports IEEE 802.15.4E-2012 Multipurpose frame reception.

bool RAIL_IEEE802154_SupportsESubsetGB868(RAIL_Handle_t railHandle)
Indicate whether this chip supports the IEEE 802.15.4E-2012 feature subset needed for Zigbee R22 GB868.

bool RAIL_IEEE802154_SupportsG4ByteCrc(RAIL_Handle_t railHandle)
Indicate whether this chip supports IEEE 802.15.4G-2012 reception and transmission of frames with 4-byte CRC.

bool RAIL_IEEE802154_SupportsGDynFec(RAIL_Handle_t railHandle)
Indicate whether this chip supports IEEE 802.15.4G dynamic FEC.

bool RAIL_SupportsProtocolWiSUN(RAIL_Handle_t railHandle)
Indicate whether this chip supports Wi-SUN.

bool RAIL_IEEE802154_SupportsGModeSwitch(RAIL_Handle_t railHandle)
Indicate whether this chip supports Wi-SUN mode switching.

bool RAIL_IEEE802154_SupportsGSubsetGB868(RAIL_Handle_t railHandle)
Indicate whether this chip supports IEEE 802.15.4G-2012 feature subset needed for Zigbee R22 GB868.

bool RAIL_IEEE802154_SupportsGUnwhitenedRx(RAIL_Handle_t railHandle)
Indicate whether this chip supports IEEE 802.15.4G-2012 reception of unwhitened frames.

bool RAIL_IEEE802154_SupportsGUnwhitenedTx(RAIL_Handle_t railHandle)
Indicate whether this chip supports IEEE 802.15.4G-2012 transmission of unwhitened frames.

bool RAIL_WMBUS_SupportsSimultaneousTCRx(RAIL_Handle_t railHandle)
Indicate whether this chip supports WMBUS simultaneous M2O RX of T and C modes.

bool RAIL_SupportsProtocolZWave(RAIL_Handle_t railHandle)
Indicate whether this chip supports the Z-Wave protocol.

bool RAIL_ZWAVE_SupportsConcPhy(RAIL_Handle_t railHandle)
Indicate whether this chip supports the Z-Wave concurrent PHY.

bool RAIL_ZWAVE_SupportsEnergyDetectPhy(RAIL_Handle_t railHandle)
Indicate whether this chip supports the Z-Wave energy detect PHY.

bool RAIL_ZWAVE_SupportsRegionPti(RAIL_Handle_t railHandle)
Indicate whether this chip supports Z-Wave Region in PTI.

bool RAIL_IEEE802154_SupportsSignalIdentifier(RAIL_Handle_t railHandle)
Indicate whether this chip supports IEEE 802.15.4 signal identifier.

bool RAIL_SupportsFastRx2Rx(RAIL_Handle_t railHandle)
Indicate whether this chip supports fast RX2RX.

bool RAIL_SupportsCollisionDetection(RAIL_Handle_t railHandle)
Indicate whether this chip supports collision detection.

bool RAIL_SupportsProtocolSidewalk(RAIL_Handle_t railHandle)
Indicate whether this chip supports Sidewalk protocol.

Macros

#define RAIL_SUPPORTS_DUAL_BAND 1
Boolean to indicate whether the selected chip supports both SubGHz and 2.4 GHz bands.

#define RAIL_FEAT_DUAL_BAND_RADIO RAIL_SUPPORTS_DUAL_BAND
Backwards-compatible synonym of RAIL_SUPPORTS_DUAL_BAND.

#define RAIL_SUPPORTS_2P4GHZ_BAND 1
Boolean to indicate whether the selected chip supports the 2.4 GHz band.

#define RAIL_FEAT_2G4_RADIO RAIL_SUPPORTS_2P4GHZ_BAND
Backwards-compatible synonym of RAIL_SUPPORTS_2P4GHZ_BAND.

#define RAIL_SUPPORTS_SUBGHZ_BAND 1
Boolean to indicate whether the selected chip supports SubGHz bands.

#define RAIL_FEAT_SUBGIG_RADIO RAIL_SUPPORTS_SUBGHZ_BAND
Backwards-compatible synonym of RAIL_SUPPORTS_SUBGHZ_BAND.

#define RAIL_SUPPORTS_OFDM_PA 0
Boolean to indicate whether the selected chip supports OFDM PA.

#define RAIL_SUPPORTS_ADDR_FILTER_ADDRESS_BIT_MASK 0
Boolean to indicate whether the selected chip supports bit masked address filtering.

#define RAIL_SUPPORTS_ADDR_FILTER_MASK 1
Boolean to indicate whether the selected chip supports address filter mask information for incoming packets in RAIL_RxPacketInfo_t::filterMask and RAIL_IEEE802154_Address_t::filterMask.

#define RAIL_SUPPORTS_ALTERNATE_TX_POWER 0
Boolean to indicate whether the selected chip supports alternate power settings for the Power Amplifier.

#define RAIL_FEAT_ALTERNATE_POWER_TX_SUPPORTED RAIL_SUPPORTS_ALTERNATE_TX_POWER
Backwards-compatible synonym of RAIL_SUPPORTS_ALTERNATE_TX_POWER.

#define RAIL_SUPPORTS_ANTENNA_DIVERSITY 0
Boolean to indicate whether the selected chip supports antenna diversity.

#define RAIL_FEAT_ANTENNA_DIVERSITY RAIL_SUPPORTS_ANTENNA_DIVERSITY
Backwards-compatible synonym of RAIL_SUPPORTS_ANTENNA_DIVERSITY.

#define RAIL_SUPPORTS_PATH_DIVERSITY 0
Boolean to indicate whether the selected chip supports path diversity.

#define RAIL_SUPPORTS_CHANNEL_HOPPING 0
Boolean to indicate whether the selected chip supports channel hopping.

#define RAIL_FEAT_CHANNEL_HOPPING RAIL_SUPPORTS_CHANNEL_HOPPING
Backwards-compatible synonym of RAIL_SUPPORTS_CHANNEL_HOPPING.

#define RAIL_SUPPORTS_DUAL_SYNC_WORDS 1
Boolean to indicate whether the selected chip supports dual sync words.

#define RAIL_SUPPORTS_TX_TO_TX 1
Boolean to indicate whether the selected chip supports automatic transitions from TX to TX.

#define RAIL_SUPPORTS_TX_REPEAT_START_TO_START 0
Boolean to indicate whether the selected chip supports RAIL_TX_REPEAT_OPTION_START_TO_START.

#define RAIL_SUPPORTS_EXTERNAL_THERMISTOR 0
Boolean to indicate whether the selected chip supports thermistor measurements.

#define RAIL_FEAT_EXTERNAL_THERMISTOR RAIL_SUPPORTS_EXTERNAL_THERMISTOR
Backwards-compatible synonym of RAIL_SUPPORTS_EXTERNAL_THERMISTOR.

#define RAIL_SUPPORTS_HFXO_COMPENSATION 0
Boolean to indicate whether the selected chip supports HFXO compensation.

#define RAIL_SUPPORTS_AUXADC 0
Boolean to indicate whether the selected chip supports AUXADC measurements.

#define RAIL_SUPPORTS_PRECISION_LFRCO 0
Boolean to indicate whether the selected chip supports a high-precision LFRCO.

#define RAIL_SUPPORTS_RADIO_ENTROPY 1
Boolean to indicate whether the selected chip supports radio entropy.

#define RAIL_SUPPORTS_RFSENSE_ENERGY_DETECTION 0
Boolean to indicate whether the selected chip supports RFSENSE Energy Detection Mode.

#define RAIL_SUPPORTS_RFSENSE_SELECTIVE_OOK 0
Boolean to indicate whether the selected chip supports RFSENSE Selective(OOK) Mode.

#define RAIL_FEAT_RFSENSE_SELECTIVE_OOK_MODE_SUPPORTED RAIL_SUPPORTS_RFSENSE_SELECTIVE_OOK
Backwards-compatible synonym of RAIL_SUPPORTS_RFSENSE_SELECTIVE_OOK.

#define RAIL_SUPPORTS_EFF 0
Boolean to indicate whether the selected chip supports the Energy Friendly Front End Module (EFF).

#define RAIL_SUPPORTS_PROTOCOL_BLE RAIL_SUPPORTS_2P4GHZ_BAND
Boolean to indicate whether the selected chip supports BLE.

#define RAIL_BLE_SUPPORTS_1MBPS_NON_VITERBI 0
Boolean to indicate whether the selected chip supports BLE 1Mbps Non-Viterbi PHY.

#define RAIL_BLE_SUPPORTS_1MBPS_VITERBI RAIL_SUPPORTS_PROTOCOL_BLE
Boolean to indicate whether the selected chip supports BLE 1Mbps Viterbi PHY.

#define RAIL_BLE_SUPPORTS_1MBPS (RAIL_BLE_SUPPORTS_1MBPS_NON_VITERBI || RAIL_BLE_SUPPORTS_1MBPS_VITERBI)
Boolean to indicate whether the selected chip supports BLE 1Mbps operation.

#define RAIL_BLE_SUPPORTS_2MBPS_NON_VITERBI 0
Boolean to indicate whether the selected chip supports BLE 2Mbps Non-Viterbi PHY.

#define RAIL_BLE_SUPPORTS_2MBPS_VITERBI RAIL_SUPPORTS_PROTOCOL_BLE
Boolean to indicate whether the selected chip supports BLE 2Mbps Viterbi PHY.

#define RAIL_BLE_SUPPORTS_2MBPS (RAIL_BLE_SUPPORTS_2MBPS_NON_VITERBI || RAIL_BLE_SUPPORTS_2MBPS_VITERBI)
Boolean to indicate whether the selected chip supports BLE 2Mbps operation.

#define RAIL_BLE_SUPPORTS_ANTENNA_SWITCHING 0
Boolean to indicate whether the selected chip supports BLE Antenna Switching needed for Angle-of-Arrival receives or Angle-of-Departure transmits.

#define RAIL_BLE_SUPPORTS_CODED_PHY 0
Boolean to indicate whether the selected chip supports the BLE Coded PHY used for Long-Range.

#define RAIL_FEAT_BLE_CODED RAIL_BLE_SUPPORTS_CODED_PHY
Backwards-compatible synonym of RAIL_BLE_SUPPORTS_CODED_PHY.

#define RAIL_BLE_SUPPORTS_SIMULSCAN_PHY 0
Boolean to indicate whether the selected chip supports the BLE Simulscan PHY used for simultaneous BLE 1Mbps and Coded PHY reception.

#define RAIL_BLE_SUPPORTS_CTE 0
Boolean to indicate whether the selected chip supports BLE CTE (Constant Tone Extension) needed for Angle-of-Arrival/Departure transmits.

#define RAIL_BLE_SUPPORTS_QUUPPA 0
Boolean to indicate whether the selected chip supports the Quuppa PHY.

#define RAIL_BLE_SUPPORTS_IQ_SAMPLING 0
Boolean to indicate whether the selected chip supports BLE IQ Sampling needed for Angle-of-Arrival/Departure receives.

#define RAIL_BLE_SUPPORTS_AOX undefined
Boolean to indicate whether the selected chip supports some BLE AOX features.

#define RAIL_FEAT_BLE_AOX_SUPPORTED RAIL_BLE_SUPPORTS_AOX
Backwards-compatible synonym of RAIL_BLE_SUPPORTS_AOX.

#define RAIL_BLE_SUPPORTS_PHY_SWITCH_TO_RX RAIL_SUPPORTS_PROTOCOL_BLE
Boolean to indicate whether the selected chip supports BLE PHY switch to RX functionality, which is used to switch BLE PHYs at a specific time to receive auxiliary packets.

#define RAIL_FEAT_BLE_PHY_SWITCH_TO_RX RAIL_BLE_SUPPORTS_PHY_SWITCH_TO_RX
Backwards-compatible synonym of RAIL_BLE_SUPPORTS_PHY_SWITCH_TO_RX.

#define RAIL_SUPPORTS_PROTOCOL_IEEE802154 1
Boolean to indicate whether the selected chip supports IEEE 802.15.4.

#define RAIL_IEEE802154_SUPPORTS_COEX_PHY 0
Boolean to indicate whether the selected chip supports the 802.15.4 Wi-Fi Coexistence PHY.

#define RAIL_FEAT_802154_COEX_PHY RAIL_IEEE802154_SUPPORTS_COEX_PHY
Backwards-compatible synonym of RAIL_IEEE802154_SUPPORTS_COEX_PHY.

#define RAIL_SUPPORTS_IEEE802154_BAND_2P4 (RAIL_SUPPORTS_PROTOCOL_IEEE802154 && RAIL_SUPPORTS_2P4GHZ_BAND)
Boolean to indicate whether the selected chip supports the IEEE 802.15.4 2.4 GHz band variant.

#define RAIL_IEEE802154_SUPPORTS_RX_CHANNEL_SWITCHING 0
Boolean to indicate whether the selected chip supports the IEEE 802.15.4 2.4 RX channel switching.

#define RAIL_IEEE802154_SUPPORTS_FEM_PHY (RAIL_SUPPORTS_IEEE802154_BAND_2P4)
Boolean to indicate whether the selected chip supports a front end module.

#define RAIL_IEEE802154_SUPPORTS_E_SUBSET_GB868 RAIL_SUPPORTS_PROTOCOL_IEEE802154
Boolean to indicate whether the selected chip supports IEEE 802.15.4E-2012 feature subset needed for Zigbee R22 GB868.

#define RAIL_FEAT_IEEE802154_E_GB868_SUPPORTED RAIL_IEEE802154_SUPPORTS_E_SUBSET_GB868
Backwards-compatible synonym of RAIL_IEEE802154_SUPPORTS_E_SUBSET_GB868.

#define RAIL_IEEE802154_SUPPORTS_E_ENHANCED_ACK RAIL_IEEE802154_SUPPORTS_E_SUBSET_GB868
Boolean to indicate whether the selected chip supports IEEE 802.15.4E-2012 Enhanced ACKing.

#define RAIL_FEAT_IEEE802154_E_ENH_ACK_SUPPORTED RAIL_IEEE802154_SUPPORTS_E_ENHANCED_ACK
Backwards-compatible synonym of RAIL_IEEE802154_SUPPORTS_E_ENHANCED_ACK.

#define RAIL_IEEE802154_SUPPORTS_E_MULTIPURPOSE_FRAMES RAIL_IEEE802154_SUPPORTS_E_SUBSET_GB868
Boolean to indicate whether the selected chip supports receiving IEEE 802.15.4E-2012 Multipurpose frames.

#define RAIL_FEAT_IEEE802154_MULTIPURPOSE_FRAME_SUPPORTED RAIL_IEEE802154_SUPPORTS_E_MULTIPURPOSE_FRAMES
Backwards-compatible synonym of RAIL_IEEE802154_SUPPORTS_E_MULTIPURPOSE_FRAMES.

#define RAIL_IEEE802154_SUPPORTS_G_SUBSET_GB868 ((RAIL_SUPPORTS_PROTOCOL_IEEE802154 != 0) && (RAIL_SUPPORTS_SUBGHZ_BAND != 0))

Boolean to indicate whether the selected chip supports IEEE 802.15.4G-2012 feature subset needed for Zigbee R22 GB868.

#define RAIL_FEAT_IEEE802154_G_GB868_SUPPORTED RAIL_IEEE802154_SUPPORTS_G_SUBSET_GB868

Backwards-compatible synonym of RAIL_IEEE802154_SUPPORTS_G_SUBSET_GB868.

#define RAIL_IEEE802154_SUPPORTS_G_DYNFEC 0

Boolean to indicate whether the selected chip supports dynamic FEC See also runtime refinement RAIL_IEEE802154_SupportsGDynFec().

#define RAIL_IEEE802154_SUPPORTS_G_MODESWITCH 0

Boolean to indicate whether the selected chip supports Wi-SUN mode switching See also runtime refinement RAIL_IEEE802154_SupportsGModeSwitch().

#define RAIL_IEEE802154_SUPPORTS_G_4BYTE_CRC RAIL_IEEE802154_SUPPORTS_G_SUBSET_GB868

Boolean to indicate whether the selected chip supports IEEE 802.15.4G-2012 reception and transmission of frames with 4-byte CRC.

#define RAIL_FEAT_IEEE802154_G_4BYTE_CRC_SUPPORTED RAIL_IEEE802154_SUPPORTS_G_4BYTE_CRC

Backwards-compatible synonym of RAIL_IEEE802154_SUPPORTS_G_4BYTE_CRC.

#define RAIL_IEEE802154_SUPPORTS_G_UNWHITENED_RX RAIL_IEEE802154_SUPPORTS_G_SUBSET_GB868

Boolean to indicate whether the selected chip supports IEEE 802.15.4G-2012 reception of unwhitened frames.

#define RAIL_FEAT_IEEE802154_G_UNWHITENED_RX_SUPPORTED RAIL_IEEE802154_SUPPORTS_G_UNWHITENED_RX

Backwards-compatible synonym of RAIL_IEEE802154_SUPPORTS_G_UNWHITENED_RX.

#define RAIL_IEEE802154_SUPPORTS_G_UNWHITENED_TX RAIL_IEEE802154_SUPPORTS_G_SUBSET_GB868

Boolean to indicate whether the selected chip supports IEEE 802.15.4G-2012 transmission of unwhitened frames.

#define RAIL_FEAT_IEEE802154_G_UNWHITENED_TX_SUPPORTED RAIL_IEEE802154_SUPPORTS_G_UNWHITENED_TX

Backwards-compatible synonym of RAIL_IEEE802154_SUPPORTS_G_UNWHITENED_TX.

#define RAIL_IEEE802154_SUPPORTS_CANCEL_FRAME_PENDING_LOOKUP RAIL_SUPPORTS_PROTOCOL_IEEE802154

Boolean to indicate whether the selected chip supports canceling the frame-pending lookup event RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND when the radio transitions to a state that renders the the reporting of this event moot (i.e., too late for the stack to influence the outgoing ACK).

#define RAIL_FEAT_IEEE802154_CANCEL_FP_LOOKUP_SUPPORTED RAIL_IEEE802154_SUPPORTS_CANCEL_FRAME_PENDING_LOOKUP

Backwards-compatible synonym of RAIL_IEEE802154_SUPPORTS_CANCEL_FRAME_PENDING_LOOKUP.

#define RAIL_IEEE802154_SUPPORTS_EARLY_FRAME_PENDING_LOOKUP RAIL_SUPPORTS_PROTOCOL_IEEE802154

Boolean to indicate whether the selected chip supports early triggering of the frame-pending lookup event RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND just after MAC address fields have been received.

#define RAIL_FEAT_IEEE802154_EARLY_FP_LOOKUP_SUPPORTED RAIL_IEEE802154_SUPPORTS_EARLY_FRAME_PENDING_LOOKUP

Backwards-compatible synonym of RAIL_IEEE802154_SUPPORTS_EARLY_FRAME_PENDING_LOOKUP.

#define RAIL_IEEE802154_SUPPORTS_DUAL_PA_CONFIG 0

Boolean to indicate whether the selected chip supports dual PA configs for mode switch or concurrent mode.

#define RAIL_SUPPORTS_DBM_POWERSETTING_MAPPING_TABLE 0

Boolean to indicate whether the selected chip supports the pa power setting table.

#define RAIL_IEEE802154_SUPPORTS_CUSTOM1_PHY 0

Boolean to indicate whether the selected chip supports IEEE 802.15.4 PHY with custom settings.

#define RAIL_SUPPORTS_PROTOCOL_WI_SUN 0
Boolean to indicate whether the selected chip supports Wi-SUN See also runtime refinement RAIL_SupportsProtocolWiSUN().

#define RAIL_WMBUS_SUPPORTS_SIMULTANEOUS_T_C_RX 0
Boolean to indicate whether the selected chip supports WMBUS simultaneous M2O RX of T and C modes set by RAIL_WMBUS_Config().

#define RAIL_SUPPORTS_PROTOCOL_ZWAVE 0
Boolean to indicate whether the selected chip supports Z-Wave.

#define RAIL_FEAT_ZWAVE_SUPPORTED RAIL_SUPPORTS_PROTOCOL_ZWAVE
Backwards-compatible synonym of RAIL_SUPPORTS_PROTOCOL_ZWAVE.

#define RAIL_ZWAVE_SUPPORTS_ED_PHY 0
Boolean to indicate whether the selected chip supports energy detect PHY.

#define RAIL_ZWAVE_SUPPORTS_CONC_PHY 0
Boolean to indicate whether the selected chip supports concurrent PHY.

#define RAIL_SUPPORTS_SQ_PHY 0
Boolean to indicate whether the selected chip supports SQ-based PHY.

#define RAIL_ZWAVE_SUPPORTS_REGION_PTI RAIL_SUPPORTS_PROTOCOL_ZWAVE
Boolean to indicate whether the code supports Z-Wave region information in PTI and newer RAIL_ZWAVE_RegionConfig_t structure See also runtime refinement RAIL_ZWAVE_SupportsRegionPti().

#define RAIL_FEAT_ZWAVE_REGION_PTI RAIL_ZWAVE_SUPPORTS_REGION_PTI
Backwards-compatible synonym of RAIL_ZWAVE_SUPPORTS_REGION_PTI.

#define RAIL_SUPPORTS_RX_RAW_DATA 1
Boolean to indicate whether the selected chip supports raw RX data sources other than RAIL_RxDataSource_t::RX_PACKET_DATA.

#define RAIL_SUPPORTS_DIRECT_MODE 0
Boolean to indicate whether the selected chip supports direct mode.

#define RAIL_SUPPORTS_RX_DIRECT_MODE_DATA_TO_FIFO 0
Boolean to indicate whether the selected chip supports RX direct mode data to FIFO.

#define RAIL_SUPPORTS_MFM 0
Boolean to indicate whether the selected chip supports MFM protocol.

#define RAIL_IEEE802154_SUPPORTS_SIGNAL_IDENTIFIER 0
Boolean to indicate whether the selected chip supports 802.15.4 signal detection.

#define RAIL_BLE_SUPPORTS_SIGNAL_IDENTIFIER 0
Boolean to indicate whether the selected chip supports BLE signal detection.

#define RAIL_SUPPORTS_RSSI_DETECT_THRESHOLD (0U)
Boolean to indicate whether the selected chip supports configurable RSSI threshold set by RAIL_SetRssiDetectThreshold().

#define RAIL_SUPPORTS_THERMAL_PROTECTION (0U)
Boolean to indicate whether the selected chip supports thermal protection set by RAIL_ConfigThermalProtection().

#define RAIL_SUPPORTS_FAST_RX2RX (0U)
Boolean to indicate whether the selected chip supports fast RX2RX enabled by RAIL_RX_OPTION_FAST_RX2RX.

#define RAIL_SUPPORTS_COLLISION_DETECTION (0U)
Boolean to indicate whether the selected chip supports collision detection enabled by RAIL_RX_OPTION_ENABLE_COLLISION_DETECTION See also runtime refinement RAIL_SupportsCollisionDetection().

#define RAIL_SUPPORTS_PROTOCOL_SIDEWALK (0U)
Boolean to indicate whether the selected chip supports Sidewalk protocol.

Function Documentation

RAIL_Supports2p4GHzBand

```
bool RAIL_Supports2p4GHzBand (RAIL_Handle_t railHandle)
```

Indicate whether RAIL supports 2.4 GHz band operation on this chip.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if the 2.4 GHz band is supported; false otherwise.

Runtime refinement of compile-time RAIL_SUPPORTS_2P4GHZ_BAND.

Definition at line `6639` of file `common/rail.h`

RAIL_SupportsSubGHzBand

```
bool RAIL_SupportsSubGHzBand (RAIL_Handle_t railHandle)
```

Indicate whether RAIL supports SubGHz band operation on this chip.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if the SubGHz band is supported; false otherwise.

Runtime refinement of compile-time RAIL_SUPPORTS_SUBGHZ_BAND.

Definition at line `6649` of file `common/rail.h`

RAIL_SupportsDualBand

```
bool RAIL_SupportsDualBand (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports dual 2.4 GHz and SubGHz band operation.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if the dual band is supported; false otherwise.

Runtime refinement of compile-time RAIL_SUPPORTS_DUAL_BAND.

Definition at line `6659` of file `common/rail.h`

RAIL_SupportsAddrFilterAddressBitMask

```
bool RAIL_SupportsAddrFilterAddressBitMask (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports bit masked address filtering.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if bit masked address filtering is supported; false otherwise.

Runtime refinement of compile-time RAIL_SUPPORTS_ADDR_FILTER_ADDRESS_BIT_MASK.

Definition at line `6670` of file `common/rail.h`

RAIL_SupportsAddrFilterMask

```
bool RAIL_SupportsAddrFilterMask (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports address filter mask information for incoming packets in RAIL_RxPacketInfo_t::filterMask and RAIL_IEEE802154_Address_t::filterMask.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if address filter information is supported; false otherwise (in which case RAIL_RxPacketInfo_t::filterMask value is undefined).

Runtime refinement of compile-time RAIL_SUPPORTS_ADDR_FILTER_MASK.

Definition at line `6684` of file `common/rail.h`

RAIL_SupportsAlternateTxPower

```
bool RAIL_SupportsAlternateTxPower (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports alternate TX power settings.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if alternate TX power settings are supported; false otherwise.

Runtime refinement of compile-time RAIL_SUPPORTS_ALTERNATE_TX_POWER.

Definition at line `6695` of file `common/rail.h`

RAIL_SupportsAntennaDiversity

bool RAIL_SupportsAntennaDiversity (RAIL_Handle_t railHandle)

Indicate whether this chip supports antenna diversity.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if antenna diversity is supported; false otherwise.

Runtime refinement of compile-time RAIL_SUPPORTS_ANTENNA_DIVERSITY.

Note

- Certain radio configurations may not support this feature even if the chip in general claims to support it.

Definition at line 6708 of file common/rail.h

RAIL_SupportsAuxAdc

bool RAIL_SupportsAuxAdc (RAIL_Handle_t railHandle)

Indicate whether RAIL supports AUXADC measurements on this chip.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if AUXADC measurements are supported; false otherwise.

Runtime refinement of compile-time RAIL_SUPPORTS_AUXADC.

Definition at line 6718 of file common/rail.h

RAIL_SupportsChannelHopping

bool RAIL_SupportsChannelHopping (RAIL_Handle_t railHandle)

Indicate whether RAIL supports channel hopping on this chip.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if channel hopping is supported; false otherwise.

Runtime refinement of compile-time RAIL_SUPPORTS_CHANNEL_HOPPING.

Definition at line 6728 of file common/rail.h

RAIL_SupportsDirectMode

```
bool RAIL_SupportsDirectMode (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports direct mode.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if direct mode is supported; false otherwise.

Runtime refinement of compile-time RAIL_SUPPORTS_DIRECT_MODE.

Definition at line `6739` of file `common/rail.h`

RAIL_SupportsDualSyncWords

```
bool RAIL_SupportsDualSyncWords (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports dual sync words.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if dual sync words are supported; false otherwise.

Runtime refinement of compile-time RAIL_SUPPORTS_DUAL_SYNC_WORDS.

Note

- Certain radio configurations may not support this feature even if the chip in general claims to support it.

Definition at line `6752` of file `common/rail.h`

RAIL_SupportsTxRepeatStartToStart

```
bool RAIL_SupportsTxRepeatStartToStart (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports start to start TX repeats.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if start to start TX repeats are supported; false otherwise.

Runtime refinement of compile-time RAIL_SUPPORTS_TX_REPEAT_START_TO_START.

Definition at line `6763` of file `common/rail.h`

RAIL_SupportsEff

bool RAIL_SupportsEff (RAIL_Handle_t railHandle)

Indicate whether this chip supports EFF.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if EFF identifier is supported; false otherwise.

Definition at line `6771` of file `common/rail.h`

RAIL_SupportsExternalThermistor

bool RAIL_SupportsExternalThermistor (RAIL_Handle_t railHandle)

Indicate whether RAIL supports thermistor measurements on this chip.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if thermistor measurements are supported; false otherwise.

Runtime refinement of compile-time RAIL_SUPPORTS_EXTERNAL_THERMISTOR.

Definition at line `6781` of file `common/rail.h`

RAIL_SupportsHFXOCompensation

bool RAIL_SupportsHFXOCompensation (RAIL_Handle_t railHandle)

Indicate whether RAIL supports HFXO compensation on this chip.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if HFXO compensation is supported and RAIL_ConfigHFXOThermistor() has been successfully called; false otherwise.

Runtime refinement of compile-time RAIL_SUPPORTS_HFXO_COMPENSATION.

Definition at line `6793` of file `common/rail.h`

RAIL_SupportsMfm

bool RAIL_SupportsMfm (RAIL_Handle_t railHandle)

Indicate whether this chip supports MFM protocol.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if MFM protocol is supported; false otherwise.

Runtime refinement of compile-time RAIL_SUPPORTS_MFM.

Definition at line 6803 of file common/rail.h

RAIL_SupportsOFDMPA

```
bool RAIL_SupportsOFDMPA (RAIL_Handle_t railHandle)
```

Indicate whether RAIL supports OFDM band operation on this chip.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if OFDM operation is supported; false otherwise.

Runtime refinement of compile-time RAIL_SUPPORTS_OFDM_PA.

Definition at line 6813 of file common/rail.h

RAIL_SupportsPrecisionLFRCO

```
bool RAIL_SupportsPrecisionLFRCO (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports a high-precision LFRCO.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if high-precision LFRCO is supported; false otherwise.

Runtime refinement of compile-time RAIL_SUPPORTS_PRECISION_LFRCO.

Definition at line 6823 of file common/rail.h

RAIL_SupportsRadioEntropy

```
bool RAIL_SupportsRadioEntropy (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports radio entropy.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if radio entropy is supported; false otherwise.

Runtime refinement of compile-time RAIL_SUPPORTS_RADIO_ENTROPY.

Definition at line 6833 of file common/rail.h

RAIL_SupportsRfSenseEnergyDetection

```
bool RAIL_SupportsRfSenseEnergyDetection (RAIL_Handle_t railHandle)
```

Indicate whether RAIL supports RFSENSE Energy Detection Mode on this chip.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if RFSENSE Energy Detection Mode is supported; false otherwise.

Runtime refinement of compile-time RAIL_SUPPORTS_RFSENSE_ENERGY_DETECTION.

Definition at line 6844 of file common/rail.h

RAIL_SupportsRfSenseSelectiveOok

```
bool RAIL_SupportsRfSenseSelectiveOok (RAIL_Handle_t railHandle)
```

Indicate whether RAIL supports RFSENSE Selective(OOK) Mode on this chip.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if RFSENSE Selective(OOK) Mode is supported; false otherwise.

Runtime refinement of compile-time RAIL_SUPPORTS_RFSENSE_SELECTIVE_OOK.

Definition at line 6854 of file common/rail.h

RAIL_SupportsRssiDetectThreshold

```
bool RAIL_SupportsRssiDetectThreshold (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports configurable RSSI threshold set by RAIL_SetRssiDetectThreshold().

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if setting configurable RSSI is supported; false otherwise.

Runtime refinement of compile-time RAIL_SUPPORTS_RSSI_DETECT_THRESHOLD.

Definition at line 6865 of file common/rail.h

RAIL_SupportsRxDirectModeDataToFifo

```
bool RAIL_SupportsRxDirectModeDataToFifo (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports RX direct mode data to FIFO.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if direct mode data to FIFO is supported; false otherwise.

Runtime refinement of compile-time RAIL_SUPPORTS_RX_DIRECT_MODE_DATA_TO_FIFO.

Definition at line `6876` of file `common/rail.h`

RAIL_SupportsRxRawData

```
bool RAIL_SupportsRxRawData (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports raw RX data sources other than RAIL_RxDataSource_t::RX_PACKET_DATA.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if direct mode is supported; false otherwise.

Runtime refinement of compile-time RAIL_SUPPORTS_RX_RAW_DATA.

Definition at line `6887` of file `common/rail.h`

RAIL_SupportsSQPhy

```
bool RAIL_SupportsSQPhy (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports SQ-based PHY.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if the SQ-based PHY is supported; false otherwise.

Runtime refinement of compile-time RAIL_SUPPORTS_SQ_PHY.

Definition at line `6897` of file `common/rail.h`

RAIL_SupportsTxPowerMode

```
bool RAIL_SupportsTxPowerMode (RAIL_Handle_t railHandle, RAIL_TxPowerMode_t powerMode, RAIL_TxPowerLevel_t
*pMaxPowerLevel)
```

Indicate whether this chip supports a particular power mode (PA).

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	powerMode	The power mode to check if supported.
[out]	pMaxPowerLevel	A pointer to a RAIL_TxPowerLevel_t that if non-NULL will be filled in with the power mode's highest power level allowed if this function returns true.

Note

- Consider using RAIL_SupportsTxPowerModeAlt to also get the power mode's lowest allowed power level.

Returns

- true if the powerMode is supported; false otherwise.

This function has no compile-time equivalent.

Definition at line 6913 of file common/rail.h

RAIL_SupportsTxPowerModeAlt

```
bool RAIL_SupportsTxPowerModeAlt (RAIL_Handle_t railHandle, RAIL_TxPowerMode_t *powerMode, RAIL_TxPowerLevel_t *maxPowerLevel, RAIL_TxPowerLevel_t *minPowerLevel)
```

Indicate whether this chip supports a particular power mode (PA) and provides the maximum and minimum power level for that power mode if supported by the chip.

Parameters

[in]	railHandle	A RAIL instance handle.
[inout]	powerMode	A pointer to PA power mode to check if supported. For platforms that support RAIL_TX_POWER_MODE_2P4_HIGHEST or RAIL_TX_POWER_MODE_SUBGIG_HIGHEST the powerMode is updated to the highest corresponding PA available on the chip.
[out]	maxPowerLevel	A pointer to a RAIL_TxPowerLevel_t that if non-NULL will be filled in with the power mode's highest power level allowed if this function returns true.
[out]	minPowerLevel	A pointer to a RAIL_TxPowerLevel_t that if non-NULL will be filled in with the power mode's lowest power level allowed if this function returns true.

Returns

- true if powerMode is supported; false otherwise.

Definition at line 6935 of file common/rail.h

RAIL_SupportsTxToTx

```
bool RAIL_SupportsTxToTx (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports automatic TX to TX transitions.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

true if TX to TX transitions are supported; false otherwise.

Runtime refinement of compile-time RAIL_SUPPORTS_TX_TO_TX.

Definition at line 6948 of file common/rail.h

RAIL_SupportsProtocolBLE

```
bool RAIL_SupportsProtocolBLE (RAIL_Handle_t railHandle)
```

Indicate whether RAIL supports the BLE protocol on this chip.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if BLE is supported; false otherwise.

Runtime refinement of compile-time RAIL_SUPPORTS_PROTOCOL_BLE.

Definition at line 6958 of file common/rail.h

RAIL_BLE_Supports1MbpsNonViterbi

```
bool RAIL_BLE_Supports1MbpsNonViterbi (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports BLE 1Mbps Non-Viterbi PHY.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if BLE 1Mbps Non-Viterbi is supported; false otherwise.

Runtime refinement of compile-time RAIL_BLE_SUPPORTS_1MBPS_NON_VITERBI.

Definition at line 6968 of file common/rail.h

RAIL_BLE_Supports1MbpsViterbi

```
bool RAIL_BLE_Supports1MbpsViterbi (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports BLE 1Mbps Viterbi PHY.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if BLE 1Mbps Viterbi is supported; false otherwise.

Runtime refinement of compile-time RAIL_BLE_SUPPORTS_1MBPS_VITERBI.

Definition at line 6978 of file common/rail.h

RAIL_BLE_Supports1Mbps

```
static bool RAIL_BLE_Supports1Mbps (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports BLE 1Mbps operation.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if BLE 1Mbps operation is supported; false otherwise.

Runtime refinement of compile-time RAIL_BLE_SUPPORTS_1MBPS.

Definition at line `6989` of file `common/rail.h`

RAIL_BLE_Supports2MbpsNonViterbi

```
bool RAIL_BLE_Supports2MbpsNonViterbi (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports BLE 2Mbps Non-Viterbi PHY.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if BLE 2Mbps Non-Viterbi is supported; false otherwise.

Runtime refinement of compile-time RAIL_BLE_SUPPORTS_2MBPS_NON_VITERBI.

Definition at line `7004` of file `common/rail.h`

RAIL_BLE_Supports2MbpsViterbi

```
bool RAIL_BLE_Supports2MbpsViterbi (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports BLE 2Mbps Viterbi PHY.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if BLE 2Mbps Viterbi is supported; false otherwise.

Runtime refinement of compile-time RAIL_BLE_SUPPORTS_2MBPS_VITERBI.

Definition at line `7014` of file `common/rail.h`

RAIL_BLE_Supports2Mbps

```
static bool RAIL_BLE_Supports2Mbps (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports BLE 2Mbps operation.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if BLE 2Mbps operation is supported; false otherwise.

Runtime refinement of compile-time RAIL_BLE_SUPPORTS_2MBPS.

Definition at line 7025 of file common/rail.h

RAIL_BLE_SupportsAntennaSwitching

```
bool RAIL_BLE_SupportsAntennaSwitching (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports BLE Antenna Switching needed for Angle-of-Arrival receives or Angle-of-Departure transmits.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if BLE Antenna Switching is supported; false otherwise.

Runtime refinement of compile-time RAIL_BLE_SUPPORTS_ANTENNA_SWITCHING.

Definition at line 7040 of file common/rail.h

RAIL_BLE_SupportsCodedPhy

```
bool RAIL_BLE_SupportsCodedPhy (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports BLE Coded PHY used for Long-Range.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if BLE Coded PHY is supported; false otherwise.

Runtime refinement of compile-time RAIL_BLE_SUPPORTS_CODED_PHY.

Definition at line 7050 of file common/rail.h

RAIL_BLE_SupportsCte

```
bool RAIL_BLE_SupportsCte (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports BLE CTE (Constant Tone Extension) needed for Angle-of-Arrival/Departure transmits.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if BLE CTE is supported; false otherwise.

Runtime refinement of compile-time RAIL_BLE_SUPPORTS_CTE.

Definition at line 7061 of file common/rail.h

RAIL_BLE_SupportsIQSampling

```
bool RAIL_BLE_SupportsIQSampling (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports BLE IQ Sampling needed for Angle-of-Arrival/Departure receives.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if BLE IQ Sampling is supported; false otherwise.

Runtime refinement of compile-time RAIL_BLE_SUPPORTS_IQ_SAMPLING.

Definition at line 7084 of file common/rail.h

RAIL_BLE_SupportsPhySwitchToRx

```
bool RAIL_BLE_SupportsPhySwitchToRx (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports BLE PHY switch to RX functionality, which is used to switch BLE PHYs at a specific time to receive auxiliary packets.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if BLE PHY switch to RX is supported; false otherwise.

Runtime refinement of compile-time RAIL_BLE_SUPPORTS_PHY_SWITCH_TO_RX.

Definition at line 7096 of file common/rail.h

RAIL_BLE_SupportsQuuppa

```
bool RAIL_BLE_SupportsQuuppa (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports the Quuppa PHY.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

true if the Quuppa is supported; false otherwise.

Runtime refinement of compile-time RAIL_BLE_SUPPORTS_QUUPPA.

Definition at line 7106 of file common/rail.h

RAIL_BLE_SupportsSignalIdentifier

```
bool RAIL_BLE_SupportsSignalIdentifier (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports BLE signal identifier.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if signal identifier is supported; false otherwise.

Definition at line 7114 of file common/rail.h

RAIL_BLE_SupportsSimulscanPhy

```
bool RAIL_BLE_SupportsSimulscanPhy (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports BLE Simulscan PHY used for simultaneous BLE 1Mbps and Coded PHY reception.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if BLE Simulscan PHY is supported; false otherwise.

Runtime refinement of compile-time RAIL_BLE_SUPPORTS_SIMULSCAN_PHY.

Definition at line 7125 of file common/rail.h

RAIL_SupportsProtocolIEEE802154

```
bool RAIL_SupportsProtocolIEEE802154 (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports the IEEE 802.15.4 protocol.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if the 802.15.4 protocol is supported; false otherwise.

Runtime refinement of compile-time RAIL_SUPPORTS_PROTOCOL_IEEE802154.

Definition at line 7135 of file common/rail.h

RAIL_IEEE802154_SupportsCoexPhy

```
bool RAIL_IEEE802154_SupportsCoexPhy (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports the IEEE 802.15.4 Wi-Fi Coexistence PHY.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if the 802.15.4 COEX PHY is supported; false otherwise.

Runtime refinement of compile-time RAIL_IEEE802154_SUPPORTS_COEX_PHY.

Definition at line 7157 of file common/rail.h

RAIL_SupportsIEEE802154Band2P4

```
bool RAIL_SupportsIEEE802154Band2P4 (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports the IEEE 802.15.4 2.4 GHz band variant.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if IEEE 802.15.4 2.4 GHz band variant is supported; false otherwise.

Runtime refinement of compile-time RAIL_SUPPORTS_IEEE802154_BAND_2P4.

Definition at line 7168 of file common/rail.h

RAIL_SupportsThermalProtection

```
bool RAIL_SupportsThermalProtection (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports the thermal protection.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if thermal protection is supported; false otherwise.

Runtime refinement of compile-time RAIL_SUPPORTS_THERMAL_PROTECTION.

Definition at line 7179 of file common/rail.h

RAIL_IEEE802154_SupportsRxChannelSwitching

```
bool RAIL_IEEE802154_SupportsRxChannelSwitching (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports the IEEE 802.15.4 2.4 RX channel switching.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if IEEE 802.15.4 2.4 GHz RX channel switching is supported; false otherwise.

Runtime refinement of compile-time RAIL_IEEE802154_SUPPORTS_RX_CHANNEL_SWITCHING.

Definition at line `7190` of file `common/rail.h`

RAIL_IEEE802154_SupportsCustom1Phy

```
bool RAIL_IEEE802154_SupportsCustom1Phy (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports the IEEE 802.15.4 PHY with custom settings.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if the 802.15.4 PHY with custom settings is supported; false otherwise.

Runtime refinement of compile-time RAIL_IEEE802154_SUPPORTS_CUSTOM1_PHY.

Definition at line `7200` of file `common/rail.h`

RAIL_IEEE802154_SupportsFemPhy

```
bool RAIL_IEEE802154_SupportsFemPhy (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports the IEEE 802.15.4 front end module optimized PHY.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if a front end module is supported; false otherwise.

Runtime refinement of compile-time RAIL_IEEE802154_SUPPORTS_FEM_PHY.

Definition at line `7211` of file `common/rail.h`

RAIL_IEEE802154_SupportsCancelFramePendingLookup

```
bool RAIL_IEEE802154_SupportsCancelFramePendingLookup (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports canceling the frame-pending lookup event RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND when the radio transitions to a state that renders the the reporting of this event moot (i.e., too late for the stack to influence the outgoing ACK).

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if canceling the lookup event is supported; false otherwise.

Runtime refinement of compile-time RAIL_IEEE802154_SUPPORTS_CANCEL_FRAME_PENDING_LOOKUP.

Definition at line 7225 of file common/rail.h

RAIL_IEEE802154_SupportsEarlyFramePendingLookup

```
bool RAIL_IEEE802154_SupportsEarlyFramePendingLookup (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports early triggering of the frame-pending lookup event RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND just after MAC address fields have been received.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if early triggering is supported; false otherwise.

Runtime refinement of compile-time RAIL_IEEE802154_SUPPORTS_EARLY_FRAME_PENDING_LOOKUP.

Definition at line 7238 of file common/rail.h

RAIL_IEEE802154_SupportsDualPaConfig

```
bool RAIL_IEEE802154_SupportsDualPaConfig (RAIL_Handle_t railHandle)
```

Indicate whether RAIL supports dual PA mode on this chip.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if the dual PA mode is supported; false otherwise.

Runtime refinement of compile-time RAIL_IEEE802154_SUPPORTS_DUAL_PA_CONFIG.

Definition at line 7248 of file common/rail.h

RAIL_IEEE802154_SupportsEEnhancedAck

```
bool RAIL_IEEE802154_SupportsEEnhancedAck (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports IEEE 802.15.4E-2012 Enhanced ACKing.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

true if 802.15.4E Enhanced ACKing is supported; false otherwise.

Runtime refinement of compile-time RAIL_IEEE802154_SUPPORTS_E_ENHANCED_ACK.

Definition at line 7259 of file common/rail.h

RAIL_IEEE802154_SupportsEMultipurposeFrames

```
bool RAIL_IEEE802154_SupportsEMultipurposeFrames (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports IEEE 802.15.4E-2012 Multipurpose frame reception.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if Multipurpose frame reception is supported; false otherwise.

Runtime refinement of compile-time RAIL_IEEE802154_SUPPORTS_E_MULTIPURPOSE_FRAMES.

Definition at line 7271 of file common/rail.h

RAIL_IEEE802154_SupportsESubsetGB868

```
bool RAIL_IEEE802154_SupportsESubsetGB868 (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports the IEEE 802.15.4E-2012 feature subset needed for Zigbee R22 GB868.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if 802.15.4E GB868 subset is supported; false otherwise.

Runtime refinement of compile-time RAIL_IEEE802154_SUPPORTS_E_SUBSET_GB868.

Definition at line 7283 of file common/rail.h

RAIL_IEEE802154_SupportsG4ByteCrc

```
bool RAIL_IEEE802154_SupportsG4ByteCrc (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports IEEE 802.15.4G-2012 reception and transmission of frames with 4-byte CRC.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if 802.15.4G 4-byte CRC is supported; false otherwise.

Runtime refinement of compile-time RAIL_IEEE802154_SUPPORTS_G_4BYTE_CRC.

Definition at line 7295 of file common/rail.h

RAIL_IEEE802154_SupportsGDynFec

```
bool RAIL_IEEE802154_SupportsGDynFec (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports IEEE 802.15.4G dynamic FEC.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if dynamic FEC is supported; false otherwise.

Runtime refinement of compile-time RAIL_IEEE802154_SUPPORTS_G_DYNFEC.

Definition at line `7306` of file `common/rail.h`

RAIL_SupportsProtocolWiSUN

```
bool RAIL_SupportsProtocolWiSUN (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports Wi-SUN.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if Wi-SUN is supported; false otherwise.

Runtime refinement of compile-time RAIL_SUPPORTS_PROTOCOL_WI_SUN.

Definition at line `7317` of file `common/rail.h`

RAIL_IEEE802154_SupportsGModeSwitch

```
bool RAIL_IEEE802154_SupportsGModeSwitch (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports Wi-SUN mode switching.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if Wi-SUN mode switching is supported; false otherwise.

Runtime refinement of compile-time RAIL_IEEE802154_SUPPORTS_G_MODESWITCH.

Definition at line `7328` of file `common/rail.h`

RAIL_IEEE802154_SupportsGSubsetGB868

```
bool RAIL_IEEE802154_SupportsGSubsetGB868 (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports IEEE 802.15.4G-2012 feature subset needed for Zigbee R22 GB868.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if 802.15.4G GB868 subset is supported; false otherwise.

Runtime refinement of compile-time RAIL_IEEE802154_SUPPORTS_G_SUBSET_GB868.

Definition at line `7340` of file `common/rail.h`

RAIL_IEEE802154_SupportsGUnwhitenedRx

```
bool RAIL_IEEE802154_SupportsGUnwhitenedRx (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports IEEE 802.15.4G-2012 reception of unwhitened frames.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if 802.15.4G unwhitened frame reception is supported; false otherwise.

Runtime refinement of compile-time RAIL_IEEE802154_SUPPORTS_G_UNWHITENED_RX.

Definition at line `7353` of file `common/rail.h`

RAIL_IEEE802154_SupportsGUnwhitenedTx

```
bool RAIL_IEEE802154_SupportsGUnwhitenedTx (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports IEEE 802.15.4G-2012 transmission of unwhitened frames.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if 802.15.4G unwhitened frame transmit is supported; false otherwise.

Runtime refinement of compile-time RAIL_IEEE802154_SUPPORTS_G_UNWHITENED_TX.

Definition at line `7366` of file `common/rail.h`

RAIL_WMBUS_SupportsSimultaneousTCRx

```
bool RAIL_WMBUS_SupportsSimultaneousTCRx (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports WMBUS simultaneous M2O RX of T and C modes.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if the WMBUS simultaneous M2O RX of T and C modes is supported; false otherwise.

Runtime refinement of compile-time RAIL_WMBUS_SUPPORTS_SIMULTANEOUS_T_C_RX.

Definition at line 7376 of file common/rail.h

RAIL_SupportsProtocolZWave

```
bool RAIL_SupportsProtocolZWave (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports the Z-Wave protocol.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if the Z-Wave protocol is supported; false otherwise.

Runtime refinement of compile-time RAIL_SUPPORTS_PROTOCOL_ZWAVE.

Definition at line 7386 of file common/rail.h

RAIL_ZWAVE_SupportsConcPhy

```
bool RAIL_ZWAVE_SupportsConcPhy (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports the Z-Wave concurrent PHY.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if the Z-Wave concurrent PHY is supported; false otherwise.

Runtime refinement of compile-time RAIL_ZWAVE_SUPPORTS_CONC_PHY.

Definition at line 7396 of file common/rail.h

RAIL_ZWAVE_SupportsEnergyDetectPhy

```
bool RAIL_ZWAVE_SupportsEnergyDetectPhy (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports the Z-Wave energy detect PHY.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if the Z-Wave energy detect PHY is supported; false otherwise.

Runtime refinement of compile-time RAIL_ZWAVE_SUPPORTS_ED_PHY.

Definition at line 7406 of file common/rail.h

RAIL_ZWAVE_SupportsRegionPti

```
bool RAIL_ZWAVE_SupportsRegionPti (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports Z-Wave Region in PTI.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if ZWAVE Region in PTI is supported; false otherwise.

Runtime refinement of compile-time RAIL_ZWAVE_SUPPORTS_REGION_PTI.

Definition at line 7416 of file common/rail.h

RAIL_IEEE802154_SupportsSignalIdentifier

```
bool RAIL_IEEE802154_SupportsSignalIdentifier (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports IEEE 802.15.4 signal identifier.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if signal identifier is supported; false otherwise.

Definition at line 7424 of file common/rail.h

RAIL_SupportsFastRx2Rx

```
bool RAIL_SupportsFastRx2Rx (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports fast RX2RX.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if fast RX2RX is supported; false otherwise.

Runtime refinement of compile-time RAIL_SUPPORTS_FAST_RX2RX.

Definition at line 7434 of file common/rail.h

RAIL_SupportsCollisionDetection

```
bool RAIL_SupportsCollisionDetection (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports collision detection.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if collision detection is supported; false otherwise.

Runtime refinement of compile-time RAIL_SUPPORTS_COLLISION_DETECTION.

Definition at line 7444 of file common/rail.h

RAIL_SupportsProtocolSidewalk

```
bool RAIL_SupportsProtocolSidewalk (RAIL_Handle_t railHandle)
```

Indicate whether this chip supports Sidewalk protocol.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if Sidewalk protocol is supported; false otherwise.

Runtime refinement of compile-time RAIL_SUPPORTS_PROTOCOL_SIDEWALK.

Definition at line 7454 of file common/rail.h

Macro Definition Documentation

RAIL_SUPPORTS_DUAL_BAND

```
#define RAIL_SUPPORTS_DUAL_BAND
```

Value:

```
1
```

Boolean to indicate whether the selected chip supports both SubGHz and 2.4 GHz bands.

See also runtime refinement RAIL_SupportsDualBand().

Definition at line 69 of file common/rail_features.h

RAIL_FEAT_DUAL_BAND_RADIO

```
#define RAIL_FEAT_DUAL_BAND_RADIO
```

Value:

```
RAIL_SUPPORTS_DUAL_BAND
```

Backwards-compatible synonym of RAIL_SUPPORTS_DUAL_BAND.

Definition at line 74 of file common/rail_features.h

RAIL_SUPPORTS_2P4GHZ_BAND

```
#define RAIL_SUPPORTS_2P4GHZ_BAND
```

Value:

```
1
```

Boolean to indicate whether the selected chip supports the 2.4 GHz band.

See also runtime refinement RAIL_Supports2p4GHzBand().

Definition at line 81 of file common/rail_features.h

RAIL_FEAT_2G4_RADIO

```
#define RAIL_FEAT_2G4_RADIO
```

Value:

```
RAIL_SUPPORTS_2P4GHZ_BAND
```

Backwards-compatible synonym of RAIL_SUPPORTS_2P4GHZ_BAND.

Definition at line 86 of file common/rail_features.h

RAIL_SUPPORTS_SUBGHZ_BAND

```
#define RAIL_SUPPORTS_SUBGHZ_BAND
```

Value:

```
1
```

Boolean to indicate whether the selected chip supports SubGHz bands.

See also runtime refinement RAIL_SupportsSubGHzBand().

Definition at line 93 of file common/rail_features.h

RAIL_FEAT_SUBGIG_RADIO

```
#define RAIL_FEAT_SUBGIG_RADIO
```

Value:

```
RAIL_SUPPORTS_SUBGHZ_BAND
```

Backwards-compatible synonym of RAIL_SUPPORTS_SUBGHZ_BAND.

Definition at line 98 of file common/rail_features.h

RAIL_SUPPORTS_OFDM_PA

```
#define RAIL_SUPPORTS_OFDM_PA
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports OFDM PA.

See also runtime refinement RAIL_SupportsOFDMPA().

Definition at line 105 of file common/rail_features.h

RAIL_SUPPORTS_ADDR_FILTER_ADDRESS_BIT_MASK

```
#define RAIL_SUPPORTS_ADDR_FILTER_ADDRESS_BIT_MASK
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports bit masked address filtering.

See also runtime refinement RAIL_SupportsAddrFilterAddressBitMask().

Definition at line 114 of file common/rail_features.h

RAIL_SUPPORTS_ADDR_FILTER_MASK

```
#define RAIL_SUPPORTS_ADDR_FILTER_MASK
```

Value:

```
1
```

Boolean to indicate whether the selected chip supports address filter mask information for incoming packets in RAIL_RxPacketInfo_t::filterMask and RAIL_IEEE802154_Address_t::filterMask.

See also runtime refinement RAIL_SupportsAddrFilterMask().

Definition at line 123 of file common/rail_features.h

RAIL_SUPPORTS_ALTERNATE_TX_POWER

```
#define RAIL_SUPPORTS_ALTERNATE_TX_POWER
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports alternate power settings for the Power Amplifier.

See also runtime refinement RAIL_SupportsAlternateTxPower().

Definition at line 136 of file common/rail_features.h

RAIL_FEAT_ALTERNATE_POWER_TX_SUPPORTED

```
#define RAIL_FEAT_ALTERNATE_POWER_TX_SUPPORTED
```

Value:

```
RAIL_SUPPORTS_ALTERNATE_TX_POWER
```

Backwards-compatible synonym of RAIL_SUPPORTS_ALTERNATE_TX_POWER.

Definition at line 139 of file common/rail_features.h

RAIL_SUPPORTS_ANTENNA_DIVERSITY

```
#define RAIL_SUPPORTS_ANTENNA_DIVERSITY
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports antenna diversity.

See also runtime refinement RAIL_SupportsAntennaDiversity().

Definition at line 147 of file common/rail_features.h

RAIL_FEAT_ANTENNA_DIVERSITY

```
#define RAIL_FEAT_ANTENNA_DIVERSITY
```

Value:

```
RAIL_SUPPORTS_ANTENNA_DIVERSITY
```

Backwards-compatible synonym of RAIL_SUPPORTS_ANTENNA_DIVERSITY.

Definition at line 150 of file common/rail_features.h

RAIL_SUPPORTS_PATH_DIVERSITY

```
#define RAIL_SUPPORTS_PATH_DIVERSITY
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports path diversity.

Definition at line 156 of file common/rail_features.h

RAIL_SUPPORTS_CHANNEL_HOPPING

```
#define RAIL_SUPPORTS_CHANNEL_HOPPING
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports channel hopping.

See also runtime refinement RAIL_SupportsChannelHopping().

Definition at line 164 of file common/rail_features.h

RAIL_FEAT_CHANNEL_HOPPING

```
#define RAIL_FEAT_CHANNEL_HOPPING
```

Value:

```
RAIL_SUPPORTS_CHANNEL_HOPPING
```

Backwards-compatible synonym of RAIL_SUPPORTS_CHANNEL_HOPPING.

Definition at line 167 of file common/rail_features.h

RAIL_SUPPORTS_DUAL_SYNC_WORDS

```
#define RAIL_SUPPORTS_DUAL_SYNC_WORDS
```

Value:

```
1
```

Boolean to indicate whether the selected chip supports dual sync words.

See also runtime refinement RAIL_SupportsDualSyncWords().

Definition at line 172 of file common/rail_features.h

RAIL_SUPPORTS_TX_TO_TX

```
#define RAIL_SUPPORTS_TX_TO_TX
```

Value:

```
1
```

Boolean to indicate whether the selected chip supports automatic transitions from TX to TX.

See also runtime refinement RAIL_SupportsTxToTx().

Definition at line 181 of file common/rail_features.h

RAIL_SUPPORTS_TX_REPEAT_START_TO_START

```
#define RAIL_SUPPORTS_TX_REPEAT_START_TO_START
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports RAIL_TX_REPEAT_OPTION_START_TO_START.

See also runtime refinement RAIL_SupportsTxRepeatStartToStart().

Definition at line 191 of file common/rail_features.h

RAIL_SUPPORTS_EXTERNAL_THERMISTOR

```
#define RAIL_SUPPORTS_EXTERNAL_THERMISTOR
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports thermistor measurements.

See also runtime refinement RAIL_SupportsExternalThermistor().

Definition at line 203 of file common/rail_features.h

RAIL_FEAT_EXTERNAL_THERMISTOR

```
#define RAIL_FEAT_EXTERNAL_THERMISTOR
```

Value:

```
RAIL_SUPPORTS_EXTERNAL_THERMISTOR
```

Backwards-compatible synonym of RAIL_SUPPORTS_EXTERNAL_THERMISTOR.

Definition at line 206 of file common/rail_features.h

RAIL_SUPPORTS_HFXO_COMPENSATION

```
#define RAIL_SUPPORTS_HFXO_COMPENSATION
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports HFXO compensation.

See also runtime refinement RAIL_SupportsHFXOCompensation().

Definition at line 213 of file common/rail_features.h

RAIL_SUPPORTS_AUXADC

```
#define RAIL_SUPPORTS_AUXADC
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports AUXADC measurements.

See also runtime refinement RAIL_SupportsAuxAdc().

Definition at line 223 of file common/rail_features.h

RAIL_SUPPORTS_PRECISION_LFRCO

```
#define RAIL_SUPPORTS_PRECISION_LFRCO
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports a high-precision LFRCO.

Best to use the runtime refinement RAIL_SupportsPrecisionLFRCO() because some chip revisions do not support it.

Definition at line 233 of file common/rail_features.h

RAIL_SUPPORTS_RADIO_ENTROPY

```
#define RAIL_SUPPORTS_RADIO_ENTROPY
```

Value:

```
1
```

Boolean to indicate whether the selected chip supports radio entropy.

See also runtime refinement RAIL_SupportsRadioEntropy().

Definition at line 239 of file common/rail_features.h

RAIL_SUPPORTS_RFSENSE_ENERGY_DETECTION

```
#define RAIL_SUPPORTS_RFSENSE_ENERGY_DETECTION
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports RFSENSE Energy Detection Mode.

See also runtime refinement RAIL_SupportsRfSenseEnergyDetection().

Definition at line 250 of file common/rail_features.h

RAIL_SUPPORTS_RFSENSE_SELECTIVE_OOK

```
#define RAIL_SUPPORTS_RFSENSE_SELECTIVE_OOK
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports RFSENSE Selective(OOK) Mode.

See also runtime refinement RAIL_SupportsRfSenseSelectiveOok().

Definition at line 259 of file common/rail_features.h

RAIL_FEAT_RFSENSE_SELECTIVE_OOK_MODE_SUPPORTED

```
#define RAIL_FEAT_RFSENSE_SELECTIVE_OOK_MODE_SUPPORTED
```

Value:

```
RAIL_SUPPORTS_RFSENSE_SELECTIVE_OOK
```

Backwards-compatible synonym of RAIL_SUPPORTS_RFSENSE_SELECTIVE_OOK.

Definition at line 262 of file common/rail_features.h

RAIL_SUPPORTS_EFF

```
#define RAIL_SUPPORTS_EFF
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports the Energy Friendly Front End Module (EFF).

See also runtime refinement RAIL_SupportsEff().

Definition at line 271 of file common/rail_features.h

RAIL_SUPPORTS_PROTOCOL_BLE

```
#define RAIL_SUPPORTS_PROTOCOL_BLE
```

Value:

```
RAIL_SUPPORTS_2P4GHZ_BAND
```

Boolean to indicate whether the selected chip supports BLE.

See also runtime refinement RAIL_SupportsProtocolBLE().

Definition at line 281 of file common/rail_features.h

RAIL_BLE_SUPPORTS_1MBPS_NON_VITERBI

```
#define RAIL_BLE_SUPPORTS_1MBPS_NON_VITERBI
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports BLE 1Mbps Non-Viterbi PHY.

See also runtime refinement RAIL_BLE_Supports1MbpsNonViterbi().

Definition at line 292 of file common/rail_features.h

RAIL_BLE_SUPPORTS_1MBPS_VITERBI

```
#define RAIL_BLE_SUPPORTS_1MBPS_VITERBI
```

Value:

```
RAIL_SUPPORTS_PROTOCOL_BLE
```

Boolean to indicate whether the selected chip supports BLE 1Mbps Viterbi PHY.

See also runtime refinement RAIL_BLE_Supports1MbpsViterbi().

Definition at line 299 of file common/rail_features.h

RAIL_BLE_SUPPORTS_1MBPS

```
#define RAIL_BLE_SUPPORTS_1MBPS
```

Value:

```
(RAIL_BLE_SUPPORTS_1MBPS_NON_VITERBI || RAIL_BLE_SUPPORTS_1MBPS_VITERBI)
```

Boolean to indicate whether the selected chip supports BLE 1Mbps operation.

See also runtime refinement RAIL_BLE_Supports1Mbps().

Definition at line 306 of file common/rail_features.h

RAIL_BLE_SUPPORTS_2MBPS_NON_VITERBI

```
#define RAIL_BLE_SUPPORTS_2MBPS_NON_VITERBI
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports BLE 2Mbps Non-Viterbi PHY.

See also runtime refinement RAIL_BLE_Supports2MbpsNonViterbi().

Definition at line 315 of file common/rail_features.h

RAIL_BLE_SUPPORTS_2MBPS_VITERBI

```
#define RAIL_BLE_SUPPORTS_2MBPS_VITERBI
```

Value:

```
RAIL_SUPPORTS_PROTOCOL_BLE
```

Boolean to indicate whether the selected chip supports BLE 2Mbps Viterbi PHY.

See also runtime refinement RAIL_BLE_Supports2MbpsViterbi().

Definition at line 322 of file common/rail_features.h

RAIL_BLE_SUPPORTS_2MBPS

```
#define RAIL_BLE_SUPPORTS_2MBPS
```

Value:

```
(RAIL_BLE_SUPPORTS_2MBPS_NON_VITERBI || RAIL_BLE_SUPPORTS_2MBPS_VITERBI)
```

Boolean to indicate whether the selected chip supports BLE 2Mbps operation.

See also runtime refinement RAIL_BLE_Supports2Mbps().

Definition at line 329 of file common/rail_features.h

RAIL_BLE_SUPPORTS_ANTENNA_SWITCHING

```
#define RAIL_BLE_SUPPORTS_ANTENNA_SWITCHING
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports BLE Antenna Switching needed for Angle-of-Arrival receives or Angle-of-Departure transmits.

See also runtime refinement RAIL_BLE_SupportsAntennaSwitching().

Definition at line 340 of file common/rail_features.h

RAIL_BLE_SUPPORTS_CODED_PHY

```
#define RAIL_BLE_SUPPORTS_CODED_PHY
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports the BLE Coded PHY used for Long-Range.

See also runtime refinement RAIL_BLE_SupportsCodedPhy().

Definition at line 353 of file common/rail_features.h

RAIL_FEAT_BLE_CODED

```
#define RAIL_FEAT_BLE_CODED
```

Value:

```
RAIL_BLE_SUPPORTS_CODED_PHY
```

Backwards-compatible synonym of RAIL_BLE_SUPPORTS_CODED_PHY.

Definition at line 356 of file common/rail_features.h

RAIL_BLE_SUPPORTS_SIMULSCAN_PHY

```
#define RAIL_BLE_SUPPORTS_SIMULSCAN_PHY
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports the BLE Simulscan PHY used for simultaneous BLE 1Mbps and Coded PHY reception.

See also runtime refinement RAIL_BLE_SupportsSimulscanPhy().

Definition at line 366 of file common/rail_features.h

RAIL_BLE_SUPPORTS_CTE

```
#define RAIL_BLE_SUPPORTS_CTE
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports BLE CTE (Constant Tone Extension) needed for Angle-of-Arrival/Departure transmits.

See also runtime refinement RAIL_BLE_SupportsCte().

Definition at line 378 of file common/rail_features.h

RAIL_BLE_SUPPORTS_QUUPPA

```
#define RAIL_BLE_SUPPORTS_QUUPPA
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports the Quuppa PHY.

See also runtime refinement RAIL_BLE_SupportsQuuppa().

Definition at line 387 of file common/rail_features.h

RAIL_BLE_SUPPORTS_IQ_SAMPLING

```
#define RAIL_BLE_SUPPORTS_IQ_SAMPLING
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports BLE IQ Sampling needed for Angle-of-Arrival/Departure receives.

See also runtime refinement RAIL_BLE_SupportsIQSampling().

Definition at line 397 of file common/rail_features.h

RAIL_BLE_SUPPORTS_AOX

```
#define RAIL_BLE_SUPPORTS_AOX
```

Value:

```
0 (RAIL_BLE_SUPPORTS_ANTENNA_SWITCHING \
0  || RAIL_BLE_SUPPORTS_IQ_SAMPLING   \
0  || RAIL_BLE_SUPPORTS_CTE)
```

Boolean to indicate whether the selected chip supports some BLE AOX features.

Definition at line 402 of file common/rail_features.h

RAIL_FEAT_BLE_AOX_SUPPORTED

```
#define RAIL_FEAT_BLE_AOX_SUPPORTED
```

Value:

```
RAIL_BLE_SUPPORTS_AOX
```

Backwards-compatible synonym of RAIL_BLE_SUPPORTS_AOX.

Definition at line 408 of file common/rail_features.h

RAIL_BLE_SUPPORTS_PHY_SWITCH_TO_RX

```
#define RAIL_BLE_SUPPORTS_PHY_SWITCH_TO_RX
```

Value:

```
RAIL_SUPPORTS_PROTOCOL_BLE
```

Boolean to indicate whether the selected chip supports BLE PHY switch to RX functionality, which is used to switch BLE PHYs at a specific time to receive auxiliary packets.

See also runtime refinement RAIL_BLE_SupportsPhySwitchToRx().

Definition at line 428 of file common/rail_features.h

RAIL_FEAT_BLE_PHY_SWITCH_TO_RX

```
#define RAIL_FEAT_BLE_PHY_SWITCH_TO_RX
```

Value:

```
RAIL_BLE_SUPPORTS_PHY_SWITCH_TO_RX
```

Backwards-compatible synonym of RAIL_BLE_SUPPORTS_PHY_SWITCH_TO_RX.

Definition at line 433 of file common/rail_features.h

RAIL_SUPPORTS_PROTOCOL_IEEE802154

```
#define RAIL_SUPPORTS_PROTOCOL_IEEE802154
```

Value:

```
1
```

Boolean to indicate whether the selected chip supports IEEE 802.15.4.

See also runtime refinement RAIL_SupportsProtocolIEEE802154().

Definition at line 442 of file common/rail_features.h

RAIL_IEEE802154_SUPPORTS_COEX_PHY

```
#define RAIL_IEEE802154_SUPPORTS_COEX_PHY
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports the 802.15.4 Wi-Fi Coexistence PHY.

See also runtime refinement RAIL_IEEE802154_SupportsCoexPhy().

Definition at line 453 of file common/rail_features.h

RAIL_FEAT_802154_COEX_PHY

```
#define RAIL_FEAT_802154_COEX_PHY
```

Value:

```
RAIL_IEEE802154_SUPPORTS_COEX_PHY
```

Backwards-compatible synonym of RAIL_IEEE802154_SUPPORTS_COEX_PHY.

Definition at line 456 of file common/rail_features.h

RAIL_SUPPORTS_IEEE802154_BAND_2P4

```
#define RAIL_SUPPORTS_IEEE802154_BAND_2P4
```

Value:

```
(RAIL_SUPPORTS_PROTOCOL_IEEE802154 && RAIL_SUPPORTS_2P4GHZ_BAND)
```

Boolean to indicate whether the selected chip supports the IEEE 802.15.4 2.4 GHz band variant.

See also runtime refinement RAIL_SupportsIEEE802154Band2P4().

Definition at line 462 of file common/rail_features.h

RAIL_IEEE802154_SUPPORTS_RX_CHANNEL_SWITCHING

```
#define RAIL_IEEE802154_SUPPORTS_RX_CHANNEL_SWITCHING
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports the IEEE 802.15.4 2.4 RX channel switching.

See also runtime refinement RAIL_IEEE802154_SupportsRxChannelSwitching().

Definition at line 474 of file common/rail_features.h

RAIL_IEEE802154_SUPPORTS_FEM_PHY

```
#define RAIL_IEEE802154_SUPPORTS_FEM_PHY
```

Value:

```
(RAIL_SUPPORTS_IEEE802154_BAND_2P4)
```

Boolean to indicate whether the selected chip supports a front end module.

See also runtime refinement RAIL_IEEE802154_SupportsFemPhy().

Definition at line 480 of file common/rail_features.h

RAIL_IEEE802154_SUPPORTS_E_SUBSET_GB868

```
#define RAIL_IEEE802154_SUPPORTS_E_SUBSET_GB868
```

Value:

```
RAIL_SUPPORTS_PROTOCOL_IEEE802154
```

Boolean to indicate whether the selected chip supports IEEE 802.15.4E-2012 feature subset needed for Zigbee R22 GB868.

See also runtime refinement RAIL_IEEE802154_SupportsESubsetGB868().

Definition at line 490 of file common/rail_features.h

RAIL_FEAT_IEEE802154_E_GB868_SUPPORTED

```
#define RAIL_FEAT_IEEE802154_E_GB868_SUPPORTED
```

Value:

```
RAIL_IEEE802154_SUPPORTS_E_SUBSET_GB868
```

Backwards-compatible synonym of RAIL_IEEE802154_SUPPORTS_E_SUBSET_GB868.

Definition at line 496 of file common/rail_features.h

RAIL_IEEE802154_SUPPORTS_E_ENHANCED_ACK

```
#define RAIL_IEEE802154_SUPPORTS_E_ENHANCED_ACK
```

Value:

```
RAIL_IEEE802154_SUPPORTS_E_SUBSET_GB868
```

Boolean to indicate whether the selected chip supports IEEE 802.15.4E-2012 Enhanced ACKing.

See also runtime refinement RAIL_IEEE802154_SupportsEEnhancedAck().

Definition at line 504 of file common/rail_features.h

RAIL_FEAT_IEEE802154_E_ENH_ACK_SUPPORTED

```
#define RAIL_FEAT_IEEE802154_E_ENH_ACK_SUPPORTED
```

Value:

```
RAIL_IEEE802154_SUPPORTS_E_ENHANCED_ACK
```

Backwards-compatible synonym of RAIL_IEEE802154_SUPPORTS_E_ENHANCED_ACK.

Definition at line 510 of file common/rail_features.h

RAIL_IEEE802154_SUPPORTS_E_MULTIPURPOSE_FRAMES

```
#define RAIL_IEEE802154_SUPPORTS_E_MULTIPURPOSE_FRAMES
```

Value:

```
RAIL_IEEE802154_SUPPORTS_E_SUBSET_GB868
```

Boolean to indicate whether the selected chip supports receiving IEEE 802.15.4E-2012 Multipurpose frames.

See also runtime refinement RAIL_IEEE802154_SupportsEMultipurposeFrames().

Definition at line 518 of file common/rail_features.h

RAIL_FEAT_IEEE802154_MULTIPURPOSE_FRAME_SUPPORTED

```
#define RAIL_FEAT_IEEE802154_MULTIPURPOSE_FRAME_SUPPORTED
```

Value:

```
RAIL_IEEE802154_SUPPORTS_E_MULTIPURPOSE_FRAMES
```

Backwards-compatible synonym of RAIL_IEEE802154_SUPPORTS_E_MULTIPURPOSE_FRAMES.

Definition at line 524 of file common/rail_features.h

RAIL_IEEE802154_SUPPORTS_G_SUBSET_GB868

```
#define RAIL_IEEE802154_SUPPORTS_G_SUBSET_GB868
```

Value:

```
((RAIL_SUPPORTS_PROTOCOL_IEEE802154 != 0) && (RAIL_SUPPORTS_SUBGHZ_BAND != 0))
```

Boolean to indicate whether the selected chip supports IEEE 802.15.4G-2012 feature subset needed for Zigbee R22 GB868.

See also runtime refinement RAIL_IEEE802154_SupportsGSubsetGB868().

Definition at line 532 of file common/rail_features.h

RAIL_FEAT_IEEE802154_G_GB868_SUPPORTED

```
#define RAIL_FEAT_IEEE802154_G_GB868_SUPPORTED
```

Value:

```
RAIL_IEEE802154_SUPPORTS_G_SUBSET_GB868
```

Backwards-compatible synonym of RAIL_IEEE802154_SUPPORTS_G_SUBSET_GB868.

Definition at line 539 of file common/rail_features.h

RAIL_IEEE802154_SUPPORTS_G_DYNFEC

```
#define RAIL_IEEE802154_SUPPORTS_G_DYNFEC
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports dynamic FEC See also runtime refinement RAIL_IEEE802154_SupportsGDynFec().

Definition at line 550 of file common/rail_features.h

RAIL_IEEE802154_SUPPORTS_G_MODESWITCH

```
#define RAIL_IEEE802154_SUPPORTS_G_MODESWITCH
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports Wi-SUN mode switching See also runtime refinement RAIL_IEEE802154_SupportsGModeSwitch().

Definition at line 562 of file common/rail_features.h

RAIL_IEEE802154_SUPPORTS_G_4BYTE_CRC

```
#define RAIL_IEEE802154_SUPPORTS_G_4BYTE_CRC
```

Value:

```
RAIL_IEEE802154_SUPPORTS_G_SUBSET_GB868
```

Boolean to indicate whether the selected chip supports IEEE 802.15.4G-2012 reception and transmission of frames with 4-byte CRC.

See also runtime refinement RAIL_IEEE802154_SupportsG4ByteCrc().

Definition at line 570 of file common/rail_features.h

RAIL_FEAT_IEEE802154_G_4BYTE_CRC_SUPPORTED

```
#define RAIL_FEAT_IEEE802154_G_4BYTE_CRC_SUPPORTED
```

Value:

```
RAIL_IEEE802154_SUPPORTS_G_4BYTE_CRC
```

Backwards-compatible synonym of RAIL_IEEE802154_SUPPORTS_G_4BYTE_CRC.

Definition at line 575 of file common/rail_features.h

RAIL_IEEE802154_SUPPORTS_G_UNWHITENED_RX

```
#define RAIL_IEEE802154_SUPPORTS_G_UNWHITENED_RX
```

Value:

```
RAIL_IEEE802154_SUPPORTS_G_SUBSET_GB868
```

Boolean to indicate whether the selected chip supports IEEE 802.15.4G-2012 reception of unwhitened frames.

See also runtime refinement RAIL_IEEE802154_SupportsGUnwhitenedRx().

Definition at line 583 of file common/rail_features.h

RAIL_FEAT_IEEE802154_G_UNWHITENED_RX_SUPPORTED

```
#define RAIL_FEAT_IEEE802154_G_UNWHITENED_RX_SUPPORTED
```

Value:

```
RAIL_IEEE802154_SUPPORTS_G_UNWHITENED_RX
```

Backwards-compatible synonym of RAIL_IEEE802154_SUPPORTS_G_UNWHITENED_RX.

Definition at line 589 of file common/rail_features.h

RAIL_IEEE802154_SUPPORTS_G_UNWHITENED_TX

```
#define RAIL_IEEE802154_SUPPORTS_G_UNWHITENED_TX
```

Value:

```
RAIL_IEEE802154_SUPPORTS_G_SUBSET_GB868
```

Boolean to indicate whether the selected chip supports IEEE 802.15.4G-2012 transmission of unwhitened frames.

See also runtime refinement RAIL_IEEE802154_SupportsGUnwhitenedTx().

Definition at line 597 of file common/rail_features.h

RAIL_FEAT_IEEE802154_G_UNWHITENED_TX_SUPPORTED

```
#define RAIL_FEAT_IEEE802154_G_UNWHITENED_TX_SUPPORTED
```

Value:

```
RAIL_IEEE802154_SUPPORTS_G_UNWHITENED_TX
```

Backwards-compatible synonym of RAIL_IEEE802154_SUPPORTS_G_UNWHITENED_TX.

Definition at line 603 of file common/rail_features.h

RAIL_IEEE802154_SUPPORTS_CANCEL_FRAME_PENDING_LOOKUP

```
#define RAIL_IEEE802154_SUPPORTS_CANCEL_FRAME_PENDING_LOOKUP
```

Value:

```
RAIL_SUPPORTS_PROTOCOL_IEEE802154
```

Boolean to indicate whether the selected chip supports canceling the frame-pending lookup event RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND when the radio transitions to a state that renders the the reporting of this event moot (i.e., too late for the stack to influence the outgoing ACK).

See also runtime refinement RAIL_IEEE802154_SupportsCancelFramePendingLookup().

Definition at line 615 of file common/rail_features.h

RAIL_FEAT_IEEE802154_CANCEL_FP_LOOKUP_SUPPORTED

```
#define RAIL_FEAT_IEEE802154_CANCEL_FP_LOOKUP_SUPPORTED
```

Value:

```
RAIL_IEEE802154_SUPPORTS_CANCEL_FRAME_PENDING_LOOKUP
```

Backwards-compatible synonym of RAIL_IEEE802154_SUPPORTS_CANCEL_FRAME_PENDING_LOOKUP.

Definition at line 621 of file common/rail_features.h

RAIL_IEEE802154_SUPPORTS_EARLY_FRAME_PENDING_LOOKUP

```
#define RAIL_IEEE802154_SUPPORTS_EARLY_FRAME_PENDING_LOOKUP
```

Value:

```
RAIL_SUPPORTS_PROTOCOL_IEEE802154
```

Boolean to indicate whether the selected chip supports early triggering of the frame-pending lookup event RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND just after MAC address fields have been received.

See also runtime refinement RAIL_IEEE802154_SupportsEarlyFramePendingLookup().

Definition at line 631 of file common/rail_features.h

RAIL_FEAT_IEEE802154_EARLY_FP_LOOKUP_SUPPORTED

```
#define RAIL_FEAT_IEEE802154_EARLY_FP_LOOKUP_SUPPORTED
```

Value:

```
RAIL_IEEE802154_SUPPORTS_EARLY_FRAME_PENDING_LOOKUP
```

Backwards-compatible synonym of RAIL_IEEE802154_SUPPORTS_EARLY_FRAME_PENDING_LOOKUP.

Definition at line 637 of file common/rail_features.h

RAIL_IEEE802154_SUPPORTS_DUAL_PA_CONFIG

```
#define RAIL_IEEE802154_SUPPORTS_DUAL_PA_CONFIG
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports dual PA configs for mode switch or concurrent mode.

See also runtime refinement RAIL_IEEE802154_SupportsDualPaConfig().

Definition at line 646 of file common/rail_features.h

RAIL_SUPPORTS_DBM_POWERSETTING_MAPPING_TABLE

```
#define RAIL_SUPPORTS_DBM_POWERSETTING_MAPPING_TABLE
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports the pa power setting table.

Definition at line 653 of file common/rail_features.h

RAIL_IEEE802154_SUPPORTS_CUSTOM1_PHY

```
#define RAIL_IEEE802154_SUPPORTS_CUSTOM1_PHY
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports IEEE 802.15.4 PHY with custom settings.

Definition at line 674 of file common/rail_features.h

RAIL_SUPPORTS_PROTOCOL_WI_SUN

```
#define RAIL_SUPPORTS_PROTOCOL_WI_SUN
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports Wi-SUN See also runtime refinement RAIL_SupportsProtocolWiSUN().

Definition at line 687 of file common/rail_features.h

RAIL_WMBUS_SUPPORTS_SIMULTANEOUS_T_C_RX

```
#define RAIL_WMBUS_SUPPORTS_SIMULTANEOUS_T_C_RX
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports WMBUS simultaneous M2O RX of T and C modes set by RAIL_WMBUS_Config().

See also runtime refinement RAIL_WMBUS_SupportsSimultaneousTCRx().

Definition at line 699 of file common/rail_features.h

RAIL_SUPPORTS_PROTOCOL_ZWAVE

```
#define RAIL_SUPPORTS_PROTOCOL_ZWAVE
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports Z-Wave.

See also runtime refinement RAIL_SupportsProtocolZWave().

Definition at line 713 of file common/rail_features.h

RAIL_FEAT_ZWAVE_SUPPORTED

```
#define RAIL_FEAT_ZWAVE_SUPPORTED
```

Value:

```
RAIL_SUPPORTS_PROTOCOL_ZWAVE
```

Backwards-compatible synonym of RAIL_SUPPORTS_PROTOCOL_ZWAVE.

Definition at line 716 of file common/rail_features.h

RAIL_ZWAVE_SUPPORTS_ED_PHY

```
#define RAIL_ZWAVE_SUPPORTS_ED_PHY
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports energy detect PHY.

See also runtime refinement RAIL_ZWAVE_SupportsEnergyDetectPhy().

Definition at line 723 of file common/rail_features.h

RAIL_ZWAVE_SUPPORTS_CONC_PHY

```
#define RAIL_ZWAVE_SUPPORTS_CONC_PHY
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports concurrent PHY.

See also runtime refinement RAIL_ZWAVE_SupportsConcPhy().

Definition at line 731 of file common/rail_features.h

RAIL_SUPPORTS_SQ_PHY

```
#define RAIL_SUPPORTS_SQ_PHY
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports SQ-based PHY.

See also runtime refinement RAIL_SupportsSQPhy().

Definition at line 740 of file common/rail_features.h

RAIL_ZWAVE_SUPPORTS_REGION_PTI

```
#define RAIL_ZWAVE_SUPPORTS_REGION_PTI
```

Value:

```
RAIL_SUPPORTS_PROTOCOL_ZWAVE
```

Boolean to indicate whether the code supports Z-Wave region information in PTI and newer RAIL_ZWAVE_RegionConfig_t structure See also runtime refinement RAIL_ZWAVE_SupportsRegionPti().

Definition at line 748 of file common/rail_features.h

RAIL_FEAT_ZWAVE_REGION_PTI

```
#define RAIL_FEAT_ZWAVE_REGION_PTI
```

Value:

```
RAIL_ZWAVE_SUPPORTS_REGION_PTI
```

Backwards-compatible synonym of RAIL_ZWAVE_SUPPORTS_REGION_PTI.

Definition at line 753 of file common/rail_features.h

RAIL_SUPPORTS_RX_RAW_DATA

```
#define RAIL_SUPPORTS_RX_RAW_DATA
```

Value:

```
1
```

Boolean to indicate whether the selected chip supports raw RX data sources other than RAIL_RxDataSource_t::RX_PACKET_DATA.

See also runtime refinement RAIL_SupportsRxRawData().

Definition at line 759 of file common/rail_features.h

RAIL_SUPPORTS_DIRECT_MODE

```
#define RAIL_SUPPORTS_DIRECT_MODE
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports direct mode.

See also runtime refinement RAIL_SupportsDirectMode().

Definition at line 772 of file common/rail_features.h

RAIL_SUPPORTS_RX_DIRECT_MODE_DATA_TO_FIFO

```
#define RAIL_SUPPORTS_RX_DIRECT_MODE_DATA_TO_FIFO
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports RX direct mode data to FIFO.

See also runtime refinement RAIL_SupportsRxDirectModeDataToFifo().

Definition at line 781 of file common/rail_features.h

RAIL_SUPPORTS_MFM

```
#define RAIL_SUPPORTS_MFM
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports MFM protocol.

See also runtime refinement RAIL_SupportsMfm().

Definition at line 790 of file common/rail_features.h

RAIL_IEEE802154_SUPPORTS_SIGNAL_IDENTIFIER

```
#define RAIL_IEEE802154_SUPPORTS_SIGNAL_IDENTIFIER
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports 802.15.4 signal detection.

Definition at line 803 of file common/rail_features.h

RAIL_BLE_SUPPORTS_SIGNAL_IDENTIFIER

```
#define RAIL_BLE_SUPPORTS_SIGNAL_IDENTIFIER
```

Value:

```
0
```

Boolean to indicate whether the selected chip supports BLE signal detection.

Definition at line 806 of file common/rail_features.h

RAIL_SUPPORTS_RSSI_DETECT_THRESHOLD

```
#define RAIL_SUPPORTS_RSSI_DETECT_THRESHOLD
```

Value:

```
(0U)
```

Boolean to indicate whether the selected chip supports configurable RSSI threshold set by RAIL_SetRssiDetectThreshold().

See also runtime refinement RAIL_SupportsRssiDetectThreshold().

Definition at line 816 of file common/rail_features.h

RAIL_SUPPORTS_THERMAL_PROTECTION

```
#define RAIL_SUPPORTS_THERMAL_PROTECTION
```

Value:

```
(0U)
```

Boolean to indicate whether the selected chip supports thermal protection set by RAIL_ConfigThermalProtection().

See also runtime refinement RAIL_SupportsThermalProtection().

Definition at line 825 of file common/rail_features.h

RAIL_SUPPORTS_FAST_RX2RX

```
#define RAIL_SUPPORTS_FAST_RX2RX
```

Value:

```
(0U)
```

Boolean to indicate whether the selected chip supports fast RX2RX enabled by RAIL_RX_OPTION_FAST_RX2RX.

See also runtime refinement RAIL_SupportsFastRx2Rx().

Definition at line 834 of file common/rail_features.h

RAIL_SUPPORTS_COLLISION_DETECTION

```
#define RAIL_SUPPORTS_COLLISION_DETECTION
```

Value:

```
(0U)
```

Boolean to indicate whether the selected chip supports collision detection enabled by RAIL_RX_OPTION_ENABLE_COLLISION_DETECTION See also runtime refinement RAIL_SupportsCollisionDetection().

Definition at line 843 of file common/rail_features.h

RAIL_SUPPORTS_PROTOCOL_SIDEWALK

```
#define RAIL_SUPPORTS_PROTOCOL_SIDEWALK
```

Value:

```
(0U)
```

Boolean to indicate whether the selected chip supports Sidewalk protocol.

See also runtime refinement RAIL_SupportsProtocolSidewalk().

Definition at line 852 of file common/rail_features.h

General

General

Basic APIs to set up and interact with the RAIL library.

Modules

RAIL_Version_t

RAILSched_Config_t

RAIL_Config_t

EFR32xG1

Enumerations

enum RAIL_Status_t {

RAIL_STATUS_NO_ERROR
RAIL_STATUS_INVALID_PARAMETER
RAIL_STATUS_INVALID_STATE
RAIL_STATUS_INVALID_CALL
RAIL_STATUS_SUSPENDED
RAIL_STATUS_SCHED_ERROR

}

A status returned by many RAIL API calls indicating their success or failure.

Typedefs

typedef void * RAIL_Handle_t

A generic handle to a particular radio (e.g.

typedef void(* RAIL_InitCompleteCallbackPtr_t)(RAIL_Handle_t railHandle)

A pointer to init complete callback function.

typedef uint8_t RAIL_StateBuffer_t[1]

Provided for backwards compatibility.

Functions

void RAIL_GetVersion(RAIL_Version_t *version, bool verbose)

Get the version information for the compiled RAIL library.

RAIL_Status_t RAIL_AddStateBuffer3(RAIL_Handle_t genericRailHandle)

Add a 3rd multiprotocol internal state buffer for use by RAIL_Init().

RAIL_Status_t RAIL_AddStateBuffer4(RAIL_Handle_t genericRailHandle)

Add a 4th multiprotocol internal state buffer for use by RAIL_Init().

RAIL_Status_t RAIL_UseDma(uint8_t channel)

Allocate a DMA channel for RAIL to work with.

RAIL_Handle_t RAIL_Init(RAIL_Config_t *railCfg, RAIL_InitCompleteCallbackPtr_t cb)
Initialize RAIL.

bool RAIL_IsInitialized(void)
Get RAIL initialization status.

uint16_t RAIL_GetRadioEntropy(RAIL_Handle_t railHandle, uint8_t *buffer, uint16_t bytes)
Collect entropy from the radio if available.

Macros

#define RAIL_EFR32_HANDLE ((RAIL_Handle_t)0xFFFFFFFFUL)
A placeholder for a chip-specific RAIL handle.

#define RAIL_DMA_INVALID (0xFFU)
A value to signal that RAIL should not use DMA.

Enumeration Documentation

RAIL_Status_t

> RAIL_Status_t

A status returned by many RAIL API calls indicating their success or failure.

Enumerator	
RAIL_STATUS_NO_ERROR	RAIL function reports no error.
RAIL_STATUS_INVALID_PARAMETER	Call to RAIL function threw an error because of an invalid parameter.
RAIL_STATUS_INVALID_STATE	Call to RAIL function threw an error because it was called during an invalid radio state.
RAIL_STATUS_INVALID_CALL	RAIL function is called in an invalid order.
RAIL_STATUS_SUSPENDED	RAIL function did not finish in the allotted time.
RAIL_STATUS_SCHED_ERROR	RAIL function could not be scheduled by the Radio scheduler.

Definition at line 119 of file common/rail_types.h

Typedef Documentation

RAIL_Handle_t

> RAIL_Handle_t

A generic handle to a particular radio (e.g.

RAIL_EFR32_HANDLE), or a real handle of a RAIL instance, as returned from RAIL_Init().

Generic handles should be used for certain RAIL APIs that are called prior to RAIL initialization. However, once RAIL has been initialized, the real handle returned by RAIL_Init() should be used instead.

Definition at line 102 of file common/rail_types.h

RAIL_InitCompleteCallbackPtr_t

> typedef void(* RAIL_InitCompleteCallbackPtr_t) (RAIL_Handle_t railHandle))(RAIL_Handle_t railHandle)

A pointer to init complete callback function.

Parameters

[in]	railHandle	The initialized RAIL instance handle.

Definition at line 150 of file common/rail_types.h

RAIL_StateBuffer_t

```
RAIL_StateBuffer_t [1]
```

Provided for backwards compatibility.

Definition at line 167 of file common/rail_types.h

Function Documentation

RAIL_GetVersion

```
void RAIL_GetVersion (RAIL_Version_t *version, bool verbose)
```

Get the version information for the compiled RAIL library.

Parameters

[out]	version	A pointer to RAIL_Version_t structure to populate with version information.
[in]	verbose	Populate RAIL_Version_t struct with verbose information.

The version information contains a major version number, a minor version number, and a rev (revision) number.

Definition at line 82 of file common/rail.h

RAIL_AddStateBuffer3

```
RAIL_Status_t RAIL_AddStateBuffer3 (RAIL_Handle_t genericRailHandle)
```

Add a 3rd multiprotocol internal state buffer for use by RAIL_Init().

Parameters

[in]	genericRailHandle	A generic RAIL instance handle.

Returns

- Status code indicating success of the function call. An error is returned if the 3rd state buffer was previously added or this isn't the RAIL multiprotocol library.

Definition at line 151 of file common/rail.h

RAIL_AddStateBuffer4

```
RAIL_Status_t RAIL_AddStateBuffer4 (RAIL_Handle_t genericRailHandle)
```

Add a 4th multiprotocol internal state buffer for use by RAIL_Init().

Parameters

[in]	genericRailHandle	A generic RAIL instance handle.

Returns

- Status code indicating success of the function call. An error is returned if the 4th state buffer was previously added. or this isn't the RAIL multiprotocol library.

Definition at line 161 of file common/rail.h

RAIL_UseDma

> RAIL_Status_t RAIL_UseDma (uint8_t channel)

Allocate a DMA channel for RAIL to work with.

Parameters

[in]	channel	The DMA channel to use when copying memory. If a value of RAIL_DMA_INVALID is passed, RAIL will stop using any DMA channel.

Returns

- Status code indicating success of the function call.

To use this API, the application must initialize the DMA engine on the chip and allocate a DMA channel. This channel will be used periodically to copy memory more efficiently. Call this function before RAIL_Init to have the most benefit. If the application needs to take back control of the DMA channel that RAIL is using, this API may be called with a channel of RAIL_DMA_INVALID to tell RAIL to stop using DMA.

Definition at line 177 of file common/rail.h

RAIL_Init

> RAIL_Handle_t RAIL_Init (RAIL_Config_t *railCfg, RAIL_InitCompleteCallbackPtr_t cb)

Initialize RAIL.

Parameters

[inout]	railCfg	The configuration and state structure for setting up the library, which contains memory and other options that RAIL needs. This structure must be allocated in application global read-write memory. RAIL may modify fields within or referenced by this structure during its operation.
[in]	cb	A callback that notifies the application when the radio is finished initializing and is ready for further configuration. This callback is useful for potential transceiver products that require a power up sequence before further configuration is available. After the callback fires, the radio is ready for additional configuration before transmit and receive operations.

Returns

- Handle for initialized rail instance or NULL if an invalid value was passed in the railCfg.

Note

- Call this function only once per protocol. If called again, it will do nothing and return NULL.

Definition at line 318 of file common/rail.h

RAIL_IsInitialized

```
bool RAIL_IsInitialized (void)
```

Get RAIL initialization status.

Parameters

N/A		

Returns

- True if the radio has finished initializing and false otherwise.

RAIL APIs, e.g., RAIL_GetTime(), which work only if RAIL_Init() has been called, can use RAIL_IsInitialized() to determine whether RAIL has been initialized or not.

Definition at line 330 of file common/rail.h

RAIL_GetRadioEntropy

```
uint16_t RAIL_GetRadioEntropy (RAIL_Handle_t railHandle, uint8_t *buffer, uint16_t bytes)
```

Collect entropy from the radio if available.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	buffer	The buffer to write the collected entropy.
[in]	bytes	The number of bytes to fill in the input buffer.

Returns

- Returns the number of bytes of entropy collected. For chips that don't support entropy collection, the function returns 0. Values less than the requested amount may also be returned on platforms that use entropy pools to collect random data periodically.

Attempts to fill the provided buffer with the requested number of bytes of entropy. If the requested number of bytes can't be provided, as many bytes as possible will be filled and returned. For chips that do not support this function, 0 bytes are always returned. For information about the specific mechanism for gathering entropy, see documentation for the chip family.

Definition at line 350 of file common/rail.h

Macro Definition Documentation

RAIL_EFR32_HANDLE

```
#define RAIL_EFR32_HANDLE
```

Value:

```
((RAIL_Handle_t)0xFFFFFFFFUL)
```

A placeholder for a chip-specific RAIL handle.

Using NULL as a RAIL handle is not recommended. As a result, another value that can't be de-referenced is used.

This generic handle can and should be used for RAIL APIs that are called prior to RAIL initialization.

Definition at line 112 of file common/rail_types.h

RAIL_DMA_INVALID

```
#define RAIL_DMA_INVALID
```

Value:

```
(0xFFU)
```

A value to signal that RAIL should not use DMA.

Definition at line 153 of file common/rail_types.h

RAIL_Version_t

Contains RAIL Library Version Information.

It is filled in by RAIL_GetVersion().

Public Attributes

uint32_t	hash Git hash.
uint8_t	major Major number
uint8_t	minor Minor number
uint8_t	rev Revision number.
uint8_t	build Build number.
uint8_t	flags Build flags.
bool	multiprotocol Boolean to indicate whether this is a multiprotocol library or not.

Public Attribute Documentation

hash

```
uint32_t RAIL_Version_t::hash
```

Git hash.

Definition at line 83 of file common/rail_types.h

major

```
uint8_t RAIL_Version_t::major
```

Major number

Definition at line 84 of file common/rail_types.h

minor

```
uint8_t RAIL_Version_t::minor
```

Minor number

Definition at line 85 of file common/rail_types.h

rev

```
uint8_t RAIL_Version_t::rev
```

Revision number.

Definition at line 86 of file common/rail_types.h

build

```
uint8_t RAIL_Version_t::build
```

Build number.

Definition at line 87 of file common/rail_types.h

flags

```
uint8_t RAIL_Version_t::flags
```

Build flags.

Definition at line 88 of file common/rail_types.h

multiprotocol

```
bool RAIL_Version_t::multiprotocol
```

Boolean to indicate whether this is a multiprotocol library or not.

Definition at line 90 of file common/rail_types.h

RAILSched_Config_t

Provided for backwards compatibility.

Public Attributes

uint8_t	buffer Dummy buffer no longer used.

Public Attribute Documentation

buffer

```
uint8_t RAILSched_Config_t::buffer[1]
```

Dummy buffer no longer used.

Definition at line 160 of file common/rail_types.h

RAIL_Config_t

RAIL configuration structure.

Public Attributes

void(*	eventsCallback A pointer to a function, which is called whenever a RAIL event occurs.
void *	protocol Provided for backwards compatibility.
RAILSched_Config_t *	scheduler Provided for backwards compatibility.
RAIL_StateBuffer_t	buffer Provided for backwards compatibility.

Public Attribute Documentation

eventsCallback

```
void(* RAIL_Config_t::eventsCallback) (RAIL_Handle_t railHandle, RAIL_Events_t events)
```

A pointer to a function, which is called whenever a RAIL event occurs.

See the RAIL_Events_t documentation for the list of RAIL events.

Definition at line 1656 of file common/rail_types.h

protocol

```
void* RAIL_Config_t::protocol
```

Provided for backwards compatibility.

Ignored.

Definition at line 1660 of file common/rail_types.h

scheduler

```
RAILSched_Config_t* RAIL_Config_t::scheduler
```

Provided for backwards compatibility.

Ignored.

Definition at line 1664 of file common/rail_types.h

buffer

```
RAIL_StateBuffer_t RAIL_Config_t::buffer
```

Provided for backwards compatibility.

Ignored.

Definition at line 1668 of file common/rail_types.h

EFR32xG1

EFR32xG1

EFR32xG1-specific initialization data types.

Multiprotocol

Multiprotocol

Multiprotocol scheduler APIs to support multiple time-sliced PHYs.

Modules

RAIL_SchedulerInfo_t

EFR32

Enumerations

enum RAIL_SchedulerStatus_t {

RAIL_SCHEDULER_STATUS_NO_ERROR = (0U << 0)
RAIL_SCHEDULER_STATUS_UNSUPPORTED = (1U << 0)
RAIL_SCHEDULER_STATUS_EVENT_INTERRUPTED = (2U << 0)
RAIL_SCHEDULER_STATUS_SCHEDULE_FAIL = (3U << 0)
RAIL_SCHEDULER_STATUS_TASK_FAIL = (4U << 0)
RAIL_SCHEDULER_STATUS_INTERNAL_ERROR = (5U << 0)
RAIL_SCHEDULER_TASK_EMPTY = (0U << 4)
RAIL_SCHEDULER_TASK_SCHEDULED_RX = (1U << 4)
RAIL_SCHEDULER_TASK_SCHEDULED_TX = (2U << 4)
RAIL_SCHEDULER_TASK_SINGLE_TX = (3U << 4)
RAIL_SCHEDULER_TASK_SINGLE_CCA_CSMA_TX = (4U << 4)
RAIL_SCHEDULER_TASK_SINGLE_CCA_LBT_TX = (5U << 4)
RAIL_SCHEDULER_TASK_SCHEDULED_CCA_CSMA_TX = (6U << 4)
RAIL_SCHEDULER_TASK_SCHEDULED_CCA_LBT_TX = (7U << 4)
RAIL_SCHEDULER_TASK_TX_STREAM = (8U << 4)
RAIL_SCHEDULER_TASK_AVERAGE_RSSI = (9U << 4)
RAIL_SCHEDULER_STATUS_SCHEDULED_TX_FAIL = (RAIL_SCHEDULER_TASK_SCHEDULED_TX |
RAIL_SCHEDULER_STATUS_TASK_FAIL)
RAIL_SCHEDULER_STATUS_SINGLE_TX_FAIL = (RAIL_SCHEDULER_TASK_SINGLE_TX |
RAIL_SCHEDULER_STATUS_TASK_FAIL)
RAIL_SCHEDULER_STATUS_CCA_CSMA_TX_FAIL = (RAIL_SCHEDULER_TASK_SINGLE_CCA_CSMA_TX |
RAIL_SCHEDULER_STATUS_TASK_FAIL)
RAIL_SCHEDULER_STATUS_CCA_LBT_TX_FAIL = (RAIL_SCHEDULER_TASK_SINGLE_CCA_LBT_TX |
RAIL_SCHEDULER_STATUS_TASK_FAIL)
RAIL_SCHEDULER_STATUS_SCHEDULED_RX_FAIL = (RAIL_SCHEDULER_TASK_SCHEDULED_RX |
RAIL_SCHEDULER_STATUS_TASK_FAIL)
RAIL_SCHEDULER_STATUS_TX_STREAM_FAIL = (RAIL_SCHEDULER_TASK_TX_STREAM |
RAIL_SCHEDULER_STATUS_TASK_FAIL)
RAIL_SCHEDULER_STATUS_AVERAGE_RSSI_FAIL = (RAIL_SCHEDULER_TASK_AVERAGE_RSSI |
RAIL_SCHEDULER_STATUS_TASK_FAIL)
RAIL_SCHEDULER_SCHEDULED_RX_INTERNAL_ERROR = (RAIL_SCHEDULER_TASK_SCHEDULED_RX |
RAIL_SCHEDULER_STATUS_INTERNAL_ERROR)
RAIL_SCHEDULER_SCHEDULED_RX_SCHEDULING_ERROR = (RAIL_SCHEDULER_TASK_SCHEDULED_RX |
RAIL_SCHEDULER_STATUS_SCHEDULE_FAIL)
RAIL_SCHEDULER_SCHEDULED_RX_INTERRUPTED = (RAIL_SCHEDULER_TASK_SCHEDULED_RX |
RAIL_SCHEDULER_STATUS_EVENT_INTERRUPTED)
RAIL_SCHEDULER_SCHEDULED_TX_INTERNAL_ERROR = (RAIL_SCHEDULER_TASK_SCHEDULED_TX |
RAIL_SCHEDULER_STATUS_INTERNAL_ERROR)
RAIL_SCHEDULER_SCHEDULED_TX_SCHEDULING_ERROR = (RAIL_SCHEDULER_TASK_SCHEDULED_TX |
RAIL_SCHEDULER_STATUS_SCHEDULE_FAIL)
RAIL_SCHEDULER_SCHEDULED_TX_INTERRUPTED = (RAIL_SCHEDULER_TASK_SCHEDULED_TX |
RAIL_SCHEDULER_STATUS_EVENT_INTERRUPTED)
RAIL_SCHEDULER_SINGLE_TX_INTERNAL_ERROR = (RAIL_SCHEDULER_TASK_SINGLE_TX |
RAIL_SCHEDULER_STATUS_INTERNAL_ERROR)
RAIL_SCHEDULER_SINGLE_TX_SCHEDULING_ERROR = (RAIL_SCHEDULER_TASK_SINGLE_TX |
RAIL_SCHEDULER_STATUS_SCHEDULE_FAIL)
RAIL_SCHEDULER_SINGLE_TX_INTERRUPTED = (RAIL_SCHEDULER_TASK_SINGLE_TX |
RAIL_SCHEDULER_STATUS_EVENT_INTERRUPTED)
RAIL_SCHEDULER_SINGLE_CCA_CSMA_TX_INTERNAL_ERROR =
(RAIL_SCHEDULER_TASK_SINGLE_CCA_CSMA_TX | RAIL_SCHEDULER_STATUS_INTERNAL_ERROR)
RAIL_SCHEDULER_SINGLE_CCA_CSMA_TX_SCHEDULING_ERROR =
(RAIL_SCHEDULER_TASK_SINGLE_CCA_CSMA_TX | RAIL_SCHEDULER_STATUS_SCHEDULE_FAIL)
RAIL_SCHEDULER_SINGLE_CCA_CSMA_TX_INTERRUPTED =
(RAIL_SCHEDULER_TASK_SINGLE_CCA_CSMA_TX | RAIL_SCHEDULER_STATUS_EVENT_INTERRUPTED)
RAIL_SCHEDULER_SINGLE_CCA_LBT_TX_INTERNAL_ERROR =
(RAIL_SCHEDULER_TASK_SINGLE_CCA_LBT_TX | RAIL_SCHEDULER_STATUS_INTERNAL_ERROR)
RAIL_SCHEDULER_SINGLE_CCA_LBT_TX_SCHEDULING_ERROR =
(RAIL_SCHEDULER_TASK_SINGLE_CCA_LBT_TX | RAIL_SCHEDULER_STATUS_SCHEDULE_FAIL)
RAIL_SCHEDULER_SINGLE_CCA_LBT_TX_INTERRUPTED = (RAIL_SCHEDULER_TASK_SINGLE_CCA_LBT_TX |
RAIL_SCHEDULER_STATUS_EVENT_INTERRUPTED)
RAIL_SCHEDULER_SCHEDULED_CCA_CSMA_TX_INTERNAL_ERROR =
(RAIL_SCHEDULER_TASK_SCHEDULED_CCA_CSMA_TX | RAIL_SCHEDULER_STATUS_INTERNAL_ERROR)
RAIL_SCHEDULER_SCHEDULED_CCA_CSMA_TX_FAIL =
(RAIL_SCHEDULER_TASK_SCHEDULED_CCA_CSMA_TX | RAIL_SCHEDULER_STATUS_TASK_FAIL)
RAIL_SCHEDULER_SCHEDULED_CCA_CSMA_TX_SCHEDULING_ERROR =
(RAIL_SCHEDULER_TASK_SCHEDULED_CCA_CSMA_TX | RAIL_SCHEDULER_STATUS_SCHEDULE_FAIL)
RAIL_SCHEDULER_SCHEDULED_CCA_CSMA_TX_INTERRUPTED =
(RAIL_SCHEDULER_TASK_SCHEDULED_CCA_CSMA_TX |
RAIL_SCHEDULER_STATUS_EVENT_INTERRUPTED)

RAIL_SCHEDULER_SCHEDULED_CCA_LBT_TX_INTERNAL_ERROR =
(RAIL_SCHEDULER_TASK_SCHEDULED_CCA_LBT_TX |
RAIL_SCHEDULER_STATUS_INTERNAL_ERROR)
RAIL_SCHEDULER_SCHEDULED_CCA_LBT_TX_FAIL =
(RAIL_SCHEDULER_TASK_SCHEDULED_CCA_LBT_TX |
RAIL_SCHEDULER_STATUS_TASK_FAIL)
RAIL_SCHEDULER_SCHEDULED_CCA_LBT_TX_SCHEDULING_ERROR
= (RAIL_SCHEDULER_TASK_SCHEDULED_CCA_LBT_TX |
RAIL_SCHEDULER_STATUS_SCHEDULE_FAIL)
RAIL_SCHEDULER_SCHEDULED_CCA_LBT_TX_INTERRUPTED =
(RAIL_SCHEDULER_TASK_SCHEDULED_CCA_LBT_TX |
RAIL_SCHEDULER_STATUS_EVENT_INTERRUPTED)
RAIL_SCHEDULER_TX_STREAM_INTERNAL_ERROR =
(RAIL_SCHEDULER_TASK_TX_STREAM |
RAIL_SCHEDULER_STATUS_INTERNAL_ERROR)
RAIL_SCHEDULER_TX_STREAM_SCHEDULING_ERROR =
(RAIL_SCHEDULER_TASK_TX_STREAM |
RAIL_SCHEDULER_STATUS_SCHEDULE_FAIL)
RAIL_SCHEDULER_TX_STREAM_INTERRUPTED =
(RAIL_SCHEDULER_TASK_TX_STREAM |
RAIL_SCHEDULER_STATUS_EVENT_INTERRUPTED)
RAIL_SCHEDULER_AVERAGE_RSSI_INTERNAL_ERROR =
(RAIL_SCHEDULER_TASK_AVERAGE_RSSI |
RAIL_SCHEDULER_STATUS_INTERNAL_ERROR)
RAIL_SCHEDULER_AVERAGE_RSSI_SCHEDULING_ERROR =
(RAIL_SCHEDULER_TASK_AVERAGE_RSSI |
RAIL_SCHEDULER_STATUS_SCHEDULE_FAIL)
RAIL_SCHEDULER_AVERAGE_RSSI_INTERRUPTED =
(RAIL_SCHEDULER_TASK_AVERAGE_RSSI |
RAIL_SCHEDULER_STATUS_EVENT_INTERRUPTED)

}

Multiprotocol scheduler status returned by RAIL_GetSchedulerStatus().

enum RAIL_TaskType_t {

RAIL_TASK_TYPE_START_RX
RAIL_TASK_TYPE_OTHER

}

Multiprotocol radio operation task types, used with RAIL_SetTaskPriority.

Functions

void RAIL_YieldRadio(RAIL_Handle_t railHandle)

Yield the radio to other configurations.

RAIL_SchedulerStatus_t RAIL_GetSchedulerStatus(RAIL_Handle_t railHandle)

Get the status of the RAIL scheduler.

RAIL_Status_t RAIL_GetSchedulerStatusAlt(RAIL_Handle_t railHandle, RAIL_SchedulerStatus_t *pSchedulerStatus, RAIL_Status_t *pRailStatus)

Get the status of the RAIL scheduler, specific to the radio operation, along with RAIL_Status_t returned by RAIL API invoked by the RAIL scheduler.

RAIL_Status_t RAIL_SetTaskPriority(RAIL_Handle_t railHandle, uint8_t priority, RAIL_TaskType_t taskType)

Change the priority of a specified task type in multiprotocol.

RAIL_Time_t RAIL_GetTransitionTime(void)

Get time needed to switch between protocols.

void RAIL_SetTransitionTime(RAIL_Time_t transitionTime)

Set time needed to switch between protocols.

Macros

#define RAIL_SCHEDULER_STATUS_MASK 0x0FU
Radio Scheduler Status mask.

#define RAIL_SCHEDULER_STATUS_SHIFT 0
Radio Scheduler Status shift.

#define RAIL_SCHEDULER_TASK_MASK 0xF0U
Radio Scheduler Task mask.

#define RAIL_SCHEDULER_TASK_SHIFT 4
Radio Scheduler Task shift.

Enumeration Documentation

RAIL_SchedulerStatus_t

```
RAIL_SchedulerStatus_t
```

Multiprotocol scheduler status returned by RAIL_GetSchedulerStatus().

Multiprotocol scheduler status is a combination of the upper 4 bits which constitute the type of scheduler task and the lower 4 bits which constitute the type of scheduler error.

	Enumerator
RAIL_SCHEDULER_STATUS_NO_ERROR	Lower 4 bits of uint8_t capture the different Radio Scheduler errors.
RAIL_SCHEDULER_STATUS_UNSUPPORTED	The scheduler is disabled or the requested scheduler operation is unsupported.
RAIL_SCHEDULER_STATUS_EVENT_INTERRUPTED	The scheduled task was started but was interrupted by a higher-priority event before it could be completed.
RAIL_SCHEDULER_STATUS_SCHEDULE_FAIL	Scheduled task could not be scheduled given its priority and the other tasks running on the system.
RAIL_SCHEDULER_STATUS_TASK_FAIL	Calling the RAIL API associated with the Radio scheduler task returned an error code.
RAIL_SCHEDULER_STATUS_INTERNAL_ERROR	An internal error occurred in scheduler data structures, which should not happen and indicates a problem.
RAIL_SCHEDULER_TASK_EMPTY	Upper 4 bits of uint8_t capture the different Radio Scheduler tasks.
RAIL_SCHEDULER_TASK_SCHEDULED_RX	Radio scheduler calls RAIL_ScheduleRx().
RAIL_SCHEDULER_TASK_SCHEDULED_TX	Radio scheduler calls RAIL_StartScheduledTx().
RAIL_SCHEDULER_TASK_SINGLE_TX	Radio scheduler calls RAIL_StartTx().
RAIL_SCHEDULER_TASK_SINGLE_CCA_CSMA_TX	Radio scheduler calls RAIL_StartCcaCsmaTx().
RAIL_SCHEDULER_TASK_SINGLE_CCA_LBT_TX	Radio scheduler calls RAIL_StartCcaLbtTx().
RAIL_SCHEDULER_TASK_SCHEDULED_CCA_CSMA_TX	Radio scheduler calls RAIL_StartScheduledCcaCsmaTx().
RAIL_SCHEDULER_TASK_SCHEDULED_CCA_LBT_TX	Radio scheduler calls RAIL_StartScheduledCcaLbtTx().
RAIL_SCHEDULER_TASK_TX_STREAM	Radio scheduler calls RAIL_StartTxStream().
RAIL_SCHEDULER_TASK_AVERAGE_RSSI	Radio scheduler calls RAIL_StartAverageRssi().
RAIL_SCHEDULER_STATUS_SCHEDULED_TX_FAIL	RAIL_StartScheduledTx() returned error status.
RAIL_SCHEDULER_STATUS_SINGLE_TX_FAIL	RAIL_StartTx() returned error status.
RAIL_SCHEDULER_STATUS_CCA_CSMA_TX_FAIL	RAIL_StartCcaCsmaTx() returned error status.
RAIL_SCHEDULER_STATUS_CCA_LBT_TX_FAIL	RAIL_StartCcaLbtTx() returned error status.

RAIL_SCHEDULER_STATUS_SCHEDULED_RX_FAIL	RAIL_ScheduleRx() returned error status.
RAIL_SCHEDULER_STATUS_TX_STREAM_FAIL	RAIL_StartTxStream() returned error status.
RAIL_SCHEDULER_STATUS_AVERAGE_RSSI_FAIL	RAIL_StartAverageRssi() returned error status.
RAIL_SCHEDULER_SCHEDULED_RX_INTERNAL_ERROR	Multiprotocol scheduled receive function internal error.
RAIL_SCHEDULER_SCHEDULED_RX_SCHEDULING_ERROR	Multiprotocol scheduled receive scheduling error.
RAIL_SCHEDULER_SCHEDULED_RX_INTERRUPTED	RAIL_ScheduleRx() operation interrupted
RAIL_SCHEDULER_SCHEDULED_TX_INTERNAL_ERROR	Multiprotocol scheduled TX internal error.
RAIL_SCHEDULER_SCHEDULED_TX_SCHEDULING_ERROR	Multiprotocol scheduled TX scheduling error.
RAIL_SCHEDULER_SCHEDULED_TX_INTERRUPTED	RAIL_StartScheduledTx() operation interrupted
RAIL_SCHEDULER_SINGLE_TX_INTERNAL_ERROR	Multiprotocol instantaneous TX internal error.
RAIL_SCHEDULER_SINGLE_TX_SCHEDULING_ERROR	Multiprotocol instantaneous TX scheduling error.
RAIL_SCHEDULER_SINGLE_TX_INTERRUPTED	RAIL_StartTx() operation interrupted
RAIL_SCHEDULER_SINGLE_CCA_CSMA_TX_INTERNAL_ERROR	Multiprotocol single CSMA transmit function internal error.
RAIL_SCHEDULER_SINGLE_CCA_CSMA_TX_SCHEDULING_ERROR	Multiprotocol single CSMA transmit scheduling error.
RAIL_SCHEDULER_SINGLE_CCA_CSMA_TX_INTERRUPTED	RAIL_StartCcaCsmaTx() operation interrupted
RAIL_SCHEDULER_SINGLE_CCA_LBT_TX_INTERNAL_ERROR	Multiprotocol single LBT transmit function internal error.
RAIL_SCHEDULER_SINGLE_CCA_LBT_TX_SCHEDULING_ERROR	Multiprotocol single LBT transmit scheduling error.
RAIL_SCHEDULER_SINGLE_CCA_LBT_TX_INTERRUPTED	RAIL_StartCcaLbtTx() operation interrupted
RAIL_SCHEDULER_SCHEDULED_CCA_CSMA_TX_INTERNAL_ERROR	Multiprotocol scheduled CSMA transmit function internal error.
RAIL_SCHEDULER_SCHEDULED_CCA_CSMA_TX_FAIL	RAIL_StartScheduledCcaCsmaTx() returned error status.
RAIL_SCHEDULER_SCHEDULED_CCA_CSMA_TX_SCHEDULING_ERROR	Multiprotocol scheduled CSMA transmit scheduling error.
RAIL_SCHEDULER_SCHEDULED_CCA_CSMA_TX_INTERRUPTED	RAIL_StartScheduledCcaCsmaTx() operation interrupted
RAIL_SCHEDULER_SCHEDULED_CCA_LBT_TX_INTERNAL_ERROR	Multiprotocol scheduled LBT transmit function internal error.
RAIL_SCHEDULER_SCHEDULED_CCA_LBT_TX_FAIL	RAIL_StartScheduledCcaLbtTx() returned error status.
RAIL_SCHEDULER_SCHEDULED_CCA_LBT_TX_SCHEDULING_ERROR	Multiprotocol scheduled LBT transmit scheduling error.
RAIL_SCHEDULER_SCHEDULED_CCA_LBT_TX_INTERRUPTED	RAIL_StartScheduledCcaLbtTx() operation interrupted
RAIL_SCHEDULER_TX_STREAM_INTERNAL_ERROR	Multiprotocol stream transmit function internal error.
RAIL_SCHEDULER_TX_STREAM_SCHEDULING_ERROR	Multiprotocol stream transmit scheduling error.
RAIL_SCHEDULER_TX_STREAM_INTERRUPTED	RAIL_StartTxStream() operation interrupted
RAIL_SCHEDULER_AVERAGE_RSSI_INTERNAL_ERROR	Multiprotocol RSSI averaging function internal error.
RAIL_SCHEDULER_AVERAGE_RSSI_SCHEDULING_ERROR	Multiprotocol RSSI average scheduling error.
RAIL_SCHEDULER_AVERAGE_RSSI_INTERRUPTED	RAIL_StartAverageRssi() operation interrupted

Definition at line 516 of file common/rail_types.h

RAIL_TaskType_t

```
RAIL_TaskType_t
```

Multiprotocol radio operation task types, used with RAIL_SetTaskPriority.

Enumerator	
RAIL_TASK_TYPE_START_RX	Indicate a task started using RAIL_StartRx.
RAIL_TASK_TYPE_OTHER	Indicate a task started functions other than RAIL_StartRx.

Definition at line `749` of file `common/rail_types.h`

Function Documentation

RAIL_YieldRadio

```
void RAIL_YieldRadio (RAIL_Handle_t railHandle)
```

Yield the radio to other configurations.

Parameters

[in]	railHandle	A RAIL instance handle.

This function is used to indicate that the previous transmit or scheduled receive operation has completed. It must be used in multiprotocol RAIL because the scheduler assumes that any transmit or receive operation that is started can go on indefinitely based on state transitions and your protocol. RAIL will not allow a lower priority tasks to run until this is called so it can negatively impact performance of those protocols if this is omitted or delayed. It is also possible to call the RAIL_Idle() API to both terminate the operation and idle the radio. In single protocol RAIL this API does nothing, however, if RAIL Power Manager is initialized, calling RAIL_YieldRadio after scheduled TX/RX and instantaneous TX completion, is required, to indicate to the Power Manager that the the radio is no longer busy and can be idled for sleeping.

See Yielding the Radio for more details.

Definition at line `5597` of file `common/rail.h`

RAIL_GetSchedulerStatus

```
RAIL_SchedulerStatus_t RAIL_GetSchedulerStatus (RAIL_Handle_t railHandle)
```

Get the status of the RAIL scheduler.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- RAIL_SchedulerStatus_t status.

This function can only be called from a callback context after the RAIL_EVENT_SCHEDULER_STATUS event occurs.

Definition at line `5608` of file `common/rail.h`

RAIL_GetSchedulerStatusAlt

RAIL_Status_t RAIL_GetSchedulerStatusAlt (RAIL_Handle_t railHandle, RAIL_SchedulerStatus_t *pSchedulerStatus, RAIL_Status_t *pRailStatus)

Get the status of the RAIL scheduler, specific to the radio operation, along with RAIL_Status_t returned by RAIL API invoked by the RAIL scheduler.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	pSchedulerStatus	An application-provided pointer to store RAIL_SchedulerStatus_t status. Can be NULL as long as RAIL_Status_t pointer is not NULL.
[out]	pRailStatus	An application-provided pointer to store RAIL_Status_t of the RAIL API invoked by the RAIL scheduler. Can be NULL as long as RAIL_SchedulerStatus_t pointer is not NULL.

Returns

- RAIL_Status_t indicating success of the function call.

This function can only be called from a callback context after the RAIL_EVENT_SCHEDULER_STATUS event occurs.

Definition at line `5627` of file `common/rail.h`

RAIL_SetTaskPriority

RAIL_Status_t RAIL_SetTaskPriority (RAIL_Handle_t railHandle, uint8_t priority, RAIL_TaskType_t taskType)

Change the priority of a specified task type in multiprotocol.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	priority	Desired new priority for the railHandle's active task
[in]	taskType	Type of task whose priority should be updated

Returns

- RAIL_Status_t indicating success of the function call.

While the application can use this function however it likes, a major use case is being able to increase an infinite receive priority while receiving a packet. In other words, a given RAIL_Handle_t can maintain a very low priority background receive, but upon getting a RAIL_EVENT_RX_SYNC1_DETECT_SHIFT or RAIL_EVENT_RX_SYNC2_DETECT_SHIFT event, the app can call this function to increase the background RX priority to lower the risk another protocol might preempt during packet reception.

Definition at line `5648` of file `common/rail.h`

RAIL_GetTransitionTime

RAIL_Time_t RAIL_GetTransitionTime (void)

Get time needed to switch between protocols.

Parameters

N/A		

Returns

- RAIL_Time_t Time needed to switch between protocols.

Definition at line 5657 of file common/rail.h

RAIL_SetTransitionTime

```
void RAIL_SetTransitionTime (RAIL_Time_t transitionTime)
```

Set time needed to switch between protocols.

Parameters

[in]	transitionTime	Time needed to switch between protocols.

Call this API only once, before any protocol is initialized via RAIL_Init(). Changing this value during normal operation can result in improper scheduling behavior.

Definition at line 5667 of file common/rail.h

Macro Definition Documentation

RAIL_SCHEDULER_STATUS_MASK

```
#define RAIL_SCHEDULER_STATUS_MASK
```

Value:

```
0x0FU
```

Radio Scheduler Status mask.

Definition at line 500 of file common/rail_types.h

RAIL_SCHEDULER_STATUS_SHIFT

```
#define RAIL_SCHEDULER_STATUS_SHIFT
```

Value:

```
0
```

Radio Scheduler Status shift.

Definition at line 502 of file common/rail_types.h

RAIL_SCHEDULER_TASK_MASK

```
#define RAIL_SCHEDULER_TASK_MASK
```

Value:

```
0xF0U
```

Radio Scheduler Task mask.

Definition at line 505 of file common/rail_types.h

RAIL_SCHEDULER_TASK_SHIFT

```
#define RAIL_SCHEDULER_TASK_SHIFT
```

Value:

```
4
```

Radio Scheduler Task shift.

Definition at line 507 of file common/rail_types.h

RAIL_SchedulerInfo_t

A structure to hold information used by the scheduler.

For multiprotocol versions of RAIL, this can be used to control how a receive or transmit operation is run. It's not necessary in single-protocol applications.

Public Attributes

uint8_t	priority The scheduler priority to use for this operation.
RAIL_Time_t	slipTime The amount of time in us that this operation can slip by into the future and still be run.
RAIL_Time_t	transactionTime The transaction time in us for this operation.

Public Attribute Documentation

priority

```
uint8_t RAIL_SchedulerInfo_t::priority
```

The scheduler priority to use for this operation.

This priority is used to preempt a long running lower-priority task to ensure higher-priority operations complete in time. A lower numerical value represents a higher logical priority meaning 0 is the highest priority and 255 is the lowest.

Definition at line `481` of file `common/rail_types.h`

slipTime

```
RAIL_Time_t RAIL_SchedulerInfo_t::slipTime
```

The amount of time in us that this operation can slip by into the future and still be run.

This time is relative to the start time which may be the current time for relative transmits. If the scheduler can't start the operation by this time, it will be considered a failure.

Definition at line `488` of file `common/rail_types.h`

transactionTime

```
RAIL_Time_t RAIL_SchedulerInfo_t::transactionTime
```

The transaction time in us for this operation.

Since transaction times may not be known exactly, use a minimum or an expected guess for this time. The scheduler will use the value entered here to look for overlaps between low-priority and high-priority tasks and attempt to find a schedule where all tasks get to run.

Definition at line `496` of file `common/rail_types.h`

EFR32

EFR32

EFR32-specific multiprotocol support defines.

Macros

#define TRANSITION_TIME_US 430
Time it takes to take care of protocol switching.

Macro Definition Documentation

TRANSITION_TIME_US

```
#define TRANSITION_TIME_US
```

Value:

```
430
```

Time it takes to take care of protocol switching.

Definition at line 191 of file chip/efr32/efr32xg1x/rail_chip_specific.h

Packet Trace (PTI)

Packet Trace (PTI)

Basic APIs to set up and interact with PTI settings.

These enumerations and structures are used with RAIL PTI API.

EFR32 supports SPI and UART PTI and is configurable in terms of baud rates and PTI pin locations.

Modules

RAIL_PtiConfig_t

Enumerations

enum RAIL_PtiMode_t {

RAIL_PTI_MODE_DISABLED
RAIL_PTI_MODE_SPI
RAIL_PTI_MODE_UART
RAIL_PTI_MODE_UART_ONEWIRE

}

A channel type enumeration.

enum RAIL_PtiProtocol_t {

RAIL_PTI_PROTOCOL_CUSTOM = 0
RAIL_PTI_PROTOCOL_THREAD = 2
RAIL_PTI_PROTOCOL_BLE = 3
RAIL_PTI_PROTOCOL_CONNECT = 4
RAIL_PTI_PROTOCOL_ZIGBEE = 5
RAIL_PTI_PROTOCOL_ZWAVE = 6
RAIL_PTI_PROTOCOL_WISUN = 7
RAIL_PTI_PROTOCOL_802154 = 8
RAIL_PTI_PROTOCOL_SIDEWALK = 9

}

The protocol that RAIL outputs via the Packet Trace Interface (PTI).

Functions

RAIL_Status_t RAIL_ConfigPti(RAIL_Handle_t railHandle, const RAIL_PtiConfig_t *ptiConfig)

Configure PTI pin locations, serial protocols, and baud rates.

RAIL_Status_t RAIL_GetPtiConfig(RAIL_Handle_t railHandle, RAIL_PtiConfig_t *ptiConfig)

Get the currently-active PTI configuration.

RAIL_Status_t RAIL_EnablePti(RAIL_Handle_t railHandle, bool enable)

Enable Packet Trace Interface (PTI) output of packet data.

RAIL_Status_t RAIL_SetPtiProtocol(RAIL_Handle_t railHandle, RAIL_PtiProtocol_t protocol)

Set a protocol that RAIL outputs on PTI.

RAIL_PtiProtocol_t	RAIL_GetPtiProtocol(RAIL_Handle_t railHandle) Get the protocol that RAIL outputs on PTI.

Enumeration Documentation

RAIL_PtiMode_t

> RAIL_PtiMode_t

A channel type enumeration.

Enumerator	
RAIL_PTI_MODE_DISABLED	Turn PTI off entirely.
RAIL_PTI_MODE_SPI	8-bit SPI mode.
RAIL_PTI_MODE_UART	8-bit UART mode.
RAIL_PTI_MODE_UART_ONEWIRE	9-bit UART mode.

Definition at line 2371 of file common/rail_types.h

RAIL_PtiProtocol_t

> RAIL_PtiProtocol_t

The protocol that RAIL outputs via the Packet Trace Interface (PTI).

Enumerator	
RAIL_PTI_PROTOCOL_CUSTOM	PTI output for a custom protocol.
RAIL_PTI_PROTOCOL_THREAD	PTI output for the Thread protocol.
RAIL_PTI_PROTOCOL_BLE	PTI output for the Bluetooth Smart protocol.
RAIL_PTI_PROTOCOL_CONNECT	PTI output for the Connect protocol.
RAIL_PTI_PROTOCOL_ZIGBEE	PTI output for the Zigbee protocol.
RAIL_PTI_PROTOCOL_ZWAVE	PTI output for the Z-Wave protocol.
RAIL_PTI_PROTOCOL_WISUN	PTI output for the Wi-SUN protocol.
RAIL_PTI_PROTOCOL_802154	PTI output for a custom protocol using a built-in 802.15.4 radio config.
RAIL_PTI_PROTOCOL_SIDEWALK	

Definition at line 2426 of file common/rail_types.h

Function Documentation

RAIL_ConfigPti

> RAIL_Status_t RAIL_ConfigPti (RAIL_Handle_t railHandle, const RAIL_PtiConfig_t *ptiConfig)

Configure PTI pin locations, serial protocols, and baud rates.

Parameters

[in]	railHandle	A RAIL instance handle (currently not used).
[in]	ptiConfig	A configuration structure applied to the relevant PTI registers. A NULL ptiConfig will produce undefined behavior.

Returns

- Status code indicating success of the function call.

This method must be called before RAIL_EnablePti() is called. Although a RAIL handle is included for potential future expansion of this function, it is currently not used. That is, there is only one PTI configuration that can be active on a chip, regardless of the number of protocols (unless the application updates the configuration upon a protocol switch), and the configuration is not saved in the RAIL instance. For optimal future compatibility, pass in a chip-specific handle, such as RAIL_EFR32_HANDLE.

PTI should be configured only when the radio is off (idle).

Note

- On EFR32 platforms GPIO configuration must be unlocked (see GPIO->LOCK register) to configure or use PTI.

Definition at line 389 of file common/rail.h

RAIL_GetPtiConfig

> RAIL_Status_t RAIL_GetPtiConfig (RAIL_Handle_t railHandle, RAIL_PtiConfig_t *ptiConfig)

Get the currently-active PTI configuration.

Parameters

[in]	railHandle	A RAIL instance handle (currently not used).
[out]	ptiConfig	A configuration structure filled with the active PTI configuration.

Returns

- RAIL status indicating success of the function call.

Although most combinations of configurations can be set, it is safest to call this method after configuration to confirm which values were actually set. As in RAIL_ConfigPti, railHandle is not used. This function always returns the single active PTI configuration regardless of the active protocol. For optimal future compatibility, pass in a chip-specific handle, such as RAIL_EFR32_HANDLE.

Definition at line 407 of file common/rail.h

RAIL_EnablePti

> RAIL_Status_t RAIL_EnablePti (RAIL_Handle_t railHandle, bool enable)

Enable Packet Trace Interface (PTI) output of packet data.

Parameters

[in]	railHandle	A RAIL instance handle (currently not used).
[in]	enable	PTI is enabled if true; disable if false.

Returns

- RAIL status indicating success of the function call.

Similarly to having only one PTI configuration per chip, PTI can only be enabled or disabled for all protocols. It cannot be individually set to enabled and disabled per protocol (unless the application switches it when the protocol switches), and enable/disable is not saved as part of the RAIL instance. For optimal future compatibility, pass in a chip-specific handle, such as RAIL_EFR32_HANDLE.

PTI should be enabled or disabled only when the radio is off (idle).

Warnings

- On EFR32 platforms GPIO configuration must be unlocked (see GPIO->LOCK register) to configure or use PTI, otherwise a fault or assert might occur. If GPIO configuration locking is desired, PTI must be disabled beforehand either with this function or with RAIL_ConfigPti() using RAIL_PTI_MODE_DISABLED.

Definition at line 434 of file common/rail.h

RAIL_SetPtiProtocol

```
RAIL_Status_t RAIL_SetPtiProtocol (RAIL_Handle_t railHandle, RAIL_PtiProtocol_t protocol)
```

Set a protocol that RAIL outputs on PTI.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	protocol	The enumeration representing which protocol the node is using.

Returns

- Status code indicating success of the function call.

The protocol is output via PTI for each packet. Before any protocol is set, the default value is RAIL_PTI_PROTOCOL_CUSTOM. Use one of the enumeration values so that the Network Analyzer can decode the packet.

Note

- This function cannot be called unless the radio is currently in the RAIL_RF_STATE_IDLE or RAIL_RF_STATE_INACTIVE states. For this reason, call this function early on before starting radio operations and not changed later.

Definition at line 454 of file common/rail.h

RAIL_GetPtiProtocol

```
RAIL_PtiProtocol_t RAIL_GetPtiProtocol (RAIL_Handle_t railHandle)
```

Get the protocol that RAIL outputs on PTI.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- PTI protocol in use.

Definition at line 463 of file common/rail.h

RAIL_PtiConfig_t

A configuration for PTI.

Public Attributes

RAIL_PtiMode_t	mode Packet Trace mode (UART or SPI).
uint32_t	baud Output baudrate for PTI in Hz.
uint8_t	doutLoc Data output (DOUT) location for doutPort/Pin.
uint8_t	doutPort Data output (DOUT) GPIO port.
uint8_t	doutPin Data output (DOUT) GPIO pin.
uint8_t	dclkLoc Data clock (DCLK) location for dclkPort/Pin.
uint8_t	dclkPort Data clock (DCLK) GPIO port.
uint8_t	dclkPin Data clock (DCLK) GPIO pin.
uint8_t	dframeLoc Data frame (DFRAME) location for dframePort/Pin.
uint8_t	dframePort Data frame (DFRAME) GPIO port.
uint8_t	dframePin Data frame (DFRAME) GPIO pin.

Public Attribute Documentation

mode

```
RAIL_PtiMode_t RAIL_PtiConfig_t::mode
```

Packet Trace mode (UART or SPI).

Definition at line `2396` of file `common/rail_types.h`

baud

```
uint32_t RAIL_PtiConfig_t::baud
```

Output baudrate for PTI in Hz.

Definition at line 2398 of file common/rail_types.h

doutLoc

```
uint8_t RAIL_PtiConfig_t::doutLoc
```

Data output (DOUT) location for doutPort/Pin.

Only needed on EFR32 Series 1; ignored on other platforms.

Definition at line 2401 of file common/rail_types.h

doutPort

```
uint8_t RAIL_PtiConfig_t::doutPort
```

Data output (DOUT) GPIO port.

Definition at line 2403 of file common/rail_types.h

doutPin

```
uint8_t RAIL_PtiConfig_t::doutPin
```

Data output (DOUT) GPIO pin.

Definition at line 2405 of file common/rail_types.h

dclkLoc

```
uint8_t RAIL_PtiConfig_t::dclkLoc
```

Data clock (DCLK) location for dclkPort/Pin.

Only used in SPI mode. Only needed on EFR32 Series 1; ignored on other platforms.

Definition at line 2408 of file common/rail_types.h

dclkPort

```
uint8_t RAIL_PtiConfig_t::dclkPort
```

Data clock (DCLK) GPIO port.

Only used in SPI mode.

Definition at line 2410 of file common/rail_types.h

dclkPin

```
uint8_t RAIL_PtiConfig_t::dclkPin
```

Data clock (DCLK) GPIO pin.

Only used in SPI mode.

Definition at line 2412 of file common/rail_types.h

dframeLoc

```
uint8_t RAIL_PtiConfig_t::dframeLoc
```

Data frame (DFRAME) location for dframePort/Pin.

Only needed on EFR32 Series 1; ignored on other platforms.

Definition at line 2415 of file common/rail_types.h

dframePort

```
uint8_t RAIL_PtiConfig_t::dframePort
```

Data frame (DFRAME) GPIO port.

Definition at line 2417 of file common/rail_types.h

dframePin

```
uint8_t RAIL_PtiConfig_t::dframePin
```

Data frame (DFRAME) GPIO pin.

Definition at line 2419 of file common/rail_types.h

Protocol-specific

Protocol-specific

Protocol-Specific RAIL APIs.

Modules

BLE

IEEE 802.15.4

Multi-Level Frequency Modulation

Sidewalk Radio Configurations

Z-Wave

BLE

BLE

Accelerator routines for Bluetooth Low Energy (BLE).

The APIs in this module configure the radio for BLE operation and provide additional helper routines necessary for normal BLE send/receive that aren't available directly in RAIL. RAIL APIs should be used to set up the application. However, RAIL_ConfigChannels() and RAIL_ConfigRadio() should not be called to set up the PHY. Instead, RAIL_BLE_Config* APIs should be used to set up the 1 Mbps, 2 Mbps, or Coded PHY configurations needed by the application. These APIs will configure the hardware and also configure the set of valid BLE channels.

To implement a standard BLE link layer, you will also need to handle tight turnaround times and send packets at specific instants. This can all be managed through general RAIL functions, such as RAIL_ScheduleTx(), RAIL_ScheduleRx(), and RAIL_SetStateTiming(). See RAIL APIs for more useful functions.

A simple example to set up the application to be in BLE mode is shown below. Note that this will put the radio on the first advertising channel with the advertising Access Address. In any full-featured BLE application you will need to use the RAIL_BLE_ConfigChannelRadioParams() function to change the sync word and other parameters as needed based on your connection.

```
// RAIL Handle set at initialization time.
static RAIL_Handle_t gRailHandle = NULL;

static void radioEventHandler(RAIL_Handle_t railHandle,
                              RAIL_Events_t events)
{
  // ... handle RAIL events, e.g., receive and transmit completion
}

#if MULTIPROTOCOL
// Allocate memory for RAIL to hold BLE-specific state information
static RAIL_BLE_State_t bleState; // Must never be const
static RAILSched_Config_t schedCfg; // Must never be const
static RAIL_Config_t railCfg = {  // Must never be const
  .eventsCallback = &radioEventHandler,
  .protocol = &bleState, // For BLE, RAIL needs additional state memory
  .scheduler = &schedCfg, // For MultiProtocol, additional scheduler memory
};
#else
static RAIL_Config_t railCfg = {  // Must never be const
  .eventsCallback = &radioEventHandler,
  .protocol = NULL,
  .scheduler = NULL,
};
#endif

// Set the radio to receive on the first BLE advertising channel.
int bleAdvertiseEnable(void)
{
  // Initializes the RAIL library and any internal state it requires.
  gRailHandle = RAIL_Init(&railCfg, NULL);

  // Calls the BLE initialization function to load the right radio configuration.
  RAIL_BLE_Init(gRailHandle);

  // Always choose the Viterbi PHY configuration if available on your chip
  // for performance reasons.
  RAIL_BLE_ConfigPhy1MbpsViterbi(gRailHandle);

  // Configures us for the first advertising channel (Physical: 0, Logical: 37).
  // The CRC init value and Access Address come from the BLE specification.
  RAIL_BLE_ConfigChannelRadioParams(gRailHandle,
                                    0x555555,
                                    0x8E89BED6,
                                    37,
                                    false);

  // Starts receiving on physical channel 0 (logical channel 37).
  RAIL_StartRx(gRailHandle, 0, NULL);
 }
```

Modules

RAIL_BLE_State_t

Angle of Arrival/Departure

BLE Radio Configurations

BLE TX Channel Hopping

Enumerations

enum RAIL_BLE_Coding_t {

RAIL_BLE_Coding_125kbps = 0
RAIL_BLE_Coding_125kbps_DSA = 1
RAIL_BLE_Coding_500kbps = 2
RAIL_BLE_Coding_500kbps_DSA = 3

}

The variant of the BLE Coded PHY.

enum RAIL_BLE_Phy_t {

RAIL_BLE_1Mbps
RAIL_BLE_2Mbps
RAIL_BLE_Coded125kbps
RAIL_BLE_Coded500kbps

}

The variant of the BLE PHY.

enum RAIL_BLE_SignalIdentifierMode_t {

RAIL_BLE_SIGNAL_IDENTIFIER_MODE_DISABLE = 0
RAIL_BLE_SIGNAL_IDENTIFIER_MODE_1MBPS
RAIL_BLE_SIGNAL_IDENTIFIER_MODE_2MBPS

}

Available Signal Identifier modes.

Functions

void RAIL_BLE_Init(RAIL_Handle_t railHandle)

Configure RAIL to run in BLE mode.

void RAIL_BLE_Deinit(RAIL_Handle_t railHandle)

Take RAIL out of BLE mode.

bool RAIL_BLE_IsEnabled(RAIL_Handle_t railHandle)

Determine whether BLE mode is enabled or not.

RAIL_Status_t RAIL_BLE_ConfigPhyQuuppa(RAIL_Handle_t railHandle)

Switch to the 1 Mbps Quuppa PHY.

RAIL_Status_t RAIL_BLE_ConfigPhy1MbpsViterbi(RAIL_Handle_t railHandle)

Switch to the Viterbi 1 Mbps BLE PHY.

RAIL_Status_t RAIL_BLE_ConfigPhy1Mbps(RAIL_Handle_t railHandle)

Switch to the legacy non-Viterbi 1 Mbps BLE PHY.

RAIL_Status_t RAIL_BLE_ConfigPhy2MbpsViterbi(RAIL_Handle_t railHandle)

Switch to the Viterbi 2 Mbps BLE PHY.

RAIL_Status_t RAIL_BLE_ConfigPhy2Mbps(RAIL_Handle_t railHandle)

Switch to the legacy non-Viterbi 2 Mbps BLE PHY.

RAIL_Status_t RAIL_BLE_ConfigPhyCoded(RAIL_Handle_t railHandle, RAIL_BLE_Coding_t bleCoding)

Switch to the BLE Coded PHY.

RAIL_Status_t RAIL_BLE_ConfigPhySimulscan(RAIL_Handle_t railHandle)

Switch to the Simulscan PHY.

RAIL_Status_t RAIL_BLE_ConfigChannelRadioParams(RAIL_Handle_t railHandle, uint32_t crcInit, uint32_t accessAddress, uint16_t channel, bool disableWhitening)

Change BLE radio parameters.

RAIL_Status_t RAIL_BLE_PhySwitchToRx(RAIL_Handle_t railHandle, RAIL_BLE_Phy_t phy, uint16_t railChannel, uint32_t startRxTime, uint32_t crcInit, uint32_t accessAddress, uint16_t logicalChannel, bool disableWhitening)
Change the current BLE PHY and go into receive.

RAIL_Status_t RAIL_BLE_ConfigSignalIdentifier(RAIL_Handle_t railHandle, RAIL_BLE_SignalIdentifierMode_t signalIdentifierMode)
Configure and enable signal identifier for BLE signal detection.

RAIL_Status_t RAIL_BLE_EnableSignalDetection(RAIL_Handle_t railHandle, bool enable)
Enable or disable signal identifier interrupt for BLE signal detection.

Macros

#define RAIL_BLE_RX_SUBPHY_ID_500K (0U)
subPhyId indicating a 500kbps packet

#define RAIL_BLE_RX_SUBPHY_ID_125K (1U)
subPhyId indicating a 125kbps packet

#define RAIL_BLE_RX_SUBPHY_ID_1M (2U)
subPhyId value indicating a 1Mbps packet

#define RAIL_BLE_RX_SUBPHY_ID_INVALID (3U)
Invalid subPhyId value.

#define RAIL_BLE_RX_SUBPHY_COUNT (4U)
subPhyId indicating the total count

#define RAIL_BLE_EnableSignalIdentifier RAIL_BLE_EnableSignalDetection
Backward compatible name for the RAIL_BLE_EnableSignalDetection API.

Enumeration Documentation

RAIL_BLE_Coding_t

> RAIL_BLE_Coding_t

The variant of the BLE Coded PHY.

Enumerator	
RAIL_BLE_Coding_125kbps	Enables the 125 kbps variant of the BLE Coded PHY.
RAIL_BLE_Coding_125kbps_DSA	
RAIL_BLE_Coding_500kbps	Enables the 500 kbps variant of the BLE Coded PHY.
RAIL_BLE_Coding_500kbps_DSA	

Definition at line 131 of file protocol/ble/rail_ble.h

RAIL_BLE_Phy_t

> RAIL_BLE_Phy_t

The variant of the BLE PHY.

Enumerator	
RAIL_BLE_1Mbps	Use the standard BLE 1Mbps PHY.
RAIL_BLE_2Mbps	Use the high data rate BLE 2Mbps PHY.

RAIL_BLE_Coded125kbps	Enables the 125 kbps variant of the BLE Coded PHY.
RAIL_BLE_Coded500kbps	Enables the 500 kbps variant of the BLE Coded PHY.

Definition at line 154 of file protocol/ble/rail_ble.h

RAIL_BLE_SignalIdentifierMode_t

> RAIL_BLE_SignalIdentifierMode_t

Available Signal Identifier modes.

Enumerator	
RAIL_BLE_SIGNAL_IDENTIFIER_MODE_DISABLE	
RAIL_BLE_SIGNAL_IDENTIFIER_MODE_1MBPS	
RAIL_BLE_SIGNAL_IDENTIFIER_MODE_2MBPS	

Definition at line 276 of file protocol/ble/rail_ble.h

Function Documentation

RAIL_BLE_Init

> void RAIL_BLE_Init (RAIL_Handle_t railHandle)

Configure RAIL to run in BLE mode.

Parameters

[in]	railHandle	A handle for RAIL instance. This function changes your radio, channel configuration, and other parameters to match what is needed for BLE. To switch back to a default RAIL mode, call RAIL_BLE_Deinit() first. This function will configure the protocol output on PTI to RAIL_PTI_PROTOCOL_BLE.

Note

- BLE may not be enabled while Auto-ACKing is enabled.

Definition at line 318 of file protocol/ble/rail_ble.h

RAIL_BLE_Deinit

> void RAIL_BLE_Deinit (RAIL_Handle_t railHandle)

Take RAIL out of BLE mode.

Parameters

[in]	railHandle	A handle for RAIL instance. This function will undo some of the configuration that happens when you call RAIL_BLE_Init(). After this you can safely run your normal radio initialization code to use a non-BLE configuration. This function does **not** change back your radio or channel configurations so you must do this by manually reinitializing. This also resets the protocol output on PTI to RAIL_PTI_PROTOCOL_CUSTOM.

Definition at line 331 of file protocol/ble/rail_ble.h

RAIL_BLE_IsEnabled

```
bool RAIL_BLE_IsEnabled (RAIL_Handle_t railHandle)
```

Determine whether BLE mode is enabled or not.

Parameters

[in]	railHandle	A handle for RAIL instance.

Returns

- True if BLE mode is enabled and false otherwise. This function returns the current status of RAIL's BLE mode. It is enabled by a call to RAIL_BLE_Init() and disabled by a call to RAIL_BLE_Deinit().

Definition at line `341` of file `protocol/ble/rail_ble.h`

RAIL_BLE_ConfigPhyQuuppa

```
RAIL_Status_t RAIL_BLE_ConfigPhyQuuppa (RAIL_Handle_t railHandle)
```

Switch to the 1 Mbps Quuppa PHY.

Parameters

[in]	railHandle	A handle for RAIL instance.

Returns

- Status code indicating success of the function call.

You can use this function to switch to the Quuppa PHY.

Note

- Not all chips support the 1Mbps Quuppa PHY. This API should return RAIL_STATUS_INVALID_CALL if unsupported by the hardware we're building for.

Definition at line `354` of file `protocol/ble/rail_ble.h`

RAIL_BLE_ConfigPhy1MbpsViterbi

```
RAIL_Status_t RAIL_BLE_ConfigPhy1MbpsViterbi (RAIL_Handle_t railHandle)
```

Switch to the Viterbi 1 Mbps BLE PHY.

Parameters

[in]	railHandle	A handle for RAIL instance.

Returns

- A status code indicating success of the function call.

Use this function to switch back to the default BLE 1 Mbps PHY if you have switched to the 2 Mbps or another configuration. You may only call this function after initializing BLE and while the radio is idle.

Note

-

The EFR32XG1 family does not support BLE Viterbi PHYs. However, calls to this function from that family will be silently redirected to the legacy RAIL_BLE_ConfigPhy1Mbps().

Definition at line 370 of file protocol/ble/rail_ble.h

RAIL_BLE_ConfigPhy1Mbps

```
RAIL_Status_t RAIL_BLE_ConfigPhy1Mbps (RAIL_Handle_t railHandle)
```

Switch to the legacy non-Viterbi 1 Mbps BLE PHY.

Parameters

[in]	railHandle	A handle for RAIL instance.

Returns

- A status code indicating success of the function call.

Use this function to switch back to the legacy BLE 1 Mbps PHY if you have switched to the 2 Mbps or another configuration. You may only call this function after initializing BLE and while the radio is idle.

Note

- The EFR32XG2x family does not support BLE non-Viterbi PHYs.

Definition at line 384 of file protocol/ble/rail_ble.h

RAIL_BLE_ConfigPhy2MbpsViterbi

```
RAIL_Status_t RAIL_BLE_ConfigPhy2MbpsViterbi (RAIL_Handle_t railHandle)
```

Switch to the Viterbi 2 Mbps BLE PHY.

Parameters

[in]	railHandle	A handle for RAIL instance.

Returns

- A status code indicating success of the function call.

Use this function to switch back to the BLE 2 Mbps PHY from the default 1 Mbps option. You may only call this function after initializing BLE and while the radio is idle.

Note

- The EFR32XG1 family does not support BLE Viterbi PHYs.

Definition at line 398 of file protocol/ble/rail_ble.h

RAIL_BLE_ConfigPhy2Mbps

```
RAIL_Status_t RAIL_BLE_ConfigPhy2Mbps (RAIL_Handle_t railHandle)
```

Switch to the legacy non-Viterbi 2 Mbps BLE PHY.

Parameters

[in]	railHandle	A handle for RAIL instance.

Returns

- A status code indicating success of the function call.

Use this function to switch back to legacy BLE 2Mbps PHY from the default 1 Mbps option. You may only call this function after initializing BLE and while the radio is idle.

Note

- The EFR32XG1 and EFR32XG2x families do not support BLE non-Viterbi 2 Mbps PHY.

Definition at line `413` of file `protocol/ble/rail_ble.h`

RAIL_BLE_ConfigPhyCoded

> RAIL_Status_t RAIL_BLE_ConfigPhyCoded (RAIL_Handle_t railHandle, RAIL_BLE_Coding_t bleCoding)

Switch to the BLE Coded PHY.

Parameters

[in]	railHandle	A handle for RAIL instance.
[in]	bleCoding	The RAIL_BLE_Coding_t to use

Returns

- A status code indicating success of the function call.

Use this function to switch back to BLE Coded PHY from the default 1 Mbps option. You may only call this function after initializing BLE and while the radio is idle. When using a BLE Coded PHY, the RAIL_RxPacketDetails_t::subPhyId marks the coding of the received packet. A subPhyId of 0 marks a 500 kbps packet, and a subPhyId of 1 marks a 125 kbps packet.

Note

- The EFR32XG1, EFR32XG12, and EFR32XG14 families do not support BLE Coded PHYs.

Definition at line `432` of file `protocol/ble/rail_ble.h`

RAIL_BLE_ConfigPhySimulscan

> RAIL_Status_t RAIL_BLE_ConfigPhySimulscan (RAIL_Handle_t railHandle)

Switch to the Simulscan PHY.

Parameters

[in]	railHandle	A handle for RAIL instance.

Returns

- A status code indicating success of the function call.

Use this function to switch to the BLE Simulscan PHY. You may only call this function after initializing BLE and while the radio is idle. When using Simulscan PHY, the RAIL_RxPacketDetails_t::subPhyId marks the coding of the received packet. A subPhyId of 0 marks a 500 kbps packet, a subPhyId of 1 marks a 125 kbps packet, and a subPhyId of 2 marks a 1 Mbps packet.

Note

: The Simulscan PHY is supported only on some 2.4 GHz Series-2 parts. The preprocessor symbol RAIL_BLE_SUPPORTS_SIMULSCAN_PHY and the runtime function RAIL_BLE_SupportsSimulscanPhy() may be used to test for support of the Simulscan PHY.

Definition at line 453 of file protocol/ble/rail_ble.h

RAIL_BLE_ConfigChannelRadioParams

> RAIL_Status_t RAIL_BLE_ConfigChannelRadioParams (RAIL_Handle_t railHandle, uint32_t crcInit, uint32_t accessAddress, uint16_t channel, bool disableWhitening)

Change BLE radio parameters.

Parameters

[in]	railHandle	A handle for RAIL instance.
[in]	crcInit	The value to use for CRC initialization.
[in]	accessAddress	The access address to use for the connection. The bits of this parameter are transmitted or received LSB first.
[in]	channel	The logical channel that you're changing to, which initializes the whitener if used.
[in]	disableWhitening	This can turn off the whitening engine and is useful for sending BLE test mode packets that don't have this turned on.

Returns

- A status code indicating success of the function call.

This function can be used to switch radio parameters on every connection and/or channel change. It is BLE-aware and will set the access address, preamble, CRC initialization value, and whitening configuration without requiring you to load a new radio configuration. This function should not be called while the radio is active.

Definition at line 504 of file protocol/ble/rail_ble.h

RAIL_BLE_PhySwitchToRx

> RAIL_Status_t RAIL_BLE_PhySwitchToRx (RAIL_Handle_t railHandle, RAIL_BLE_Phy_t phy, uint16_t railChannel, uint32_t startRxTime, uint32_t crcInit, uint32_t accessAddress, uint16_t logicalChannel, bool disableWhitening)

Change the current BLE PHY and go into receive.

Parameters

[in]	railHandle	A handle for RAIL instance.
[in]	phy	Indicates which PHY to receive on
[in]	railChannel	Which channel of the given PHY to receive on
[in]	startRxTime	When to enter RX
[in]	crcInit	The value to use for CRC initialization.
[in]	accessAddress	The access address to use for the connection. The bits of this parameter are transmitted or received LSB first.
[in]	logicalChannel	The logical channel that you're changing to, which initializes the whitener if used.
[in]	disableWhitening	This can turn off the whitening engine and is useful for sending BLE test mode packets that don't have this turned on.

Returns

A status code indicating success of the function call.

This function is used to implement auxiliary packet reception, as defined in the BLE specification. The radio will be put into IDLE, the PHY and channel will be changed, and then receive will be entered at the start time given. The new receive will have a timeout of 30 us, which means that this function should only be called if the offset unit is 30 us.

This function is extremely time-sensitive, and may only be called within the interrupt context of a RAIL_EVENT_RX_PACKET_RECEIVED event.

Definition at line `535` of file `protocol/ble/rail_ble.h`

RAIL_BLE_ConfigSignalIdentifier

> RAIL_Status_t RAIL_BLE_ConfigSignalIdentifier (RAIL_Handle_t railHandle, RAIL_BLE_SignalIdentifierMode_t signalIdentifierMode)

Configure and enable signal identifier for BLE signal detection.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	signalIdentifierMode	Mode of signal identifier operation.

This features allows detection of BLE signal on air based on the mode. This function must be called once before RAIL_BLE_EnableSignalDetection to configure and enable signal identifier.

To enable event for signal detection RAIL_ConfigEvents() must be called for enabling RAIL_EVENT_SIGNAL_DETECTED.

This function is only supported by chips where RAIL_BLE_SUPPORTS_SIGNAL_IDENTIFIER and RAIL_BLE_SupportsSignalIdentifier() are true.

Returns

- Status code indicating success of the function call.

Definition at line `563` of file `protocol/ble/rail_ble.h`

RAIL_BLE_EnableSignalDetection

> RAIL_Status_t RAIL_BLE_EnableSignalDetection (RAIL_Handle_t railHandle, bool enable)

Enable or disable signal identifier interrupt for BLE signal detection.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	enable	Signal detection is enabled if true, disabled if false.

RAIL_BLE_ConfigSignalIdentifier must be called once before calling this function to configure and enable signal identifier. Once a signal is detected signal detection will be turned off and this function should be called to re-enable the signal detection without needing to call RAIL_BLE_ConfigSignalIdentifier if the signal identifier is already configured and enabled.

This function is only supported by chips where RAIL_BLE_SUPPORTS_SIGNAL_IDENTIFIER and RAIL_BLE_SupportsSignalIdentifier() are true.

Returns

- Status code indicating success of the function call.

Definition at line `585` of file `protocol/ble/rail_ble.h`

Macro Definition Documentation

RAIL_BLE_RX_SUBPHY_ID_500K

```
#define RAIL_BLE_RX_SUBPHY_ID_500K
```

Value:

```
(0U)
```

subPhyId indicating a 500kbps packet

Definition at line 262 of file protocol/ble/rail_ble.h

RAIL_BLE_RX_SUBPHY_ID_125K

```
#define RAIL_BLE_RX_SUBPHY_ID_125K
```

Value:

```
(1U)
```

subPhyId indicating a 125kbps packet

Definition at line 264 of file protocol/ble/rail_ble.h

RAIL_BLE_RX_SUBPHY_ID_1M

```
#define RAIL_BLE_RX_SUBPHY_ID_1M
```

Value:

```
(2U)
```

subPhyId value indicating a 1Mbps packet

Definition at line 266 of file protocol/ble/rail_ble.h

RAIL_BLE_RX_SUBPHY_ID_INVALID

```
#define RAIL_BLE_RX_SUBPHY_ID_INVALID
```

Value:

```
(3U)
```

Invalid subPhyId value.

Definition at line 268 of file protocol/ble/rail_ble.h

RAIL_BLE_RX_SUBPHY_COUNT

```
#define RAIL_BLE_RX_SUBPHY_COUNT
```

Value:

```
(4U)
```

subPhyId indicating the total count

Definition at line 270 of file protocol/ble/rail_ble.h

RAIL_BLE_EnableSignalIdentifier

```
#define RAIL_BLE_EnableSignalIdentifier
```

Value:

```
RAIL_BLE_EnableSignalDetection
```

Backward compatible name for the RAIL_BLE_EnableSignalDetection API.

Definition at line 592 of file protocol/ble/rail_ble.h

RAIL_BLE_State_t

A state structure for BLE.

This structure must be allocated in application global read-write memory that persists for the duration of BLE usage. It cannot be allocated in read-only memory or on the call stack.

Public Attributes

uint32_t	crcInit The value used to initialize the CRC algorithm.
uint32_t	accessAddress The access address used for the connection.
uint16_t	channel The logical channel used.
bool	disableWhitening Indicates whether the whitening engine should be off.

Public Attribute Documentation

crcInit

> uint32_t RAIL_BLE_State_t::crcInit

The value used to initialize the CRC algorithm.

Definition at line `301` of file `protocol/ble/rail_ble.h`

accessAddress

> uint32_t RAIL_BLE_State_t::accessAddress

The access address used for the connection.

Definition at line `302` of file `protocol/ble/rail_ble.h`

channel

> uint16_t RAIL_BLE_State_t::channel

The logical channel used.

Definition at line `303` of file `protocol/ble/rail_ble.h`

disableWhitening

bool RAIL_BLE_State_t::disableWhitening

Indicates whether the whitening engine should be off.

Definition at line 304 of file protocol/ble/rail_ble.h

Angle of Arrival/Departure

Angle of Arrival/Departure

These APIs are to a stack implementing BLE's angle of arrival and angle of departure functionality.

They are designed for use by the Silicon Labs BLE stack only at this time and may cause problems if accessed directly.

Modules

RAIL_BLE_AoxConfig_t

RAIL_BLE_AoxAntennaPortPins_t

RAIL_BLE_AoxAntennaConfig_t

Enumerations

enum RAIL_BLE_AoxOptions_t {

RAIL_BLE_AOX_OPTIONS_SAMPLE_MODE_SHIFT = 0
RAIL_BLE_AOX_OPTIONS_CONNLESS_SHIFT = 1
RAIL_BLE_AOX_OPTIONS_CONN_SHIFT = 2
RAIL_BLE_AOX_OPTIONS_DISABLE_BUFFER_LOCK_SHIFT = 3

}

Angle of Arrival/Departure options bit fields.

Functions

bool RAIL_BLE_LockCteBuffer(RAIL_Handle_t railHandle, bool lock)

Lock/unlock the CTE buffer from the application's perspective.

bool RAIL_BLE_CteBufferIsLocked(RAIL_Handle_t railHandle)

Determine whether the CTE buffer is currently locked or not.

uint8_t RAIL_BLE_GetCteSampleOffset(RAIL_Handle_t railHandle)

Get the offset into CTE sample of CTE data.

uint32_t RAIL_BLE_GetCteSampleRate(RAIL_Handle_t railHandle)

Get the effective sample rate used by the ADC to capture the CTE samples.

RAIL_Status_t RAIL_BLE_ConfigAox(RAIL_Handle_t railHandle, const RAIL_BLE_AoxConfig_t *aoxConfig)

Configure Angle of Arrival/Departure (AoX) functionality.

RAIL_Status_t RAIL_BLE_InitCte(RAIL_Handle_t railHandle)

Perform one time initialization of AoX registers.

RAIL_Status_t RAIL_BLE_ConfigAoxAntenna(RAIL_Handle_t railHandle, RAIL_BLE_AoxAntennaConfig_t *antennaConfig)

Perform initialization of AoX antenna GPIO pins.

Macros

#define RAIL_BLE_AOX_ANTENNA_PIN_COUNT (6U)
The maximum number of GPIO pins used for AoX Antenna switching.

#define RAIL_BLE_AOX_OPTIONS_DO_SWITCH (0U)

#define RAIL_BLE_AOX_OPTIONS_TX_ENABLED (0U)

#define RAIL_BLE_AOX_OPTIONS_RX_ENABLED (0U)

#define RAIL_BLE_AOX_OPTIONS_LOCK_CTE_BUFFER_SHIFT RAIL_BLE_AOX_OPTIONS_DISABLE_BUFFER_LOCK_SHIFT

#define RAIL_BLE_AOX_OPTIONS_DISABLED (0U)
Disable the AoX feature.

#define RAIL_BLE_AOX_OPTIONS_SAMPLE_MODE (1U << RAIL_BLE_AOX_OPTIONS_SAMPLE_MODE_SHIFT)
Sets one of the two AoX sampling/switching modes: 1 us or 2 us window.

#define RAIL_BLE_AOX_OPTIONS_CONNLESS (1U << RAIL_BLE_AOX_OPTIONS_CONNLESS_SHIFT)
Enables connectionless AoX Rx packets.

#define RAIL_BLE_AOX_OPTIONS_CONN (1U << RAIL_BLE_AOX_OPTIONS_CONN_SHIFT)
Enables connection based AoX Rx packets.

#define RAIL_BLE_AOX_OPTIONS_DISABLE_BUFFER_LOCK (1U << RAIL_BLE_AOX_OPTIONS_DISABLE_BUFFER_LOCK_SHIFT)
Disables CTE buffer lock.

#define RAIL_BLE_AOX_OPTIONS_ENABLED (RAIL_BLE_AOX_OPTIONS_CONN | RAIL_BLE_AOX_OPTIONS_CONNLESS)
Enables connection based or connectionless AoX Rx packets.

Enumeration Documentation

RAIL_BLE_AoxOptions_t

```
RAIL_BLE_AoxOptions_t
```

Angle of Arrival/Departure options bit fields.

Enumerator	
RAIL_BLE_AOX_OPTIONS_SAMPLE_MODE_SHIFT	Shift position of RAIL_BLE_AOX_OPTIONS_SAMPLE_MODE bit.
RAIL_BLE_AOX_OPTIONS_CONNLESS_SHIFT	Shift position of RAIL_BLE_AOX_OPTIONS_CONNLESS bit.
RAIL_BLE_AOX_OPTIONS_CONN_SHIFT	Shift position of RAIL_BLE_AOX_OPTIONS_CONN bit.
RAIL_BLE_AOX_OPTIONS_DISABLE_BUFFER_LOCK_SHIFT	Shift position of RAIL_BLE_AOX_OPTIONS_DISABLE_BUFFER_LOCK bit.

Definition at line 634 of file protocol/ble/rail_ble.h

Function Documentation

RAIL_BLE_LockCteBuffer

```
bool RAIL_BLE_LockCteBuffer (RAIL_Handle_t railHandle, bool lock)
```

Lock/unlock the CTE buffer from the application's perspective.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	lock	Lock the CTE buffer if true and unlock it if false.

The radio will write to the buffer only if the bit is NOT set at the beginning of the sampling period. The radio will set the bit once the sampling period starts to indicate that some CTE data has been collected, which will not be overwritten during the next sampling period, unless the buffer is unlocked by the application.

Returns

- True if the CTE buffer is locked after the call, otherwise false.

Definition at line `763` of file `protocol/ble/rail_ble.h`

RAIL_BLE_CteBufferIsLocked

```
bool RAIL_BLE_CteBufferIsLocked (RAIL_Handle_t railHandle)
```

Determine whether the CTE buffer is currently locked or not.

Parameters

[in]	railHandle	A handle for RAIL instance.

Returns

- True if CTE buffer is locked and false otherwise.

Definition at line `771` of file `protocol/ble/rail_ble.h`

RAIL_BLE_GetCteSampleOffset

```
uint8_t RAIL_BLE_GetCteSampleOffset (RAIL_Handle_t railHandle)
```

Get the offset into CTE sample of CTE data.

Parameters

[in]	railHandle	A handle for RAIL instance.

Returns

- The offset of CTE data in a CTE sample in bytes. On unsupported platforms this returns 0.

Definition at line `780` of file `protocol/ble/rail_ble.h`

RAIL_BLE_GetCteSampleRate

```
uint32_t RAIL_BLE_GetCteSampleRate (RAIL_Handle_t railHandle)
```

Get the effective sample rate used by the ADC to capture the CTE samples.

Parameters

[in]	railHandle	A handle for RAIL instance.

Returns

- The actual sample rate used to capture the CTE in samples per second. On unsupported platforms this returns 0.

Definition at line 789 of file protocol/ble/rail_ble.h

RAIL_BLE_ConfigAox

> RAIL_Status_t RAIL_BLE_ConfigAox (RAIL_Handle_t railHandle, const RAIL_BLE_AoxConfig_t *aoxConfig)

Configure Angle of Arrival/Departure (AoX) functionality.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	aoxConfig	Configuration options for AoX

AoX is a method of radio localization which infers angle of arrival/departure of the signal based on different phases of the raw I/Q signal from different antennas by controlling external RF switch during the continuous tone extension (CTE). Connection based AoX packets are different than normal BLE packets in that they have 3 header bytes instead of 2 and they have CTE appended after the payload's CRC. 3rd byte or CTE info contains CTE length. Connectionless AoX packets have 2 header bytes and CTE info is part of the payload. AoX is supported on EFR32XG12/13/14 only on legacy 1Mbps BLE PHY. Note that calling RAIL_GetRadioEntropy during AoX reception may break receiving packets.

Returns

- RAIL_Status_t indicating success or failure of the call.

Definition at line 808 of file protocol/ble/rail_ble.h

RAIL_BLE_InitCte

> RAIL_Status_t RAIL_BLE_InitCte (RAIL_Handle_t railHandle)

Perform one time initialization of AoX registers.

Parameters

[in]	railHandle	A RAIL instance handle.

This function must be called before RAIL_BLE_ConfigAox and before configuring the BLE PHY.

Returns

- RAIL_Status_t indicating success or failure of the call.

Definition at line 818 of file protocol/ble/rail_ble.h

RAIL_BLE_ConfigAoxAntenna

> RAIL_Status_t RAIL_BLE_ConfigAoxAntenna (RAIL_Handle_t railHandle, RAIL_BLE_AoxAntennaConfig_t *antennaConfig)

Perform initialization of AoX antenna GPIO pins.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	antennaConfig	structure to hold the set of ports and pins to configure Antenna pins for AoX Antenna switching.

This function must be called before calls to RAIL_BLE_InitCte and RAIL_BLE_ConfigAox, and before configuring the BLE PHY, else a RAIL_STATUS_INVALID_CALL is returned.

If user configures more pins, i.e., antCount in RAIL_BLE_AoxAntennaConfig_t, than allowed RAIL_BLE_AOX_ANTENNA_PIN_COUNT, then the API returns RAIL_STATUS_INVALID_PARAMETER.

If user configures lesser than or equal to number of pins allowed by RAIL_BLE_AOX_ANTENNA_PIN_COUNT, then the requested number of pins are configured and RAIL_STATUS_NO_ERROR is returned.

If AoX antenna switching is inactive, non-AoX transmits and receives will occur on the first antenna specified by the antenna pattern or on the default antenna if no antenna pattern is provided.

Returns

- RAIL_Status_t indicating success or failure of the call.

Definition at line 844 of file protocol/ble/rail_ble.h

Macro Definition Documentation

RAIL_BLE_AOX_ANTENNA_PIN_COUNT

```
#define RAIL_BLE_AOX_ANTENNA_PIN_COUNT
```

Value:

```
(6U)
```

The maximum number of GPIO pins used for AoX Antenna switching.

If the user configures more pins using RAIL_BLE_ConfigAoxAntenna than allowed RAIL_BLE_AOX_ANTENNA_PIN_COUNT, then RAIL_STATUS_INVALID_PARAMETER status will be returned.

RAIL_STATUS_INVALID_CALL is returned if : RAIL_BLE_AOX_ANTENNA_PIN_COUNT is set to 0 or The user configures no pins.

The maximum value RAIL_BLE_AOX_ANTENNA_PIN_COUNT can take depends on number of Antenna route pins , a chip provides. For EFR32XG22, the maximum value of RAIL_BLE_AOX_ANTENNA_PIN_COUNT is 6. If the user configures fewer pins than RAIL_BLE_AOX_ANTENNA_PIN_COUNT, then only number of pins asked by user will be configured with RAIL_STATUS_NO_ERROR.

Definition at line 628 of file protocol/ble/rail_ble.h

RAIL_BLE_AOX_OPTIONS_DO_SWITCH

```
#define RAIL_BLE_AOX_OPTIONS_DO_SWITCH
```

Value:

```
(0U)
```

DeprecatedObsolete AOX option

Definition at line 648 of file protocol/ble/rail_ble.h

RAIL_BLE_AOX_OPTIONS_TX_ENABLED

```
#define RAIL_BLE_AOX_OPTIONS_TX_ENABLED
```

Value:

```
(0U)
```

DeprecatedObsolete AOX option

Definition at line `652` of file `protocol/ble/rail_ble.h`

RAIL_BLE_AOX_OPTIONS_RX_ENABLED

```
#define RAIL_BLE_AOX_OPTIONS_RX_ENABLED
```

Value:

```
(0U)
```

DeprecatedObsolete AOX option

Definition at line `656` of file `protocol/ble/rail_ble.h`

RAIL_BLE_AOX_OPTIONS_LOCK_CTE_BUFFER_SHIFT

```
#define RAIL_BLE_AOX_OPTIONS_LOCK_CTE_BUFFER_SHIFT
```

Value:

```
RAIL_BLE_AOX_OPTIONS_DISABLE_BUFFER_LOCK_SHIFT
```

DeprecatedPlease use RAIL_BLE_AOX_OPTIONS_DISABLE_BUFFER_LOCK_SHIFT instead.

Definition at line `660` of file `protocol/ble/rail_ble.h`

RAIL_BLE_AOX_OPTIONS_DISABLED

```
#define RAIL_BLE_AOX_OPTIONS_DISABLED
```

Value:

```
(0U)
```

Disable the AoX feature.

Definition at line `665` of file `protocol/ble/rail_ble.h`

RAIL_BLE_AOX_OPTIONS_SAMPLE_MODE

```
#define RAIL_BLE_AOX_OPTIONS_SAMPLE_MODE
```

Value:

```
(1U << RAIL_BLE_AOX_OPTIONS_SAMPLE_MODE_SHIFT)
```

Sets one of the two AoX sampling/switching modes: 1 us or 2 us window.

Definition at line 669 of file protocol/ble/rail_ble.h

RAIL_BLE_AOX_OPTIONS_CONNLESS

```
#define RAIL_BLE_AOX_OPTIONS_CONNLESS
```

Value:

```
(1U << RAIL_BLE_AOX_OPTIONS_CONNLESS_SHIFT)
```

Enables connectionless AoX Rx packets.

Definition at line 673 of file protocol/ble/rail_ble.h

RAIL_BLE_AOX_OPTIONS_CONN

```
#define RAIL_BLE_AOX_OPTIONS_CONN
```

Value:

```
(1U << RAIL_BLE_AOX_OPTIONS_CONN_SHIFT)
```

Enables connection based AoX Rx packets.

Definition at line 677 of file protocol/ble/rail_ble.h

RAIL_BLE_AOX_OPTIONS_DISABLE_BUFFER_LOCK

```
#define RAIL_BLE_AOX_OPTIONS_DISABLE_BUFFER_LOCK
```

Value:

```
(1U << RAIL_BLE_AOX_OPTIONS_DISABLE_BUFFER_LOCK_SHIFT)
```

Disables CTE buffer lock.

Definition at line 681 of file protocol/ble/rail_ble.h

RAIL_BLE_AOX_OPTIONS_ENABLED

```
#define RAIL_BLE_AOX_OPTIONS_ENABLED
```

Value:

```
(RAIL_BLE_AOX_OPTIONS_CONN | RAIL_BLE_AOX_OPTIONS_CONNLESS)
```

Enables connection based or connectionless AoX Rx packets.

Definition at line 685 of file protocol/ble/rail_ble.h

RAIL_BLE_AoxConfig_t

Contains arguments for RAIL_BLE_ConfigAox function.

Public Attributes

RAIL_BLE_AoxOptions_t	aoxOptions See RAIL_BLE_AOX_OPTIONS_* for bitfield defines for different AoX features.
uint16_t	cteBuffSize Size of the raw AoX CTE (continuous tone extension) data capture buffer in bytes.
uint32_t *	cteBuffAddr Address to where the received CTE is written.
uint8_t *	antArrayAddr Address to first element of antenna pattern array.
uint8_t	antArraySize Size of the antenna pattern array.

Public Attribute Documentation

aoxOptions

> RAIL_BLE_AoxOptions_t RAIL_BLE_AoxConfig_t::aoxOptions

See RAIL_BLE_AOX_OPTIONS_* for bitfield defines for different AoX features.

Definition at line 695 of file protocol/ble/rail_ble.h

cteBuffSize

> uint16_t RAIL_BLE_AoxConfig_t::cteBuffSize

Size of the raw AoX CTE (continuous tone extension) data capture buffer in bytes.

Note this value should be a multiple of 4 as each IQ sample requires 4 bytes.

Definition at line 701 of file protocol/ble/rail_ble.h

cteBuffAddr

> uint32_t* RAIL_BLE_AoxConfig_t::cteBuffAddr

Address to where the received CTE is written.

Buffer must be 32-bit aligned.

Definition at line 706 of file protocol/ble/rail_ble.h

antArrayAddr

```
uint8_t* RAIL_BLE_AoxConfig_t::antArrayAddr
```

Address to first element of antenna pattern array.

Array must be in RAM. Each element of the array contains an antenna number. The switching pattern is defined by the order of antennas in this array.

Definition at line `712` of file `protocol/ble/rail_ble.h`

antArraySize

```
uint8_t RAIL_BLE_AoxConfig_t::antArraySize
```

Size of the antenna pattern array.

Definition at line `716` of file `protocol/ble/rail_ble.h`

RAIL_BLE_AoxAntennaPortPins_t

Contains elements of RAIL_BLE_AoxAntennaConfig_t struct.

Public Attributes

uint8_t	antPort The port which is used for AoX antenna switching.
uint8_t	antPin The pin which is used for AoX antenna switching.

Public Attribute Documentation

antPort

```
uint8_t RAIL_BLE_AoxAntennaPortPins_t::antPort
```

The port which is used for AoX antenna switching.

Definition at line 727 of file protocol/ble/rail_ble.h

antPin

```
uint8_t RAIL_BLE_AoxAntennaPortPins_t::antPin
```

The pin which is used for AoX antenna switching.

Definition at line 731 of file protocol/ble/rail_ble.h

RAIL_BLE_AoxAntennaConfig_t

Contains arguments for RAIL_BLE_ConfigAoxAntenna function for EFR32XG22.

Public Attributes

RAIL_BLE_AoxAntennaPortPins_t *	antPortPin A pointer to an array containing struct of port and pin used for AoX antenna switching.
uint8_t	antCount Number of antenna pins to be configured.

Public Attribute Documentation

antPortPin

```
RAIL_BLE_AoxAntennaPortPins_t* RAIL_BLE_AoxAntennaConfig_t::antPortPin
```

A pointer to an array containing struct of port and pin used for AoX antenna switching.

Definition at line 744 of file protocol/ble/rail_ble.h

antCount

```
uint8_t RAIL_BLE_AoxAntennaConfig_t::antCount
```

Number of antenna pins to be configured.

Definition at line 748 of file protocol/ble/rail_ble.h

BLE Radio Configurations

BLE Radio Configurations

Radio configurations for the RAIL BLE Accelerator.

These radio configurations are used to configure BLE when a function such as RAIL_BLE_ConfigPhy1MbpsViterbi() is called. Each radio configuration listed below is compiled into the RAIL library as a weak symbol that will take into account per-die defaults. If the board configuration in use has different settings than the default, such as a different radio subsystem clock frequency, these radio configurations can be overriden to account for those settings.

Variables

const RAIL_ChannelConfig_t *const	RAIL_BLE_Phy1Mbps Default PHY to use for BLE 1M non-Viterbi.
const RAIL_ChannelConfig_t *const	RAIL_BLE_Phy2Mbps Default PHY to use for BLE 2M non-Viterbi.
const RAIL_ChannelConfig_t *const	RAIL_BLE_Phy1MbpsViterbi Default PHY to use for BLE 1M Viterbi.
const RAIL_ChannelConfig_t *const	RAIL_BLE_Phy2MbpsViterbi Default PHY to use for BLE 2M Viterbi.
const RAIL_ChannelConfig_t *const	RAIL_BLE_Phy2MbpsAox PHY to use for BLE 2M with AoX functionality.
const RAIL_ChannelConfig_t *const	RAIL_BLE_Phy125kbps Default PHY to use for BLE Coded 125kbps.
const RAIL_ChannelConfig_t *const	RAIL_BLE_Phy500kbps Default PHY to use for BLE Coded 500kbps.
const RAIL_ChannelConfig_t *const	RAIL_BLE_PhySimulscan Default PHY to use for BLE Simulscan.
const RAIL_ChannelConfig_t *const	RAIL_BLE_PhyQuuppa Default 1Mbps Quuppa PHY.

Variable Documentation

RAIL_BLE_Phy1Mbps

```
const RAIL_ChannelConfig_t* const RAIL_BLE_Phy1Mbps
```

Default PHY to use for BLE 1M non-Viterbi.

Will be NULL if RAIL_BLE_SUPPORTS_1MBPS_NON_VITERBI is 0.

Definition at line 189 of file protocol/ble/rail_ble.h

RAIL_BLE_Phy2Mbps

```
const RAIL_ChannelConfig_t* const RAIL_BLE_Phy2Mbps
```

Default PHY to use for BLE 2M non-Viterbi.

Will be NULL if RAIL_BLE_SUPPORTS_2MBPS_NON_VITERBI is 0.

Definition at line 195 of file protocol/ble/rail_ble.h

RAIL_BLE_Phy1MbpsViterbi

```
const RAIL_ChannelConfig_t* const RAIL_BLE_Phy1MbpsViterbi
```

Default PHY to use for BLE 1M Viterbi.

Will be NULL if RAIL_BLE_SUPPORTS_1MBPS_VITERBI is 0.

Definition at line 201 of file protocol/ble/rail_ble.h

RAIL_BLE_Phy2MbpsViterbi

```
const RAIL_ChannelConfig_t* const RAIL_BLE_Phy2MbpsViterbi
```

Default PHY to use for BLE 2M Viterbi.

Will be NULL if RAIL_BLE_SUPPORTS_2MBPS_VITERBI is 0.

Definition at line 207 of file protocol/ble/rail_ble.h

RAIL_BLE_Phy2MbpsAox

```
const RAIL_ChannelConfig_t* const RAIL_BLE_Phy2MbpsAox
```

PHY to use for BLE 2M with AoX functionality.

Will be NULL if either RAIL_BLE_SUPPORTS_2MBPS_VITERBI or RAIL_BLE_SUPPORTS_AOX is 0.

Definition at line 229 of file protocol/ble/rail_ble.h

RAIL_BLE_Phy125kbps

```
const RAIL_ChannelConfig_t* const RAIL_BLE_Phy125kbps
```

Default PHY to use for BLE Coded 125kbps.

Will be NULL if RAIL_BLE_SUPPORTS_CODED_PHY is 0. This PHY can receive on both 125kbps and 500kbps BLE Coded, but will only transmit at 125kbps.

Definition at line 236 of file protocol/ble/rail_ble.h

RAIL_BLE_Phy500kbps

```
const RAIL_ChannelConfig_t* const RAIL_BLE_Phy500kbps
```

Default PHY to use for BLE Coded 500kbps.

Will be NULL if RAIL_BLE_SUPPORTS_CODED_PHY is 0. This PHY can receive on both 125kbps and 500kbps BLE Coded, but will only transmit at 125kbps.

Definition at line 243 of file protocol/ble/rail_ble.h

RAIL_BLE_PhySimulscan

```
const RAIL_ChannelConfig_t* const RAIL_BLE_PhySimulscan
```

Default PHY to use for BLE Simulscan.

Will be NULL if RAIL_BLE_SUPPORTS_SIMULSCAN_PHY is 0. This PHY can receive on 1Mbps as well as 125kbps and 500kbps BLE Coded, but will only transmit at 1Mbps.

Definition at line 250 of file protocol/ble/rail_ble.h

RAIL_BLE_PhyQuuppa

```
const RAIL_ChannelConfig_t* const RAIL_BLE_PhyQuuppa
```

Default 1Mbps Quuppa PHY.

Will be NULL if RAIL_BLE_SUPPORTS_QUUPPA is 0.

Definition at line 256 of file protocol/ble/rail_ble.h

BLE TX Channel Hopping

BLE TX Channel Hopping

```
// Configuration to send one additional packet
static RAIL_BLE_TxChannelHoppingConfigEntry_t entry[1];
static uint32_t buffer[BUFFER_SIZE];
static RAIL_BLE_TxRepeatConfig_t repeat = {
  .iterations = 1,
  .repeatOptions = RAIL_TX_REPEAT_OPTION_HOP,
  .delayOrHop.channelHopping = {
    .buffer = buffer,
    .bufferLength = BUFFER_SIZE,
    .numberOfChannels = 1,
    .entries = &entry[0],
  },
};

// Send a normal packet on the current channel, then a packet on a new channel
int bleSendThenAdvertise(uint8_t *firstPacket, uint8_t *secondPacket)
{
  // Load both packets into the FIFO
  RAIL_WriteTxFifo(railHandle, firstPacket, FIRST_PACKET_LEN, true);
  RAIL_WriteTxFifo(railHandle, secondPacket, SECOND_PACKET_LEN, false);

  // Configure a 300 us turnaround between transmits
  entry[0].delayMode = RAIL_CHANNEL_HOPPING_DELAY_MODE_STATIC;
  entry[0].delay = 300; // microseconds

  // Use default advertising parameters
  entry[0].disableWhitening = false;
  entry[0].crcInit = 0x00555555;
  entry[0].accessAddress = 0x8E89BED6;

  // Transmit the repeated packet on the first advertising channel
  entry[0].phy = RAIL_BLE_1Mbps;
  entry[0].railChannel = 0;
  entry[0].logicalChannel = 37;

 // Configure repeated transmit in RAIL, then transmit, sending both packets
 RAIL_BLE_SetNextTxRepeat(railHandle, &repeat);
 RAIL_StartTx(railHandle, currentChannel, RAIL_TX_OPTIONS_DEFAULT, NULL);
}
```

Modules

RAIL_BLE_TxChannelHoppingConfigEntry_t

RAIL_BLE_TxChannelHoppingConfig_t

RAIL_BLE_TxRepeatConfig_t

Functions

RAIL_Status_t RAIL_BLE_SetNextTxRepeat(RAIL_Handle_t railHandle, const RAIL_BLE_TxRepeatConfig_t *repeatConfig)
Set up automatic repeated transmits after the next transmit.

Function Documentation

RAIL_BLE_SetNextTxRepeat

> RAIL_Status_t RAIL_BLE_SetNextTxRepeat (RAIL_Handle_t railHandle, const RAIL_BLE_TxRepeatConfig_t *repeatConfig)

Set up automatic repeated transmits after the next transmit.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	repeatConfig	The configuration structure for repeated transmits.

Returns

- Status code indicating a success of the function call.

Repeated transmits will occur after an application-initiated transmit caused by calling one of the Packet Transmit APIs. The repetition will only occur after the first application-initiated transmit after this function is called. Future repeated transmits must be requested by calling this function again.

Each repeated transmit that occurs will have full Packet Trace (PTI) information and will receive events such as RAIL_EVENT_TX_PACKET_SENT as normal.

If a TX error occurs during the repetition, the process will abort and the TX error transition from RAIL_SetTxTransitions will be used. If the repetition completes successfully, the TX success transition from RAIL_SetTxTransitions will be used.

Any call to RAIL_Idle or RAIL_StopTx will clear the pending repeated transmits. The state will also be cleared by another call to this function. To clear the repeated transmits before they've started without stopping other radio actions, call this function with a RAIL_BLE_TxRepeatConfig_t::iterations count of 0. A DMP switch will clear this state only if the initial transmit triggering the repeated transmits has started.

The application is responsible for populating the transmit data to be used by the repeated transmits via RAIL_SetTxFifo or RAIL_WriteTxFifo. Data will be transmitted from the TX FIFO. If the TX FIFO does not have sufficient data to transmit, a TX error and a RAIL_EVENT_TX_UNDERFLOW will occur. To avoid an underflow, the application should queue data to be transmitted as early as possible.

This function will fail to configure the repetition if a transmit of any kind is ongoing, including during the time between an initial transmit and the end of a previously-configured repetition.

Note

- This feature/API is not supported on the EFR32XG1 family of chips. Use the compile time symbol RAIL_SUPPORTS_TX_TO_TX or the runtime call RAIL_SupportsTxToTx() to check whether the platform supports this feature.

Definition at line `1649` of file `protocol/ble/rail_ble.h`

RAIL_BLE_TxChannelHoppingConfigEntry_t

Structure that represents one of the channels that is part of a RAIL_BLE_TxChannelHoppingConfig_t sequence of channels used in channel hopping.

Public Attributes

uint32_t	delay Idle time in microseconds to wait before hopping into the channel indicated by this entry.
RAIL_BLE_Phy_t	phy The BLE PHY to use for this hop's transmit.
uint8_t	logicalChannel The logical channel to use for this hop's transmit.
uint8_t	railChannel The channel number to be used for this hop's transmit.
bool	disableWhitening This can turn off the whitening engine and is useful for sending BLE test mode packets that don't have this turned on.
uint32_t	crcInit The value to use for CRC initialization.
uint32_t	accessAddress The access address to use for the connection.

Public Attribute Documentation

delay

```
uint32_t RAIL_BLE_TxChannelHoppingConfigEntry_t::delay
```

Idle time in microseconds to wait before hopping into the channel indicated by this entry.

Definition at line 1502 of file protocol/ble/rail_ble.h

phy

```
RAIL_BLE_Phy_t RAIL_BLE_TxChannelHoppingConfigEntry_t::phy
```

The BLE PHY to use for this hop's transmit.

Definition at line 1506 of file protocol/ble/rail_ble.h

logicalChannel

```
uint8_t RAIL_BLE_TxChannelHoppingConfigEntry_t::logicalChannel
```

The logical channel to use for this hop's transmit.

The whitener will be reinitialized if used.

Definition at line 1511 of file protocol/ble/rail_ble.h

railChannel

```
uint8_t RAIL_BLE_TxChannelHoppingConfigEntry_t::railChannel
```

The channel number to be used for this hop's transmit.

If this is an invalid channel for the chosen PHY, the call to RAIL_SetNextTxRepeat() will fail.

Definition at line 1517 of file protocol/ble/rail_ble.h

disableWhitening

```
bool RAIL_BLE_TxChannelHoppingConfigEntry_t::disableWhitening
```

This can turn off the whitening engine and is useful for sending BLE test mode packets that don't have this turned on.

Definition at line 1522 of file protocol/ble/rail_ble.h

crcInit

```
uint32_t RAIL_BLE_TxChannelHoppingConfigEntry_t::crcInit
```

The value to use for CRC initialization.

Definition at line 1526 of file protocol/ble/rail_ble.h

accessAddress

```
uint32_t RAIL_BLE_TxChannelHoppingConfigEntry_t::accessAddress
```

The access address to use for the connection.

Definition at line 1530 of file protocol/ble/rail_ble.h

RAIL_BLE_TxChannelHoppingConfig_t

Wrapper struct that will contain the sequence of RAIL_BLE_TxChannelHoppingConfigEntry_t that represents the channel sequence to use during TX Channel Hopping.

Public Attributes

uint32_t *	buffer Pointer to contiguous global read-write memory that will be used by RAIL to store channel hopping information throughout its operation.
uint16_t	bufferLength This parameter must be set to the length of the buffer array.
uint8_t	numberOfChannels The number of channels that is in the channel hopping sequence.
uint8_t	reserved Pad bytes reserved for future use and currently ignored.
RAIL_BLE_TxChannelHoppingConfigEntry_t *	entries A pointer to the first element of an array of RAIL_BLE_TxChannelHoppingConfigEntry_t that represents the channels used during channel hopping.

Public Attribute Documentation

buffer

```
uint32_t* RAIL_BLE_TxChannelHoppingConfig_t::buffer
```

Pointer to contiguous global read-write memory that will be used by RAIL to store channel hopping information throughout its operation.

It need not be initialized and applications should never write data anywhere in this buffer.

Definition at line 1546 of file protocol/ble/rail_ble.h

bufferLength

```
uint16_t RAIL_BLE_TxChannelHoppingConfig_t::bufferLength
```

This parameter must be set to the length of the buffer array.

This way, during configuration, the software can confirm it's writing within the range of the buffer. The configuration API will return an error if bufferLength is insufficient.

Definition at line 1553 of file protocol/ble/rail_ble.h

numberOfChannels

uint8_t RAIL_BLE_TxChannelHoppingConfig_t::numberOfChannels

The number of channels that is in the channel hopping sequence.

Definition at line 1555 of file protocol/ble/rail_ble.h

reserved

uint8_t RAIL_BLE_TxChannelHoppingConfig_t::reserved

Pad bytes reserved for future use and currently ignored.

Definition at line 1559 of file protocol/ble/rail_ble.h

entries

RAIL_BLE_TxChannelHoppingConfigEntry_t* RAIL_BLE_TxChannelHoppingConfig_t::entries

A pointer to the first element of an array of RAIL_BLE_TxChannelHoppingConfigEntry_t that represents the channels used during channel hopping.

The length of this array must be numberOfChannels.

Definition at line 1566 of file protocol/ble/rail_ble.h

RAIL_BLE_TxRepeatConfig_t

A configuration structure for repeated transmits.

Public Attributes

uint16_t	iterations The number of repeated transmits to run.
RAIL_TxRepeatOptions_t	repeatOptions Repeat option(s) to apply.
RAIL_TransitionTime_t	delay When RAIL_TX_REPEAT_OPTION_HOP is not set, this specifies the delay time between each repeated transmit.
RAIL_BLE_TxChannelHoppingConfig_t	channelHopping When RAIL_TX_REPEAT_OPTION_HOP is set, this specifies the channel hopping configuration to use when hopping between repeated transmits.
union RAIL_BLE_TxRepeatConfig_t::@1	delayOrHop Per-repeat delay or hopping configuration, depending on repeatOptions.

Public Attribute Documentation

iterations

```
uint16_t RAIL_BLE_TxRepeatConfig_t::iterations
```

The number of repeated transmits to run.

A total of (iterations + 1) transmits will go on-air in the absence of errors.

Definition at line 1577 of file protocol/ble/rail_ble.h

repeatOptions

```
RAIL_TxRepeatOptions_t RAIL_BLE_TxRepeatConfig_t::repeatOptions
```

Repeat option(s) to apply.

Definition at line 1581 of file protocol/ble/rail_ble.h

delay

```
RAIL_TransitionTime_t RAIL_BLE_TxRepeatConfig_t::delay
```

When RAIL_TX_REPEAT_OPTION_HOP is not set, this specifies the delay time between each repeated transmit.

Specify RAIL_TRANSITION_TIME_KEEP to use the current RAIL_StateTiming_t::txToTx transition time setting.

Definition at line 1592 of file protocol/ble/rail_ble.h

channelHopping

> RAIL_BLE_TxChannelHoppingConfig_t RAIL_BLE_TxRepeatConfig_t::channelHopping

When RAIL_TX_REPEAT_OPTION_HOP is set, this specifies the channel hopping configuration to use when hopping between repeated transmits.

Per-hop delays are configured within each RAIL_BLE_TxChannelHoppingConfigEntry_t::delay rather than this union's delay field.

Definition at line 1600 of file protocol/ble/rail_ble.h

delayOrHop

> union RAIL_BLE_TxRepeatConfig_t::@1 RAIL_BLE_TxRepeatConfig_t::delayOrHop

Per-repeat delay or hopping configuration, depending on repeatOptions.

Definition at line 1601 of file protocol/ble/rail_ble.h

IEEE 802.15.4

IEEE 802.15.4

IEEE 802.15.4 configuration routines.

The functions in this group configure RAIL IEEE 802.15.4 hardware acceleration which includes IEEE 802.15.4 format filtering, address filtering, ACKing, and filtering based on the frame type.

To configure IEEE 802.15.4 functionality, the application must first set up a RAIL instance with RAIL_Init() and other setup functions. Instead of RAIL_ConfigChannels(), however, an application may use RAIL_IEEE802154_Config2p4GHzRadio() to set up the official IEEE 2.4 GHz 802.15.4 PHY. This configuration is shown below.

802.15.4 defines its macAckWaitDuration from the end of the transmitted packet to complete reception of the ACK. RAIL's ackTimeout only covers sync word detection of the ACK. Therefore, subtract the ACK's PHY header and payload time to get RAIL's ackTimeout setting. For 2.4 GHz OQPSK, macAckWaitDuration is specified as 54 symbols; subtracting 2-symbol PHY header and 10-symbol payload yields a RAIL ackTimeout of 42 symbols or 672 microseconds at 16 microseconds/symbol.

```
static RAIL_Handle_t railHandle = NULL; // Initialized somewhere else.

static const RAIL_IEEE802154_Config_t rail154Config = {
  .addresses = NULL,
  .ackConfig = {
    .enable = true,     // Turn on auto ACK for IEEE 802.15.4.
    .ackTimeout = 672,  // See note above: 54-12 sym * 16 us/sym = 672 us.
    .rxTransitions = {
      .success = RAIL_RF_STATE_RX,  // Return to RX after ACK processing
      .error = RAIL_RF_STATE_RX,    // Ignored
    },
    .txTransitions = {
      .success = RAIL_RF_STATE_RX,  // Return to RX after ACK processing
      .error = RAIL_RF_STATE_RX,    // Ignored
    },
  },
  .timings = {
    .idleToRx = 100,
    .idleToTx = 100,
    .rxToTx = 192,    // 12 symbols * 16 us/symbol = 192 us
    .txToRx = 192,    // 12 symbols * 16 us/symbol = 192 us
    .rxSearchTimeout = 0, // Not used
    .txToRxSearchTimeout = 0, // Not used
  },
  .framesMask = RAIL_IEEE802154_ACCEPT_STANDARD_FRAMES,
  .promiscuousMode = false,  // Enable format and address filtering.
  .isPanCoordinator = false,
  .defaultFramePendingInOutgoingAcks = false,
};

void config154(void)
{
  // Configure the radio and channels for 2.4 GHz IEEE 802.15.4.
  RAIL_IEEE802154_Config2p4GHzRadio(railHandle);
  // Initialize the IEEE 802.15.4 configuration using the static configuration above.
  RAIL_IEEE802154_Init(railHandle, &rail154Config);
}
```

To configure address filtering, call RAIL_IEEE802154_SetAddresses() with a structure containing all addresses or call the individual RAIL_IEEE802154_SetPanId(), RAIL_IEEE802154_SetShortAddress(), and RAIL_IEEE802154_SetLongAddress() APIs. RAIL supports RAIL_IEEE802154_MAX_ADDRESSES number of address pairs to receive packets from multiple IEEE 802.15.4 networks at the same time. Broadcast addresses are supported by default without any additional configuration so they do not consume one of these slots. If the application does not require all address pairs, be sure to set unused ones to the proper disabled value for each type. These can be found in the RAIL_IEEE802154_AddrConfig_t documentation. Below is an example of setting filtering for one set of addresses.

```
// PanID OTA value of 0x34 0x12.
// Short Address OTA byte order of 0x78 0x56.
// Long address with OTA byte order of 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88.

// Set up all addresses simultaneously.
RAIL_Status_t setup1(void)
{
  RAIL_IEEE802154_AddrConfig_t nodeAddress = {
    { 0x1234, 0xFFFF, 0xFFFF },
    { 0x5678, 0xFFFF, 0xFFFF },
    { { 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88 },
      { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
      { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 } }
  };
  return RAIL_IEEE802154_SetAddresses(railHandle, &nodeAddress);
}

// Alternatively, the addresses can be set up individually as follows:
RAIL_Status_t setup2(void)
{
  RAIL_Status_t status;
  const uint8_t longAddress[] = { 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88 };

  status = RAIL_IEEE802154_SetPanId(railHandle, 0x1234, 0);
  if (status != RAIL_STATUS_NO_ERROR) {
    return status
  }
  status = RAIL_IEEE802154_SetShortAddress(railHandle, 0x5678, 0);
  if (status != RAIL_STATUS_NO_ERROR) {
    return status
  }
  status = RAIL_IEEE802154_SetLongAddress(railHandle, longAddress, 0);
  if (status != RAIL_STATUS_NO_ERROR) {
    return status
  }

  return RAIL_STATUS_NO_ERROR;
}
```

Address filtering will be enabled except when in promiscuous mode, which can be set with RAIL_IEEE802154_SetPromiscuousMode(). The addresses may be changed at runtime. However, if you are receiving a packet while reconfiguring the address filters, you may get undesired behavior so it's safest to do this while not in receive.

Auto ACK is controlled by the ackConfig and timings fields passed to RAIL_IEEE802154_Init(). After initialization, they may be controlled using the normal Auto-ACK and State Transitions APIs. When in IEEE 802.15.4 mode, the ACK will generally have a 5 byte length, its Frame Type will be ACK, its Frame Version 0 (2003), and its Frame Pending bit will be false unless the RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND event is triggered in which case it will default to the RAIL_IEEE802154_Config_t::defaultFramePendingInOutgoingAcks setting. If the default Frame Pending setting is incorrect, the app must call RAIL_IEEE802154_ToggleFramePending (formerly RAIL_IEEE802154_SetFramePending) while handling the RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND event.

This event must be turned on by the user and will fire whenever a data request is being received so that the stack can determine if there is pending data. Note that if the default Frame Pending bit needs to be changed, it must be done quickly.

Otherwise, the ACK may already have been transmitted with the default setting. Check the return code of RAIL_IEEE802154_ToggleFramePending() to be sure that the bit was changed in time.

Transmit and receive operations are done using the standard RAIL APIs in IEEE 802.15.4 mode. To send packets using the correct CSMA configuration, use RAIL_CSMA_CONFIG_802_15_4_2003_2p4_GHz_OQPSK_CSMA define that can initialize the csmaConfig structure passed to RAIL_StartCcaCsmaTx().

Modules

RAIL_IEEE802154_Address_t

RAIL_IEEE802154_AddrConfig_t

RAIL_IEEE802154_Config_t

RAIL_IEEE802154_RxChannelSwitchingCfg_t

RAIL_IEEE802154_ModeSwitchPhr_t

IEEE 802.15.4 Radio Configurations

Enumerations

enum RAIL_IEEE802154_AddressLength_t {

RAIL_IEEE802154_ShortAddress = 2
RAIL_IEEE802154_LongAddress = 3

}

Different lengths that an 802.15.4 address can have.

enum RAIL_IEEE802154_PtiRadioConfig_t {

RAIL_IEEE802154_PTI_RADIO_CONFIG_2P4GHZ = 0x00U
RAIL_IEEE802154_PTI_RADIO_CONFIG_2P4GHZ_ANTDIV = 0x01U
RAIL_IEEE802154_PTI_RADIO_CONFIG_2P4GHZ_COEX = 0x02U
RAIL_IEEE802154_PTI_RADIO_CONFIG_2P4GHZ_ANTDIV_COEX = 0x03U
RAIL_IEEE802154_PTI_RADIO_CONFIG_2P4GHZ_FEM = 0x08U
RAIL_IEEE802154_PTI_RADIO_CONFIG_2P4GHZ_FEM_ANTDIV = 0x09U
RAIL_IEEE802154_PTI_RADIO_CONFIG_2P4GHZ_FEM_COEX = 0x0AU
RAIL_IEEE802154_PTI_RADIO_CONFIG_2P4GHZ_FEM_ANTDIV_COEX = 0x0BU
RAIL_IEEE802154_PTI_RADIO_CONFIG_863MHZ_GB868 = 0x85U
RAIL_IEEE802154_PTI_RADIO_CONFIG_915MHZ_GB868 = 0x86U
RAIL_IEEE802154_PTI_RADIO_CONFIG_915MHZ_R23_NA_EXT = 0x97U

}

802.15.4 PTI radio configuration mode

enum RAIL_IEEE802154_EOptions_t {

RAIL_IEEE802154_E_OPTION_GB868_SHIFT = 0
RAIL_IEEE802154_E_OPTION_ENH_ACK_SHIFT
RAIL_IEEE802154_E_OPTION_IMPLICIT_BROADCAST_SHIFT

}

802.15.4E-2012 options, in reality a bitmask.

enum RAIL_IEEE802154_GOptions_t {

RAIL_IEEE802154_G_OPTION_GB868_SHIFT = 0
RAIL_IEEE802154_G_OPTION_DYNFEC_SHIFT
RAIL_IEEE802154_G_OPTION_WISUN_MODESWITCH_SHIFT

}

802.15.4G-2012 options, in reality a bitmask.

enum RAIL_IEEE802154_CcaMode_t {

RAIL_IEEE802154_CCA_MODE_RSSI = 0
RAIL_IEEE802154_CCA_MODE_SIGNAL
RAIL_IEEE802154_CCA_MODE_SIGNAL_OR_RSSI
RAIL_IEEE802154_CCA_MODE_SIGNAL_AND_RSSI
RAIL_IEEE802154_CCA_MODE_ALWAYS_TRANSMIT
RAIL_IEEE802154_CCA_MODE_COUNT

}

Available CCA modes.

enum RAIL_IEEE802154_SignalIdentifierMode_t {

RAIL_IEEE802154_SIGNAL_IDENTIFIER_MODE_DISABLE = 0
RAIL_IEEE802154_SIGNAL_IDENTIFIER_MODE_154

}

Available Signal identifier modes.

Functions

RAIL_Status_t RAIL_IEEE802154_Init(RAIL_Handle_t railHandle, const RAIL_IEEE802154_Config_t *config)

Initialize RAIL for IEEE802.15.4 features.

RAIL_Status_t RAIL_IEEE802154_Config2p4GHzRadio(RAIL_Handle_t railHandle)

Configure the radio for 2.4 GHz 802.15.4 operation.

RAIL_Status_t RAIL_IEEE802154_Config2p4GHzRadioAntDiv(RAIL_Handle_t railHandle)

Configure the radio for 2.4 GHz 802.15.4 operation with antenna diversity.

RAIL_Status_t RAIL_IEEE802154_Config2p4GHzRadioAntDivCoex(RAIL_Handle_t railHandle)

Configure the radio for 2.4 GHz 802.15.4 operation with antenna diversity optimized for radio coexistence.

RAIL_Status_t RAIL_IEEE802154_Config2p4GHzRadioCoex(RAIL_Handle_t railHandle)

Configure the radio for 2.4 GHz 802.15.4 operation optimized for radio coexistence.

RAIL_Status_t RAIL_IEEE802154_Config2p4GHzRadioFem(RAIL_Handle_t railHandle)

Configure the radio for 2.4 GHz 802.15.4 operation with a front end module.

RAIL_Status_t RAIL_IEEE802154_Config2p4GHzRadioAntDivFem(RAIL_Handle_t railHandle)

Configure the radio for 2.4 GHz 802.15.4 operation with antenna diversity optimized for a front end module.

RAIL_Status_t RAIL_IEEE802154_Config2p4GHzRadioCoexFem(RAIL_Handle_t railHandle)

Configure the radio for 2.4 GHz 802.15.4 operation optimized for radio coexistence and a front end module.

RAIL_Status_t RAIL_IEEE802154_Config2p4GHzRadioAntDivCoexFem(RAIL_Handle_t railHandle)

Configure the radio for 2.4 GHz 802.15.4 operation with antenna diversity optimized for radio coexistence and a front end module.

RAIL_Status_t RAIL_IEEE802154_Config2p4GHzRadioCustom1(RAIL_Handle_t railHandle)

Configure the radio for 2.4 GHz 802.15.4 operation with custom settings.

RAIL_Status_t RAIL_IEEE802154_ConfigGB863MHzRadio(RAIL_Handle_t railHandle)

Configure the radio for SubGHz GB868 863 MHz 802.15.4 operation.

RAIL_Status_t RAIL_IEEE802154_ConfigGB915MHzRadio(RAIL_Handle_t railHandle)

Configure the radio for SubGHz GB868 915 MHz 802.15.4 operation.

RAIL_Status_t RAIL_IEEE802154_Deinit(RAIL_Handle_t railHandle)

De-initialize IEEE802.15.4 hardware acceleration.

bool RAIL_IEEE802154_IsEnabled(RAIL_Handle_t railHandle)

Return whether IEEE802.15.4 hardware acceleration is currently enabled.

RAIL_IEEE802154_PtiRadioConfig_t	RAIL_IEEE802154_GetPtiRadioConfig(RAIL_Handle_t railHandle) Return IEEE802.15.4 PTI radio config.
RAIL_Status_t	RAIL_IEEE802154_SetAddresses(RAIL_Handle_t railHandle, const RAIL_IEEE802154_AddrConfig_t *addresses) Configure the RAIL Address Filter for 802.15.4 filtering.
RAIL_Status_t	RAIL_IEEE802154_SetPanId(RAIL_Handle_t railHandle, uint16_t panId, uint8_t index) Set a PAN ID for 802.15.4 address filtering.
RAIL_Status_t	RAIL_IEEE802154_SetShortAddress(RAIL_Handle_t railHandle, uint16_t shortAddr, uint8_t index) Set a short address for 802.15.4 address filtering.
RAIL_Status_t	RAIL_IEEE802154_SetLongAddress(RAIL_Handle_t railHandle, const uint8_t *longAddr, uint8_t index) Set a long address for 802.15.4 address filtering.
RAIL_Status_t	RAIL_IEEE802154_SetPanCoordinator(RAIL_Handle_t railHandle, bool isPanCoordinator) Set whether the current node is a PAN coordinator.
RAIL_Status_t	RAIL_IEEE802154_SetPromiscuousMode(RAIL_Handle_t railHandle, bool enable) Set whether to enable 802.15.4 promiscuous mode.
RAIL_Status_t	RAIL_IEEE802154_ConfigEOptions(RAIL_Handle_t railHandle, RAIL_IEEE802154_EOptions_t mask, RAIL_IEEE802154_EOptions_t options) Configure certain 802.15.4E-2012 / 802.15.4-2015 Frame Version 2 features.
RAIL_Status_t	RAIL_IEEE802154_ConfigGOptions(RAIL_Handle_t railHandle, RAIL_IEEE802154_GOptions_t mask, RAIL_IEEE802154_GOptions_t options) Configure certain 802.15.4G-2012 / 802.15.4-2015 SUN PHY features (only for radio configurations designed accordingly).
RAIL_Status_t	RAIL_IEEE802154_ComputeChannelFromPhyModeId(RAIL_Handle_t railHandle, uint8_t newPhyModeId, uint16_t *pChannel) Compute channel to switch to given a targeted PhyMode ID in the context of Wi-SUN mode switching.
RAIL_Status_t	RAILCb_IEEE802154_IsModeSwitchNewChannelValid(uint32_t currentBaseFreq, uint8_t newPhyModeId, const RAIL_ChannelConfigEntry_t *configEntryNewPhyModeId, uint16_t *pChannel) Manage forbidden channels during mode switch.
RAIL_Status_t	RAIL_IEEE802154_AcceptFrames(RAIL_Handle_t railHandle, uint8_t framesMask) Set which 802.15.4 frame types to accept.
RAIL_Status_t	RAIL_IEEE802154_EnableEarlyFramePending(RAIL_Handle_t railHandle, bool enable) Enable early Frame Pending lookup event notification (RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND).
RAIL_Status_t	RAIL_IEEE802154_EnableDataFramePending(RAIL_Handle_t railHandle, bool enable) Enable Frame Pending lookup event notification (RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND) for MAC Data frames.
RAIL_Status_t	RAIL_IEEE802154_SetFramePending(RAIL_Handle_t railHandle) Change the Frame Pending bit on the outgoing legacy Immediate ACK from the default specified by RAIL_IEEE802154_Config_t::defaultFramePendingInOutgoingAcks.
RAIL_Status_t	RAIL_IEEE802154_GetAddress(RAIL_Handle_t railHandle, RAIL_IEEE802154_Address_t *pAddress) Get the source address of the incoming data request.
RAIL_Status_t	RAIL_IEEE802154_WriteEnhAck(RAIL_Handle_t railHandle, const uint8_t *ackData, uint8_t ackDataLen) Write the AutoACK FIFO for the next outgoing 802.15.4E Enhanced ACK.
RAIL_Status_t	RAIL_IEEE802154_SetRxToEnhAckTx(RAIL_Handle_t railHandle, RAIL_TransitionTime_t *pRxToEnhAckTx) Set a separate RX packet to TX state transition turnaround time for sending an Enhanced ACK.
uint8_t	RAIL_IEEE802154_ConvertRssiToLqi(uint8_t origLqi, int8_t rssiDbm) Convert RSSI into 802.15.4 Link Quality Indication (LQI) metric compatible with the Silicon Labs Zigbee stack.

uint8_t RAIL_IEEE802154_ConvertRssiToEd(int8_t rssiDbm)
Convert RSSI into 802.15.4 Energy Detection (ED) metric compatible with the Silicon Labs Zigbee stack.

RAIL_Status_t RAIL_IEEE802154_ConfigSignalIdentifier(RAIL_Handle_t railHandle, RAIL_IEEE802154_SignalIdentifierMode_t signalIdentifierMode)
Configure signal identifier for 802.15.4 signal detection.

RAIL_Status_t RAIL_IEEE802154_EnableSignalDetection(RAIL_Handle_t railHandle, bool enable)
Enable or disable signal identifier for 802.15.4 signal detection.

RAIL_Status_t RAIL_IEEE802154_ConfigCcaMode(RAIL_Handle_t railHandle, RAIL_IEEE802154_CcaMode_t ccaMode)
Set 802.15.4 CCA mode.

RAIL_Status_t RAIL_IEEE802154_ConfigRxChannelSwitching(RAIL_Handle_t railHandle, const RAIL_IEEE802154_RxChannelSwitchingCfg_t *pConfig)
Configure RX channel switching for 802.15.4.

Macros

#define RAIL_IEEE802154_MAX_ADDRESSES (3U)
The maximum number of allowed addresses of each type.

#define RAIL_IEEE802154_RX_CHANNEL_SWITCHING_BUF_BYTES (0U)
RX channel switching buffer size, in bytes.

#define RAIL_IEEE802154_RX_CHANNEL_SWITCHING_BUF_ALIGNMENT_TYPE uint32_t
Fixed-width type indicating the needed alignment for RX channel switching buffer.

#define RAIL_IEEE802154_RX_CHANNEL_SWITCHING_BUF_ALIGNMENT (sizeof(RAIL_IEEE802154_RX_CHANNEL_SWITCHING_BUF_ALIGNMENT_TYPE))
Alignment that is needed for RX channel switching buffer.

#define RAIL_IEEE802154_RX_CHANNEL_SWITCHING_NUM_CHANNELS (2U)
Maximum numbers of channels supported for RX channel switching.

#define RAIL_IEEE802154_E_OPTIONS_NONE 0UL
A value representing no options enabled.

#define RAIL_IEEE802154_E_OPTIONS_DEFAULT RAIL_IEEE802154_E_OPTIONS_NONE
All options disabled by default .

#define RAIL_IEEE802154_E_OPTION_GB868 (1UL << RAIL_IEEE802154_E_OPTION_GB868_SHIFT)
An option to enable/disable 802.15.4E-2012 features needed for GB868.

#define RAIL_IEEE802154_E_OPTION_ENH_ACK (1UL << RAIL_IEEE802154_E_OPTION_ENH_ACK_SHIFT)
An option to enable/disable 802.15.4E-2012 features needed for Enhanced ACKs.

#define RAIL_IEEE802154_E_OPTION_IMPLICIT_BROADCAST (1UL << RAIL_IEEE802154_E_OPTION_IMPLICIT_BROADCAST_SHIFT)
An option to enable/disable 802.15.4E-2012 macImplicitBroadcast feature.

#define RAIL_IEEE802154_E_OPTIONS_ALL 0xFFFFFFFFUL
A value representing all possible options.

#define RAIL_IEEE802154_G_OPTIONS_NONE 0UL
A value representing no options enabled.

#define RAIL_IEEE802154_G_OPTIONS_DEFAULT RAIL_IEEE802154_G_OPTIONS_NONE
All options disabled by default .

#define RAIL_IEEE802154_G_OPTION_GB868 (1UL << RAIL_IEEE802154_G_OPTION_GB868_SHIFT)
An option to enable/disable 802.15.4G-2012 features needed for GB868.

#define RAIL_IEEE802154_G_OPTION_DYNFEC (1UL << RAIL_IEEE802154_G_OPTION_DYNFEC_SHIFT)
An option to enable/disable 802.15.4G dynamic FEC feature (SUN FSK only).

#define RAIL_IEEE802154_G_OPTION_WISUN_MODESWITCH (1UL << RAIL_IEEE802154_G_OPTION_WISUN_MODESWITCH_SHIFT)
An option to enable/disable Wi-SUN Mode Switch feature.

#define RAIL_IEEE802154_G_OPTIONS_ALL 0xFFFFFFFFUL
A value representing all possible options.

#define RAIL_IEEE802154_ACCEPT_BEACON_FRAMES (0x01)
When receiving packets, accept 802.15.4 BEACON frame types.

#define RAIL_IEEE802154_ACCEPT_DATA_FRAMES (0x02)
When receiving packets, accept 802.15.4 DATA frame types.

#define RAIL_IEEE802154_ACCEPT_ACK_FRAMES (0x04)
When receiving packets, accept 802.15.4 ACK frame types.

#define RAIL_IEEE802154_ACCEPT_COMMAND_FRAMES (0x08)
When receiving packets, accept 802.15.4 COMMAND frame types.

#define RAIL_IEEE802154_ACCEPT_MULTIPURPOSE_FRAMES (0x20)
When receiving packets, accept 802.15.4-2015 Multipurpose frame types.

#define RAIL_IEEE802154_ACCEPT_STANDARD_FRAMES undefined
In standard operation, accept BEACON, DATA and COMMAND frames.

#define RAIL_IEEE802154_ToggleFramePending RAIL_IEEE802154_SetFramePending
Alternate naming for function RAIL_IEEE802154_SetFramePending to depict it is used for changing the default setting specified by RAIL_IEEE802154_Config_t::defaultFramePendingInOutgoingAcks in an outgoing ACK.

#define RAIL_IEEE802154_EnableSignalIdentifier RAIL_IEEE802154_EnableSignalDetection
Backward compatible name for the RAIL_IEEE802154_EnableSignalDetection API.

Enumeration Documentation

RAIL_IEEE802154_AddressLength_t

> RAIL_IEEE802154_AddressLength_t

Different lengths that an 802.15.4 address can have.

Enumerator	
RAIL_IEEE802154_ShortAddress	2 byte short address.
RAIL_IEEE802154_LongAddress	8 byte extended address.

Definition at line 194 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_PtiRadioConfig_t

> RAIL_IEEE802154_PtiRadioConfig_t

802.15.4 PTI radio configuration mode

Enumerator	
RAIL_IEEE802154_PTI_RADIO_CONFIG_2P4GHZ	Built-in 2.4 GHz 802.15.4 radio configuration.

RAIL_IEEE802154_PTI_RADIO_CONFIG_2P4GHZ_ANTDIV	Built-in 2.4 GHz 802.15.4 radio configuration with RX antenna diversity support.
RAIL_IEEE802154_PTI_RADIO_CONFIG_2P4GHZ_COEX	Built-in 2.4 GHz 802.15.4 radio configuration optimized for radio coexistence.
RAIL_IEEE802154_PTI_RADIO_CONFIG_2P4GHZ_ANTDIV_COEX	Built-in 2.4 GHz 802.15.4 radio configuration with RX antenna diversity support optimized for radio coexistence.
RAIL_IEEE802154_PTI_RADIO_CONFIG_2P4GHZ_FEM	Built-in 2.4 GHz 802.15.4 radio configuration optimized for front end modules.
RAIL_IEEE802154_PTI_RADIO_CONFIG_2P4GHZ_FEM_ANTDIV	Built-in 2.4 GHz 802.15.4 radio configuration with RX antenna diversity support optimized for front end modules.
RAIL_IEEE802154_PTI_RADIO_CONFIG_2P4GHZ_FEM_COEX	Built-in 2.4 GHz 802.15.4 radio configuration optimized for radio coexistence and front end modules.
RAIL_IEEE802154_PTI_RADIO_CONFIG_2P4GHZ_FEM_ANTDIV_COEX	Built-in 2.4 GHz 802.15.4 radio configuration with RX antenna diversity support optimized for radio coexistence and front end modules.
RAIL_IEEE802154_PTI_RADIO_CONFIG_863MHZ_GB868	Built-in 863 MHz GB868 802.15.4 radio configuration.
RAIL_IEEE802154_PTI_RADIO_CONFIG_915MHZ_GB868	Built-in 915 MHz GB868 802.15.4 radio configuration.
RAIL_IEEE802154_PTI_RADIO_CONFIG_915MHZ_R23_NA_EXT	External 915 MHz Zigbee R23 802.15.4 NA radio configuration.

Definition at line `736` of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_EOptions_t

> RAIL_IEEE802154_EOptions_t

802.15.4E-2012 options, in reality a bitmask.

Enumerator	
RAIL_IEEE802154_E_OPTION_GB868_SHIFT	Shift position of RAIL_IEEE802154_E_OPTION_GB868 bit.
RAIL_IEEE802154_E_OPTION_ENH_ACK_SHIFT	
RAIL_IEEE802154_E_OPTION_IMPLICIT_BROADCAST_SHIFT	

Definition at line `931` of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_GOptions_t

> RAIL_IEEE802154_GOptions_t

802.15.4G-2012 options, in reality a bitmask.

Enumerator	
RAIL_IEEE802154_G_OPTION_GB868_SHIFT	Shift position of RAIL_IEEE802154_G_OPTION_GB868 bit.
RAIL_IEEE802154_G_OPTION_DYNFEC_SHIFT	Shift position of RAIL_IEEE802154_G_OPTION_DYNFEC bit.
RAIL_IEEE802154_G_OPTION_WISUN_MODESWITCH_SHIFT	Shift position of RAIL_IEEE802154_G_OPTION_WISUN_MODESWITCH bit.

Definition at line `1046` of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_CcaMode_t

> RAIL_IEEE802154_CcaMode_t

Available CCA modes.

Enumerator	
RAIL_IEEE802154_CCA_MODE_RSSI	RSSI-based CCA.
RAIL_IEEE802154_CCA_MODE_SIGNAL	Signal Identifier-based CCA.
RAIL_IEEE802154_CCA_MODE_SIGNAL_OR_RSSI	RSSI or signal identifier-based CCA.
RAIL_IEEE802154_CCA_MODE_SIGNAL_AND_RSSI	RSSI and signal identifier-based CCA.
RAIL_IEEE802154_CCA_MODE_ALWAYS_TRANSMIT	ALOHA.
RAIL_IEEE802154_CCA_MODE_COUNT	Number of CCA modes.

Definition at line `1487` of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_SignalIdentifierMode_t

> RAIL_IEEE802154_SignalIdentifierMode_t

Available Signal identifier modes.

Enumerator	
RAIL_IEEE802154_SIGNAL_IDENTIFIER_MODE_DISABLE	
RAIL_IEEE802154_SIGNAL_IDENTIFIER_MODE_154	

Definition at line `1537` of file `protocol/ieee802154/rail_ieee802154.h`

Function Documentation

RAIL_IEEE802154_Init

> RAIL_Status_t RAIL_IEEE802154_Init (RAIL_Handle_t railHandle, const RAIL_IEEE802154_Config_t *config)

Initialize RAIL for IEEE802.15.4 features.

Parameters

[in]	railHandle	A handle of RAIL instance.
[in]	config	An IEEE802154 configuration structure.

Returns

- A status code indicating success of the function call.

This function calls the following RAIL functions to configure the radio for IEEE802.15.4 features.

Initializes the following:

- Enables IEEE802154 hardware acceleration
- Configures RAIL Auto ACK functionality
- Configures RAIL Address Filter for 802.15.4 address filtering

It saves having to call the following functions individually:

RAIL_ConfigAutoAck()
- RAIL_SetRxTransitions()
- RAIL_SetTxTransitions()
- RAIL_WriteAutoAckFifo()
- RAIL_SetStateTiming()
- RAIL_ConfigAddressFilter()
- RAIL_EnableAddressFilter()

It must be called before most of RAIL_IEEE802154_* function.

Definition at line 502 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_Config2p4GHzRadio

> RAIL_Status_t RAIL_IEEE802154_Config2p4GHzRadio (RAIL_Handle_t railHandle)

Configure the radio for 2.4 GHz 802.15.4 operation.

Parameters

[in]	railHandle	A handle of RAIL instance.

Returns

- A status code indicating success of the function call.

This initializes the radio for 2.4 GHz operation. It takes the place of calling RAIL_ConfigChannels. After this call, channels 11-26 will be available, giving the frequencies of those channels on channel page 0, as defined by IEEE 802.15.4-2011 section 8.1.2.2.

Note

- This call implicitly disables all RAIL_IEEE802154_GOptions_t.

Definition at line 518 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_Config2p4GHzRadioAntDiv

> RAIL_Status_t RAIL_IEEE802154_Config2p4GHzRadioAntDiv (RAIL_Handle_t railHandle)

Configure the radio for 2.4 GHz 802.15.4 operation with antenna diversity.

Parameters

[in]	railHandle	A handle of RAIL instance.

Returns

- A status code indicating success of the function call.

This initializes the radio for 2.4 GHz operation, but with a configuration that supports antenna diversity. It takes the place of calling RAIL_ConfigChannels. After this call, channels 11-26 will be available, giving the frequencies of those channels on channel page 0, as defined by IEEE 802.15.4-2011 section 8.1.2.2.

Note

- This call implicitly disables all RAIL_IEEE802154_GOptions_t.

Definition at line 534 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_Config2p4GHzRadioAntDivCoex

```
RAIL_Status_t RAIL_IEEE802154_Config2p4GHzRadioAntDivCoex (RAIL_Handle_t railHandle)
```

Configure the radio for 2.4 GHz 802.15.4 operation with antenna diversity optimized for radio coexistence.

Parameters

[in]	railHandle	A handle of RAIL instance.

Returns

- A status code indicating success of the function call.

This initializes the radio for 2.4 GHz operation, but with a configuration that supports antenna diversity optimized for radio coexistence. It takes the place of calling RAIL_ConfigChannels. After this call, channels 11-26 will be available, giving the frequencies of those channels on channel page 0, as defined by IEEE 802.15.4-2011 section 8.1.2.2.

Note

- This call implicitly disables all RAIL_IEEE802154_GOptions_t.

Definition at line 551 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_Config2p4GHzRadioCoex

```
RAIL_Status_t RAIL_IEEE802154_Config2p4GHzRadioCoex (RAIL_Handle_t railHandle)
```

Configure the radio for 2.4 GHz 802.15.4 operation optimized for radio coexistence.

Parameters

[in]	railHandle	A handle of RAIL instance.

Returns

- A status code indicating success of the function call.

This initializes the radio for 2.4 GHz operation, but with a configuration that supports radio coexistence. It takes the place of calling RAIL_ConfigChannels. After this call, channels 11-26 will be available, giving the frequencies of those channels on channel page 0, as defined by IEEE 802.15.4-2011 section 8.1.2.2.

Note

- This call implicitly disables all RAIL_IEEE802154_GOptions_t.

Definition at line 567 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_Config2p4GHzRadioFem

```
RAIL_Status_t RAIL_IEEE802154_Config2p4GHzRadioFem (RAIL_Handle_t railHandle)
```

Configure the radio for 2.4 GHz 802.15.4 operation with a front end module.

Parameters

[in]	railHandle	A handle of RAIL instance.

Returns

A status code indicating success of the function call.

This initializes the radio for 2.4 GHz operation, but with a configuration that supports a front end module. It takes the place of calling RAIL_ConfigChannels. After this call, channels 11-26 will be available, giving the frequencies of those channels on channel page 0, as defined by IEEE 802.15.4-2011 section 8.1.2.2.

Note

- This call implicitly disables all RAIL_IEEE802154_GOptions_t.

Definition at line 583 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_Config2p4GHzRadioAntDivFem

```
RAIL_Status_t RAIL_IEEE802154_Config2p4GHzRadioAntDivFem (RAIL_Handle_t railHandle)
```

Configure the radio for 2.4 GHz 802.15.4 operation with antenna diversity optimized for a front end module.

Parameters

[in]	railHandle	A handle of RAIL instance.

Returns

- A status code indicating success of the function call.

This initializes the radio for 2.4 GHz operation, but with a configuration that supports antenna diversity and a front end module. It takes the place of calling RAIL_ConfigChannels. After this call, channels 11-26 will be available, giving the frequencies of those channels on channel page 0, as defined by IEEE 802.15.4-2011 section 8.1.2.2.

Note

- This call implicitly disables all RAIL_IEEE802154_GOptions_t.

Definition at line 600 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_Config2p4GHzRadioCoexFem

```
RAIL_Status_t RAIL_IEEE802154_Config2p4GHzRadioCoexFem (RAIL_Handle_t railHandle)
```

Configure the radio for 2.4 GHz 802.15.4 operation optimized for radio coexistence and a front end module.

Parameters

[in]	railHandle	A handle of RAIL instance.

Returns

- A status code indicating success of the function call.

This initializes the radio for 2.4 GHz operation, but with a configuration that supports radio coexistence and a front end module. It takes the place of calling RAIL_ConfigChannels. After this call, channels 11-26 will be available, giving the frequencies of those channels on channel page 0, as defined by IEEE 802.15.4-2011 section 8.1.2.2.

Note

- This call implicitly disables all RAIL_IEEE802154_GOptions_t.

Definition at line 617 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_Config2p4GHzRadioAntDivCoexFem

> RAIL_Status_t RAIL_IEEE802154_Config2p4GHzRadioAntDivCoexFem (RAIL_Handle_t railHandle)

Configure the radio for 2.4 GHz 802.15.4 operation with antenna diversity optimized for radio coexistence and a front end module.

Parameters

[in]	railHandle	A handle of RAIL instance.

Returns

- A status code indicating success of the function call.

This initializes the radio for 2.4 GHz operation, but with a configuration that supports antenna diversity, radio coexistence and a front end module. It takes the place of calling RAIL_ConfigChannels. After this call, channels 11-26 will be available, giving the frequencies of those channels on channel page 0, as defined by IEEE 802.15.4-2011 section 8.1.2.2.

Note

- This call implicitly disables all RAIL_IEEE802154_GOptions_t.

Definition at line `634` of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_Config2p4GHzRadioCustom1

> RAIL_Status_t RAIL_IEEE802154_Config2p4GHzRadioCustom1 (RAIL_Handle_t railHandle)

Configure the radio for 2.4 GHz 802.15.4 operation with custom settings.

Parameters

[in]	railHandle	A handle of RAIL instance.

It enables better interoperability with some proprietary PHYs, but doesn't guarantee data sheet performance.

Returns

- A status code indicating success of the function call.

This initializes the radio for 2.4 GHz operation with custom settings. It replaces needing to call RAIL_ConfigChannels. Do not call this function unless instructed by Silicon Labs.

Note

- This feature is only available on platforms where RAIL_IEEE802154_SUPPORTS_CUSTOM1_PHY is true.

Definition at line `673` of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_ConfigGB863MHzRadio

> RAIL_Status_t RAIL_IEEE802154_ConfigGB863MHzRadio (RAIL_Handle_t railHandle)

Configure the radio for SubGHz GB868 863 MHz 802.15.4 operation.

Parameters

[in]	railHandle	A handle of RAIL instance.

Returns

A status code indicating success of the function call.

This initializes the radio for SubGHz GB868 863 MHz operation. It takes the place of calling RAIL_ConfigChannels. After this call, GB868 channels in the 863 MHz band (channel pages 28, 29, and 30 – logical channels 0x80..0x9A, 0xA0..0xA8, 0xC0..0xDA, respectively) will be available, as defined by Rev 22 of the Zigbee Specification, 2017 document 05-3474-22, section D.10.2.1.3.2.

Note

- This call implicitly enables RAIL_IEEE802154_G_OPTION_GB868.

Definition at line `690` of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_ConfigGB915MHzRadio

> RAIL_Status_t RAIL_IEEE802154_ConfigGB915MHzRadio (RAIL_Handle_t railHandle)

Configure the radio for SubGHz GB868 915 MHz 802.15.4 operation.

Parameters

[in]	railHandle	A handle of RAIL instance.

Returns

- A status code indicating success of the function call.

This initializes the radio for SubGHz GB868 915 MHz operation. It takes the place of calling RAIL_ConfigChannels. After this call, GB868 channels in the 915 MHz band (channel page 31 – logical channels 0xE0..0xFA) will be available, as defined by Rev 22 of the Zigbee Specification, 2017 document 05-3474-22, section D.10.2.1.3.2.

Note

- This call implicitly enables RAIL_IEEE802154_G_OPTION_GB868.

Definition at line `706` of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_Deinit

> RAIL_Status_t RAIL_IEEE802154_Deinit (RAIL_Handle_t railHandle)

De-initialize IEEE802.15.4 hardware acceleration.

Parameters

[in]	railHandle	A handle of RAIL instance.

Returns

- A status code indicating success of the function call.

Disables and resets all IEE802.15.4 hardware acceleration features. This function should only be called when the radio is IDLE. This calls the following:

- RAIL_SetStateTiming(), to reset all timings to 100 us
- RAIL_EnableAddressFilter(false)
- RAIL_ResetAddressFilter()

Definition at line `721` of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_IsEnabled

```
bool RAIL_IEEE802154_IsEnabled (RAIL_Handle_t railHandle)
```

Return whether IEEE802.15.4 hardware acceleration is currently enabled.

Parameters

[in]	railHandle	A handle of RAIL instance.

Returns

- True if IEEE802.15.4 hardware acceleration was enabled to start with and false otherwise.

Definition at line `730` of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_GetPtiRadioConfig

```
RAIL_IEEE802154_PtiRadioConfig_t RAIL_IEEE802154_GetPtiRadioConfig (RAIL_Handle_t railHandle)
```

Return IEEE802.15.4 PTI radio config.

Parameters

[in]	railHandle	A handle of RAIL instance.

Returns

- PTI (Packet Trace Information) radio config ID.

Definition at line `809` of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_SetAddresses

```
RAIL_Status_t RAIL_IEEE802154_SetAddresses (RAIL_Handle_t railHandle, const RAIL_IEEE802154_AddrConfig_t
*addresses)
```

Configure the RAIL Address Filter for 802.15.4 filtering.

Parameters

[in]	railHandle	A handle of RAIL instance.
[in]	addresses	The address information that should be used.

Returns

- A status code indicating success of the function call. If this returns an error, the 802.15.4 address filter is in an undefined state.

Set up the 802.15.4 address filter to accept messages to the given addresses. This will return false if any of the addresses failed to be set. If NULL is passed in for addresses, all addresses will be set to their reset value.

Definition at line `836` of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_SetPanId

```
RAIL_Status_t RAIL_IEEE802154_SetPanId (RAIL_Handle_t railHandle, uint16_t panId, uint8_t index)
```

Set a PAN ID for 802.15.4 address filtering.

Parameters

[in]	railHandle	A handle of RAIL instance.
[in]	panId	The 16-bit PAN ID information. This will be matched against the destination PAN ID of incoming messages. The PAN ID is sent little endian over the air, meaning panId[7:0] is first in the payload followed by panId[15:8]. Set to 0xFFFF to disable for this index.
[in]	index	Indicates which PAN ID to set. Must be below RAIL_IEEE802154_MAX_ADDRESSES.

Returns

- A status code indicating success of the function call.

Set up the 802.15.4 address filter to accept messages to the given PAN ID.

Definition at line `853` of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_SetShortAddress

> RAIL_Status_t RAIL_IEEE802154_SetShortAddress (RAIL_Handle_t railHandle, uint16_t shortAddr, uint8_t index)

Set a short address for 802.15.4 address filtering.

Parameters

[in]	railHandle	A handle of RAIL instance
[in]	shortAddr	16 bit short address value. This will be matched against the destination short address of incoming messages. The short address is sent little endian over the air meaning shortAddr[7:0] is first in the payload followed by shortAddr[15:8]. Set to 0xFFFF to disable for this index.
[in]	index	Which short address to set. Must be below RAIL_IEEE802154_MAX_ADDRESSES.

Returns

- A status code indicating success of the function call.

Set up the 802.15.4 address filter to accept messages to the given short address.

Definition at line `872` of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_SetLongAddress

> RAIL_Status_t RAIL_IEEE802154_SetLongAddress (RAIL_Handle_t railHandle, const uint8_t *longAddr, uint8_t index)

Set a long address for 802.15.4 address filtering.

Parameters

[in]	railHandle	A handle of RAIL instance.
[in]	longAddr	A pointer to an 8-byte array containing the long address information. The long address must be in over-the-air byte order. This will be matched against the destination long address of incoming messages. Set to 0x00 00 00 00 00 00 00 00 to disable for this index.
[in]	index	Indicates which long address to set. Must be below RAIL_IEEE802154_MAX_ADDRESSES.

Returns

- A status code indicating success of the function call.

Set up the 802.15.4 address filter to accept messages to the given long address.

Definition at line 891 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_SetPanCoordinator

> RAIL_Status_t RAIL_IEEE802154_SetPanCoordinator (RAIL_Handle_t railHandle, bool isPanCoordinator)

Set whether the current node is a PAN coordinator.

Parameters

[in]	railHandle	A handle of RAIL instance.
[in]	isPanCoordinator	True if this device is a PAN coordinator.

Returns

- A status code indicating success of the function call.

If the device is a PAN Coordinator, it will accept data and command frames with no destination address. This function will fail if 802.15.4 hardware acceleration is not currently enabled by calling RAIL_IEEE802154_Init(). This setting may be changed at any time when 802.15.4 hardware acceleration is enabled.

Definition at line 908 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_SetPromiscuousMode

> RAIL_Status_t RAIL_IEEE802154_SetPromiscuousMode (RAIL_Handle_t railHandle, bool enable)

Set whether to enable 802.15.4 promiscuous mode.

Parameters

[in]	railHandle	A handle of RAIL instance.
[in]	enable	True if all frames and addresses should be accepted.

Returns

- A status code indicating success of the function call.

If promiscuous mode is enabled, no frame or address filtering steps will be performed other than checking the CRC. This function will fail if 802.15.4 hardware acceleration is not currently enabled by calling RAIL_IEEE802154_Init(). This setting may be changed at any time when 802.15.4 hardware acceleration is enabled.

Definition at line 924 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_ConfigEOptions

> RAIL_Status_t RAIL_IEEE802154_ConfigEOptions (RAIL_Handle_t railHandle, RAIL_IEEE802154_EOptions_t mask, RAIL_IEEE802154_EOptions_t options)

Configure certain 802.15.4E-2012 / 802.15.4-2015 Frame Version 2 features.

Parameters

[in]	railHandle	A handle of RAIL instance.

[in]	mask	A bitmask containing which options should be modified.
[in]	options	A bitmask containing desired options settings. Bit positions for each option are found in the RAIL_IEEE802154_EOptions_t.

Returns

- A status code indicating success of the function call.

This function will fail if 802.15.4 hardware acceleration is not currently enabled by calling RAIL_IEEE802154_Init() or the platform does not support the feature(s). These settings may be changed at any time when 802.15.4 hardware acceleration is enabled.

Definition at line 1038 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_ConfigGOptions

> RAIL_Status_t RAIL_IEEE802154_ConfigGOptions (RAIL_Handle_t railHandle, RAIL_IEEE802154_GOptions_t mask, RAIL_IEEE802154_GOptions_t options)

Configure certain 802.15.4G-2012 / 802.15.4-2015 SUN PHY features (only for radio configurations designed accordingly).

Parameters

[in]	railHandle	A handle of RAIL instance.
[in]	mask	A bitmask containing which options should be modified.
[in]	options	A bitmask containing desired options settings. Bit positions for each option are found in the RAIL_IEEE802154_GOptions_t.

Returns

- A status code indicating success of the function call.

This function will fail if 802.15.4 hardware acceleration is not currently enabled by calling RAIL_IEEE802154_Init(), the platform does not support the feature(s), the radio configuration is not appropriate, or the radio is not idle.

Definition at line 1150 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_ComputeChannelFromPhyModeId

> RAIL_Status_t RAIL_IEEE802154_ComputeChannelFromPhyModeId (RAIL_Handle_t railHandle, uint8_t newPhyModeId, uint16_t *pChannel)

Compute channel to switch to given a targeted PhyMode ID in the context of Wi-SUN mode switching.

Parameters

[in]	railHandle	A handle of RAIL instance.
[in]	newPhyModeId	A targeted PhyMode ID.
[out]	pChannel	A pointer to the channel to switch to.

Returns

- A status code indicating success of the function call.

This function will fail if:

- the targeted PhyModeID is the same as the current PhyMode ID
- called on a platform that lacks RAIL_IEEE802154_SUPPORTS_G_MODESWITCH
-

called on a platform that doesn't have 802154G options enabled by RAIL_IEEE802154_ConfigGOptions(). For newPhyModeId associated with a FSK FEC_off PHY, if dynamic FEC is activated (see RAIL_IEEE802154_G_OPTION_DYNFEC), the returned channel can correspond to the associated FSK FEC_on PHY corresponding then to PhyModeID = newPhyModeId + 16

Definition at line 1191 of file protocol/ieee802154/rail_ieee802154.h

RAILCb_IEEE802154_IsModeSwitchNewChannelValid

> RAIL_Status_t RAILCb_IEEE802154_IsModeSwitchNewChannelValid (uint32_t currentBaseFreq, uint8_t newPhyModeId, const RAIL_ChannelConfigEntry_t *configEntryNewPhyModeId, uint16_t *pChannel)

Manage forbidden channels during mode switch.

Parameters

[in]	currentBaseFreq	The current frequency of the base channel.
[in]	newPhyModeId	A targeted PhyMode ID.
[in]	configEntryNewPhyModeId	A pointer to RAIL_ChannelConfigEntry_t structure corresponding to the new PHY configEntry.
[inout]	pChannel	A pointer to the channel to switch to. If channel is valid, the function must just return. If channel is forbidden, the function must update it with the closest valid channel. The highest channel must be selected in case of two valid channels being equidistant to a forbidden channel.

Returns

- A status code indicating success of the function call. It must return RAIL_STATUS_INVALID_PARAMETER for failure or RAIL_STATUS_NO_ERROR for success.

This function must fail if no valid channel has been found. If so, RAIL will abort the mode switch.

Note

- This callback will only be called on platforms where RAIL_IEEE802154_SUPPORTS_G_MODESWITCH is true, RAIL_IEEE802154_G_OPTION_WISUN_MODESWITCH was successfully enabled, and a valid mode switch PHY header is received.

Definition at line 1219 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_AcceptFrames

> RAIL_Status_t RAIL_IEEE802154_AcceptFrames (RAIL_Handle_t railHandle, uint8_t framesMask)

Set which 802.15.4 frame types to accept.

Parameters

[in]	railHandle	A handle of RAIL instance.
[in]	framesMask	A mask containing which 802.15.4 frame types to receive.

Returns

- A status code indicating success of the function call.

This function will fail if 802.15.4 hardware acceleration is not currently enabled by calling RAIL_IEEE802154_Init() or framesMask requests an unsupported frame type. This setting may be changed at any time when 802.15.4 hardware acceleration is enabled. Only Beacon, Data, ACK, Command, and Multipurpose (except on EFR32XG1) frames may be

received. The RAIL_IEEE802154_ACCEPT_XXX_FRAMES defines may be combined to create a bitmask to pass into this function.

RAIL_IEEE802154_ACCEPT_ACK_FRAMES behaves slightly different than the other defines. If RAIL_IEEE802154_ACCEPT_ACK_FRAMES is set, the radio will accept an ACK frame during normal packet reception, but only a truly expected ACK will have its RAIL_RxPacketDetails_t::isAck true. If RAIL_IEEE802154_ACCEPT_ACK_FRAMES is not set, ACK frames will be filtered unless they're expected when the radio is waiting for an ACK.

Definition at line 1270 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_EnableEarlyFramePending

> RAIL_Status_t RAIL_IEEE802154_EnableEarlyFramePending (RAIL_Handle_t railHandle, bool enable)

Enable early Frame Pending lookup event notification (RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND).

Parameters

[in]	railHandle	A handle of RAIL instance.
[in]	enable	True to enable, false to disable.

Returns

- A status code indicating success of the function call.

Normally, RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND is triggered after receiving the entire MAC header and MAC command byte for an ACK-requesting MAC command frame. Version 0/1 frames also require that command to be a Data Request for this event to occur. Enabling this feature causes this event to be triggered earlier to allow for more time to determine the type of ACK needed (Immediate or Enhanced) and/or perform frame pending lookup to influence the outgoing ACK by using RAIL_IEEE802154_WriteEnhAck() or RAIL_IEEE802154_ToggleFramePending().

For Frame Version 0/1 packets and for Frame Version 2 packets when RAIL_IEEE802154_E_OPTION_ENH_ACK is not in use, "early" means right after receiving the source address information in the MAC header.

For Frame Version 2 packets when RAIL_IEEE802154_E_OPTION_ENH_ACK is in use, "early" means right after receiving any Auxiliary Security header which follows the source address information in the MAC header.

This feature is useful when the protocol knows an ACK-requesting MAC Command must be a data poll without needing to receive the MAC Command byte, giving it a bit more time to adjust Frame Pending or generate an Enhanced ACK.

This function will fail if 802.15.4 hardware acceleration is not currently enabled by calling RAIL_IEEE802154_Init(), or on platforms that do not support this feature. This setting may be changed at any time when 802.15.4 hardware acceleration is enabled.

Definition at line 1309 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_EnableDataFramePending

> RAIL_Status_t RAIL_IEEE802154_EnableDataFramePending (RAIL_Handle_t railHandle, bool enable)

Enable Frame Pending lookup event notification (RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND) for MAC Data frames.

Parameters

[in]	railHandle	A handle of RAIL instance.
[in]	enable	True to enable, false to disable.

Returns

A status code indicating success of the function call.

Normally RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND is triggered only for ACK-requesting MAC command frames. Enabling this feature causes this event to also be triggered for MAC data frames, at the same point in the packet as RAIL_IEEE802154_EnableEarlyFramePending() would trigger. This feature is necessary to support the Thread Basil-Hayden Enhanced Frame Pending feature in Version 0/1 frames, and to support Version 2 Data frames which require an Enhanced ACK.

This function will fail if 802.15.4 hardware acceleration is not currently enabled by calling RAIL_IEEE802154_Init(). This setting may be changed at any time when 802.15.4 hardware acceleration is enabled.

Definition at line 1334 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_SetFramePending

> RAIL_Status_t RAIL_IEEE802154_SetFramePending (RAIL_Handle_t railHandle)

Change the Frame Pending bit on the outgoing legacy Immediate ACK from the default specified by RAIL_IEEE802154_Config_t::defaultFramePendingInOutgoingAcks.

Parameters

[in]	railHandle	A handle of RAIL instance

Returns

- A status code indicating success of the function call.

This function must only be called while processing the RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND if the ACK for this packet should go out with its Frame Pending bit set differently than what was specified by RAIL_IEEE802154_Config_t::defaultFramePendingInOutgoingAcks.

It's intended only for use with 802.15.4 legacy immediate ACKs and not 802.15.4E enhanced ACKs. This will return RAIL_STATUS_INVALID_STATE if it is too late to modify the outgoing Immediate ACK.

Note

- This function is used to set the Frame Pending bit but its meaning depends on the value of RAIL_IEEE802154_Config_t::defaultFramePendingInOutgoingAcks while transmitting ACK. If RAIL_IEEE802154_Config_t::defaultFramePendingInOutgoingAcks is not set, then Frame Pending bit is set in outgoing ACK. Whereas, if RAIL_IEEE802154_Config_t::defaultFramePendingInOutgoingAcks is set, then Frame Pending bit is cleared in outgoing ACK.

Therefore, this function is to be called if the frame is pending when RAIL_IEEE802154_Config_t::defaultFramePendingInOutgoingAcks is not set or if there is no frame pending when RAIL_IEEE802154_Config_t::defaultFramePendingInOutgoingAcks is set.

Definition at line 1378 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_GetAddress

> RAIL_Status_t RAIL_IEEE802154_GetAddress (RAIL_Handle_t railHandle, RAIL_IEEE802154_Address_t *pAddress)

Get the source address of the incoming data request.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	pAddress	A pointer to RAIL_IEEE802154_Address_t structure to populate with source address information.

Returns

- A status code indicating success of the function call.

This function must only be called when handling the RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND event. This will return RAIL_STATUS_INVALID_STATE if the address information is stale (i.e., it is too late to affect the outgoing ACK).

Definition at line 1393 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_WriteEnhAck

> RAIL_Status_t RAIL_IEEE802154_WriteEnhAck (RAIL_Handle_t railHandle, const uint8_t *ackData, uint8_t ackDataLen)

Write the AutoACK FIFO for the next outgoing 802.15.4E Enhanced ACK.

Parameters

[in]	railHandle	A handle of RAIL instance.
[in]	ackData	Pointer to ACK data to transmit This may be NULL, in which case it's assumed the data has already been emplaced into the ACK buffer and RAIL just needs to be told how many bytes are there. Use RAIL_GetAutoAckFifo() to get the address of RAIL's AutoACK buffer in RAM and its size.
[in]	ackDataLen	Length of ACK data, in bytes. If this exceeds RAIL_AUTOACK_MAX_LENGTH the function will return RAIL_STATUS_INVALID_PARAMETER.

Returns

- A status code indicating success of the function call.

This function sets the AutoACK data to use in acknowledging the frame being received. It must only be called while processing the RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND, and is intended for use when packet information from RAIL_GetRxIncomingPacketInfo() indicates an 802.15.4E Enhanced ACK must be sent instead of a legacy Immediate ACK. RAIL_IEEE802154_ToggleFramePending() should not be called for an Enhanced ACK; instead the Enhanced ACK's Frame Control Field should have the Frame Pending bit set appropriately in its ackData. This will return RAIL_STATUS_INVALID_STATE if it is too late to write the outgoing ACK – a situation that will likely trigger a RAIL_EVENT_TXACK_UNDERFLOW event. When successful, the Enhanced ackData will only be sent once. Subsequent packets needing an Enhanced ACK will each need to call this function to write their ACK information.

Definition at line 1424 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_SetRxToEnhAckTx

> RAIL_Status_t RAIL_IEEE802154_SetRxToEnhAckTx (RAIL_Handle_t railHandle, RAIL_TransitionTime_t *pRxToEnhAckTx)

Set a separate RX packet to TX state transition turnaround time for sending an Enhanced ACK.

Parameters

[in]	railHandle	A RAIL instance handle.
[inout]	pRxToEnhAckTx	Pointer to the turnaround transition requested for Enhanced ACKs. It will be updated with the actual time set. Requesting a time of 0 will sync the Enhanced ACK turnaround time with that used for immediate ACKs (and output 0). Requesting a time of RAIL_TRANSITION_TIME_KEEP will output the current Enhanced ACK timing parameter (0 if it is the same as that used for Immediate ACKs).

Returns

- Status code indicating a success of the function call. An error will not update the pRxToEnhAckTx output parameter.

Normally Immediate and Enhanced ACKs are both sent using the RAIL_IEEE802154_Config_t::timings rxToTx turnaround time. If the stack needs more time to prepare an Enhanced ACK, it can call this function after RAIL_IEEE802154_Init() to set a longer turnaround time used just for Enhanced ACK transmits.

This function will fail on platforms that lack RAIL_IEEE802154_SUPPORTS_E_ENHANCED_ACK.

Definition at line 1451 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_ConvertRssiToLqi

```
uint8_t RAIL_IEEE802154_ConvertRssiToLqi (uint8_t origLqi, int8_t rssiDbm)
```

Convert RSSI into 802.15.4 Link Quality Indication (LQI) metric compatible with the Silicon Labs Zigbee stack.

Parameters

[in]	origLqi	The original LQI, for example from RAIL_RxPacketDetails_t::lqi. This parameter is not currently used but may be used in the future.
[in]	rssiDbm	The RSSI in dBm, for example from RAIL_RxPacketDetails_t::rssi.

Returns

- An LQI value (range 0..255 but not all intermediate values are possible) based on the rssiDbm and the chip's RSSI sensitivity range.

This function is compatible with RAIL_ConvertLqiCallback_t and is suitable to pass to RAIL_ConvertLqi().

Definition at line 1469 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_ConvertRssiToEd

```
uint8_t RAIL_IEEE802154_ConvertRssiToEd (int8_t rssiDbm)
```

Convert RSSI into 802.15.4 Energy Detection (ED) metric compatible with the Silicon Labs Zigbee stack.

Parameters

[in]	rssiDbm	The RSSI in dBm, for example from RAIL_RxPacketDetails_t::rssi.

Returns

- An Energy Detect value (range 0..255 but not all intermediate values are possible) based on the rssiDbm and the chip's RSSI sensitivity range.

Definition at line 1481 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_ConfigSignalIdentifier

```
RAIL_Status_t RAIL_IEEE802154_ConfigSignalIdentifier (RAIL_Handle_t railHandle, RAIL_IEEE802154_SignalIdentifierMode_t signalIdentifierMode)
```

Configure signal identifier for 802.15.4 signal detection.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	signalIdentifierMode	Mode of signal identifier operation.

This features allows detection of 2.4Ghz 802.15.4 signal on air. This function must be called once before RAIL_IEEE802154_EnableSignalDetection to configure and enable signal identifier.

To enable event for signal detection RAIL_ConfigEvents() must be called for enabling RAIL_EVENT_SIGNAL_DETECTED.

This function is only supported by chips where RAIL_IEEE802154_SUPPORTS_SIGNAL_IDENTIFIER and RAIL_IEEE802154_SupportsSignalIdentifier() are true.

Returns

- Status code indicating success of the function call.

Definition at line `1569` of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_EnableSignalDetection

> RAIL_Status_t RAIL_IEEE802154_EnableSignalDetection (RAIL_Handle_t railHandle, bool enable)

Enable or disable signal identifier for 802.15.4 signal detection.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	enable	Signal detection is enabled if true, disabled if false.

RAIL_IEEE802154_ConfigSignalIdentifier must be called once before calling this function to configure and enable signal identifier. Once a signal is detected signal detection will be turned off and this function should be called to re-enable the signal detection without needing to call RAIL_IEEE802154_ConfigSignalIdentifier if the signal identifier is already configured and enabled.

This function is only supported by chips where RAIL_IEEE802154_SUPPORTS_SIGNAL_IDENTIFIER and RAIL_IEEE802154_SupportsSignalIdentifier() are true.

Returns

- Status code indicating success of the function call.

Definition at line `1591` of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_ConfigCcaMode

> RAIL_Status_t RAIL_IEEE802154_ConfigCcaMode (RAIL_Handle_t railHandle, RAIL_IEEE802154_CcaMode_t ccaMode)

Set 802.15.4 CCA mode.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	ccaMode	Mode of CCA operation.

This function sets the CCA mode RAIL_IEEE802154_CcaMode_t. If not called, RAIL_IEEE802154_CCA_MODE_RSSI (RSSI-based CCA) is used for CCA.

In RAIL_IEEE802154_CCA_MODE_SIGNAL, RAIL_IEEE802154_CCA_MODE_SIGNAL_OR_RSSI and RAIL_IEEE802154_CCA_MODE_SIGNAL_AND_RSSI signal identifier is enabled for the duration of LBT. If previously enabled by RAIL_IEEE802154_ConfigSignalIdentifier, the signal identifier will remain active until triggered.

This function is only supported by chips where RAIL_IEEE802154_SUPPORTS_SIGNAL_IDENTIFIER and RAIL_IEEE802154_SupportsSignalIdentifier() are true.

Returns

- Status code indicating success of the function call. An error should be returned if ccaMode is unsuppported on a given device.

Definition at line 1622 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_ConfigRxChannelSwitching

```
RAIL_Status_t RAIL_IEEE802154_ConfigRxChannelSwitching (RAIL_Handle_t railHandle, const
RAIL_IEEE802154_RxChannelSwitchingCfg_t *pConfig)
```

Configure RX channel switching for 802.15.4.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	pConfig	A pointer to RAIL_IEEE802154_RxChannelSwitchingCfg_t structure. Use NULL to disable any switching previously set up.

Returns

- Status code indicating success of the function call.

This function configures RX channel switching, allowing reception of 2.4Ghz 802.15.4 signals on two different radio channels within the same PHY. (If the two channels are same, the function behaves the same as if pConfig was NULL.) This function should be called once before RAIL_StartRx and/or enabling RAIL_RX_OPTION_CHANNEL_SWITCHING.

When RAIL_RX_OPTION_CHANNEL_SWITCHING is enabled, channel switching will occur during normal listening but is suspended (and the radio is idled) when starting any kind of transmit, including scheduled or CSMA transmits. It remains suspended after a RAIL_TX_OPTION_WAIT_FOR_ACK transmit until the ACK is received or times out.

When RAIL_RX_OPTION_CHANNEL_SWITCHING is disabled after switching has been active, the radio could be left listening on either channel, so the application should call RAIL_StartRx() to put it on the desired non-switching channel.

Note

- IEEE 802.15.4 must be enabled via RAIL_IEEE802154_Init, and the radio must be in the idle state when configuring RX channel switching. A DMA channel must be allocated with RAIL_UseDma; otherwise this API will return RAIL_STATUS_INVALID_CALL. This feature also requires a PRS channel, internally allocated by the RAIL library, to use and hold onto for future use. If no PRS channel is available, the function returns RAIL_STATUS_INVALID_PARAMETER.
- When RX channel switching is active, receive sensitivity and performance are slightly impacted.
- This function internally uses RAIL_EnableCacheSynthCal to enable/disable the sequencer cache to store the synth calibration value.
- Switching is cancelled on any PHY change, so this function would need to be re-called to reestablish switching after such a change.

Definition at line 1691 of file protocol/ieee802154/rail_ieee802154.h

Macro Definition Documentation

RAIL_IEEE802154_MAX_ADDRESSES

```
#define RAIL_IEEE802154_MAX_ADDRESSES
```

Value:

```
(3U)
```

The maximum number of allowed addresses of each type.

Definition at line 229 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_RX_CHANNEL_SWITCHING_BUF_BYTES

```
#define RAIL_IEEE802154_RX_CHANNEL_SWITCHING_BUF_BYTES
```

Value:

```
(0U)
```

RX channel switching buffer size, in bytes.

SLI_LIBRARY_BUILD

Definition at line 326 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_RX_CHANNEL_SWITCHING_BUF_ALIGNMENT_TYPE

```
#define RAIL_IEEE802154_RX_CHANNEL_SWITCHING_BUF_ALIGNMENT_TYPE
```

Value:

```
uint32_t
```

Fixed-width type indicating the needed alignment for RX channel switching buffer.

Definition at line 331 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_RX_CHANNEL_SWITCHING_BUF_ALIGNMENT

```
#define RAIL_IEEE802154_RX_CHANNEL_SWITCHING_BUF_ALIGNMENT
```

Value:

```
(sizeof(RAIL_IEEE802154_RX_CHANNEL_SWITCHING_BUF_ALIGNMENT_TYPE))
```

Alignment that is needed for RX channel switching buffer.

Definition at line 334 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_RX_CHANNEL_SWITCHING_NUM_CHANNELS

```
#define RAIL_IEEE802154_RX_CHANNEL_SWITCHING_NUM_CHANNELS
```

Value:

```
(2U)
```

Maximum numbers of channels supported for RX channel switching.

Definition at line 337 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_E_OPTIONS_NONE

```
#define RAIL_IEEE802154_E_OPTIONS_NONE
```

Value:

```
0UL
```

A value representing no options enabled.

Definition at line 939 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_E_OPTIONS_DEFAULT

```
#define RAIL_IEEE802154_E_OPTIONS_DEFAULT
```

Value:

```
RAIL_IEEE802154_E_OPTIONS_NONE
```

All options disabled by default .

Definition at line 941 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_E_OPTION_GB868

```
#define RAIL_IEEE802154_E_OPTION_GB868
```

Value:

```
(1UL << RAIL_IEEE802154_E_OPTION_GB868_SHIFT)
```

An option to enable/disable 802.15.4E-2012 features needed for GB868.

When not promiscuous, RAIL normally accepts only 802.15.4 MAC frames whose MAC header Frame Version is 0 (802.15.4-2003) or 1 (802.15.4-2006), filtering out higher Frame Version packets (as RAIL_RX_PACKET_ABORT_FORMAT). Enabling this feature additionally allows Frame Version 2 (802.15.4E-2012 / 802.15.4-2015) packets to be accepted and passed to the application.

Note

- Enabling this feature also automatically enables RAIL_IEEE802154_E_OPTION_ENH_ACK on platforms that support that feature.
- This feature does not automatically enable receiving Multipurpose frames; that can be enabled via RAIL_IEEE802154_AcceptFrames()'s RAIL_IEEE802154_ACCEPT_MULTIPURPOSE_FRAMES.

Definition at line 960 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_E_OPTION_ENH_ACK

```
#define RAIL_IEEE802154_E_OPTION_ENH_ACK
```

Value:

```
(1UL << RAIL_IEEE802154_E_OPTION_ENH_ACK_SHIFT)
```

An option to enable/disable 802.15.4E-2012 features needed for Enhanced ACKs.

This option requires that RAIL_IEEE802154_E_OPTION_GB868 also be enabled, and is enabled automatically on platforms that support this feature. It exists as a separate flag to allow runtime detection of whether the platform supports this feature or not.

When enabled, only an Enhanced ACK is expected in response to a transmitted ACK-requesting 802.15.4E Version 2 frame. RAIL only knows how to construct 802.15.4 Immediate ACKs but not Enhanced ACKs.

This option causes RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND to be issued for ACK-requesting Version 2 MAC Command frames, Data frames (if RAIL_IEEE802154_EnableDataFramePending() is enabled), and Multipurpose Frames (if RAIL_IEEE802154_ACCEPT_MULTIPURPOSE_FRAMES is enabled).

The application is expected to handle this event by calling RAIL_GetRxIncomingPacketInfo() and parsing the partly-received incoming frame to determine the type of ACK needed:

- If an Immediate ACK, determine Frame Pending needs based on the packet type and addressing information and call RAIL_IEEE802154_ToggleFramePending() if necessary;
- If an Enhanced ACK, generate the complete payload of the Enhanced ACK including any Frame Pending information and call RAIL_IEEE802154_WriteEnhAck() in time for that Enhanced ACK to be sent. If not called in time, RAIL_EVENT_TXACK_UNDERFLOW will likely result. Note that if 802.15.4 MAC-level encryption is used with Version 2 frames, the application should decrypt the MAC Command byte in a MAC Command frame to determine whether it is a Data Request or other MAC Command.

An application can also enable RAIL_IEEE802154_EnableEarlyFramePending() if the protocol doesn't need to examine the MAC Command byte of MAC Command frames but can infer it to be a Data Request.

On 802.15.4E GB868 platforms that lack this support, legacy Immediate ACKs are sent/expected for received/transmitted ACK-requesting 802.15.4E Frame Version 2 frames; calls to RAIL_IEEE802154_WriteEnhAck() have no effect. Attempting to use this feature via RAIL_IEEE802154_ConfigEOptions() returns an error.

Definition at line 1006 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_E_OPTION_IMPLICIT_BROADCAST

```
#define RAIL_IEEE802154_E_OPTION_IMPLICIT_BROADCAST
```

Value:

```
(1UL << RAIL_IEEE802154_E_OPTION_IMPLICIT_BROADCAST_SHIFT)
```

An option to enable/disable 802.15.4E-2012 macImplicitBroadcast feature.

When enabled, received Frame Version 2 frames without a destination PAN ID or destination address are treated as though they are addressed to the broadcast PAN ID and broadcast short address. When disabled, such frames are filtered unless the device is the PAN coordinator and appropriate source addressing information exists in the packet

Definition at line 1017 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_E_OPTIONS_ALL

```
#define RAIL_IEEE802154_E_OPTIONS_ALL
```

Value:

```
0xFFFFFFFFUL
```

A value representing all possible options.

Definition at line 1020 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_G_OPTIONS_NONE

```
#define RAIL_IEEE802154_G_OPTIONS_NONE
```

Value:

```
0UL
```

A value representing no options enabled.

Definition at line 1056 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_G_OPTIONS_DEFAULT

```
#define RAIL_IEEE802154_G_OPTIONS_DEFAULT
```

Value:

```
RAIL_IEEE802154_G_OPTIONS_NONE
```

All options disabled by default .

Definition at line 1058 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_G_OPTION_GB868

```
#define RAIL_IEEE802154_G_OPTION_GB868
```

Value:

```
(1UL << RAIL_IEEE802154_G_OPTION_GB868_SHIFT)
```

An option to enable/disable 802.15.4G-2012 features needed for GB868.

Normally RAIL supports 802.15.4-2003 and -2006 radio configurations that have the single-byte PHY header allowing frames up to 128 bytes in size. This feature must be enabled for 802.15.4G-2012 or 802.15.4-2015 SUN PHY radio configurations with the two-byte bit-reversed-length PHY header format.

While GB868 only supports whitened non-FEC non-mode-switch frames up to 129 bytes including 2-byte CRC, this option also enables:

- On platforms where RAIL_FEAT_IEEE802154_G_4BYTE_CRC_SUPPORTED is true: automatic per-packet 2/4-byte Frame Check Sequence (FCS) reception and transmission based on the FCS Type bit in the received/transmitted PHY header. This includes ACK reception and automatically-generated ACKs reflect the CRC size of the incoming frame being acknowledged (i.e., their MAC payload will be increased to 7 bytes when sending 4-byte FCS). On other platforms, only the 2-byte FCS is supported.
- On platforms where RAIL_FEAT_IEEE802154_G_UNWHITENED_RX_SUPPORTED and/or RAIL_FEAT_IEEE802154_G_UNWHITENED_TX_SUPPORTED are true: automatic per-packet whitened/unwhitened reception and transmission, respectively, based on the Data Whitening bit in the received/transmitted PHY header. This includes ACK reception and automatically-generated ACKs which reflect the whitening of the incoming frame being acknowledged. On other platforms, only whitened frames are supported.

- Support for frames up to 2049 bytes per the radio configuration's maximum packet length setting.

Note

- Sending/receiving whitened frames assumes the radio configuration has established an appropriate 802.15.4-compliant whitening algorithm. RAIL does not itself override the radio configuration's whitening settings other than to enable/disable it per-packet based on the packet's PHY header Data Whitening flag.

Definition at line 1094 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_G_OPTION_DYNFEC

```
#define RAIL_IEEE802154_G_OPTION_DYNFEC
```

Value:

```
(1UL << RAIL_IEEE802154_G_OPTION_DYNFEC_SHIFT)
```

An option to enable/disable 802.15.4G dynamic FEC feature (SUN FSK only).

The syncWord, called start-of-frame delimiter (SFD) in the 15.4 spec, indicates whether the rest of the packet is FEC encoded or not. This feature requires per-packet dual syncWord detection and specific receiver pausing. Note that this feature is only available on platforms where RAIL_IEEE802154_SUPPORTS_G_DYNFEC is true.

This option is only valid for SUN PHYs that have the FEC configured and enabled.

The syncWord used during transmit is selected with RAIL_TX_OPTION_SYNC_WORD_ID.

The syncWord corresponding to the FEC encoded mode must be SYNC1, with SYNC2 indicating non-FEC. SyncWords are set appropriately in all Sun FEC-enabled PHYs so there should never be a need to call RAIL_ConfigSyncWords() when this option is enabled.

Also, dual syncWord detection is set in all SUN FEC enabled PHYs, then there is no need to change RAIL_RX_OPTION_ENABLE_DUALSYNC .

Note

- EFR32xG12 support for 802.15.4 FEC-capable PHYs and dynamic FEC are incompatible with 802.15.4 filtering and AutoACK. RAIL_IEEE802154_Config_t::promiscuousMode must be true and RAIL_IEEE802154_Config_t::ackConfig's RAIL_AutoAckConfig_t::enable must be false on these platforms when using a FEC-capable PHY.

Definition at line 1120 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_G_OPTION_WISUN_MODESWITCH

```
#define RAIL_IEEE802154_G_OPTION_WISUN_MODESWITCH
```

Value:

```
(1UL << RAIL_IEEE802154_G_OPTION_WISUN_MODESWITCH_SHIFT)
```

An option to enable/disable Wi-SUN Mode Switch feature.

This feature consists in switching to a new PHY mode with a higher rate typically by sending/receiving a specific Mode Switch packet that indicates the incoming new PHY mode. The Mode Switch packet is an FSK-modulated 2-byte PHY header with no payload. Because this feature relies on specific receiver pausing, note that it is only available on platforms where RAIL_IEEE802154_SUPPORTS_G_MODESWITCH is true.

Definition at line 1129 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_G_OPTIONS_ALL

```
#define RAIL_IEEE802154_G_OPTIONS_ALL
```

Value:

```
0xFFFFFFFFUL
```

A value representing all possible options.

Definition at line 1132 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_ACCEPT_BEACON_FRAMES

```
#define RAIL_IEEE802154_ACCEPT_BEACON_FRAMES
```

Value:

```
(0x01)
```

When receiving packets, accept 802.15.4 BEACON frame types.

Definition at line 1225 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_ACCEPT_DATA_FRAMES

```
#define RAIL_IEEE802154_ACCEPT_DATA_FRAMES
```

Value:

```
(0x02)
```

When receiving packets, accept 802.15.4 DATA frame types.

Definition at line 1227 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_ACCEPT_ACK_FRAMES

```
#define RAIL_IEEE802154_ACCEPT_ACK_FRAMES
```

Value:

```
(0x04)
```

When receiving packets, accept 802.15.4 ACK frame types.

Note

- Expected ACK frame types will still be accepted regardless of this setting when waiting for an ACK after a transmit that used RAIL_TX_OPTION_WAIT_FOR_ACK and auto-ACK is enabled.

Definition at line 1232 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_ACCEPT_COMMAND_FRAMES

```
#define RAIL_IEEE802154_ACCEPT_COMMAND_FRAMES
```

Value:

```
(0x08)
```

When receiving packets, accept 802.15.4 COMMAND frame types.

Definition at line 1234 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_ACCEPT_MULTIPURPOSE_FRAMES

```
#define RAIL_IEEE802154_ACCEPT_MULTIPURPOSE_FRAMES
```

Value:

```
(0x20)
```

When receiving packets, accept 802.15.4-2015 Multipurpose frame types.

(Not supported on EFR32XG1.)

Definition at line 1238 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_ACCEPT_STANDARD_FRAMES

```
#define RAIL_IEEE802154_ACCEPT_STANDARD_FRAMES
```

Value:

```
(RAIL_IEEE802154_ACCEPT_BEACON_FRAMES \
 | RAIL_IEEE802154_ACCEPT_DATA_FRAMES \
 | RAIL_IEEE802154_ACCEPT_COMMAND_FRAMES)
```

In standard operation, accept BEACON, DATA and COMMAND frames.

Don't receive ACK frames unless waiting for ACK (i.e., only receive expected ACKs).

Definition at line 1243 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_ToggleFramePending

```
#define RAIL_IEEE802154_ToggleFramePending
```

Value:

```
RAIL_IEEE802154_SetFramePending
```

Alternate naming for function RAIL_IEEE802154_SetFramePending to depict it is used for changing the default setting specified by RAIL_IEEE802154_Config_t::defaultFramePendingInOutgoingAcks in an outgoing ACK.

Definition at line 1343 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_EnableSignalIdentifier

```
#define RAIL_IEEE802154_EnableSignalIdentifier
```

Value:

```
RAIL_IEEE802154_EnableSignalDetection
```

Backward compatible name for the RAIL_IEEE802154_EnableSignalDetection API.

Definition at line 1598 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_Address_t

Representation of 802.15.4 address This structure is only used for received source address information needed to perform Frame Pending lookup.

Public Attributes

uint16_t	shortAddress Present for 2 byte addresses.
uint8_t	longAddress Present for 8 byte addresses.
union RAIL_IEEE802154_Address_t::@2	@3 Convenient storage for different address types.
RAIL_IEEE802154_AddressLength_t	length Enumeration of the received address length.
RAIL_AddrFilterMask_t	filterMask A bitmask representing which address filter(s) this packet has passed.

Public Attribute Documentation

shortAddress

```
uint16_t RAIL_IEEE802154_Address_t::shortAddress
```

Present for 2 byte addresses.

Definition at line 214 of file protocol/ieee802154/rail_ieee802154.h

longAddress

```
uint8_t RAIL_IEEE802154_Address_t::longAddress[8]
```

Present for 8 byte addresses.

Definition at line 215 of file protocol/ieee802154/rail_ieee802154.h

@3

```
union RAIL_IEEE802154_Address_t::@2 RAIL_IEEE802154_Address_t::@3
```

Convenient storage for different address types.

Definition at line 216 of file protocol/ieee802154/rail_ieee802154.h

length

> RAIL_IEEE802154_AddressLength_t RAIL_IEEE802154_Address_t::length

Enumeration of the received address length.

Definition at line 220 of file protocol/ieee802154/rail_ieee802154.h

filterMask

> RAIL_AddrFilterMask_t RAIL_IEEE802154_Address_t::filterMask

A bitmask representing which address filter(s) this packet has passed.

It is undefined on platforms lacking RAIL_SUPPORTS_ADDR_FILTER_MASK.

Definition at line 225 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_AddrConfig_t

A configuration structure for IEEE 802.15.4 Address Filtering.

The broadcast addresses are handled separately and do not need to be specified here. Any address to be ignored should be set with all bits high.

This structure allows configuration of multi-PAN functionality by specifying multiple PAN IDs and short addresses. A packet will be received if it matches an address and its corresponding PAN ID. Long address 0 and short address 0 match against PAN ID 0, etc. The broadcast PAN ID and address will work with any address or PAN ID, respectively.

Public Attributes

uint16_t	panId PAN IDs for destination filtering.
uint16_t	shortAddr A short network addresses for destination filtering.
uint8_t	longAddr A 64-bit address for destination filtering.

Public Attribute Documentation

panId

```
uint16_t RAIL_IEEE802154_AddrConfig_t::panId[(3U)]
```

PAN IDs for destination filtering.

All must be specified. To disable a PAN ID, set it to the broadcast value, 0xFFFF.

Definition at line 248 of file protocol/ieee802154/rail_ieee802154.h

shortAddr

```
uint16_t RAIL_IEEE802154_AddrConfig_t::shortAddr[(3U)]
```

A short network addresses for destination filtering.

All must be specified. To disable a short address, set it to the broadcast value, 0xFFFF.

Definition at line 253 of file protocol/ieee802154/rail_ieee802154.h

longAddr

```
uint8_t RAIL_IEEE802154_AddrConfig_t::longAddr[(3U)][8]
```

A 64-bit address for destination filtering.

All must be specified. This field is parsed in over-the-air (OTA) byte order. To disable a long address, set it to the reserved value of 0x00 00 00 00 00 00 00 00.

Definition at line 259 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_Config_t

A configuration structure for IEEE 802.15.4 in RAIL.

Note

- 802.15.4 radio configurations with Forward Error Correction (FEC) enabled are incompatible with 802.15.4 filtering and AutoACK on EFR32xG1 platforms. AutoACK should be disabled and promiscuous mode enabled when using such a configuration. This is enforced implicitly on EFR32xG1 platforms with RAIL_IEEE802154_SUPPORTS_G_DYNFEC true when RAIL_IEEE802154_ConfigGOptions() is called to enable any G options.

Public Attributes

const RAIL_IEEE802154_AddrConfig_t *	addresses Configure the RAIL Address Filter to allow the given destination addresses.
RAIL_AutoAckConfig_t	ackConfig Define the ACKing configuration for the IEEE 802.15.4 implementation.
RAIL_StateTiming_t	timings Define state timings for the IEEE 802.15.4 implementation.
uint8_t	framesMask Set which 802.15.4 frame types will be received, of Beacon, Data, ACK, and Command.
bool	promiscuousMode Enable promiscuous mode during configuration.
bool	isPanCoordinator Set whether the device is a PAN Coordinator during configuration.
bool	defaultFramePendingInOutgoingAcks The default value for the Frame Pending bit in outgoing ACKs for packets that triggered the RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND event.

Public Attribute Documentation

addresses

```
const RAIL_IEEE802154_AddrConfig_t* RAIL_IEEE802154_Config_t::addresses
```

Configure the RAIL Address Filter to allow the given destination addresses.

If this pointer is NULL, defer destination address configuration. If a member of addresses is NULL, defer configuration of just that member. This can be overridden via RAIL_IEEE802154_SetAddresses(), or the individual members can be changed via RAIL_IEEE802154_SetPanId(), RAIL_IEEE802154_SetShortAddress(), and RAIL_IEEE802154_SetLongAddress().

Definition at line `283` of file `protocol/ieee802154/rail_ieee802154.h`

ackConfig

> RAIL_AutoAckConfig_t RAIL_IEEE802154_Config_t::ackConfig

Define the ACKing configuration for the IEEE 802.15.4 implementation.

Definition at line `287` of file `protocol/ieee802154/rail_ieee802154.h`

timings

> RAIL_StateTiming_t RAIL_IEEE802154_Config_t::timings

Define state timings for the IEEE 802.15.4 implementation.

Definition at line `291` of file `protocol/ieee802154/rail_ieee802154.h`

framesMask

> uint8_t RAIL_IEEE802154_Config_t::framesMask

Set which 802.15.4 frame types will be received, of Beacon, Data, ACK, and Command.

This setting can be overridden via RAIL_IEEE802154_AcceptFrames().

Definition at line `296` of file `protocol/ieee802154/rail_ieee802154.h`

promiscuousMode

> bool RAIL_IEEE802154_Config_t::promiscuousMode

Enable promiscuous mode during configuration.

This can be overridden via RAIL_IEEE802154_SetPromiscuousMode() afterwards.

Definition at line `301` of file `protocol/ieee802154/rail_ieee802154.h`

isPanCoordinator

> bool RAIL_IEEE802154_Config_t::isPanCoordinator

Set whether the device is a PAN Coordinator during configuration.

This can be overridden via RAIL_IEEE802154_SetPanCoordinator() afterwards.

Definition at line `306` of file `protocol/ieee802154/rail_ieee802154.h`

defaultFramePendingInOutgoingAcks

> bool RAIL_IEEE802154_Config_t::defaultFramePendingInOutgoingAcks

The default value for the Frame Pending bit in outgoing ACKs for packets that triggered the RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND event.

Such an ACK's Frame Pending bit can be inverted if necessary during the handling of that event by calling RAIL_IEEE802154_ToggleFramePending (formerly RAIL_IEEE802154_SetFramePending).

Definition at line 314 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_RxChannelSwitchingCfg_t

A configuration structure for RX channel switching.

Public Attributes

uint32_t *	buffer Pointer to contiguous global read-write memory that will be used by RAIL to store channel specific settings for concurrent listening.
uint16_t	bufferBytes This parameter must be set to the length of the buffer array, in bytes.
uint16_t	channels Array to hold the channel numbers for RX channel switching.

Public Attribute Documentation

buffer

```
uint32_t* RAIL_IEEE802154_RxChannelSwitchingCfg_t::buffer
```

Pointer to contiguous global read-write memory that will be used by RAIL to store channel specific settings for concurrent listening.

It need not be initialized and applications should never write data anywhere in this buffer.

Note

- the size of this buffer must be at least as large as the RAIL_IEEE802154_RX_CHANNEL_SWITCHING_BUF_BYTES and needs to be word aligned.

Definition at line `354` of file `protocol/ieee802154/rail_ieee802154.h`

bufferBytes

```
uint16_t RAIL_IEEE802154_RxChannelSwitchingCfg_t::bufferBytes
```

This parameter must be set to the length of the buffer array, in bytes.

This way, during configuration, the software can confirm it's writing within the range of the buffer. The configuration API will return an error if bufferBytes is insufficient.

Definition at line `361` of file `protocol/ieee802154/rail_ieee802154.h`

channels

```
uint16_t RAIL_IEEE802154_RxChannelSwitchingCfg_t::channels[(2U)]
```

Array to hold the channel numbers for RX channel switching.

Note

- Radio will switch between the exact channels specified, and not across an inclusive range of channels between the specified channels.

Definition at line `367` of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_ModeSwitchPhr_t

A structure containing the PHYModeID value and the corresponding mode switch PHR as defined in Wi-SUN spec.

These structures are usually generated by the radio configurator.

Public Attributes

uint8_t	phyModeId PHY mode Id.
uint16_t	phr Corresponding Mode Switch PHY header.

Public Attribute Documentation

phyModeId

```
uint8_t RAIL_IEEE802154_ModeSwitchPhr_t::phyModeId
```

PHY mode Id.

Definition at line 1161 of file protocol/ieee802154/rail_ieee802154.h

phr

```
uint16_t RAIL_IEEE802154_ModeSwitchPhr_t::phr
```

Corresponding Mode Switch PHY header.

Definition at line 1162 of file protocol/ieee802154/rail_ieee802154.h

IEEE 802.15.4 Radio Configurations

IEEE 802.15.4 Radio Configurations

Radio configurations for the RAIL 802.15.4 Accelerator.

These radio configurations are used to configure 802.15.4 when a function such as RAIL_IEEE802154_Config2p4GHzRadio() is called. Each radio configuration listed below is compiled into the RAIL library as a weak symbol that will take into account per-die defaults. If the board configuration in use has different settings than the default, such as a different radio subsystem clock frequency, these radio configurations can be overridden to account for those settings.

Variables

const RAIL_ChannelConfig_t *const — **RAIL_IEEE802154_Phy2p4GHz**
Default PHY to use for 2.4 GHz 802.15.4.

const RAIL_ChannelConfig_t *const — **RAIL_IEEE802154_Phy2p4GHzAntDiv**
Default PHY to use for 2.4 GHz 802.15.4 with antenna diversity.

const RAIL_ChannelConfig_t *const — **RAIL_IEEE802154_Phy2p4GHzCoex**
Default PHY to use for 2.4 GHz 802.15.4 optimized for coexistence.

const RAIL_ChannelConfig_t *const — **RAIL_IEEE802154_Phy2p4GHzAntDivCoex**
Default PHY to use for 2.4 GHz 802.15.4 optimized for coexistence, while supporting antenna diversity.

const RAIL_ChannelConfig_t *const — **RAIL_IEEE802154_Phy2p4GHzFem**
Default PHY to use for 2.4 GHz 802.15.4 with a configuration that supports a front-end module.

const RAIL_ChannelConfig_t *const — **RAIL_IEEE802154_Phy2p4GHzAntDivFem**
Default PHY to use for 2.4 GHz 802.15.4 with a configuration that supports a front-end module and antenna diversity.

const RAIL_ChannelConfig_t *const — **RAIL_IEEE802154_Phy2p4GHzCoexFem**
Default PHY to use for 2.4 GHz 802.15.4 with a configuration that supports a front-end module and is optimized for radio coexistence.

const RAIL_ChannelConfig_t *const — **RAIL_IEEE802154_Phy2p4GHzAntDivCoexFem**
Default PHY to use for 2.4 GHz 802.15.4 with a configuration that supports a front-end module and antenna diversity, and is optimized for radio coexistence.

const RAIL_ChannelConfig_t *const — **RAIL_IEEE802154_Phy2p4GHzCustom1**
Default PHY to use for 2.4 GHz 802.15.4 with custom settings.

const RAIL_ChannelConfig_t *const — **RAIL_IEEE802154_PhyGB863MHz**
Default PHY to use for 863MHz GB868 802.15.4.

const RAIL_ChannelConfig_t *const — **RAIL_IEEE802154_PhyGB915MHz**
Default PHY to use for 915MHz GB868 802.15.4.

const RAIL_ChannelConfig_t *const	RAIL_IEEE802154_Phy2p4GHzRxChSwitching Default PHY to use for 2.4 GHz 802.15.4 with RX channel switching.

Variable Documentation

RAIL_IEEE802154_Phy2p4GHz

```
const RAIL_ChannelConfig_t* const RAIL_IEEE802154_Phy2p4GHz
```

Default PHY to use for 2.4 GHz 802.15.4.

Will be NULL if RAIL_SUPPORTS_PROTOCOL_IEEE802154 or RAIL_SUPPORTS_2P4GHZ_BAND is 0.

Definition at line 387 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_Phy2p4GHzAntDiv

```
const RAIL_ChannelConfig_t* const RAIL_IEEE802154_Phy2p4GHzAntDiv
```

Default PHY to use for 2.4 GHz 802.15.4 with antenna diversity.

Will be NULL if RAIL_SUPPORTS_PROTOCOL_IEEE802154, RAIL_SUPPORTS_2P4GHZ_BAND, or RAIL_SUPPORTS_ANTENNA_DIVERSITY is 0.

Definition at line 402 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_Phy2p4GHzCoex

```
const RAIL_ChannelConfig_t* const RAIL_IEEE802154_Phy2p4GHzCoex
```

Default PHY to use for 2.4 GHz 802.15.4 optimized for coexistence.

Will be NULL if RAIL_IEEE802154_SUPPORTS_COEX_PHY is 0.

Definition at line 408 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_Phy2p4GHzAntDivCoex

```
const RAIL_ChannelConfig_t* const RAIL_IEEE802154_Phy2p4GHzAntDivCoex
```

Default PHY to use for 2.4 GHz 802.15.4 optimized for coexistence, while supporting antenna diversity.

Will be NULL if RAIL_SUPPORTS_ANTENNA_DIVERSITY or RAIL_IEEE802154_SUPPORTS_COEX_PHY is 0.

Definition at line 416 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_Phy2p4GHzFem

```
const RAIL_ChannelConfig_t* const RAIL_IEEE802154_Phy2p4GHzFem
```

Default PHY to use for 2.4 GHz 802.15.4 with a configuration that supports a front-end module.

Will be NULL if RAIL_IEEE802154_SUPPORTS_FEM_PHY is 0.

Definition at line 423 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_Phy2p4GHzAntDivFem

```
const RAIL_ChannelConfig_t* const RAIL_IEEE802154_Phy2p4GHzAntDivFem
```

Default PHY to use for 2.4 GHz 802.15.4 with a configuration that supports a front-end module and antenna diversity.

Will be NULL if RAIL_IEEE802154_SUPPORTS_FEM_PHY or RAIL_SUPPORTS_ANTENNA_DIVERSITY is 0.

Definition at line 431 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_Phy2p4GHzCoexFem

```
const RAIL_ChannelConfig_t* const RAIL_IEEE802154_Phy2p4GHzCoexFem
```

Default PHY to use for 2.4 GHz 802.15.4 with a configuration that supports a front-end module and is optimized for radio coexistence.

Will be NULL if RAIL_IEEE802154_SUPPORTS_FEM_PHY or RAIL_IEEE802154_SUPPORTS_COEX_PHY is 0.

Definition at line 439 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_Phy2p4GHzAntDivCoexFem

```
const RAIL_ChannelConfig_t* const RAIL_IEEE802154_Phy2p4GHzAntDivCoexFem
```

Default PHY to use for 2.4 GHz 802.15.4 with a configuration that supports a front-end module and antenna diversity, and is optimized for radio coexistence.

Will be NULL if RAIL_IEEE802154_SUPPORTS_FEM_PHY, RAIL_IEEE802154_SUPPORTS_COEX_PHY, or RAIL_SUPPORTS_ANTENNA_DIVERSITY is 0.

Definition at line 448 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_Phy2p4GHzCustom1

```
const RAIL_ChannelConfig_t* const RAIL_IEEE802154_Phy2p4GHzCustom1
```

Default PHY to use for 2.4 GHz 802.15.4 with custom settings.

Will be NULL if RAIL_IEEE802154_SUPPORTS_CUSTOM1_PHY is 0.

Definition at line 454 of file protocol/ieee802154/rail_ieee802154.h

RAIL_IEEE802154_PhyGB863MHz

> const RAIL_ChannelConfig_t* const RAIL_IEEE802154_PhyGB863MHz

Default PHY to use for 863MHz GB868 802.15.4.

Will be NULL if RAIL_IEEE802154_SUPPORTS_G_SUBSET_GB868 is 0.

Definition at line `460` of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_PhyGB915MHz

> const RAIL_ChannelConfig_t* const RAIL_IEEE802154_PhyGB915MHz

Default PHY to use for 915MHz GB868 802.15.4.

Will be NULL if RAIL_IEEE802154_SUPPORTS_G_SUBSET_GB868 is 0.

Definition at line `466` of file `protocol/ieee802154/rail_ieee802154.h`

RAIL_IEEE802154_Phy2p4GHzRxChSwitching

> const RAIL_ChannelConfig_t* const RAIL_IEEE802154_Phy2p4GHzRxChSwitching

Default PHY to use for 2.4 GHz 802.15.4 with RX channel switching.

Will be NULL if RAIL_IEEE802154_SUPPORTS_RX_CHANNEL_SWITCHING is 0.

Definition at line `472` of file `protocol/ieee802154/rail_ieee802154.h`

Multi-Level Frequency Modulation

Multi-Level Frequency Modulation

MFM configuration routines Note that this feature is only supported on EFR32xG23 devices.

This feature can be used to directly control the TX interpolation filter input to allow for a more flexible frequency modulation scheme than the standard MODEM. When doing this, the MFM buffer is treated as an array of 8-bit signed data used as normalized frequency deviation to the SYNTH frequency to directly control the interpolation filter input. No support for frame handling, coding, nor shaping is supported. Only compatible with FSK modulations.

The functions in this group configure RAIL Multi-Level Frequency Modulation (MFM) hardware acceleration features.

To configure MFM functionality, the application must first set up a RAIL instance with RAIL_Init() and other setup functions. Before enabling MFM, a ping-pong buffer (called buffer0 and buffer1 below) must be configured via RAIL_SetMfmPingPongFifo() and populated with the initial buffer content. MFM is enabled by setting RAIL_TxDataSource_t::TX_MFM_DATA using RAIL_ConfigData() and is activated when transmit is started by RAIL_StartTx(). Once transmitting the data in the ping-pong buffers, RAIL will manage them so it looks like a continuous transmission to the receiver. Every time one of the ping-ping buffers has been transmitted, RAIL_EVENT_MFM_TX_BUFFER_DONE is triggered so the application can update the data in that buffer without the need to start/stop the transmission. RAIL_EVENT_MFM_TX_BUFFER_DONE can be enable with RAIL_ConfigEvents(). Use RAIL_StopTx() to finish transmitting.

```
uint8_t txCount = 0;

typedef struct RAIL_MFM_Config_App {
  RAIL_MFM_PingPongBufferConfig_t buffer;
  RAIL_StateTiming_t timings;
} RAIL_MFM_Config_App_t;

// Main RAIL_EVENT callback
static void RAILCb_Event(RAIL_Handle_t railHandle, RAIL_Events_t events)
{
  // Increment TX counter
  if (events & RAIL_EVENT_MFM_BUF_DONE) {
      txCount++;
      return;
    }
  }
}

static const RAIL_MFM_Config_App_t mfmConfig = {
  .buffer = {
    .pBuffer0 = (&channelHoppingBufferSpace[0]),
    .pBuffer1 = (&channelHoppingBufferSpace[MFM_RAW_BUF_SZ_BYTES / 4]),
    .bufferSizeWords = (MFM_RAW_BUF_SZ_BYTES / 4)
  },
  .timings = {
    .idleToTx = 100,
    .idleToRx = 0,
    .rxToTx = 0,
    .txToRx = 0,
    .rxSearchTimeout = 0,
    .txToRxSearchTimeout = 0
};

RAIL_Status_t mfmInit(void)
{
  // initialize MFM
  uint32_t idx;
  uint32_t *pDst0 = mfmConfig.pBuffer0;
  uint32_t *pDst1 = mfmConfig.pBuffer1;
  RAIL_Status_t status;
  for (idx = 0; idx < (MFM_RAW_BUF_SZ_BYTES / 16); idx++) {
    pDst0[4 * idx + 0] = 0x755A3100;
    pDst1[4 * idx + 0] = 0x755A3100;
    pDst0[4 * idx + 1] = 0x315A757F;
    pDst1[4 * idx + 1] = 0x315A757F;
    pDst0[4 * idx + 2] = 0x8BA6CF00;
    pDst1[4 * idx + 2] = 0x8BA6CF00;
    pDst0[4 * idx + 3] = 0xCFA68B81;
    pDst1[4 * idx + 3] = 0xCFA68B81;
  }

  RAIL_Status_t status;
  railDataConfig.txSource = TX_MFM_DATA;
  status = RAIL_SetMfmPingPongFifo(railHandle,
                                   &(config->buffer));
  if (status != RAIL_STATUS_NO_ERROR) {
    return (status);
  }


  status = RAIL_ConfigData(railHandle, &railDataConfig);
  if (status != RAIL_STATUS_NO_ERROR) {
    return (status);
  }

  status = RAIL_SetStateTiming(railHandle, &(config->timings));
```

```
return(status);}// start transmittingreturn(RAIL_StartTx(railHandle,0,0,&schedulerInfo));}

RAIL_Status_t mfmDeInit(void){
  RAIL_Status_t status;
  status =RAIL_StopTx(railHandle, RAIL_STOP_MODES_ALL);if(status != RAIL_STATUS_NO_ERROR){return(status);}

  railDataConfig.txSource = TX_PACKET_DATA;return(RAIL_ConfigData(railHandle,&railDataConfig));}
```

Modules

RAIL_MFM_PingPongBufferConfig_t

Functions

RAIL_Status_t	RAIL_SetMfmPingPongFifo(RAIL_Handle_t railHandle, const RAIL_MFM_PingPongBufferConfig_t *config) Set MFM ping-pong buffer.

Function Documentation

RAIL_SetMfmPingPongFifo

> RAIL_Status_t RAIL_SetMfmPingPongFifo (RAIL_Handle_t railHandle, const RAIL_MFM_PingPongBufferConfig_t *config)

Set MFM ping-pong buffer.

Parameters

[in]	railHandle	A handle of RAIL instance.
[in]	config	A MFM ping-pong buffer configuration structure.

Returns

- A status code indicating success of the function call.

Definition at line 184 of file common/rail_mfm.h

RAIL_MFM_PingPongBufferConfig_t

A configuration structure for MFM Ping-pong buffer in RAIL.

Public Attributes

uint32_t *	pBuffer0 pointer to buffer0.
uint32_t *	pBuffer1 pointer to buffer1.
uint32_t	bufferSizeWords size of each buffer A and B in 32-bit words.

Public Attribute Documentation

pBuffer0

```
uint32_t* RAIL_MFM_PingPongBufferConfig_t::pBuffer0
```

pointer to buffer0.

Must be 32-bit aligned.

Definition at line 169 of file common/rail_mfm.h

pBuffer1

```
uint32_t* RAIL_MFM_PingPongBufferConfig_t::pBuffer1
```

pointer to buffer1.

Must be 32-bit aligned.

Definition at line 171 of file common/rail_mfm.h

bufferSizeWords

```
uint32_t RAIL_MFM_PingPongBufferConfig_t::bufferSizeWords
```

size of each buffer A and B in 32-bit words.

Definition at line 173 of file common/rail_mfm.h

Sidewalk Radio Configurations

Sidewalk Radio Configurations

Radio configurations for the RAIL Sidewalk Accelerator.

These radio configurations are used to configure Sidewalk when a function such as RAIL_Sidewalk_ConfigPhy2GFSK50kbps() is called. Each radio configuration listed below is compiled into the RAIL library as a weak symbol that will take into account per-die defaults. If the board configuration in use has different settings than the default, such as a different radio subsystem clock frequency, these radio configurations can be overriden to account for those settings.

Variables

const RAIL_ChannelConfig_t *const	RAIL_Sidewalk_Phy2GFSK50kbps Default PHY to use for Sidewalk 2GFSK 50kbps.

Functions

RAIL_Status_t	RAIL_Sidewalk_ConfigPhy2GFSK50kbps(RAIL_Handle_t railHandle) Switch to the 2GFSK 50kbps Sidewalk PHY.

Variable Documentation

RAIL_Sidewalk_Phy2GFSK50kbps

```
const RAIL_ChannelConfig_t* const RAIL_Sidewalk_Phy2GFSK50kbps
```

Default PHY to use for Sidewalk 2GFSK 50kbps.

Will be NULL if RAIL_SUPPORTS_PROTOCOL_SIDEWALK is 0.

Definition at line 62 of file protocol/sidewalk/rail_sidewalk.h

Function Documentation

RAIL_Sidewalk_ConfigPhy2GFSK50kbps

```
RAIL_Status_t RAIL_Sidewalk_ConfigPhy2GFSK50kbps (RAIL_Handle_t railHandle)
```

Switch to the 2GFSK 50kbps Sidewalk PHY.

Parameters

[in]	railHandle	A handle for RAIL instance.

Returns

- A status code indicating success of the function call.

Use this function to switch to the 2GFSK 50kbps Sidewalk PHY.

Note

- The Sidewalk PHY is supported only on some parts. The preprocessor symbol RAIL_SUPPORTS_PROTOCOL_SIDEWALK and the runtime function RAIL_SupportsProtocolSidewalk() may be used to test for support of the Sidewalk PHY.

Definition at line `77` of file `protocol/sidewalk/rail_sidewalk.h`

Z-Wave

Z-Wave

Z-Wave configuration routines.

The functions in this group configure RAIL Z-Wave hardware acceleration features.

To configure Z-Wave functionality, the application must first set up a RAIL instance with RAIL_Init() and other setup functions.

```
RAIL_ZWAVE_NodeId_t gRecentBeamNodeId;
uint8_t gRecentBeamChannelIndex;

// Main RAIL_EVENT callback
static void RAILCb_Event(RAIL_Handle_t railHandle, RAIL_Events_t events)
{
  // Get beamNodeId and channel index from beam packet
  if (events & RAIL_EVENT_ZWAVE_BEAM) {
    if (RAIL_ZWAVE_IsEnabled(railHandle)) {
      if ((RAIL_ZWAVE_GetBeamNodeId(railHandle, &gRecentBeamNodeId)
           != RAIL_STATUS_NO_ERROR)
          || (RAIL_ZWAVE_GetBeamChannelIndex(railHandle, &gRecentBeamChannelIndex)
              != RAIL_STATUS_NO_ERROR)) {
        return;
      }
    }
  }
}

static const RAIL_ZWAVE_Config_t zwaveConfig = {
  .options = RAIL_ZWAVE_OPTIONS_DEFAULT
};

RAIL_Status_t zwaveInit(void)
{
  // initialize Z-Wave
  RAIL_Status_t status = RAIL_ZWAVE_Init(railHandle, &zwaveConfig);

  if (status != RAIL_STATUS_NO_ERROR) {
    return status;
  }

  uint8_t myHomeId[4] = { 0xDE, 0xAD, 0xBE, 0xEF };
  RAIL_ZWAVE_SetNodeId(railHandle, RAIL_ZWAVE_NODE_ID_DEFAULT);
  RAIL_ZWAVE_SetHomeId(railHandle, myHomeId, RAIL_ZWAVE_HOME_ID_HASH_DONT_CARE);

  // configure region to EU(European Union)
  return RAIL_ZWAVE_ConfigRegion(railHandle, RAIL_ZWAVE_REGION_EU);
}
```

Modules

RAIL_ZWAVE_Config_t

RAIL_ZWAVE_LrAckData_t

RAIL_ZWAVE_BeamRxConfig_t

RAIL_ZWAVE_RegionConfig_t

RAIL_ZWAVE_IrcalVal_t

Enumerations

enum RAIL_ZWAVE_Options_t {

RAIL_ZWAVE_OPTION_PROMISCUOUS_MODE_SHIFT = 0
RAIL_ZWAVE_OPTION_DETECT_BEAM_FRAMES_SHIFT
RAIL_ZWAVE_OPTION_NODE_ID_FILTERING_SHIFT
RAIL_ZWAVE_OPTION_PROMISCUOUS_BEAM_MODE_SHIFT

}

Z-Wave options.

enum RAIL_ZWAVE_NodeId_t {

RAIL_ZWAVE_NODE_ID_NONE = 0×00U
RAIL_ZWAVE_NODE_ID_BROADCAST = 0xFFU
RAIL_ZWAVE_NODE_ID_DEFAULT = RAIL_ZWAVE_NODE_ID_BROADCAST
RAIL_ZWAVE_NODE_ID_BROADCAST_LONGRANGE = 0xFFFU
RAIL_ZWAVE_NODE_ID_DEFAULT_LONGRANGE = RAIL_ZWAVE_NODE_ID_BROADCAST_LONGRANGE

}

A Z-Wave Node ID.

enum RAIL_ZWAVE_HomeId_t {

RAIL_ZWAVE_HOME_ID_UNKNOWN = 0×00000000U
RAIL_ZWAVE_HOME_ID_DEFAULT = 0×54545454U

}

A Z-Wave Home ID.

enum RAIL_ZWAVE_HomeIdHash_t {

RAIL_ZWAVE_HOME_ID_HASH_ILLEGAL_1 = 0×0AU
RAIL_ZWAVE_HOME_ID_HASH_ILLEGAL_2 = 0×4AU
RAIL_ZWAVE_HOME_ID_HASH_ILLEGAL_3 = 0×55U
RAIL_ZWAVE_HOME_ID_HASH_DONT_CARE = 0×55U
RAIL_ZWAVE_HOME_ID_HASH_DEFAULT = RAIL_ZWAVE_HOME_ID_HASH_DONT_CARE

}

A Z-Wave Home ID hash.

enum RAIL_ZWAVE_Baud_t {

RAIL_ZWAVE_BAUD_9600
RAIL_ZWAVE_BAUD_40K
RAIL_ZWAVE_BAUD_100K
RAIL_ZWAVE_LR
RAIL_ZWAVE_ENERGY_DETECT = RAIL_ZWAVE_LR
RAIL_ZWAVE_BAUD_INVALID

}

Z-Wave supported baudrates or PHYs.

enum RAIL_ZWAVE_RegionId_t {

RAIL_ZWAVE_REGIONID_UNKNOWN
RAIL_ZWAVE_REGIONID_EU
RAIL_ZWAVE_REGIONID_US
RAIL_ZWAVE_REGIONID_ANZ
RAIL_ZWAVE_REGIONID_HK
RAIL_ZWAVE_REGIONID_MY
RAIL_ZWAVE_REGIONID_IN
RAIL_ZWAVE_REGIONID_JP
RAIL_ZWAVE_REGIONID_RU
RAIL_ZWAVE_REGIONID_IL
RAIL_ZWAVE_REGIONID_KR
RAIL_ZWAVE_REGIONID_CN
RAIL_ZWAVE_REGIONID_US_LR1
RAIL_ZWAVE_REGIONID_US_LR2
RAIL_ZWAVE_REGIONID_US_LR_END_DEVICE
RAIL_ZWAVE_REGIONID_EU_LR1
RAIL_ZWAVE_REGIONID_EU_LR2
RAIL_ZWAVE_REGIONID_EU_LR_END_DEVICE
RAIL_ZWAVE_REGIONID_COUNT

}

Z-Wave region identifications.

Typedefs

typedef RAIL_RxChannelHoppingParameter_t — RAIL_RxChannelHoppingParameters_t[(4U)]

Rx channel hopping on-channel time for all Z-Wave channels in a region.

Variables

const RAIL_ZWAVE_RegionConfig_t — RAIL_ZWAVE_REGION_EU

EU-European Union, RAIL_ZWAVE_REGION_EU.

const RAIL_ZWAVE_RegionConfig_t — RAIL_ZWAVE_REGION_US

US-United States, RAIL_ZWAVE_REGION_US.

const RAIL_ZWAVE_RegionConfig_t — RAIL_ZWAVE_REGION_ANZ

ANZ-Australia/New Zealand, RAIL_ZWAVE_REGION_ANZ.

const RAIL_ZWAVE_RegionConfig_t — RAIL_ZWAVE_REGION_HK

HK-Hong Kong, RAIL_ZWAVE_REGION_HK.

const RAIL_ZWAVE_RegionConfig_t — RAIL_ZWAVE_REGION_MY

MY-Malaysia, RAIL_ZWAVE_REGION_MY.

const RAIL_ZWAVE_RegionConfig_t — RAIL_ZWAVE_REGION_IN

IN-India, RAIL_ZWAVE_REGION_IN.

const RAIL_ZWAVE_RegionConfig_t — RAIL_ZWAVE_REGION_JP

JP-Japan, RAIL_ZWAVE_REGION_JP.

const RAIL_ZWAVE_RegionConfig_t	RAIL_ZWAVE_REGION_JPED JP-Japan, RAIL_ZWAVE_REGION_JP.
const RAIL_ZWAVE_RegionConfig_t	RAIL_ZWAVE_REGION_RU RU-Russia, RAIL_ZWAVE_REGION_RU.
const RAIL_ZWAVE_RegionConfig_t	RAIL_ZWAVE_REGION_IL IL-Israel, RAIL_ZWAVE_REGION_IL.
const RAIL_ZWAVE_RegionConfig_t	RAIL_ZWAVE_REGION_KR KR-Korea, RAIL_ZWAVE_REGION_KR.
const RAIL_ZWAVE_RegionConfig_t	RAIL_ZWAVE_REGION_KRED KR-Korea, RAIL_ZWAVE_REGION_KR.
const RAIL_ZWAVE_RegionConfig_t	RAIL_ZWAVE_REGION_CN CN-China, RAIL_ZWAVE_REGION_CN.
const RAIL_ZWAVE_RegionConfig_t	RAIL_ZWAVE_REGION_US_LR1 US-Long Range 1, RAIL_ZWAVE_REGION_US_LR1.
const RAIL_ZWAVE_RegionConfig_t	RAIL_ZWAVE_REGION_US_LR2 US-Long Range 2, RAIL_ZWAVE_REGION_US_LR2.
const RAIL_ZWAVE_RegionConfig_t	RAIL_ZWAVE_REGION_US_LR_END_DEVICE US-Long Range End Device, RAIL_ZWAVE_REGION_US_LR_END_DEVICE.
const RAIL_ZWAVE_RegionConfig_t	RAIL_ZWAVE_REGION_EU_LR1 EU-Long Range 1, RAIL_ZWAVE_REGION_EU_LR1.
const RAIL_ZWAVE_RegionConfig_t	RAIL_ZWAVE_REGION_EU_LR2 EU-Long Range 2, RAIL_ZWAVE_REGION_EU_LR2.
const RAIL_ZWAVE_RegionConfig_t	RAIL_ZWAVE_REGION_EU_LR_END_DEVICE EU-Long Range End Device, RAIL_ZWAVE_REGION_EU_LR_END_DEVICE.
const RAIL_ZWAVE_RegionConfig_t	RAIL_ZWAVE_REGION_INVALID Invalid Region.

Functions

RAIL_Status_t	RAIL_ZWAVE_ConfigRegion(RAIL_Handle_t railHandle, const RAIL_ZWAVE_RegionConfig_t *regionCfg) Switch the Z-Wave region.
RAIL_Status_t	RAIL_ZWAVE_PerformIrcal(RAIL_Handle_t railHandle, RAIL_ZWAVE_IrcalVal_t *pIrCalVals, bool forceIrcal) Perform image rejection calibration on all valid channels of a Z-Wave region.
RAIL_Status_t	RAIL_ZWAVE_Init(RAIL_Handle_t railHandle, const RAIL_ZWAVE_Config_t *config) Initialize RAIL for Z-Wave features.

RAIL_Status_t	RAIL_ZWAVE_Deinit(RAIL_Handle_t railHandle) De-initialize Z-Wave hardware acceleration.
bool	RAIL_ZWAVE_IsEnabled(RAIL_Handle_t railHandle) Return whether Z-Wave hardware acceleration is currently enabled.
RAIL_Status_t	RAIL_ZWAVE_ConfigOptions(RAIL_Handle_t railHandle, RAIL_ZWAVE_Options_t mask, RAIL_ZWAVE_Options_t options) Configure Z-Wave options.
RAIL_Status_t	RAIL_ZWAVE_SetNodeId(RAIL_Handle_t railHandle, RAIL_ZWAVE_NodeId_t nodeId) Inform RAIL of the Z-Wave node's NodeId for receive filtering.
RAIL_Status_t	RAIL_ZWAVE_SetHomeId(RAIL_Handle_t railHandle, RAIL_ZWAVE_HomeId_t homeId, RAIL_ZWAVE_HomeIdHash_t homeIdHash) Inform RAIL of the Z-Wave node's HomeId and its hash for receive filtering.
RAIL_Status_t	RAIL_ZWAVE_GetBeamNodeId(RAIL_Handle_t railHandle, RAIL_ZWAVE_NodeId_t *pNodeId) Get the NodeId of the most recently seen beam frame that triggered RAIL_EVENT_ZWAVE_BEAM.
RAIL_Status_t	RAIL_ZWAVE_GetBeamHomeIdHash(RAIL_Handle_t railHandle, RAIL_ZWAVE_HomeIdHash_t *pBeamHomeIdHash) Get the HomeIdHash of the most recently seen beam frame that triggered RAIL_EVENT_ZWAVE_BEAM.
RAIL_Status_t	RAIL_ZWAVE_GetBeamChannelIndex(RAIL_Handle_t railHandle, uint8_t *pChannelIndex) Get the channel hopping index of the most recently seen beam frame that triggered RAIL_EVENT_ZWAVE_BEAM.
RAIL_Status_t	RAIL_ZWAVE_GetLrBeamTxPower(RAIL_Handle_t railHandle, uint8_t *pLrBeamTxPower) Get the TX power used by the transmitter of the most recently seen long range beam frame that triggered RAIL_EVENT_ZWAVE_BEAM.
RAIL_Status_t	RAIL_ZWAVE_GetBeamRssi(RAIL_Handle_t railHandle, int8_t *pBeamRssi) Get the RSSI of the received beam frame.
RAIL_Status_t	RAIL_ZWAVE_SetTxLowPower(RAIL_Handle_t railHandle, uint8_t powerLevel) Set the Raw Low Power settings.
RAIL_Status_t	RAIL_ZWAVE_SetTxLowPowerDbm(RAIL_Handle_t railHandle, RAIL_TxPower_t powerLevel) Set the Low Power settings in dBm.
RAIL_TxPowerLevel_t	RAIL_ZWAVE_GetTxLowPower(RAIL_Handle_t railHandle) Get the TX low power in raw units (see rail_chip_specific.h for value ranges).
RAIL_TxPower_t	RAIL_ZWAVE_GetTxLowPowerDbm(RAIL_Handle_t railHandle) Get the TX low power in terms of deci-dBm instead of raw power level.
RAIL_Status_t	RAIL_ZWAVE_ReceiveBeam(RAIL_Handle_t railHandle, uint8_t *beamDetectIndex, const RAIL_SchedulerInfo_t *schedulerInfo) Implement beam detection and reception algorithms.
RAIL_Status_t	RAIL_ZWAVE_ConfigBeamRx(RAIL_Handle_t railHandle, RAIL_ZWAVE_BeamRxConfig_t *config) Configure the receive algorithm used in RAIL_ZWAVE_ReceiveBeam.
RAIL_Status_t	RAIL_ZWAVE_SetDefaultRxBeamConfig(RAIL_Handle_t railHandle) Set the default RX beam configuration.
RAIL_Status_t	RAIL_ZWAVE_GetRxBeamConfig(RAIL_ZWAVE_BeamRxConfig_t *pConfig) Get the current RX beam configuration.
RAIL_Status_t	RAIL_ZWAVE_ConfigRxChannelHopping(RAIL_Handle_t railHandle, RAIL_RxChannelHoppingConfig_t *config) Configure the channel hop timings for use in Z-Wave RX channel hop configuration.

RAIL_ZWAVE_RegionId_t RAIL_ZWAVE_GetRegion(RAIL_Handle_t railHandle)
Get the Z-Wave region.

RAIL_Status_t RAIL_ZWAVE_SetLrAckData(RAIL_Handle_t railHandle, const RAIL_ZWAVE_LrAckData_t *pLrAckData)
Write the AutoACK FIFO for the next outgoing Z-Wave Long Range ACK.

Macros

#define RAIL_ZWAVE_OPTIONS_NONE 0U
A value representing no options.

#define RAIL_ZWAVE_OPTIONS_DEFAULT RAIL_ZWAVE_OPTIONS_NONE
All options are disabled by default.

#define RAIL_ZWAVE_OPTION_PROMISCUOUS_MODE (1u << RAIL_ZWAVE_OPTION_PROMISCUOUS_MODE_SHIFT)
An option to configure promiscuous mode, accepting non-beam packets regardless of their HomeId.

#define RAIL_ZWAVE_OPTION_NODE_ID_FILTERING (1u << RAIL_ZWAVE_OPTION_NODE_ID_FILTERING_SHIFT)
An option to filter non-beam packets based on their NodeId when RAIL_ZWAVE_OPTION_PROMISCUOUS_MODE is disabled.

#define RAIL_ZWAVE_OPTION_DETECT_BEAM_FRAMES (1u << RAIL_ZWAVE_OPTION_DETECT_BEAM_FRAMES_SHIFT)
An option to configure beam frame recognition.

#define RAIL_ZWAVE_OPTION_PROMISCUOUS_BEAM_MODE (1u << RAIL_ZWAVE_OPTION_PROMISCUOUS_BEAM_MODE_SHIFT)
An option to receive all beams promiscuously when RAIL_ZWAVE_OPTION_DETECT_BEAM_FRAMES is enabled.

#define RAIL_ZWAVE_OPTIONS_ALL 0xFFFFFFFFU
A value representing all options.

#define RAIL_ZWAVE_FREQ_INVALID 0xFFFFFFFFUL
Sentinel value to indicate that a channel (and thus its frequency) are invalid.

#define RAIL_ZWAVE_LR_BEAM_TX_POWER_INVALID (0xFFU)
Invalid beam TX power value returned when RAIL_ZWAVE_GetLrBeamTxPower is called after receiving a regular non-long-range beam.

#define RAIL_NUM_ZWAVE_CHANNELS (4U)
Number of channels in each of Z-Wave's region-based PHYs.

Enumeration Documentation

RAIL_ZWAVE_Options_t

> RAIL_ZWAVE_Options_t

Z-Wave options.

Enumerator	
RAIL_ZWAVE_OPTION_PROMISCUOUS_MODE_SHIFT	Shift position of RAIL_ZWAVE_OPTION_PROMISCUOUS_MODE bit.
RAIL_ZWAVE_OPTION_DETECT_BEAM_FRAMES_SHIFT	Shift position of RAIL_ZWAVE_OPTION_DETECT_BEAM_FRAMES bit.
RAIL_ZWAVE_OPTION_NODE_ID_FILTERING_SHIFT	Shift position of RAIL_ZWAVE_OPTION_NODE_ID_FILTERING bit.
RAIL_ZWAVE_OPTION_PROMISCUOUS_BEAM_MODE_SHIFT	Shift position of RAIL_ZWAVE_OPTION_PROMISCUOUS_BEAM_MODE bit.

Definition at line 99 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_NodeId_t

> RAIL_ZWAVE_NodeId_t

A Z-Wave Node ID.

This data type is 12 bits wide when using the ZWave Long Range PHY, and 8 bits wide otherwise.

Note

- When using the Long Range PHY, values 0xFA1..0xFFE are reserved. Otherwise, values 0xE9..0xFE are reserved.

Enumerator	
RAIL_ZWAVE_NODE_ID_NONE	The unknown NodeId for uninitialized nodes.
RAIL_ZWAVE_NODE_ID_BROADCAST	The broadcast NodeId.
RAIL_ZWAVE_NODE_ID_DEFAULT	Default to the broadcast NodeId.
RAIL_ZWAVE_NODE_ID_BROADCAST_LONGRANGE	The Long Range broadcast NodeId.
RAIL_ZWAVE_NODE_ID_DEFAULT_LONGRANGE	Default to the Long Range broadcast NodeId.

Definition at line 181 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_HomeId_t

> RAIL_ZWAVE_HomeId_t

A Z-Wave Home ID.

Note

- Home IDs in the range 0x54000000..0x55FFFFFF are illegal.

Enumerator	
RAIL_ZWAVE_HOME_ID_UNKNOWN	The unknown HomeId.
RAIL_ZWAVE_HOME_ID_DEFAULT	An impossible and unlikely HomeId.

Definition at line 217 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_HomeIdHash_t

> RAIL_ZWAVE_HomeIdHash_t

A Z-Wave Home ID hash.

Note

- Certain values (as shown) are illegal.

Enumerator	
RAIL_ZWAVE_HOME_ID_HASH_ILLEGAL_1	An illegal HomeIdHash value.
RAIL_ZWAVE_HOME_ID_HASH_ILLEGAL_2	An illegal HomeIdHash value.
RAIL_ZWAVE_HOME_ID_HASH_ILLEGAL_3	An illegal HomeIdHash value.

RAIL_ZWAVE_HOME_ID_HASH_DONT_CARE	Illegal HomeIdHash value that suppresses checking the HomeIdHash field of beam packets.
RAIL_ZWAVE_HOME_ID_HASH_DEFAULT	Default to don't care.

Definition at line `234` of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_Baud_t

> RAIL_ZWAVE_Baud_t

Z-Wave supported baudrates or PHYs.

Enumerator	
RAIL_ZWAVE_BAUD_9600	9.6kbps baudrate
RAIL_ZWAVE_BAUD_40K	40kbps baudrate
RAIL_ZWAVE_BAUD_100K	100kbps baudrate
RAIL_ZWAVE_LR	Long Range PHY.
RAIL_ZWAVE_ENERGY_DETECT	Energy detection PHY.
RAIL_ZWAVE_BAUD_INVALID	Sentinel value for invalid baud rate.

Definition at line `278` of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_RegionId_t

> RAIL_ZWAVE_RegionId_t

Z-Wave region identifications.

Enumerator	
RAIL_ZWAVE_REGIONID_UNKNOWN	Unknown/Invalid.
RAIL_ZWAVE_REGIONID_EU	European Union.
RAIL_ZWAVE_REGIONID_US	United States.
RAIL_ZWAVE_REGIONID_ANZ	Australia/New Zealand.
RAIL_ZWAVE_REGIONID_HK	Hong Kong.
RAIL_ZWAVE_REGIONID_MY	Malaysia.
RAIL_ZWAVE_REGIONID_IN	India.
RAIL_ZWAVE_REGIONID_JP	Japan.
RAIL_ZWAVE_REGIONID_RU	Russian Federation.
RAIL_ZWAVE_REGIONID_IL	Israel.
RAIL_ZWAVE_REGIONID_KR	Korea.
RAIL_ZWAVE_REGIONID_CN	China.
RAIL_ZWAVE_REGIONID_US_LR1	United States, with first long range PHY.
RAIL_ZWAVE_REGIONID_US_LR2	United States, with second long range PHY.
RAIL_ZWAVE_REGIONID_US_LR_END_DEVICE	United States long range end device PHY for both LR frequencies.
RAIL_ZWAVE_REGIONID_EU_LR1	European Union, with first long range PHY.
RAIL_ZWAVE_REGIONID_EU_LR2	European Union, with second long range PHY.

RAIL_ZWAVE_REGIONID_EU_LR_END_DEVICE	European Union long range end device PHY for both LR frequencies.
RAIL_ZWAVE_REGIONID_COUNT	Count of known regions, must be last.

Definition at line 336 of file protocol/zwave/rail_zwave.h

Typedef Documentation

RAIL_RxChannelHoppingParameters_t

```
RAIL_RxChannelHoppingParameters_t [(4U)]
```

Rx channel hopping on-channel time for all Z-Wave channels in a region.

Definition at line 464 of file protocol/zwave/rail_zwave.h

Variable Documentation

RAIL_ZWAVE_REGION_EU

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_EU
```

EU-European Union, RAIL_ZWAVE_REGION_EU.

Definition at line 884 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_REGION_US

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_US
```

US-United States, RAIL_ZWAVE_REGION_US.

Definition at line 887 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_REGION_ANZ

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_ANZ
```

ANZ-Australia/New Zealand, RAIL_ZWAVE_REGION_ANZ.

Definition at line 890 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_REGION_HK

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_HK
```

HK-Hong Kong, RAIL_ZWAVE_REGION_HK.

Definition at line 893 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_REGION_MY

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_MY
```

MY-Malaysia, RAIL_ZWAVE_REGION_MY.

Definition at line `896` of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_REGION_IN

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_IN
```

IN-India, RAIL_ZWAVE_REGION_IN.

Definition at line `899` of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_REGION_JP

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_JP
```

JP-Japan, RAIL_ZWAVE_REGION_JP.

Definition at line `902` of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_REGION_JPED

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_JPED
```

JP-Japan, RAIL_ZWAVE_REGION_JP.

Definition at line `905` of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_REGION_RU

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_RU
```

RU-Russia, RAIL_ZWAVE_REGION_RU.

Definition at line `908` of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_REGION_IL

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_IL
```

IL-Israel, RAIL_ZWAVE_REGION_IL.

Definition at line `911` of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_REGION_KR

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_KR
```

KR-Korea, RAIL_ZWAVE_REGION_KR.

Definition at line `914` of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_REGION_KRED

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_KRED
```

KR-Korea, RAIL_ZWAVE_REGION_KR.

Definition at line `917` of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_REGION_CN

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_CN
```

CN-China, RAIL_ZWAVE_REGION_CN.

Definition at line `920` of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_REGION_US_LR1

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_US_LR1
```

US-Long Range 1, RAIL_ZWAVE_REGION_US_LR1.

Definition at line `923` of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_REGION_US_LR2

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_US_LR2
```

US-Long Range 2, RAIL_ZWAVE_REGION_US_LR2.

Definition at line `926` of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_REGION_US_LR_END_DEVICE

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_US_LR_END_DEVICE
```

US-Long Range End Device, RAIL_ZWAVE_REGION_US_LR_END_DEVICE.

Definition at line `929` of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_REGION_EU_LR1

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_EU_LR1
```

EU-Long Range 1, RAIL_ZWAVE_REGION_EU_LR1.

Definition at line `932` of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_REGION_EU_LR2

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_EU_LR2
```

EU-Long Range 2, RAIL_ZWAVE_REGION_EU_LR2.

Definition at line `935` of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_REGION_EU_LR_END_DEVICE

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_EU_LR_END_DEVICE
```

EU-Long Range End Device, RAIL_ZWAVE_REGION_EU_LR_END_DEVICE.

Definition at line `938` of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_REGION_INVALID

```
const RAIL_ZWAVE_RegionConfig_t RAIL_ZWAVE_REGION_INVALID
```

Invalid Region.

Definition at line `941` of file `protocol/zwave/rail_zwave.h`

Function Documentation

RAIL_ZWAVE_ConfigRegion

```
RAIL_Status_t RAIL_ZWAVE_ConfigRegion (RAIL_Handle_t railHandle, const RAIL_ZWAVE_RegionConfig_t *regionCfg)
```

Switch the Z-Wave region.

Parameters

[in]	railHandle	A handle of RAIL instance.
[in]	regionCfg	Z-Wave channel configuration for the selected region

Returns

- Status code indicating success of the function call.

Note

- Setting a new Z-Wave Region will default any Low Power values to Normal Power values for the region. Z-Wave Region configuration must always be followed by a Low Power setup in case one desires to have the Low Power ACKing

functionality.

Definition at line 478 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_PerformIrcal

> RAIL_Status_t RAIL_ZWAVE_PerformIrcal (RAIL_Handle_t railHandle, RAIL_ZWAVE_IrcalVal_t *pIrCalVals, bool forceIrcal)

Perform image rejection calibration on all valid channels of a Z-Wave region.

Parameters

[in]	railHandle	A handle of RAIL instance.
[inout]	pIrCalVals	An application-provided pointer of type RAIL_ZWAVE_IrcalVal_t. This is populated with image rejection calibration values, if not NULL or initialized with RAIL_CAL_INVALID_VALUE or if forceIrcal is true.
[in]	forceIrcal	If true, will always perform image rejection calibration and not use previously cached values.

Returns

- Status code indicating success of the function call.

Note: This function also calibrates for beam detection and should be called before RAIL_ZWAVE_ReceiveBeam() and after the Z-Wave region has been configured via RAIL_ZWAVE_ConfigRegion(). Channel hopping must be disabled otherwise this function will return RAIL_STATUS_INVALID_CALL.

Definition at line 500 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_Init

> RAIL_Status_t RAIL_ZWAVE_Init (RAIL_Handle_t railHandle, const RAIL_ZWAVE_Config_t *config)

Initialize RAIL for Z-Wave features.

Parameters

[in]	railHandle	A handle of RAIL instance.
[in]	config	A Z-Wave configuration structure.

Returns

- A status code indicating success of the function call.

This function is the entry point for working with Z-Wave within RAIL. It sets up relevant hardware acceleration for Z-Wave-specific features, such as HomeId filtering and beam packets (as specified in the configuration) and allows users to select the relevant Z-Wave region-specific PHY via RAIL_ZWAVE_ConfigRegion.

Definition at line 516 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_Deinit

> RAIL_Status_t RAIL_ZWAVE_Deinit (RAIL_Handle_t railHandle)

De-initialize Z-Wave hardware acceleration.

Parameters

[in]	railHandle	A handle of RAIL instance.

Returns

- A status code indicating success of the function call.

Disables and resets all Z-Wave hardware acceleration features. This function should only be called when the radio is IDLE.

Definition at line 528 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_IsEnabled

> bool RAIL_ZWAVE_IsEnabled (RAIL_Handle_t railHandle)

Return whether Z-Wave hardware acceleration is currently enabled.

Parameters

[in]	railHandle	A handle of RAIL instance.

Returns

- True if Z-Wave hardware acceleration was enabled to start with and false otherwise.

Definition at line 537 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_ConfigOptions

> RAIL_Status_t RAIL_ZWAVE_ConfigOptions (RAIL_Handle_t railHandle, RAIL_ZWAVE_Options_t mask, RAIL_ZWAVE_Options_t options)

Configure Z-Wave options.

Parameters

[in]	railHandle	A handle of RAIL instance.
[in]	mask	A bitmask containing which options should be modified.
[in]	options	A bitmask containing desired configuration settings. Bit positions for each option are found in the RAIL_ZWAVE_Options_t.

Returns

- Status code indicating success of the function call.

Definition at line 548 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_SetNodeId

> RAIL_Status_t RAIL_ZWAVE_SetNodeId (RAIL_Handle_t railHandle, RAIL_ZWAVE_NodeId_t nodeId)

Inform RAIL of the Z-Wave node's NodeId for receive filtering.

Parameters

[in]	railHandle	A handle of RAIL instance.
[in]	nodeId	A Z-Wave Node ID.

Returns

Status code indicating success of the function call.

Note

- Until this API is called, RAIL will assume the NodeId is RAIL_ZWAVE_NODE_ID_DEFAULT.

Definition at line 562 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_SetHomeId

```
RAIL_Status_t RAIL_ZWAVE_SetHomeId (RAIL_Handle_t railHandle, RAIL_ZWAVE_HomeId_t homeId,
RAIL_ZWAVE_HomeIdHash_t homeIdHash)
```

Inform RAIL of the Z-Wave node's HomeId and its hash for receive filtering.

Parameters

[in]	railHandle	A handle of RAIL instance.
[in]	homeId	A Z-Wave HomeId.
[in]	homeIdHash	The hash of the HomeId expected in beam frames. If this is RAIL_ZWAVE_HOME_ID_HASH_DONT_CARE, beam frame detection will not check the HomeIdHash in a received beam frame at all, and RAIL_EVENT_ZWAVE_BEAM will trigger based solely on the NodeId in the beam frame.

Returns

- Status code indicating success of the function call.

Note

- Until this API is called, RAIL will assume the HomeId is an illegal one of RAIL_ZWAVE_HOME_ID_DEFAULT, and its hash is RAIL_ZWAVE_HOME_ID_HASH_DONT_CARE.

Definition at line 581 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_GetBeamNodeId

```
RAIL_Status_t RAIL_ZWAVE_GetBeamNodeId (RAIL_Handle_t railHandle, RAIL_ZWAVE_NodeId_t *pNodeId)
```

Get the NodeId of the most recently seen beam frame that triggered RAIL_EVENT_ZWAVE_BEAM.

Parameters

[in]	railHandle	A handle of RAIL instance.
[out]	pNodeId	A pointer to RAIL_ZWAVE_NodeId_t to populate.

Returns

- Status code indicating success of the function call.

Note

- This is best called while handling the RAIL_EVENT_ZWAVE_BEAM event; if multiple beams are received only the most recent beam's NodeId is provided.

Definition at line 597 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_GetBeamHomeIdHash

> RAIL_Status_t RAIL_ZWAVE_GetBeamHomeIdHash (RAIL_Handle_t railHandle, RAIL_ZWAVE_HomeIdHash_t *pBeamHomeIdHash)

Get the HomeIdHash of the most recently seen beam frame that triggered RAIL_EVENT_ZWAVE_BEAM.

Parameters

[in]	railHandle	A handle of RAIL instance.
[out]	pBeamHomeIdHash	A pointer to RAIL_ZWAVE_HomeIdHash_t to populate.

Returns

- Status code indicating success of the function call.

Note

- This is best called while handling the RAIL_EVENT_ZWAVE_BEAM event; if multiple beams are received only the most recent beam's HomeIdHash is provided.

Definition at line `612` of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_GetBeamChannelIndex

> RAIL_Status_t RAIL_ZWAVE_GetBeamChannelIndex (RAIL_Handle_t railHandle, uint8_t *pChannelIndex)

Get the channel hopping index of the most recently seen beam frame that triggered RAIL_EVENT_ZWAVE_BEAM.

Parameters

[in]	railHandle	A handle of RAIL instance.
[out]	pChannelIndex	A pointer to a uint8_t to populate with the channel hopping index. If channel-hopping was off at the time the beam packet was received, RAIL_CHANNEL_HOPPING_INVALID_INDEX is provided.

Returns

- Status code indicating success of the function call.

Note

- This is best called while handling the RAIL_EVENT_ZWAVE_BEAM event; if multiple beams are received only the most recent beam's channel hopping index is provided.

Definition at line `630` of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_GetLrBeamTxPower

> RAIL_Status_t RAIL_ZWAVE_GetLrBeamTxPower (RAIL_Handle_t railHandle, uint8_t *pLrBeamTxPower)

Get the TX power used by the transmitter of the most recently seen long range beam frame that triggered RAIL_EVENT_ZWAVE_BEAM.

Parameters

[in]	railHandle	A handle of RAIL instance.
[out]	pLrBeamTxPower	An application provided pointer to a uint8_t to be populated with the TX power of the latest long range beam. This will be set to RAIL_ZWAVE_LR_BEAM_TX_POWER_INVALID if this API is called after receiving a regular non-long-range beam.

Returns

- Status code indicating success of the function call. This function will return RAIL_STATUS_INVALID_STATE if called after receiving a regular (non-long-range) beam.

Note

- This is best called while handling the RAIL_EVENT_ZWAVE_BEAM event; if multiple beams are received only the most recent long range beam's TX power is provided.
- The following table shows long range beam TX power value to dBm value mapping:

Tx Power Value	Description
0	-6dBm
1	-2dBm
2	+2dBm
3	+6dBm
4	+10dBm
5	+13dBm
6	+16dBm
7	+19dBm
8	+21dBm
9	+23Bm
10	+25dBm
11	+26dBm
12	+27dBm
13	+28dBm
14	+29dBm
15	+30dBm

Definition at line `673` of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_GetBeamRssi

```
RAIL_Status_t RAIL_ZWAVE_GetBeamRssi (RAIL_Handle_t railHandle, int8_t *pBeamRssi)
```

Get the RSSI of the received beam frame.

Parameters

[in]	railHandle	A handle of RAIL instance.
[out]	pBeamRssi	An application provided pointer to a int8_t to be populated with the latest beam's RSSI, in dBm.

Returns

- Status code indicating success of the function call. This function will return RAIL_STATUS_INVALID_STATE if called without ever having received a beam.

Note

- This is best called while handling the RAIL_EVENT_ZWAVE_BEAM event; if multiple beams are received only the most recent beam's RSSI is provided.

Definition at line `690` of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_SetTxLowPower

```
RAIL_Status_t RAIL_ZWAVE_SetTxLowPower (RAIL_Handle_t railHandle, uint8_t powerLevel)
```

Set the Raw Low Power settings.

Parameters

[in]	railHandle	A handle of RAIL instance.
[in]	powerLevel	Desired low power raw level.

Returns

- Status code indicating success of the function call.

Low Power settings are required during ACK transmissions when the Low Power Bit is set. This setting is only valid for one subsequent transmission, after which all transmissions will be at the nominal power setting, until re-invoked.

Definition at line `705` of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_SetTxLowPowerDbm

```
RAIL_Status_t RAIL_ZWAVE_SetTxLowPowerDbm (RAIL_Handle_t railHandle, RAIL_TxPower_t powerLevel)
```

Set the Low Power settings in dBm.

Parameters

[in]	railHandle	A handle of RAIL instance.
[in]	powerLevel	Desired low power level dBm.

Returns

- Status code indicating success of the function call.

Low Power settings are required during ACK transmissions when the Low Power Bit is set. This setting is only valid for one subsequent transmission, after which all transmissions will be at the nominal power setting, until re-invoked.

Definition at line `720` of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_GetTxLowPower

```
RAIL_TxPowerLevel_t RAIL_ZWAVE_GetTxLowPower (RAIL_Handle_t railHandle)
```

Get the TX low power in raw units (see rail_chip_specific.h for value ranges).

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- The chip-specific RAIL_TxPowerLevel_t raw value of the low transmit power.

This API returns the low raw power value that was set by RAIL_ZWAVE_SetTxLowPower.

Calling this function before configuring the Low Power PA (i.e., before a successful call to RAIL_ZWAVE_SetTxLowPowerDbm or RAIL_ZWAVE_SetTxLowPower) will return the low power value same as the nominal

power. Also, calling this function before configuring the PA (i.e., before a successful call to RAIL_ConfigTxPower) will return an error (RAIL_TX_POWER_LEVEL_INVALID).

Definition at line 742 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_GetTxLowPowerDbm

```
RAIL_TxPower_t RAIL_ZWAVE_GetTxLowPowerDbm (RAIL_Handle_t railHandle)
```

Get the TX low power in terms of deci-dBm instead of raw power level.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- The chip-specific RAIL_TxPower_t value of the low transmit power in deci-dBm.

Definition at line 751 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_ReceiveBeam

```
RAIL_Status_t RAIL_ZWAVE_ReceiveBeam (RAIL_Handle_t railHandle, uint8_t *beamDetectIndex, const
RAIL_SchedulerInfo_t *schedulerInfo)
```

Implement beam detection and reception algorithms.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	beamDetectIndex	Indicator of whether or not a beam was detected at all, regardless of if it was received, generally for use only by instruction from Silicon Labs. Can be NULL.
[out]	schedulerInfo	While Z-Wave is currently not supported in RAIL Multiprotocol, this scheduler info is added to future proof against any future version of multiprotocol which may support it. For now, this argument can be NULL.

It will take care of all configuration and radio setup to detect and receive beams in the current Z-Wave region. If a beam is detected, RAIL will provide the usual RAIL_EVENT_ZWAVE_BEAM event during which time users can process the beam as expected. However, normal packets may also be received during this time (also triggering RAIL_EVENTS_RX_COMPLETION events), in which case, this API may need to be re-called to receive a beam. Users should also listen for RAIL_EVENT_RX_CHANNEL_HOPPING_COMPLETE, which will indicate that no beam is heard. At that point, the radio will be automatically idled. Until one of these events is received, users should not try to reconfigure radio settings or start another radio operation. If an application needs to do some other operation or configuration, it must first call RAIL_Idle and wait for the radio to idle.

Note

- : The radio must be idle before calling this function.
- : RAIL_ConfigRxChannelHopping must have been called successfully in Z-Wave before this function is called to provide a valid memory buffer for internal use (see RAIL_RxChannelHoppingConfig_t::buffer).
- : This function alters radio functionality substantially. After calling it, the user should call RAIL_ZWAVE_ConfigRegion, RAIL_ConfigRxChannelHopping, RAIL_EnableRxChannelHopping, and RAIL_SetRxTransitions to reset these parameters to whatever behaviors were desired before calling this function. Additionally, this function will idle the radio upon on exit.

Returns

- status indicating whether or not the radio was able to configure beam packet detection/reception. Reasons for failure include an un-idled radio or a non-Japan non-Korea region configured before calling this function.

Definition at line 796 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_ConfigBeamRx

> RAIL_Status_t RAIL_ZWAVE_ConfigBeamRx (RAIL_Handle_t railHandle, RAIL_ZWAVE_BeamRxConfig_t *config)

Configure the receive algorithm used in RAIL_ZWAVE_ReceiveBeam.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	config	Configuration for beam detection algorithm.

This function should not be used without direct instruction by Silicon Labs.

Returns

- Status code indicating success of the function call.

Definition at line 808 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_SetDefaultRxBeamConfig

> RAIL_Status_t RAIL_ZWAVE_SetDefaultRxBeamConfig (RAIL_Handle_t railHandle)

Set the default RX beam configuration.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Status code indicating success of the function call.

Note

- This function resets any changes made to the beam configuration via RAIL_ZWAVE_ConfigBeamRx and the default beam configuration will be in effect on subsequent call(s) to RAIL_ZWAVE_ReceiveBeam.

Definition at line 820 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_GetRxBeamConfig

> RAIL_Status_t RAIL_ZWAVE_GetRxBeamConfig (RAIL_ZWAVE_BeamRxConfig_t *pConfig)

Get the current RX beam configuration.

Parameters

[out]	pConfig	A pointer to RAIL_ZWAVE_BeamRxConfig_t to be populated with the current beam configuration.

Returns

- A status code indicating success of the function call.

Definition at line 829 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_ConfigRxChannelHopping

```
RAIL_Status_t RAIL_ZWAVE_ConfigRxChannelHopping (RAIL_Handle_t railHandle, RAIL_RxChannelHoppingConfig_t *config)
```

Configure the channel hop timings for use in Z-Wave RX channel hop configuration.

Parameters

[in]	railHandle	A RAIL instance handle.
[inout]	config	Configuration for Z-Wave RX channel hopping. This structure must be allocated in application global read-write memory. RAIL will populate fields within or referenced by this structure during its operation. Be sure to allocate RAIL_RxChannelHoppingConfigEntry_t entries[] for RAIL_NUM_ZWAVE_CHANNELS. Be sure to set RAIL_RxChannelHoppingConfig_t::numberOfChannels to the desired number of channels.

This function should not be used without direct instruction by Silicon Labs.

Returns

- Status code indicating success of the function call.

Note

- : This API must be called before RAIL_EnableRxChannelHopping(). This API must never be called while the radio is on with RX Duty Cycle or Channel Hopping enabled.

Definition at line `849` of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_GetRegion

```
RAIL_ZWAVE_RegionId_t RAIL_ZWAVE_GetRegion (RAIL_Handle_t railHandle)
```

Get the Z-Wave region.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- The RAIL_ZWAVE_RegionId_t value

Note

- : RAIL_ZWAVE_ConfigRegion must have been called successfully before this function is called. Otherwise, RAIL_ZWAVE_REGIONID_UNKNOWN is returned.

Definition at line `861` of file `protocol/zwave/rail_zwave.h`

RAIL_ZWAVE_SetLrAckData

```
RAIL_Status_t RAIL_ZWAVE_SetLrAckData (RAIL_Handle_t railHandle, const RAIL_ZWAVE_LrAckData_t *pLrAckData)
```

Write the AutoACK FIFO for the next outgoing Z-Wave Long Range ACK.

Parameters

[in]	railHandle	A handle of RAIL instance.

[in]	pLrAckData	An application provided pointer to a const RAIL_ZWAVE_LrAckData_t to populate the noise floor, TX power and receive rssi bytes of the outgoing Z-Wave Long Range ACK packet.

Returns

- A status code indicating success of the function call.

This function sets the AutoACK data to use in acknowledging the frame being received. It must only be called while processing the RAIL_EVENT_ZWAVE_LR_ACK_REQUEST_COMMAND. This will return RAIL_STATUS_INVALID_STATE if it is too late to write the outgoing ACK. When successful, the ackData will only be sent once. Subsequent packets needing an Z-Wave Long Range ACK will each need to call this function to write the ACK information.

Definition at line 880 of file protocol/zwave/rail_zwave.h

Macro Definition Documentation

RAIL_ZWAVE_OPTIONS_NONE

```
#define RAIL_ZWAVE_OPTIONS_NONE
```

Value:

```
0U
```

A value representing no options.

Definition at line 115 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_OPTIONS_DEFAULT

```
#define RAIL_ZWAVE_OPTIONS_DEFAULT
```

Value:

```
RAIL_ZWAVE_OPTIONS_NONE
```

All options are disabled by default.

Definition at line 118 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_OPTION_PROMISCUOUS_MODE

```
#define RAIL_ZWAVE_OPTION_PROMISCUOUS_MODE
```

Value:

```
(1u << RAIL_ZWAVE_OPTION_PROMISCUOUS_MODE_SHIFT)
```

An option to configure promiscuous mode, accepting non-beam packets regardless of their HomeId.

By default packets are filtered by their HomeId. When true, such filtering is disabled.

Definition at line 125 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_OPTION_NODE_ID_FILTERING

```
#define RAIL_ZWAVE_OPTION_NODE_ID_FILTERING
```

Value:

```
(1u << RAIL_ZWAVE_OPTION_NODE_ID_FILTERING_SHIFT)
```

An option to filter non-beam packets based on their NodeId when RAIL_ZWAVE_OPTION_PROMISCUOUS_MODE is disabled.

Note

- This option has no effect when RAIL_ZWAVE_OPTION_PROMISCUOUS_MODE is enabled.

Definition at line 135 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_OPTION_DETECT_BEAM_FRAMES

```
#define RAIL_ZWAVE_OPTION_DETECT_BEAM_FRAMES
```

Value:

```
(1u << RAIL_ZWAVE_OPTION_DETECT_BEAM_FRAMES_SHIFT)
```

An option to configure beam frame recognition.

By default beams are not considered special and will be received as if they were normal Z-Wave frames, assuredly triggering RAIL_EVENT_RX_FRAME_ERROR. When true, beam frames that are broadcast or match the NodeId and HomeIdHash values will trigger RAIL_EVENT_ZWAVE_BEAM event. (All beams additionally trigger RAIL_EVENT_RX_PACKET_ABORTED regardless of NodeId / HomeIdHash values.)

Note

- This option takes precedence over RAIL_ZWAVE_OPTION_PROMISCUOUS_MODE when receiving a beam frame. For promiscuous beam handling see related RAIL_ZWAVE_OPTION_PROMISCUOUS_BEAM_MODE option.

Definition at line 152 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_OPTION_PROMISCUOUS_BEAM_MODE

```
#define RAIL_ZWAVE_OPTION_PROMISCUOUS_BEAM_MODE
```

Value:

```
(1u << RAIL_ZWAVE_OPTION_PROMISCUOUS_BEAM_MODE_SHIFT)
```

An option to receive all beams promiscuously when RAIL_ZWAVE_OPTION_DETECT_BEAM_FRAMES is enabled.

When true, beam frames are received regardless of their NodeId or HomeIdHash resulting in RAIL_EVENT_ZWAVE_BEAM (and also RAIL_EVENT_RX_PACKET_ABORTED) for each beam frame.

Note

- This option has no effect when RAIL_ZWAVE_OPTION_DETECT_BEAM_FRAMES is disabled.

Definition at line 165 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_OPTIONS_ALL

```
#define RAIL_ZWAVE_OPTIONS_ALL
```

Value:

```
0xFFFFFFFFU
```

A value representing all options.

Definition at line 169 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_FREQ_INVALID

```
#define RAIL_ZWAVE_FREQ_INVALID
```

Value:

```
0xFFFFFFFFUL
```

Sentinel value to indicate that a channel (and thus its frequency) are invalid.

Definition at line 320 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_LR_BEAM_TX_POWER_INVALID

```
#define RAIL_ZWAVE_LR_BEAM_TX_POWER_INVALID
```

Value:

```
(0xFFU)
```

Invalid beam TX power value returned when RAIL_ZWAVE_GetLrBeamTxPower is called after receiving a regular non-long-range beam.

Definition at line 399 of file protocol/zwave/rail_zwave.h

RAIL_NUM_ZWAVE_CHANNELS

```
#define RAIL_NUM_ZWAVE_CHANNELS
```

Value:

```
(4U)
```

Number of channels in each of Z-Wave's region-based PHYs.

Definition at line 435 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_Config_t

A configuration structure for Z-Wave in RAIL.

Public Attributes

RAIL_ZWAVE_Options_t	options Defines Z-Wave options.
RAIL_AutoAckConfig_t	ackConfig Defines Z-Wave ACKing configuration.
RAIL_StateTiming_t	timings Defines state timings for Z-Wave.

Public Attribute Documentation

options

> RAIL_ZWAVE_Options_t RAIL_ZWAVE_Config_t::options

Defines Z-Wave options.

Definition at line 263 of file protocol/zwave/rail_zwave.h

ackConfig

> RAIL_AutoAckConfig_t RAIL_ZWAVE_Config_t::ackConfig

Defines Z-Wave ACKing configuration.

Definition at line 267 of file protocol/zwave/rail_zwave.h

timings

> RAIL_StateTiming_t RAIL_ZWAVE_Config_t::timings

Defines state timings for Z-Wave.

Definition at line 271 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_LrAckData_t

Configuration structure for Z-Wave Long Range ACK.

Public Attributes

int8_t	noiseFloorDbm Radio noise level measured on the channel the frame is transmitted on.
int8_t	txPowerDbm Transmit power used to transmit the ongoing Z-Wave Long Range ACK.
int8_t	receiveRssiDbm Signal strength measured while receiving the Z-Wave Long Range frame.

Public Attribute Documentation

noiseFloorDbm

```
int8_t RAIL_ZWAVE_LrAckData_t::noiseFloorDbm
```

Radio noise level measured on the channel the frame is transmitted on.

Definition at line 407 of file protocol/zwave/rail_zwave.h

txPowerDbm

```
int8_t RAIL_ZWAVE_LrAckData_t::txPowerDbm
```

Transmit power used to transmit the ongoing Z-Wave Long Range ACK.

Definition at line 409 of file protocol/zwave/rail_zwave.h

receiveRssiDbm

```
int8_t RAIL_ZWAVE_LrAckData_t::receiveRssiDbm
```

Signal strength measured while receiving the Z-Wave Long Range frame.

Definition at line 411 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_BeamRxConfig_t

Configuration structure for Z-Wave beam detection.

This structure should not be used without direct instruction by Silicon Labs. Appropriate defaults for this are built into the RAIL library.

Public Attributes

RAIL_RxChannelHoppingConfig_t	channelHoppingConfig Channel hopping pattern to use for beam detection.
RAIL_RxDutyCycleConfig_t	receiveConfig_100 Amount of time to spend trying to receive a beam once detected.
RAIL_RxDutyCycleConfig_t	receiveConfig_40 Amount of time to spend trying to receive a beam once detected.

Public Attribute Documentation

channelHoppingConfig

```
RAIL_RxChannelHoppingConfig_t RAIL_ZWAVE_BeamRxConfig_t::channelHoppingConfig
```

Channel hopping pattern to use for beam detection.

Definition at line 423 of file protocol/zwave/rail_zwave.h

receiveConfig_100

```
RAIL_RxDutyCycleConfig_t RAIL_ZWAVE_BeamRxConfig_t::receiveConfig_100
```

Amount of time to spend trying to receive a beam once detected.

100kbps only

Definition at line 426 of file protocol/zwave/rail_zwave.h

receiveConfig_40

```
RAIL_RxDutyCycleConfig_t RAIL_ZWAVE_BeamRxConfig_t::receiveConfig_40
```

Amount of time to spend trying to receive a beam once detected.

40kbps only

Definition at line 429 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_RegionConfig_t

Each Z-Wave region supports 3 channels.

Public Attributes

uint32_t	frequency Channel frequency in hertz.
RAIL_TxPower_t	maxPower The maximum power allowed on the channel.
RAIL_ZWAVE_Baud_t	baudRate Channel baud rate index.
RAIL_ZWAVE_RegionId_t	regionId Identification number for the region.
RAIL_ZWAVE_RegionOptions_t	regionSpecific Encapsulates region specific data.

Public Attribute Documentation

frequency

```
uint32_t RAIL_ZWAVE_RegionConfig_t::frequency[(4U)]
```

Channel frequency in hertz.

Definition at line 442 of file protocol/zwave/rail_zwave.h

maxPower

```
RAIL_TxPower_t RAIL_ZWAVE_RegionConfig_t::maxPower[(4U)]
```

The maximum power allowed on the channel.

Definition at line 443 of file protocol/zwave/rail_zwave.h

baudRate

```
RAIL_ZWAVE_Baud_t RAIL_ZWAVE_RegionConfig_t::baudRate[(4U)]
```

Channel baud rate index.

Definition at line 444 of file protocol/zwave/rail_zwave.h

regionId

> RAIL_ZWAVE_RegionId_t RAIL_ZWAVE_RegionConfig_t::regionId

Identification number for the region.

Definition at line 445 of file protocol/zwave/rail_zwave.h

regionSpecific

> RAIL_ZWAVE_RegionOptions_t RAIL_ZWAVE_RegionConfig_t::regionSpecific

Encapsulates region specific data.

Definition at line 446 of file protocol/zwave/rail_zwave.h

RAIL_ZWAVE_IrcalVal_t

Structure for Z-Wave Image Rejection Calibration.

Note

- Index 0 will hold the low side image rejection calibration value (channel 0), while index 1 will hold the high side image rejection value (channel 1).

Public Attributes

RAIL_IrCalValues_t	imageRejection Low side and high side image rejection values.

Public Attribute Documentation

imageRejection

```
RAIL_IrCalValues_t RAIL_ZWAVE_IrcalVal_t::imageRejection[2]
```

Low side and high side image rejection values.

Definition at line 457 of file protocol/zwave/rail_zwave.h

RF Sense

RF Sense

Modules

RAIL_RfSenseSelectiveOokConfig_t

Enumerations

enum RAIL_RfSenseBand_t {

RAIL_RFSENSE_OFF
RAIL_RFSENSE_2_4GHZ
RAIL_RFSENSE_SUBGHZ
RAIL_RFSENSE_ANY
RAIL_RFSENSE_MAX
RAIL_RFSENSE_2_4GHZ_LOW_SENSITIVITY = (0×20U) + RAIL_RFSENSE_2_4GHZ
RAIL_RFSENSE_SUBGHZ_LOW_SENSITIVITY = (0×20U) + RAIL_RFSENSE_SUBGHZ
RAIL_RFENSE_ANY_LOW_SENSITIVITY = (0×20U) + RAIL_RFSENSE_ANY

}

An enumeration for specifying the RF Sense frequency band.

Typedefs

typedef void(* RAIL_RfSense_CallbackPtr_t)(void)

A pointer to an RF Sense callback function.

Functions

RAIL_Time_t RAIL_StartRfSense(RAIL_Handle_t railHandle, RAIL_RfSenseBand_t band, RAIL_Time_t senseTime, RAIL_RfSense_CallbackPtr_t cb)

Start/stop the RF Sense functionality in Energy Detection Mode for use during low-energy sleep modes.

RAIL_Status_t RAIL_StartSelectiveOokRfSense(RAIL_Handle_t railHandle, RAIL_RfSenseSelectiveOokConfig_t *config)

Start/stop the RF Sense functionality in Selective(OOK Based) Mode for use during low-energy sleep modes.

RAIL_Status_t RAIL_ConfigRfSenseSelectiveOokWakeupPhy(RAIL_Handle_t railHandle)

Switch to RF Sense Selective(OOK) PHY.

RAIL_Status_t RAIL_SetRfSenseSelectiveOokWakeupPayload(RAIL_Handle_t railHandle, uint8_t numSyncwordBytes, uint32_t syncword)

Set the transmit payload for waking up a node configured for RF Sense Selective(OOK).

bool RAIL_IsRfSensed(RAIL_Handle_t railHandle)

Check whether the RF was sensed.

Macros

#define RAIL_RFSENSE_LOW_SENSITIVITY_OFFSET (0×20U)

RF Sense low sensitivity offset.

#define RAIL_RFSENSE_USE_HW_SYNCWORD (0U)
Use the MODEM default sync word.

Enumeration Documentation

RAIL_RfSenseBand_t

```
RAIL_RfSenseBand_t
```

An enumeration for specifying the RF Sense frequency band.

Enumerator	
RAIL_RFSENSE_OFF	RF Sense is disabled.
RAIL_RFSENSE_2_4GHZ	RF Sense is in 2.4 G band.
RAIL_RFSENSE_SUBGHZ	RF Sense is in subgig band.
RAIL_RFSENSE_ANY	RF Sense is in both bands.
RAIL_RFSENSE_MAX	
RAIL_RFSENSE_2_4GHZ_LOW_SENSITIVITY	RF Sense is in low sensitivity 2.4 G band.
RAIL_RFSENSE_SUBGHZ_LOW_SENSITIVITY	RF Sense is in low sensitivity subgig band.
RAIL_RFENSE_ANY_LOW_SENSITIVITY	RF Sense is in low sensitivity for both bands.

Definition at line 4675 of file common/rail_types.h

Typedef Documentation

RAIL_RfSense_CallbackPtr_t

```
typedef void(* RAIL_RfSense_CallbackPtr_t) (void) )(void)
```

A pointer to an RF Sense callback function.

Consider using the event RAIL_EVENT_RF_SENSED as an alternative.

Definition at line 4664 of file common/rail_types.h

Function Documentation

RAIL_StartRfSense

```
RAIL_Time_t RAIL_StartRfSense (RAIL_Handle_t railHandle, RAIL_RfSenseBand_t band, RAIL_Time_t senseTime,
RAIL_RfSense_CallbackPtr_t cb)
```

Start/stop the RF Sense functionality in Energy Detection Mode for use during low-energy sleep modes.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	band	The frequency band(s) on which to sense the RF energy. To stop RF Sense, specify RAIL_RFSENSE_OFF.
[in]	senseTime	The time (in microseconds) the RF energy must be continually detected to be considered "sensed".
[in]	cb	RAIL_RfSense_CallbackPtr_t is called when the RF is sensed. Set null if polling via RAIL_IsRfSensed().

Returns

- The actual senseTime used, which may be different than requested due to limitations of the hardware. If 0, RF sense was disabled or could not be enabled (no callback will be issued).

The EFR32 has the ability to sense the presence of RF Energy above -20 dBm within either or both the 2.4 GHz and Sub-GHz bands and trigger an event if that energy is continuously present for certain durations of time.

Note

- After RF energy has been sensed, the RF Sense is automatically disabled and RAIL_StartRfSense() must be called again to reactivate it. If RF energy has not been sensed and to manually disable RF Sense, RAIL_StartRfSense() must be called with band specified as RAIL_RFSENSE_OFF or with senseTime set to 0 microseconds.
- Packet reception is not guaranteed to work correctly once RF Sense is enabled, both in single protocol and multiprotocol RAIL. To be safe, an application should turn this on only after idling the radio to stop receive and turn it off before attempting to restart receive. Since EM4 sleep causes the chip to come up through the reset vector any wake from EM4 must also shut off RF Sense to ensure proper receive functionality.

Warnings

- For some chips, RF Sense functionality is only guaranteed within a specified temperature range. See chip-specific documentation for more details.

Definition at line 5178 of file common/rail.h

RAIL_StartSelectiveOokRfSense

> RAIL_Status_t RAIL_StartSelectiveOokRfSense (RAIL_Handle_t railHandle, RAIL_RfSenseSelectiveOokConfig_t *config)

Start/stop the RF Sense functionality in Selective(OOK Based) Mode for use during low-energy sleep modes.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	config	RAIL_RfSenseSelectiveOokConfig_t holds the RFSENSE configuration for Selective(OOK) mode.

Returns

- Status code indicating success of the function call.

Some chips support Selective RF energy detection (OOK mode) where the user can program the chip to look for a particular sync word pattern (1byte - 4bytes) sent using OOK and wake only when that is detected. See chip-specific documentation for more details.

The following code gives an example of how to use RF Sense functionality in Selective(OOK Based) Mode.

```
// Syncword Length in bytes, 1-4 bytes.
#define NUMSYNCWORDBYTES (2U)
// Syncword Value.
#define SYNCWORD       (0xB16FU)

// Configure the transmitting node for sending the wakeup packet.
RAIL_Idle(railHandle, RAIL_IDLE_ABORT, true);
RAIL_ConfigRfSenseSelectiveOokWakeupPhy(railHandle);
RAIL_SetRfSenseSelectiveOokWakeupPayload(railHandle, NUMSYNCWORDBYTES, SYNCWORD);
RAIL_StartTx(railHandle, channel, RAIL_TX_OPTIONS_DEFAULT, NULL);

// Configure the receiving node (EFR32XG22) for RF Sense.
RAIL_RfSenseSelectiveOokConfig_t config = {
 .band = rfBand,
 .syncWordNumBytes = NUMSYNCWORDBYTES,
```

```
.cb =&RAILCb_SensedRf
};RAIL_StartSelectiveOokRfSense(railHandle,&config);
```

Note

- After RF energy has been sensed, the RF Sense is automatically disabled and RAIL_StartSelectiveOokRfSense() must be called again to reactivate. If RF energy has not been sensed and to manually disable RF Sense, RAIL_StartSelectiveOokRfSense() must be called with band specified as RAIL_RFSENSE_OFF or with RAIL_RfSenseSelectiveOokConfig_t as NULL.
- Packet reception is not guaranteed to work correctly once RF Sense is enabled, both in single protocol and multiprotocol RAIL. To be safe, an application should turn this on only after idling the radio to stop receive and turn it off before attempting to restart receive. Since EM4 sleep causes the chip to come up through the reset vector any wake from EM4 must also shut off RF Sense to ensure proper receive functionality.

Definition at line 5237 of file common/rail.h

RAIL_ConfigRfSenseSelectiveOokWakeupPhy

> RAIL_Status_t RAIL_ConfigRfSenseSelectiveOokWakeupPhy (RAIL_Handle_t railHandle)

Switch to RF Sense Selective(OOK) PHY.

Parameters

[in]	railHandle	A handle for RAIL instance.

Returns

- A status code indicating success of the function call.

This function switches to the RFSENSE Selective(OOK) PHY for transmitting a packet to wake up a chip that supports Selective RF energy detection (OOK mode). You may only call this function while the radio is idle. While the radio is configured for this PHY, receive functionality should not be used.

Note

- The user must also set up the transmit FIFO, via RAIL_SetRfSenseSelectiveOokWakeupPayload, post this function call to include the first byte as the Preamble Byte, followed by the Syncword (1byte - 4bytes). See chip-specific documentation for more details.

Definition at line 5257 of file common/rail.h

RAIL_SetRfSenseSelectiveOokWakeupPayload

> RAIL_Status_t RAIL_SetRfSenseSelectiveOokWakeupPayload (RAIL_Handle_t railHandle, uint8_t numSyncwordBytes, uint32_t syncword)

Set the transmit payload for waking up a node configured for RF Sense Selective(OOK).

Parameters

[in]	railHandle	A handle for RAIL instance.
[in]	numSyncwordBytes	Syncword Length in bytes, 1-4 bytes.
[in]	syncword	Syncword Value.

Returns

- A status code indicating success of the function call.

Note

- You must call this function after the chip has been set up with the RF Sense Selective(OOK) PHY, using RAIL_ConfigRfSenseSelectiveOokWakeupPhy.

Definition at line 5272 of file common/rail.h

RAIL_IsRfSensed

```
bool RAIL_IsRfSensed (RAIL_Handle_t railHandle)
```

Check whether the RF was sensed.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- true if RF was sensed since the last call to RAIL_StartRfSense. False otherwise.

This function is useful if RAIL_StartRfSense is called with a null callback. It is generally used after EM4 reboot but can be used any time.

Definition at line 5286 of file common/rail.h

Macro Definition Documentation

RAIL_RFSENSE_LOW_SENSITIVITY_OFFSET

```
#define RAIL_RFSENSE_LOW_SENSITIVITY_OFFSET
```

Value:

```
(0x20U)
```

RF Sense low sensitivity offset.

Definition at line 4669 of file common/rail_types.h

RAIL_RFSENSE_USE_HW_SYNCWORD

```
#define RAIL_RFSENSE_USE_HW_SYNCWORD
```

Value:

```
(0U)
```

Use the MODEM default sync word.

Definition at line 4701 of file common/rail_types.h

RAIL_RfSenseSelectiveOokConfig_t

Structure to configure RFSENSE Selective(OOK) mode.

Public Attributes

RAIL_RfSenseBand_t	band The frequency band(s) on which to sense the RF energy.
uint8_t	syncWordNumBytes Syncword Length in bytes, 1-4 bytes.
uint32_t	syncWord Sync Word Value.
RAIL_RfSense_CallbackPtr_t	cb The callback function, called when RF is sensed.

Public Attribute Documentation

band

```
RAIL_RfSenseBand_t RAIL_RfSenseSelectiveOokConfig_t::band
```

The frequency band(s) on which to sense the RF energy.

To stop RF Sense, specify RAIL_RFSENSE_OFF.

Definition at line 4712 of file common/rail_types.h

syncWordNumBytes

```
uint8_t RAIL_RfSenseSelectiveOokConfig_t::syncWordNumBytes
```

Syncword Length in bytes, 1-4 bytes.

Note

- When syncWord is set to use RAIL_RFSENSE_USE_HW_SYNCWORD, the syncWordNumBytes value will be ignored since we rely on the HW default settings for sync word.

Definition at line 4719 of file common/rail_types.h

syncWord

```
uint32_t RAIL_RfSenseSelectiveOokConfig_t::syncWord
```

Sync Word Value.

To use HW default sync word, set to RAIL_RFSENSE_USE_HW_SYNCWORD.

Definition at line 4724 of file common/rail_types.h

cb

```
RAIL_RfSense_CallbackPtr_t RAIL_RfSenseSelectiveOokConfig_t::cb
```

The callback function, called when RF is sensed.

Definition at line 4728 of file common/rail_types.h

RX Channel Hopping

RX Channel Hopping

Hardware accelerated hopping between channels while waiting for a packet in receive.

Channel hopping provides a hardware accelerated method for scanning across multiple channels quickly, as part of a receive protocol. While it is possible to call RAIL_StartRx on different channels, back to back, and listen on many channels sequentially in that way, the time it takes to switch channels with that method may be too long for some protocols. This API pre-computes necessary channel change operations for a given list of channels, so that the radio can move from channel to channel much faster. Additionally, it leads to more succinct code as channel changes will be done implicitly, without requiring numerous calls to RAIL_StartRx. Currently, while this feature is enabled, the radio will hop channels in the given sequence each time it enters RX. Note that RX Channel hopping and EFR32xG25's concurrent mode / collision detection are mutually exclusive.

The channel hopping buffer requires RAIL_CHANNEL_HOPPING_BUFFER_SIZE_PER_CHANNEL number of 32-bit words of overhead per channel, plus 3 words overall plus the twice the size of the radioConfigDeltaSubtract of the whole radio configuration, plus the twice the sum of the sizes of all the radioConfigDeltaAdds of all the channel hopping channels.

The following code gives an example of how to use the RX Channel Hopping API.

```
#define CHANNEL_HOPPING_NUMBER_OF_CHANNELS 4
#define CHANNEL_HOPPING_BUFFER_SIZE do {      \
  3 +                                 \
  (RAIL_CHANNEL_HOPPING_BUFFER_SIZE_PER_CHANNEL \
   * CHANNEL_HOPPING_NUMBER_OF_CHANNELS) +    \
  2 * (SIZEOF_UINT32_DELTA_SUBTRACT +        \
  SIZEOF_UINT32_DELTA_ADD_0 +                \
  SIZEOF_UINT32_DELTA_ADD_1 +                \
  SIZEOF_UINT32_DELTA_ADD_2 +                \
  SIZEOF_UINT32_DELTA_ADD_3)                 \
} while (0)

RAIL_RxChannelHoppingConfigEntry_t channelHoppingEntries[CHANNEL_HOPPING_NUMBER_OF_CHANNELS];
uint32_t channelHoppingBuffer[CHANNEL_HOPPING_BUFFER_SIZE];

RAIL_RxChannelHoppingConfig_t channelHoppingConfig = {
  .buffer = channelHoppingBuffer,
  .bufferLength = CHANNEL_HOPPING_BUFFER_SIZE,
  .numberOfChannels = CHANNEL_HOPPING_NUMBER_OF_CHANNELS,
  .entries = channelHoppingEntries
};

channelHoppingEntries[0].channel = 1;
channelHoppingEntries[1].channel = 2;
channelHoppingEntries[2].channel = 3;

RAIL_ConfigRxChannelHopping(railHandle, &channelHoppingConfig);
RAIL_EnableRxChannelHopping(railHandle, true, true)
```

Modules

RAIL_RxChannelHoppingConfigMultiMode_t

RAIL_RxChannelHoppingConfigEntry_t

RAIL_RxChannelHoppingConfig_t

RAIL_RxDutyCycleConfig_t

Enumerations

enum RAIL_RxChannelHoppingMode_t {

RAIL_RX_CHANNEL_HOPPING_MODE_MANUAL
RAIL_RX_CHANNEL_HOPPING_MODE_TIMEOUT
RAIL_RX_CHANNEL_HOPPING_MODE_TIMING_SENSE
RAIL_RX_CHANNEL_HOPPING_MODE_PREAMBLE_SENSE
RAIL_RX_CHANNEL_HOPPING_MODE_RESERVED1
RAIL_RX_CHANNEL_HOPPING_MODE_MULTI_SENSE
RAIL_RX_CHANNEL_HOPPING_MODE_SQ
RAIL_RX_CHANNEL_HOPPING_MODE_CONC
RAIL_RX_CHANNEL_HOPPING_MODE_VT
RAIL_RX_CHANNEL_HOPPING_MODE_TX
RAIL_RX_CHANNEL_HOPPING_MODES_COUNT
RAIL_RX_CHANNEL_HOPPING_MODES_WITH_OPTIONS_BASE = 0x80
RAIL_RX_CHANNEL_HOPPING_MODE_MANUAL_WITH_OPTIONS =
RAIL_RX_CHANNEL_HOPPING_MODES_WITH_OPTIONS_BASE
RAIL_RX_CHANNEL_HOPPING_MODE_TIMEOUT_WITH_OPTIONS
RAIL_RX_CHANNEL_HOPPING_MODE_TIMING_SENSE_WITH_OPTIONS
RAIL_RX_CHANNEL_HOPPING_MODE_PREAMBLE_SENSE_WITH_OPTIONS
RAIL_RX_CHANNEL_HOPPING_MODE_RESERVED1_WITH_OPTIONS
RAIL_RX_CHANNEL_HOPPING_MODE_MULTI_SENSE_WITH_OPTIONS
RAIL_RX_CHANNEL_HOPPING_MODE_SQ_WITH_OPTIONS
RAIL_RX_CHANNEL_HOPPING_MODE_CONC_WITH_OPTIONS
RAIL_RX_CHANNEL_HOPPING_MODE_VT_WITH_OPTIONS
RAIL_RX_CHANNEL_HOPPING_MODE_TX_WITH_OPTIONS

}

Modes by which RAIL can determine when to proceed to the next channel during channel hopping.

enum RAIL_RxChannelHoppingDelayMode_t {

RAIL_RX_CHANNEL_HOPPING_DELAY_MODE_STATIC

}

enum RAIL_RxChannelHoppingOptions_t {

RAIL_RX_CHANNEL_HOPPING_OPTION_SKIP_SYNTH_CAL_SHIFT
RAIL_RX_CHANNEL_HOPPING_OPTION_SKIP_DC_CAL_SHIFT
RAIL_RX_CHANNEL_HOPPING_OPTION_RSSI_THRESHOLD_SHIFT
RAIL_RX_CHANNEL_HOPPING_OPTION_STOP_SHIFT
RAIL_RX_CHANNEL_HOPPING_OPTIONS_COUNT

}

Options that can customize channel hopping behavior on a per-hop basis.

Typedefs

typedef uint32_t RAIL_RxChannelHoppingParameter_t

Rx channel hopping on-channel time.

Functions

RAIL_Status_t RAIL_ConfigRxChannelHopping(RAIL_Handle_t railHandle, RAIL_RxChannelHoppingConfig_t *config)

Configure RX channel hopping.

RAIL_Status_t RAIL_EnableRxChannelHopping(RAIL_Handle_t railHandle, bool enable, bool reset)

Enable RX channel hopping.

int16_t RAIL_GetChannelHoppingRssi(RAIL_Handle_t railHandle, uint8_t channelIndex)
Get RSSI of one channel in the channel hopping sequence, during channel hopping.

RAIL_Status_t RAIL_ConfigRxDutyCycle(RAIL_Handle_t railHandle, const RAIL_RxDutyCycleConfig_t *config)
Configure RX duty cycle mode.

RAIL_Status_t RAIL_EnableRxDutyCycle(RAIL_Handle_t railHandle, bool enable)
Enable RX duty cycle mode.

RAIL_Status_t RAIL_GetDefaultRxDutyCycleConfig(RAIL_Handle_t railHandle, RAIL_RxDutyCycleConfig_t *config)
Get the default RX duty cycle configuration.

Macros

#define RAIL_RX_CHANNEL_HOPPING_MAX_SENSE_TIME_US 0x08000000UL
The maximum sense time supported for those RAIL_RxChannelHoppingMode_t modes whose parameter(s) specify a sensing time.

#define RAIL_RX_CHANNEL_HOPPING_OPTIONS_NONE 0U
A value representing no options enabled.

#define RAIL_RX_CHANNEL_HOPPING_OPTIONS_DEFAULT RAIL_RX_CHANNEL_HOPPING_OPTIONS_NONE
All options disabled by default.

#define RAIL_RX_CHANNEL_HOPPING_OPTION_DEFAULT RAIL_RX_CHANNEL_HOPPING_OPTIONS_DEFAULT

#define RAIL_RX_CHANNEL_HOPPING_OPTION_SKIP_SYNTH_CAL (1U << RAIL_RX_CHANNEL_HOPPING_OPTION_SKIP_SYNTH_CAL_SHIFT)
An option to skip synth calibration while **hopping into** the channel specified in the current entry.

#define RAIL_RX_CHANNEL_HOPPING_OPTION_SKIP_DC_CAL (1U << RAIL_RX_CHANNEL_HOPPING_OPTION_SKIP_DC_CAL_SHIFT)
An option to skip DC calibration while **hopping into** the channel specified in the current entry.

#define RAIL_RX_CHANNEL_HOPPING_OPTION_RSSI_THRESHOLD (1U << RAIL_RX_CHANNEL_HOPPING_OPTION_RSSI_THRESHOLD_SHIFT)
An option to check RSSI after **hopping into** the channel specified in the current entry and hop if that RSSI is below the threshold specified in RAIL_RxChannelHoppingConfigEntry_t::rssiThresholdDbm.

#define RAIL_RX_CHANNEL_HOPPING_OPTION_STOP (1U << RAIL_RX_CHANNEL_HOPPING_OPTION_STOP_SHIFT)
An option to stop the hopping sequence at this entry in the hop table.

#define RAIL_CHANNEL_HOPPING_INVALID_INDEX (0xFEU)
A sentinel value to flag an invalid channel hopping index.

#define RAIL_CHANNEL_HOPPING_BUFFER_SIZE_PER_CHANNEL (55U)
The static amount of memory needed per channel for channel hopping, measured in 32 bit words, regardless of the size of radio configuration structures.

Enumeration Documentation

RAIL_RxChannelHoppingMode_t

> RAIL_RxChannelHoppingMode_t

Modes by which RAIL can determine when to proceed to the next channel during channel hopping.

Enumerator	
RAIL_RX_CHANNEL_HOPPING_MODE_MANUAL	Switch to the next channel each time the radio re-enters RX after packet reception or a transmit based on the corresponding State Transitions.

RAIL_RX_CHANNEL_HOPPING_MODE_TIMEOUT	Switch to the next channel after a certain amount of time passes.
RAIL_RX_CHANNEL_HOPPING_MODE_TIMING_SENSE	Listen in receive RX for at least a specified timeout.
RAIL_RX_CHANNEL_HOPPING_MODE_PREAMBLE_SENSE	Listen in receive RX for at least a specified timeout.
RAIL_RX_CHANNEL_HOPPING_MODE_RESERVED1	Placeholder for a reserved hopping mode that is not supported.
RAIL_RX_CHANNEL_HOPPING_MODE_MULTI_SENSE	A mode that combines modes TIMING_SENSE, PREAMBLE_SENSE, and TIMEOUT (sync detect) all running in parallel.
RAIL_RX_CHANNEL_HOPPING_MODE_SQ	Switch to the next channel based on the demodulation settings in the PHY config.
RAIL_RX_CHANNEL_HOPPING_MODE_CONC	Marks that the channel is concurrent with another channel, and otherwise behaves identically to RAIL_RX_CHANNEL_HOPPING_MODE_SQ.
RAIL_RX_CHANNEL_HOPPING_MODE_VT	Indicates that this channel is a virtual channel that is concurrently detected with the channel indicated by the RAIL_RxChannelHoppingConfigEntry_t::parameter.
RAIL_RX_CHANNEL_HOPPING_MODE_TX	This is the transmit channel used for auto-ACK if the regular channel, specified in RAIL_RxChannelHoppingConfigEntry::parameter, is optimized for RX which may degrade some TX performance.
RAIL_RX_CHANNEL_HOPPING_MODES_COUNT	A count of the basic choices in this enumeration.
RAIL_RX_CHANNEL_HOPPING_MODES_WITH_OPTIONS_BASE	The start of equivalent modes requiring non-default RAIL_RxDutyCycleConfig_t::options, needed for backwards-compatibility with earlier RAIL_RxDutyCycleConfig_t format.
RAIL_RX_CHANNEL_HOPPING_MODE_MANUAL_WITH_OPTIONS	Variant of RAIL_RX_CHANNEL_HOPPING_MODE_MANUAL with options.
RAIL_RX_CHANNEL_HOPPING_MODE_TIMEOUT_WITH_OPTIONS	Variant of RAIL_RX_CHANNEL_HOPPING_MODE_TIMEOUT with options.
RAIL_RX_CHANNEL_HOPPING_MODE_TIMING_SENSE_WITH_OPTIONS	Variant of RAIL_RX_CHANNEL_HOPPING_MODE_TIMING_SENSE with options.
RAIL_RX_CHANNEL_HOPPING_MODE_PREAMBLE_SENSE_WITH_OPTIONS	Variant of RAIL_RX_CHANNEL_HOPPING_MODE_PREAMBLE_SENSE with options.
RAIL_RX_CHANNEL_HOPPING_MODE_RESERVED1_WITH_OPTIONS	Variant of RAIL_RX_CHANNEL_HOPPING_MODE_RESERVED1 with options.
RAIL_RX_CHANNEL_HOPPING_MODE_MULTI_SENSE_WITH_OPTIONS	Variant of RAIL_RX_CHANNEL_HOPPING_MODE_MULTI_SENSE with options.
RAIL_RX_CHANNEL_HOPPING_MODE_SQ_WITH_OPTIONS	Variant of RAIL_RX_CHANNEL_HOPPING_MODE_SQ with options.
RAIL_RX_CHANNEL_HOPPING_MODE_CONC_WITH_OPTIONS	Variant of RAIL_RX_CHANNEL_HOPPING_MODE_CONC with options.
RAIL_RX_CHANNEL_HOPPING_MODE_VT_WITH_OPTIONS	Variant of RAIL_RX_CHANNEL_HOPPING_MODE_VT with options.

RAIL_RX_CHANNEL_HOPPING_MODE_TX_WITH_OPTIONS	Variant of RAIL_RX_CHANNEL_HOPPING_MODE_TX with options.

Definition at line 4746 of file common/rail_types.h

RAIL_RxChannelHoppingDelayMode_t

> RAIL_RxChannelHoppingDelayMode_t

DeprecatedSet only to RAIL_RX_CHANNEL_DELAY_MODE_STATIC

Enumerator	
RAIL_RX_CHANNEL_HOPPING_DELAY_MODE_STATIC	Always delay for exactly the amount of time specified in the delay parameter, regardless of how other channel hopping channels were extended via preamble sense or other means.

Definition at line 4909 of file common/rail_types.h

RAIL_RxChannelHoppingOptions_t

> RAIL_RxChannelHoppingOptions_t

Options that can customize channel hopping behavior on a per-hop basis.

Enumerator	
RAIL_RX_CHANNEL_HOPPING_OPTION_SKIP_SYNTH_CAL_SHIFT	Shift position of RAIL_RX_CHANNEL_HOPPING_OPTION_SKIP_SYNTH_CAL bit.
RAIL_RX_CHANNEL_HOPPING_OPTION_SKIP_DC_CAL_SHIFT	Shift position of RAIL_RX_CHANNEL_HOPPING_OPTION_SKIP_DC_CAL bit.
RAIL_RX_CHANNEL_HOPPING_OPTION_RSSI_THRESHOLD_SHIFT	Shift position of RAIL_RX_CHANNEL_HOPPING_OPTION_RSSI_THRESHOLD bit.
RAIL_RX_CHANNEL_HOPPING_OPTION_STOP_SHIFT	Stop hopping on this hop.
RAIL_RX_CHANNEL_HOPPING_OPTIONS_COUNT	A count of the choices in this enumeration.

Definition at line 4934 of file common/rail_types.h

Typedef Documentation

RAIL_RxChannelHoppingParameter_t

> RAIL_RxChannelHoppingParameter_t

Rx channel hopping on-channel time.

Definition at line 4922 of file common/rail_types.h

Function Documentation

RAIL_ConfigRxChannelHopping

> RAIL_Status_t RAIL_ConfigRxChannelHopping (RAIL_Handle_t railHandle, RAIL_RxChannelHoppingConfig_t *config)

Configure RX channel hopping.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	config	Configuration parameters for RX Channel Hopping.

Returns

- Status code indicating success of the function call.

Configure channel hopping channels, conditions, and parameters. This API must be called before RAIL_EnableRxChannelHopping(). This API must never be called while the radio is on with RX Duty Cycle or Channel Hopping enabled.

Note

- This feature/API is not supported on the EFR32XG1 family of chips. Use the compile time symbol RAIL_SUPPORTS_CHANNEL_HOPPING or the runtime call RAIL_SupportsChannelHopping() to check whether the platform supports this feature.
- Calling this function will overwrite any settings configured with RAIL_ConfigRxDutyCycle.

Definition at line `5375` of file `common/rail.h`

RAIL_EnableRxChannelHopping

> RAIL_Status_t RAIL_EnableRxChannelHopping (RAIL_Handle_t railHandle, bool enable, bool reset)

Enable RX channel hopping.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	enable	Enable (true) or disable (false) RX Channel Hopping.
[in]	reset	Start from the first channel of the channel hopping sequence (true) or from wherever hopping left off last time the code left RX.

Returns

- Status code indicating success of the function call.

Enable or disable Channel Hopping. Additionally, specify whether hopping should be reset to start from the channel at index zero, or continue from the channel last hopped to. The radio should not be on when this API is called. RAIL_ConfigRxChannelHopping must be called successfully before this API is called.

Note

- This feature/API is not supported on the EFR32XG1 family of chips. Use the compile time symbol RAIL_SUPPORTS_CHANNEL_HOPPING or the runtime call RAIL_SupportsChannelHopping() to check whether the platform supports this feature.
- RX Channel Hopping may not be enabled while auto-ACKing is enabled.
- Calling this function will overwrite any settings configured with RAIL_EnableRxDutyCycle.

Definition at line `5404` of file `common/rail.h`

RAIL_GetChannelHoppingRssi

int16_t RAIL_GetChannelHoppingRssi (RAIL_Handle_t railHandle, uint8_t channelIndex)

Get RSSI of one channel in the channel hopping sequence, during channel hopping.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	channelIndex	Index in the channel hopping sequence of the channel of interest

Returns

- Latest RSSI for the channel at the specified index.

Note

- This feature/API is not supported on the EFR32XG1 family of chips. Use the compile time symbol RAIL_SUPPORTS_CHANNEL_HOPPING or the runtime call RAIL_SupportsChannelHopping() to check whether the platform supports this feature.
- In multiprotocol, this function returns RAIL_RSSI_INVALID immediately if railHandle is not the current active RAIL_Handle_t.
- RAIL_ConfigRxChannelHopping must be called successfully before this API is called.
- When the Z-Wave protocol is active, it is expected that after RAIL_ConfigRxChannelHopping is called successfully on EFR32XG1 family of chips, running RAIL_GetChannelHoppingRssi() on a 40kbps PHY will not work well. Plan to use the 9.6kbps PHY for estimating channel noise instead. On EFR32XG2 family of chips, running RAIL_GetChannelHoppingRssi() on the 9.6kbps PHY returns the RSSI measurement of the 40kpbs PHY. This is because the 9.6kbps PHY has trouble with RSSI measurements on EFR32XG2 family of chips.

Definition at line `5436` of file `common/rail.h`

RAIL_ConfigRxDutyCycle

RAIL_Status_t RAIL_ConfigRxDutyCycle (RAIL_Handle_t railHandle, const RAIL_RxDutyCycleConfig_t *config)

Configure RX duty cycle mode.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	config	Configuration structure to specify duty cycle parameters.

Returns

- Status code indicating success of the function call.

Configure RX duty cycle mode. With this mode enabled, every time the radio enters RX, it will duty cycle on and off to save power. The duty cycle ratio can be altered dynamically and intelligently by the hardware by staying on longer if a preamble or other packet segments are detected in the air. This API must never be called while the radio is on with RX Duty Cycle or Channel Hopping enabled. For short delays (in the order of microseconds), RAIL_RxDutyCycleConfig_t::delay, this can be used to save receive current while having little impact on the radio performance, for protocols with long preambles. For long delays (in the order of milliseconds or higher) the chip can be put into EM2 energy mode before re-entering RX, to save extra power, with some application hooks as shown below.

```
#include <rail.h>
#include <rail_types.h>

extern RAIL_Handle_t railHandle;
RAIL_Time_t periodicWakeupUs;

void RAILCb_Event(RAIL_Handle_t railHandle, RAIL_Events_t events) {
```

```
if(events & RAIL_EVENT_RX_DUTY_CYCLE_RX_END){// Schedule the next receive.
  RAIL_ScheduleRxConfig_t rxCfg ={.start = periodicWakeupUs,.startMode = RAIL_TIME_DELAY, .end =0U,.endMode =
RAIL_TIME_DISABLED,.rxTransitionEndSchedule =0U,.hardWindowEnd =0U
  };RAIL_Idle(railHandle, RAIL_IDLE_ABORT,true);RAIL_ScheduleRx(railHandle, channel,&rxCfg, NULL);}}

void main(void){
 RAIL_Status_t status;
 bool shouldSleep =false;// This function depends on your board/chip but it must enable the LFCLK// you intend to use for RTCC sync before we
configure sleep as that// function will attempt to auto detect the clock.BoardSetupLFCLK();// Initialize Power Manager
modulesl_power_manager_init();// Initialize RAIL Power ManagerRAIL_InitPowerManager();// Configure sleep for timer synchronization
 status =RAIL_ConfigSleep(railHandle, RAIL_SLEEP_CONFIG_TIMERSYNC_ENABLED);assert(status == RAIL_STATUS_NO_ERROR);// Application main
loopwhile(1){// ... do normal app stuff and set shouldSleep when we want to sleepif(shouldSleep){// Let the CPU go to sleep if the system allows
it.sl_power_manager_sleep();}}}
```

Note

- This feature/API is not supported on the EFR32XG1 family of chips. Use the compile time symbol RAIL_SUPPORTS_CHANNEL_HOPPING or the runtime call RAIL_SupportsChannelHopping() to check whether the platform supports this feature.
- Calling this function will overwrite any settings configured with RAIL_ConfigRxChannelHopping.

Definition at line 5518 of file common/rail.h

RAIL_EnableRxDutyCycle

> RAIL_Status_t RAIL_EnableRxDutyCycle (RAIL_Handle_t railHandle, bool enable)

Enable RX duty cycle mode.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	enable	Enable (true) or disable (false) RX Duty Cycling.

Returns

- Status code indicating success of the function call.

Enable or disable RX duty cycle mode. After this is called, the radio will begin duty cycling each time it enters RX, based on the configuration passed to RAIL_ConfigRxDutyCycle. This API must not be called while the radio is on.

Note

- This feature/API is not supported on the EFR32XG1 family of chips. Use the compile time symbol RAIL_SUPPORTS_CHANNEL_HOPPING or the runtime call RAIL_SupportsChannelHopping() to check whether the platform supports this feature.
- Calling this function will overwrite any settings configured with RAIL_EnableRxChannelHopping.

Definition at line 5541 of file common/rail.h

RAIL_GetDefaultRxDutyCycleConfig

> RAIL_Status_t RAIL_GetDefaultRxDutyCycleConfig (RAIL_Handle_t railHandle, RAIL_RxDutyCycleConfig_t *config)

Get the default RX duty cycle configuration.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	config	An application-provided non-NULL pointer to store the default RX duty cycle configuration.

Returns

- Status code indicating success of the function call. Note that RAIL_STATUS_INVALID_PARAMETER will be returned if the current channel's radio configuration does not support the requested information.

To save power during RX, an application may want to go to low power as long as possible by periodically waking up and trying to "sense" if there are any incoming packets. This API returns the recommended RX duty cycle configuration, so the application can enter low power mode periodically without missing packets. To wake up earlier, the application can reduce the delay parameter. Note that these value might be different if any configuration / channel has changed.

Definition at line `5563` of file `common/rail.h`

Macro Definition Documentation

RAIL_RX_CHANNEL_HOPPING_MAX_SENSE_TIME_US

```
#define RAIL_RX_CHANNEL_HOPPING_MAX_SENSE_TIME_US
```

Value:

```
0x08000000UL
```

The maximum sense time supported for those RAIL_RxChannelHoppingMode_t modes whose parameter(s) specify a sensing time.

Definition at line `4903` of file `common/rail_types.h`

RAIL_RX_CHANNEL_HOPPING_OPTIONS_NONE

```
#define RAIL_RX_CHANNEL_HOPPING_OPTIONS_NONE
```

Value:

```
0U
```

A value representing no options enabled.

Definition at line `4948` of file `common/rail_types.h`

RAIL_RX_CHANNEL_HOPPING_OPTIONS_DEFAULT

```
#define RAIL_RX_CHANNEL_HOPPING_OPTIONS_DEFAULT
```

Value:

```
RAIL_RX_CHANNEL_HOPPING_OPTIONS_NONE
```

All options disabled by default.

Channel hopping will behave as described by other parameters as it did in RAIL 2.7 and earlier.

Definition at line `4954` of file `common/rail_types.h`

RAIL_RX_CHANNEL_HOPPING_OPTION_DEFAULT

```
#define RAIL_RX_CHANNEL_HOPPING_OPTION_DEFAULT
```

Value:

```
RAIL_RX_CHANNEL_HOPPING_OPTIONS_DEFAULT
```

DeprecatedPlease use RAIL_RX_CHANNEL_HOPPING_OPTIONS_DEFAULT instead.

Definition at line 4958 of file common/rail_types.h

RAIL_RX_CHANNEL_HOPPING_OPTION_SKIP_SYNTH_CAL

```
#define RAIL_RX_CHANNEL_HOPPING_OPTION_SKIP_SYNTH_CAL
```

Value:

```
(1U << RAIL_RX_CHANNEL_HOPPING_OPTION_SKIP_SYNTH_CAL_SHIFT)
```

An option to skip synth calibration while **hopping into** the channel specified in the current entry.

Definition at line 4963 of file common/rail_types.h

RAIL_RX_CHANNEL_HOPPING_OPTION_SKIP_DC_CAL

```
#define RAIL_RX_CHANNEL_HOPPING_OPTION_SKIP_DC_CAL
```

Value:

```
(1U << RAIL_RX_CHANNEL_HOPPING_OPTION_SKIP_DC_CAL_SHIFT)
```

An option to skip DC calibration while **hopping into** the channel specified in the current entry.

Definition at line 4968 of file common/rail_types.h

RAIL_RX_CHANNEL_HOPPING_OPTION_RSSI_THRESHOLD

```
#define RAIL_RX_CHANNEL_HOPPING_OPTION_RSSI_THRESHOLD
```

Value:

```
(1U << RAIL_RX_CHANNEL_HOPPING_OPTION_RSSI_THRESHOLD_SHIFT)
```

An option to check RSSI after **hopping into** the channel specified in the current entry and hop if that RSSI is below the threshold specified in RAIL_RxChannelHoppingConfigEntry_t::rssiThresholdDbm.

This check runs in parallel with the RAIL_RxChannelHoppingMode_t specified and may cause a hop sooner than that mode otherwise would.

Definition at line 4976 of file common/rail_types.h

RAIL_RX_CHANNEL_HOPPING_OPTION_STOP

```
#define RAIL_RX_CHANNEL_HOPPING_OPTION_STOP
```

Value:

```
(1U << RAIL_RX_CHANNEL_HOPPING_OPTION_STOP_SHIFT)
```

An option to stop the hopping sequence at this entry in the hop table.

Definition at line 4981 of file common/rail_types.h

RAIL_CHANNEL_HOPPING_INVALID_INDEX

```
#define RAIL_CHANNEL_HOPPING_INVALID_INDEX
```

Value:

```
(0xFEU)
```

A sentinel value to flag an invalid channel hopping index.

Definition at line 5251 of file common/rail_types.h

RAIL_CHANNEL_HOPPING_BUFFER_SIZE_PER_CHANNEL

```
#define RAIL_CHANNEL_HOPPING_BUFFER_SIZE_PER_CHANNEL
```

Value:

```
(55U)
```

The static amount of memory needed per channel for channel hopping, measured in 32 bit words, regardless of the size of radio configuration structures.

Definition at line 310 of file chip/efr32/efr32xg1x/rail_chip_specific.h

RAIL_RxChannelHoppingConfigMultiMode_t

Structure that parameterizes RAIL_RX_CHANNEL_HOPPING_MODE_MULTI_SENSE.

Every RAIL_RxChannelHoppingConfigEntry_t or RAIL_RxDutyCycleConfig_t that uses RAIL_RX_CHANNEL_HOPPING_MODE_MULTI_SENSE must allocate one of these structures in global read-write memory to provide the settings for this mode and for RAIL to use during hopping or duty cycling. A pointer to this structure, cast appropriately, is what is passed in the corresponding RAIL_RxChannelHoppingConfigEntry_t::parameter or RAIL_RxDutyCycleConfig_t::parameter.

The contents of this structure must be initialized prior to each RAIL_ConfigRxChannelHopping() or RAIL_ConfigRxDutyCycle() call and must not be touched thereafter until the next such call. RAIL may change these contents during configuration or operation.

This mode of operation functions algorithmically like this pseudocode:

```
extern bool channelHopping; // true if channel hopping, false if duty cycling
extern RAIL_RxChannelHoppingConfigEntry_t *hopConfigEntry; // current channel

static RAIL_RxChannelHoppingConfigMultiMode_t *multiParams;
static RAIL_Time_t rxStartTime;
static bool preambleSensed;

static void hopOrSuspendRx(uint32_t delay)
{
  disableDemodEvents();
  disableTimerEvents();
  stopTimer();
  if (channelHopping) {
    hopToNextChannel(delay, &hopConfigEntry); // updates hopConfigEntry
  } else {
    suspendRx(delay);
  }
  onStartRx(); // resume receive after delay (on new channel if hopping)
}

void onStartRx(void) // called upon entry to receive
{
  rxStartTime = RAIL_GetTime();
  multiParams = (RAIL_RxChannelHoppingConfigMultiMode_t *)
                (void *)hopConfigEntry->parameter;
  startTimer(rxStartTime + multiParams->timingSense);
  preambleSensed = false;
  enableTimerEvents(); // timer will trigger onTimerEvent() handler
  enableDemodEvents(); // demod will trigger onDemodEvent() handler
}

void onTimerEvent(void) // called when timer expires
{
  hopOrSuspendRx(hopConfigEntry->delay);
}

void onDemodEvent(void) // called when demodulator state changes
{
  if (DEMOD_TIMING_SENSED) {
    stopTimer();
    startTimer(rxStartTime + multiParams->syncDetect);
  }
  if (DEMOD_TIMING_LOST) {
    stopTimer();
    uint32_t newTimeout = RAIL_GetTime() + multiParams->timingReSense;
    uint32_t limitTimeout;
    if (preambleSensed) {
      limitTimeout = rxStartTime + multiParams->syncDetect;
    } else {
      limitTimeout = rxStartTime + multiParams->preambleSense;
    }
    if (newTimeout > limitTimeout) {
      newTimeout = limitTimeout;
    }
    if (newTimeout > RAIL_GetTime()) {
      startTimer(newTimeout);
    } else {
      hopOrSuspendRx(hopConfigEntry->delay);
    }
  }
  if (DEMOD_PREAMBLE_SENSED) {
    preambleSensed = true;
  }
  if (DEMOD_PREAMBLE_LOST) {
    preambleSensed = false;
  }
```

```
disableDemodEvents();disableTimerEvents();stopTimer();receivePacket();// stay on channel to receive framehopOrSuspendRx(0);// continue RX per state transitions with no delay}}
```

Public Attributes

uint32_t	syncDetect Switch to the next channel if sync is not detected before this time, in microseconds, measured from entry to Rx.
uint32_t	preambleSense Switch to the next channel if timing was sensed but then lost after this time, in microseconds, measured from entry to Rx – unless preamble had been sensed in which case any switching is deferred to timingReSense and, if timing is regained, to syncDetect.
uint32_t	timingSense Switch to the next channel if timing is not sensed before this time, in microseconds, measured from entry to Rx.
uint32_t	timingReSense Switch to the next channel if timing was sensed but then lost and not regained before this time, in microseconds, measured from when timing was lost.
uint32_t	status Set this to 0.

Public Attribute Documentation

syncDetect

```
uint32_t RAIL_RxChannelHoppingConfigMultiMode_t::syncDetect
```

Switch to the next channel if sync is not detected before this time, in microseconds, measured from entry to Rx.

This must be greater than preambleSense and less than RAIL_RX_CHANNEL_HOPPING_MAX_SENSE_TIME_US.

Definition at line `5086` of file `common/rail_types.h`

preambleSense

```
uint32_t RAIL_RxChannelHoppingConfigMultiMode_t::preambleSense
```

Switch to the next channel if timing was sensed but then lost after this time, in microseconds, measured from entry to Rx – unless preamble had been sensed in which case any switching is deferred to timingReSense and, if timing is regained, to syncDetect.

This must be greater than timingSense and less than RAIL_RX_CHANNEL_HOPPING_MAX_SENSE_TIME_US.

Definition at line `5095` of file `common/rail_types.h`

timingSense

```
uint32_t RAIL_RxChannelHoppingConfigMultiMode_t::timingSense
```

Switch to the next channel if timing is not sensed before this time, in microseconds, measured from entry to Rx.

This must be greater than 2 and less than RAIL_RX_CHANNEL_HOPPING_MAX_SENSE_TIME_US.

Definition at line 5102 of file common/rail_types.h

timingReSense

> uint32_t RAIL_RxChannelHoppingConfigMultiMode_t::timingReSense

Switch to the next channel if timing was sensed but then lost and not regained before this time, in microseconds, measured from when timing was lost.

This must be less than RAIL_RX_CHANNEL_HOPPING_MAX_SENSE_TIME_US.

Definition at line 5109 of file common/rail_types.h

status

> uint32_t RAIL_RxChannelHoppingConfigMultiMode_t::status

Set this to 0.

This field, along with the others, may be used internally by RAIL during configuration or operation.

Definition at line 5114 of file common/rail_types.h

RAIL_RxChannelHoppingConfigEntry_t

Structure that represents one of the channels that is part of a RAIL_RxChannelHoppingConfig_t sequence of channels used in channel hopping.

Public Attributes

uint16_t	channel The channel number to be used for this entry in the channel hopping sequence.
RAIL_RxChannelHoppingMode_t	mode The mode by which RAIL determines when to hop to the next channel.
RAIL_RxChannelHoppingParameter_t	parameter Depending on the 'mode' parameter that was specified, this member is used to parameterize that mode.
uint32_t	delay Idle time in microseconds to wait before hopping into the channel indicated by this entry.
RAIL_RxChannelHoppingDelayMode_t	delayMode
RAIL_RxChannelHoppingOptions_t	options Bitmask of various options that can be applied to the current channel hop.
int8_t	rssiThresholdDbm The RSSI threshold (in dBm) below which a hop will occur in any mode when RAIL_RX_CHANNEL_HOPPING_OPTION_RSSI_THRESHOLD is specified.
uint8_t	reserved2 Pad bytes reserved for future use and currently ignored.

Public Attribute Documentation

channel

```
uint16_t RAIL_RxChannelHoppingConfigEntry_t::channel
```

The channel number to be used for this entry in the channel hopping sequence.

If this is an invalid channel for the current PHY, the call to RAIL_ConfigRxChannelHopping() will fail.

Definition at line 5129 of file common/rail_types.h

mode

```
RAIL_RxChannelHoppingMode_t RAIL_RxChannelHoppingConfigEntry_t::mode
```

The mode by which RAIL determines when to hop to the next channel.

Definition at line 5131 of file common/rail_types.h

parameter

> RAIL_RxChannelHoppingParameter_t RAIL_RxChannelHoppingConfigEntry_t::parameter

Depending on the 'mode' parameter that was specified, this member is used to parameterize that mode.

See the comments on each value of RAIL_RxChannelHoppingMode_t to learn what to specify here.

Definition at line 5138 of file common/rail_types.h

delay

> uint32_t RAIL_RxChannelHoppingConfigEntry_t::delay

Idle time in microseconds to wait before hopping into the channel indicated by this entry.

Definition at line 5143 of file common/rail_types.h

delayMode

> RAIL_RxChannelHoppingDelayMode_t RAIL_RxChannelHoppingConfigEntry_t::delayMode

DeprecatedSet delayMode to RAIL_RX_CHANNEL_HOPPING_DELAY_MODE_STATIC.

Definition at line 5145 of file common/rail_types.h

options

> RAIL_RxChannelHoppingOptions_t RAIL_RxChannelHoppingConfigEntry_t::options

Bitmask of various options that can be applied to the current channel hop.

Definition at line 5150 of file common/rail_types.h

rssiThresholdDbm

> int8_t RAIL_RxChannelHoppingConfigEntry_t::rssiThresholdDbm

The RSSI threshold (in dBm) below which a hop will occur in any mode when RAIL_RX_CHANNEL_HOPPING_OPTION_RSSI_THRESHOLD is specified.

Definition at line 5156 of file common/rail_types.h

reserved2

> uint8_t RAIL_RxChannelHoppingConfigEntry_t::reserved2[1]

Pad bytes reserved for future use and currently ignored.

Definition at line 5160 of file common/rail_types.h

RAIL_RxChannelHoppingConfig_t

Wrapper struct that will contain the sequence of RAIL_RxChannelHoppingConfigEntry_t that represents the channel sequence to use during RX Channel Hopping.

Public Attributes

uint32_t *	buffer Pointer to contiguous global read-write memory that will be used by RAIL to store channel hopping information throughout its operation.
uint16_t	bufferLength This parameter must be set to the length of the buffer array, in 32 bit words.
uint8_t	numberOfChannels The number of channels in the channel hopping sequence, which is the number of elements in the array that entries points to.
RAIL_RxChannelHoppingConfigEntry_t *	entries A pointer to the first element of an array of RAIL_RxChannelHoppingConfigEntry_t that represents the channels used during channel hopping.

Public Attribute Documentation

buffer

```
uint32_t* RAIL_RxChannelHoppingConfig_t::buffer
```

Pointer to contiguous global read-write memory that will be used by RAIL to store channel hopping information throughout its operation.

It need not be initialized and applications should never write data anywhere in this buffer.

Note

- the size of this buffer must be at least as large as 3 + RAIL_CHANNEL_HOPPING_BUFFER_SIZE_PER_CHANNEL * numberOfChannels, plus the sum of the sizes of the radioConfigDeltaAdd's of the required channels, plus the size of the radioConfigDeltaSubtract. In the case that one channel appears two or more times in your channel sequence (e.g., 1, 2, 3, 2), you must account for the radio configuration size that number of times (i.e., need to count channel 2's radio configuration size twice for the given example). The buffer is for internal use to the library.

Definition at line 5187 of file common/rail_types.h

bufferLength

```
uint16_t RAIL_RxChannelHoppingConfig_t::bufferLength
```

This parameter must be set to the length of the buffer array, in 32 bit words.

This way, during configuration, the software can confirm it's writing within the bounds of the buffer. The configuration API will return an error or trigger RAIL_ASSERT_CHANNEL_HOPPING_BUFFER_TOO_SHORT if bufferLength is insufficient.

Definition at line 5195 of file common/rail_types.h

numberOfChannels

> uint8_t RAIL_RxChannelHoppingConfig_t::numberOfChannels

The number of channels in the channel hopping sequence, which is the number of elements in the array that entries points to.

Definition at line 5200 of file common/rail_types.h

entries

> RAIL_RxChannelHoppingConfigEntry_t* RAIL_RxChannelHoppingConfig_t::entries

A pointer to the first element of an array of RAIL_RxChannelHoppingConfigEntry_t that represents the channels used during channel hopping.

The length of this array must be numberOfChannels.

Definition at line 5207 of file common/rail_types.h

RAIL_RxDutyCycleConfig_t

Structure to configure duty cycled receive mode.

Public Attributes

RAIL_RxChannelHoppingMode_t	mode The mode by which RAIL determines when to exit RX.
RAIL_RxChannelHoppingParameter_t	parameter Depending on the 'mode' parameter that was specified, this member is used to parameterize that mode.
uint32_t	delay Idle time in microseconds to wait before re-entering RX.
RAIL_RxChannelHoppingDelayMode_t	delayMode Indicate how the timing specified in 'delay' should be applied.
RAIL_RxChannelHoppingOptions_t	options Bitmask of various options that can be applied to the current duty cycle operation when the mode is >= RAIL_RX_CHANNEL_HOPPING_MODE_MANUAL_WITH_OPTIONS (ignored otherwise).
int8_t	rssiThresholdDbm The RSSI threshold (in dBm) below which Rx will end in any mode when RAIL_RX_CHANNEL_HOPPING_OPTION_RSSI_THRESHOLD is specified.
uint8_t	reserved2 Pad bytes reserved for future use and currently ignored.

Public Attribute Documentation

mode

```
RAIL_RxChannelHoppingMode_t RAIL_RxDutyCycleConfig_t::mode
```

The mode by which RAIL determines when to exit RX.

Definition at line 5216 of file common/rail_types.h

parameter

```
RAIL_RxChannelHoppingParameter_t RAIL_RxDutyCycleConfig_t::parameter
```

Depending on the 'mode' parameter that was specified, this member is used to parameterize that mode.

See the comments on each value of RAIL_RxChannelHoppingMode_t to learn what to specify here.

Definition at line 5223 of file common/rail_types.h

delay

> uint32_t RAIL_RxDutyCycleConfig_t::delay

Idle time in microseconds to wait before re-entering RX.

Definition at line 5227 of file common/rail_types.h

delayMode

> RAIL_RxChannelHoppingDelayMode_t RAIL_RxDutyCycleConfig_t::delayMode

Indicate how the timing specified in 'delay' should be applied.

Definition at line 5231 of file common/rail_types.h

options

> RAIL_RxChannelHoppingOptions_t RAIL_RxDutyCycleConfig_t::options

Bitmask of various options that can be applied to the current duty cycle operation when the mode is >= RAIL_RX_CHANNEL_HOPPING_MODE_MANUAL_WITH_OPTIONS (ignored otherwise).

Definition at line 5237 of file common/rail_types.h

rssiThresholdDbm

> int8_t RAIL_RxDutyCycleConfig_t::rssiThresholdDbm

The RSSI threshold (in dBm) below which Rx will end in any mode when RAIL_RX_CHANNEL_HOPPING_OPTION_RSSI_THRESHOLD is specified.

Definition at line 5243 of file common/rail_types.h

reserved2

> uint8_t RAIL_RxDutyCycleConfig_t::reserved2[1]

Pad bytes reserved for future use and currently ignored.

Definition at line 5247 of file common/rail_types.h

Radio Configuration

Radio Configuration

Routines for setting up and querying radio configuration information.

These routines allow for runtime flexibility in the radio configuration. Some of the parameters, however, are meant to be generated from the radio calculator in Simplicity Studio. The basic code to configure the radio from this calculator output looks like the example below.

```
// Associate a specific channel configuration with a particular RAIL instance and
// load the settings that correspond to the first usable channel.
RAIL_ConfigChannels(railHandle, channelConfigs[0]);
```

For more information about the types of parameters that can be changed in the other functions and how to use them, see their individual documentation.

Modules

RAIL_FrameType_t

RAIL_AlternatePhy_t

RAIL_ChannelConfigEntry_t

RAIL_ChannelConfig_t

RAIL_ChannelMetadata_t

RAIL_StackInfoCommon_t

RAIL_ChannelConfigEntryAttr_t

Enumerations

enum RAIL_ChannelConfigEntryType_t {

RAIL_CH_TYPE_NORMAL
RAIL_CH_TYPE_CONC_BASE
RAIL_CH_TYPE_CONC_VIRTUAL

}

Define if the channel support using concurrent PHY during channel hopping.

Typedefs

typedef const uint32_t * RAIL_RadioConfig_t

Pointer to a radio configuration array.

typedef void(* RAIL_RadioConfigChangedCallback_t)(RAIL_Handle_t railHandle, const RAIL_ChannelConfigEntry_t *entry)

A pointer to a function called whenever a radio configuration change occurs.

Functions

RAIL_Status_t RAIL_ConfigRadio(RAIL_Handle_t railHandle, RAIL_RadioConfig_t config)
Load a static radio configuration.

uint16_t RAIL_SetFixedLength(RAIL_Handle_t railHandle, uint16_t length)
Modify the currently configured fixed frame length in bytes.

uint16_t RAIL_ConfigChannels(RAIL_Handle_t railHandle, const RAIL_ChannelConfig_t *config, RAIL_RadioConfigChangedCallback_t cb)
Configure the channels supported by this device.

RAIL_Status_t RAIL_GetChannelMetadata(RAIL_Handle_t railHandle, RAIL_ChannelMetadata_t *channelMetadata, uint16_t *length, uint16_t minChannel, uint16_t maxChannel)
Get verbose listing of channel metadata for the current channel configuration.

RAIL_Status_t RAIL_IsValidChannel(RAIL_Handle_t railHandle, uint16_t channel)
Check whether the channel exists in RAIL.

RAIL_Status_t RAIL_PrepareChannel(RAIL_Handle_t railHandle, uint16_t channel)
Cause radio settings associated with a particular channel to be applied to hardware.

RAIL_Status_t RAIL_GetChannel(RAIL_Handle_t railHandle, uint16_t *channel)
Return the current RAIL channel.

RAIL_Status_t RAIL_GetChannelAlt(RAIL_Handle_t railHandle, uint16_t *channel)
Return the current RAIL channel.

uint32_t RAIL_GetSymbolRate(RAIL_Handle_t railHandle)
Return the symbol rate for the current PHY.

uint32_t RAIL_GetBitRate(RAIL_Handle_t railHandle)
Return the bit rate for the current PHY.

RAIL_Status_t RAIL_SetPaCTune(RAIL_Handle_t railHandle, uint8_t txPaCtuneValue, uint8_t rxPaCtuneValue)
Set the PA capacitor tune value for transmit and receive.

RAIL_Status_t RAIL_GetSyncWords(RAIL_Handle_t railHandle, RAIL_SyncWordConfig_t *syncWordConfig)
Get the sync words and their length.

RAIL_Status_t RAIL_ConfigSyncWords(RAIL_Handle_t railHandle, const RAIL_SyncWordConfig_t *syncWordConfig)
Set the selected sync words and their length.

Macros

#define RAIL_SETFIXEDLENGTH_INVALID (0xFFFFU)
An invalid return value when calling RAIL_SetFixedLength().

#define RADIO_CONFIG_ENABLE_CONC_PHY 1
Indicates this version of RAIL supports concurrent PHY information in radio configurator output.

#define RADIO_CONFIG_ENABLE_STACK_INFO undefined
Indicates this version of RAIL supports stack info feature in radio configurator output.

Enumeration Documentation

RAIL_ChannelConfigEntryType_t

```
RAIL_ChannelConfigEntryType_t
```

Define if the channel support using concurrent PHY during channel hopping.

RAIL_RX_CHANNEL_HOPPING_MODE_CONC and RAIL_RX_CHANNEL_HOPPING_MODE_VT can only be used if the channel supports it.

Enumerator	
RAIL_CH_TYPE_NORMAL	Not a concurrent PHY.
RAIL_CH_TYPE_CONC_BASE	Base concurrent PHY.
RAIL_CH_TYPE_CONC_VIRTUAL	Virtual concurrent PHY.

Definition at line 2028 of file common/rail_types.h

Typedef Documentation

RAIL_RadioConfig_t

```
typedef const uint32_t* RAIL_RadioConfig_t
```

Pointer to a radio configuration array.

The radio configuration properly configures the radio for operation on a protocol. These configurations are very chip-specific should not be created or edited by hand.

Definition at line 1966 of file common/rail_types.h

RAIL_RadioConfigChangedCallback_t

```
RAIL_RadioConfigChangedCallback_t )(RAIL_Handle_t railHandle, const RAIL_ChannelConfigEntry_t *entry)
```

A pointer to a function called whenever a radio configuration change occurs.

Parameters

[in]	railHandle	A handle for RAIL instance.
[in]	entry	The radio configuration entry being changed to.

Definition at line 2353 of file common/rail_types.h

Function Documentation

RAIL_ConfigRadio

```
RAIL_Status_t RAIL_ConfigRadio (RAIL_Handle_t railHandle, RAIL_RadioConfig_t config)
```

Load a static radio configuration.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	config	A pointer to a radio configuration.

Returns

- Status code indicating success of the function call.

The configuration passed into this function should be auto-generated and not manually created or edited. By default, do not call this function in RAIL 2.x and later unless instructed by Silicon Labs because it may bypass updating certain RAIL state. In RAIL 2.x and later, the RAIL_ConfigChannels function applies the default radio configuration automatically.

Definition at line `559` of file `common/rail.h`

RAIL_SetFixedLength

```
uint16_t RAIL_SetFixedLength (RAIL_Handle_t railHandle, uint16_t length)
```

Modify the currently configured fixed frame length in bytes.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	length	The expected fixed frame length. A value of 0 is infinite. A value of RAIL_SETFIXEDLENGTH_INVALID restores the frame's length back to the length specified by the default frame type configuration.

Returns

- Length configured; The new frame length configured into the hardware for use. 0 if in infinite mode, or RAIL_SETFIXEDLENGTH_INVALID if the frame length has not yet been overridden by a valid value.

Sets the fixed-length configuration for transmit and receive. Be careful when using this function in receive and transmit as this function changes the default frame configuration and remains in force until it is called again with an input value of RAIL_SETFIXEDLENGTH_INVALID. This function will override any fixed or variable length settings from a radio configuration.

Definition at line `580` of file `common/rail.h`

RAIL_ConfigChannels

```
uint16_t RAIL_ConfigChannels (RAIL_Handle_t railHandle, const RAIL_ChannelConfig_t *config,
RAIL_RadioConfigChangedCallback_t cb)
```

Configure the channels supported by this device.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	config	A pointer to the channel configuration for your device. This pointer will be cached in the library so it must exist for the runtime of the application. Typically, this should be what is stored in Flash by the configuration tool.
[in]	cb	Function called whenever a radio configuration change occurs.

Returns

- Returns the first available channel in the configuration.

When configuring channels on EFR32, the radio tuner is reconfigured based on the frequency and channel spacing in the channel configuration.

Note

- config can be NULL to simply register or unregister the cb callback function when using RAIL internal protocol-specific radio configuration APIs for BLE, IEEE 802.15.4, or Z-Wave, which lack callback specification. In this use case, 0 is returned.

Definition at line `601` of file `common/rail.h`

RAIL_GetChannelMetadata

> RAIL_Status_t RAIL_GetChannelMetadata (RAIL_Handle_t railHandle, RAIL_ChannelMetadata_t *channelMetadata, uint16_t *length, uint16_t minChannel, uint16_t maxChannel)

Get verbose listing of channel metadata for the current channel configuration.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	channelMetadata	Allocated array that will be populated with channel metadata.
[inout]	length	Pointer to the length of the channelMetadata. This value will be updated to the number of channels written to the array.
[in]	minChannel	Minimum channel number about which to collect data.
[in]	maxChannel	Maximum channel number about which to collect data.

Returns

- Status of the call. RAIL_STATUS_INVALID_PARAMETER means that, based on the currently active radio configuration, there are more channels to write than there is space provided in the allocated channelMetadata. However, the channel metadata that was written is valid. RAIL_STATUS_INVALID_STATE indicates that the channel configuration has not been configured. RAIL_STATUS_NO_ERROR indicates complete success.

Definition at line `623` of file `common/rail.h`

RAIL_IsValidChannel

> RAIL_Status_t RAIL_IsValidChannel (RAIL_Handle_t railHandle, uint16_t channel)

Check whether the channel exists in RAIL.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	channel	A channel number to check.

Returns

- Returns RAIL_STATUS_NO_ERROR if channel exists

Returns RAIL_STATUS_INVALID_PARAMETER if the given channel does not exist in the channel configuration currently used or RAIL_STATUS_NO_ERROR if the channel is valid.

Definition at line `640` of file `common/rail.h`

RAIL_PrepareChannel

> RAIL_Status_t RAIL_PrepareChannel (RAIL_Handle_t railHandle, uint16_t channel)

Cause radio settings associated with a particular channel to be applied to hardware.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	channel	The channel to prepare for use.

Returns

- RAIL_STATUS_NO_ERROR on success or RAIL_STATUS_INVALID_PARAMETER if the given channel does not have an associated channel configuration entry.

This function walks the channelConfigEntry list and applies the configuration associated with the specified channel. This function manually changes channels without starting a TX or RX operation.

When successful, the radio is idled. When unsuccessful, the radio state will not be altered.

Definition at line `660` of file `common/rail.h`

RAIL_GetChannel

> RAIL_Status_t RAIL_GetChannel (RAIL_Handle_t railHandle, uint16_t *channel)

Return the current RAIL channel.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	channel	The channel for which RAIL is currently configured.

Returns

- RAIL_STATUS_NO_ERROR on success or RAIL_STATUS_INVALID_CALL if the radio is not configured for any channel or RAIL_STATUS_INVALID_PARAMETER if channel parameter is NULL.

This function returns the channel most recently specified in API calls that pass in a channel to tune to, namely RAIL_PrepareChannel, RAIL_StartTx, RAIL_StartScheduledTx, RAIL_StartCcaCsmaTx, RAIL_StartCcaLbtTx, RAIL_StartScheduledCcaCsmaTx, RAIL_StartScheduledCcaLbtTx, RAIL_StartRx, RAIL_ScheduleRx, RAIL_StartAverageRssi, RAIL_StartTxStream, RAIL_StartTxStreamAlt. It doesn't follow changes RAIL performs implicitly during channel hopping and mode switch.

Definition at line `680` of file `common/rail.h`

RAIL_GetChannelAlt

> RAIL_Status_t RAIL_GetChannelAlt (RAIL_Handle_t railHandle, uint16_t *channel)

Return the current RAIL channel.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	channel	The channel for which RAIL is currently configured.

Returns

- RAIL_STATUS_NO_ERROR on success or RAIL_STATUS_INVALID_CALL if the radio is not configured for any channel or RAIL_STATUS_INVALID_PARAMETER if channel parameter is NULL.

This function returns the channel the radio is currently tuned to if the specified RAIL handle is active. It returns the channel it will be tuned to during the next protocol switch if the handle is inactive. The channel returned may be different than what RAIL_GetChannel returns when channel hopping or mode switch are involved.

Definition at line `697` of file `common/rail.h`

RAIL_GetSymbolRate

```
uint32_t RAIL_GetSymbolRate (RAIL_Handle_t railHandle)
```

Return the symbol rate for the current PHY.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- The symbol rate in symbols per second or 0.

The symbol rate is the rate of symbol changes over the air. For non-DSSS PHYs, this is the same as the baudrate. For DSSS PHYs, it is the baudrate divided by the length of a chipping sequence. For more information, see the modem calculator documentation. If the rate cannot be calculated, this function returns 0.

Definition at line `711` of file `common/rail.h`

RAIL_GetBitRate

```
uint32_t RAIL_GetBitRate (RAIL_Handle_t railHandle)
```

Return the bit rate for the current PHY.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- The bit rate in bits per second or 0.

The bit rate is the effective over-the-air data rate. It does not account for extra spreading for forward error correction, and so on, but accounts for modulation schemes, DSSS, and other configurations. For more information, see the modem calculator documentation. If the rate cannot be calculated, this function returns 0.

Definition at line `739` of file `common/rail.h`

RAIL_SetPaCTune

```
RAIL_Status_t RAIL_SetPaCTune (RAIL_Handle_t railHandle, uint8_t txPaCtuneValue, uint8_t rxPaCtuneValue)
```

Set the PA capacitor tune value for transmit and receive.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	txPaCtuneValue	PA Ctune value for TX mode.
[in]	rxPaCtuneValue	PA Ctune value for RX mode.

Returns

- Status code indicating success of the function call.

Tunes the impedance of the transmit and receive modes by changing the amount of capacitance at the PA output. Changes made to the TX Power configuration, e.g., calling RAIL_ConfigTxPower, will undo changes made to PA capacitor tune value

for transmit and receive via RAIL_SetPaCTune.

Definition at line 769 of file common/rail.h

RAIL_GetSyncWords

> RAIL_Status_t RAIL_GetSyncWords (RAIL_Handle_t railHandle, RAIL_SyncWordConfig_t *syncWordConfig)

Get the sync words and their length.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	syncWordConfig	An application-provided non-NULL pointer to store RAIL_SyncWordConfig_t sync word information.

Returns

- Status code indicating success of the function call.

Definition at line 781 of file common/rail.h

RAIL_ConfigSyncWords

> RAIL_Status_t RAIL_ConfigSyncWords (RAIL_Handle_t railHandle, const RAIL_SyncWordConfig_t *syncWordConfig)

Set the selected sync words and their length.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	syncWordConfig	A non-NULL pointer to RAIL_SyncWordConfig_t specifying the sync words and their length. The desired length should be between 2 and 32 bits inclusive, however it is recommended to not change the length below what the PHY syncWord length is configured to be. Changing the syncWord length, especially to that which is lower than the default length, may result in a decrease in packet reception rate or may not work at all. Other values will result in RAIL_STATUS_INVALID_PARAMETER. The default syncWord continues to be valid.

Returns

- Status code indicating success of the function call. When the custom sync word(s) applied by this API are no longer needed, or to revert to default sync word, calling RAIL_ConfigChannels() will re-establish the sync words specified in the radio configuration.

This function will return RAIL_STATUS_INVALID_STATE if called when BLE has been enabled for this railHandle. When changing sync words in BLE mode, use RAIL_BLE_ConfigChannelRadioParams instead.

Definition at line 806 of file common/rail.h

Macro Definition Documentation

RAIL_SETFIXEDLENGTH_INVALID

> #define RAIL_SETFIXEDLENGTH_INVALID

Value:

```
(0xFFFFU)
```

An invalid return value when calling RAIL_SetFixedLength().

An invalid return value when calling RAIL_SetFixedLength() while the radio is not in fixed-length mode.

Definition at line 2013 of file common/rail_types.h

RADIO_CONFIG_ENABLE_CONC_PHY

```
#define RADIO_CONFIG_ENABLE_CONC_PHY
```

Value:

```
1
```

Indicates this version of RAIL supports concurrent PHY information in radio configurator output.

Needed for backwards compatibility.

Definition at line 2046 of file common/rail_types.h

RADIO_CONFIG_ENABLE_STACK_INFO

```
#define RADIO_CONFIG_ENABLE_STACK_INFO
```

Indicates this version of RAIL supports stack info feature in radio configurator output.

Needed for backwards compatibility.

Definition at line 2053 of file common/rail_types.h

RAIL_FrameType_t

Configures if there is a frame type in your frame and the lengths of each frame.

The number of bits set in the mask determines the number of elements in frameLen. A maximum of 8 different frame types may be specified.

Public Attributes

uint16_t *	frameLen A pointer to an array of frame lengths for each frame type.
uint8_t	offset Zero-indexed location of the byte containing the frame type field.
uint8_t	mask A bitmask of the frame type field, which determines a number of frames expected based on the number of bits set.
uint8_t	isValid A bitmask that marks if each frame is valid or should be filtered.
uint8_t	addressFilter A bitmask that marks if each frame should have the address filter applied.

Public Attribute Documentation

frameLen

```
uint16_t* RAIL_FrameType_t::frameLen
```

A pointer to an array of frame lengths for each frame type.

The length of this array should be equal to the number of frame types. The array that frameLen points to should not change location or be modified.

Definition at line `1980` of file `common/rail_types.h`

offset

```
uint8_t RAIL_FrameType_t::offset
```

Zero-indexed location of the byte containing the frame type field.

Definition at line `1984` of file `common/rail_types.h`

mask

```
uint8_t RAIL_FrameType_t::mask
```

A bitmask of the frame type field, which determines a number of frames expected based on the number of bits set.

No more than 3 bits can be set in the mask and they must be contiguous ones. For example, if the highest three bits of the byte specified by offset constitute the frame type, then mask should be 0xE0, which has 3 bits set, indicating 8 possible frame types.

Definition at line `1992` of file `common/rail_types.h`

isValid

```
uint8_t RAIL_FrameType_t::isValid
```

A bitmask that marks if each frame is valid or should be filtered.

Frame type 0 corresponds to the lowest bit in isValid. If the frame is filtered, a RAIL_EVENT_RX_PACKET_ABORTED will be raised.

Definition at line `1998` of file `common/rail_types.h`

addressFilter

```
uint8_t RAIL_FrameType_t::addressFilter
```

A bitmask that marks if each frame should have the address filter applied.

Frame type 0 corresponds to the least significant bit in addressFilter.

Definition at line `2003` of file `common/rail_types.h`

RAIL_AlternatePhy_t

Alternate PHY configuration entry structure, which gathers some info on the alternate PHY in the context of concurrent mode.

Public Attributes

uint32_t	baseFrequency A base frequency in Hz of this channel set.
uint32_t	channelSpacing A channel spacing in Hz of this channel set.
uint16_t	numberOfChannels The number of channels (and not the channel number !)
uint16_t	minIf_kHz minimum IF for the alternate PHY in kHz.
uint16_t	minBaseIf_kHz minimum IF for the base PHY in kHz.
bool	isOfdmModem Indicates that OFDM modem is used by this alternate PHY.
uint32_t	rateInfo rate Info of the alternate PHY.

Public Attribute Documentation

baseFrequency

```
uint32_t RAIL_AlternatePhy_t::baseFrequency
```

A base frequency in Hz of this channel set.

Definition at line `2061` of file `common/rail_types.h`

channelSpacing

```
uint32_t RAIL_AlternatePhy_t::channelSpacing
```

A channel spacing in Hz of this channel set.

Definition at line `2062` of file `common/rail_types.h`

numberOfChannels

```
uint16_t RAIL_AlternatePhy_t::numberOfChannels
```

The number of channels (and not the channel number !)

Definition at line 2063 of file common/rail_types.h

minIf_kHz

```
uint16_t RAIL_AlternatePhy_t::minIf_kHz
```

minimum IF for the alternate PHY in kHz.

Definition at line 2064 of file common/rail_types.h

minBaseIf_kHz

```
uint16_t RAIL_AlternatePhy_t::minBaseIf_kHz
```

minimum IF for the base PHY in kHz.

Definition at line 2065 of file common/rail_types.h

isOfdmModem

```
bool RAIL_AlternatePhy_t::isOfdmModem
```

Indicates that OFDM modem is used by this alternate PHY.

Definition at line 2066 of file common/rail_types.h

rateInfo

```
uint32_t RAIL_AlternatePhy_t::rateInfo
```

rate Info of the alternate PHY.

Definition at line 2067 of file common/rail_types.h

RAIL_ChannelConfigEntry_t

A channel configuration entry structure, which defines a channel range and parameters across which a corresponding radio configuration is valid.

operating frequency = baseFrequency

- channelSpacing * (channel - physicalChannelOffset);

Public Attributes

RAIL_RadioConfig_t	phyConfigDeltaAdd The minimum radio configuration to apply to the base configuration for this channel set.
uint32_t	baseFrequency A base frequency in Hz of this channel set.
uint32_t	channelSpacing A channel spacing in Hz of this channel set.
uint16_t	physicalChannelOffset The offset to subtract from the logical channel to align them with the zero based physical channels which are relative to baseFrequency.
uint16_t	channelNumberStart The first valid RAIL channel number for this channel set.
uint16_t	channelNumberEnd The last valid RAIL channel number for this channel set.
RAIL_TxPower_t	maxPower The maximum power allowed in this channel set.
RAIL_ChannelConfigEntryAttr_t *	attr A pointer to a structure containing attributes specific to this channel set.
RAIL_ChannelConfigEntryType_t	entryType Indicates channel config type.
uint8_t	reserved to align to 32-bit boundary.
const uint8_t *	stackInfo Array containing information according to the protocolId value, first byte of this array.
RAIL_AlternatePhy_t *	alternatePhy Pointer to alternate PHY.

Public Attribute Documentation

phyConfigDeltaAdd

> RAIL_RadioConfig_t RAIL_ChannelConfigEntry_t::phyConfigDeltaAdd

The minimum radio configuration to apply to the base configuration for this channel set.

Definition at line 2079 of file common/rail_types.h

baseFrequency

> uint32_t RAIL_ChannelConfigEntry_t::baseFrequency

A base frequency in Hz of this channel set.

Definition at line 2081 of file common/rail_types.h

channelSpacing

> uint32_t RAIL_ChannelConfigEntry_t::channelSpacing

A channel spacing in Hz of this channel set.

Definition at line 2082 of file common/rail_types.h

physicalChannelOffset

> uint16_t RAIL_ChannelConfigEntry_t::physicalChannelOffset

The offset to subtract from the logical channel to align them with the zero based physical channels which are relative to baseFrequency.

(i.e., By default ch 0 = base freq, but if offset = 11, ch 11 = base freq.)

Definition at line 2083 of file common/rail_types.h

channelNumberStart

> uint16_t RAIL_ChannelConfigEntry_t::channelNumberStart

The first valid RAIL channel number for this channel set.

Definition at line 2089 of file common/rail_types.h

channelNumberEnd

> uint16_t RAIL_ChannelConfigEntry_t::channelNumberEnd

The last valid RAIL channel number for this channel set.

Definition at line 2091 of file common/rail_types.h

maxPower

> RAIL_TxPower_t RAIL_ChannelConfigEntry_t::maxPower

The maximum power allowed in this channel set.

Definition at line 2093 of file common/rail_types.h

attr

```
RAIL_ChannelConfigEntryAttr_t* RAIL_ChannelConfigEntry_t::attr
```

A pointer to a structure containing attributes specific to this channel set.

Definition at line 2094 of file common/rail_types.h

entryType

```
RAIL_ChannelConfigEntryType_t RAIL_ChannelConfigEntry_t::entryType
```

Indicates channel config type.

Definition at line 2097 of file common/rail_types.h

reserved

```
uint8_t RAIL_ChannelConfigEntry_t::reserved[3]
```

to align to 32-bit boundary.

Definition at line 2098 of file common/rail_types.h

stackInfo

```
const uint8_t* RAIL_ChannelConfigEntry_t::stackInfo
```

Array containing information according to the protocolId value, first byte of this array.

The first 2 fields are common to all protocols and accessible by RAIL, others are ignored by RAIL and only used by the application. Common fields are listed in RAIL_StackInfoCommon_t.

Definition at line 2099 of file common/rail_types.h

alternatePhy

```
RAIL_AlternatePhy_t* RAIL_ChannelConfigEntry_t::alternatePhy
```

Pointer to alternate PHY.

Definition at line 2104 of file common/rail_types.h

RAIL_ChannelConfig_t

A channel configuration structure, which defines the channel meaning when a channel number is passed into a RAIL function, e.g., RAIL_StartTx() and RAIL_StartRx().

A RAIL_ChannelConfig_t structure defines the channel scheme that an application uses when registered in RAIL_ConfigChannels().

These are a few examples of different channel configurations:

```
// 21 channels starting at 2.45 GHz with channel spacing of 1 MHz
// ... generated by Simplicity Studio (i.e., rail_config.c) ...
const uint32_t generated[] = { ... };
RAIL_ChannelConfigEntryAttr_t generated_entryAttr = { ... };
const RAIL_ChannelConfigEntry_t generated_channels[] = {
 {
  .phyConfigDeltaAdd = NULL, // Add this to default configuration for this entry
  .baseFrequency = 2450000000,
  .channelSpacing = 1000000,
  .physicalChannelOffset = 0,
  .channelNumberStart = 0,
  .channelNumberEnd = 20,
  .maxPower = RAIL_TX_POWER_MAX,
  .attr = &generated_entryAttr
 },
};
const RAIL_ChannelConfig_t generated_channelConfig = {
 .phyConfigBase = generated, // Default radio configuration for all entries
 .phyConfigDeltaSubtract = NULL, // Subtract this to restore the default configuration
 .configs = generated_channels,
 .length = 1 // There are this many channel configuration entries
};
const RAIL_ChannelConfig_t *channelConfigs[] = {
 &generated_channelConfig,
 NULL
};
// ... in main code ...
// Associate a specific channel configuration with a particular RAIL instance.
RAIL_ConfigChannels(railHandle, channelConfigs[0]);

// 4 nonlinear channels
// ... in rail_config.c ...
const uint32_t generated[] = { ... };
RAIL_ChannelConfigEntryAttr_t generated_entryAttr = { ... };
const RAIL_ChannelConfigEntry_t generated_channels[] = {
 {
  .phyConfigDeltaAdd = NULL, // Add this to default configuration for this entry
  .baseFrequency = 910123456,
  .channelSpacing = 0,
  .physicalChannelOffset = 0,
  .channelNumberStart = 0,
  .channelNumberEnd = 0,
  .maxPower = RAIL_TX_POWER_MAX,
  .attr = &generated_entryAttr
 },
 {
  .phyConfigDeltaAdd = NULL,
  .baseFrequency = 911654789,
  .channelSpacing = 0,
  .physicalChannelOffset = 0, // Since ch spacing = 0, offset can be 0
  .channelNumberStart = 1,
  .channelNumberEnd = 1,
  .maxPower = RAIL_TX_POWER_MAX,
  .attr = &generated_entryAttr
 },
 {
  .phyConfigDeltaAdd = NULL,
  .baseFrequency = 912321456,
  .channelSpacing = 100000,
  .physicalChannelOffset = 2, // Since ch spacing != 0, offset = 2
  .channelNumberStart = 2, // ch 2 = baseFrequency
  .channelNumberEnd = 2,
  .maxPower = RAIL_TX_POWER_MAX,
  .attr = &generated_entryAttr
 },
 {
```

```
.baseFrequency =913147852,.channelSpacing =0,.physicalChannelOffset =0,.channelNumberStart =3,.channelNumberEnd =3,.maxPower =
RAIL_TX_POWER_MAX,.attr =&generated_entryAttr
  },};
const RAIL_ChannelConfig_t generated_channelConfig ={.phyConfigBase = generated,// Default radio configuration for all
entries.phyConfigDeltaSubtract = NULL,// Subtract this to restore the default configuration.configs = generated_channels,.length =4// There are
this many channel configuration entries};
const RAIL_ChannelConfig_t *channelConfigs[]={&generated_channelConfig,
  NULL
};// ... in main code ...// Associate a specific channel configuration with a particular RAIL instance.RAIL_ConfigChannels(railHandle,
channelConfigs[0]);// Multiple radio configurations// ... in rail_config.c ...
const uint32_t generated0[]={...};
RAIL_ChannelConfigEntryAttr_t generated0_entryAttr ={...};
const RAIL_ChannelConfigEntry_t generated0_channels[]={{.phyConfigDeltaAdd = NULL,// Add this to the default configuration for this
entry.baseFrequency =2450000000,.channelSpacing =1000000,.physicalChannelOffset =0,.channelNumberStart =0,.channelNumberEnd
=20,.maxPower = RAIL_TX_POWER_MAX,.attr =&generated0_entryAttr
  },};
const RAIL_ChannelConfig_t generated0_channelConfig ={.phyConfigBase = generated0,// Default radio configuration for all
entries.phyConfigDeltaSubtract = NULL,// Subtract this to restore default configuration.configs = generated0_channels,.length =1// There are this
many channel configuration entries};
const uint32_t generated1[]={...};
RAIL_ChannelConfigEntryAttr_t generated1_entryAttr ={...};
const RAIL_ChannelConfigEntry_t generated1_channels[]={{.phyConfigDeltaAdd = NULL,.baseFrequency =2450000000,.channelSpacing
=1000000,.physicalChannelOffset =0,.channelNumberStart =0,.channelNumberEnd =20,.maxPower =-100,// Use this entry when TX power <=
-10dBm.attr =&generated1_entryAttr
  },{.phyConfigDeltaAdd = NULL,.baseFrequency =2450000000,.channelSpacing =1000000,.physicalChannelOffset =0,.channelNumberStart
=0,.channelNumberEnd =20,.maxPower =15,// Use this entry when TX power > -10dBm// and TX power <= 1.5dBm.attr =&generated1_entryAttr
  },{.phyConfigDeltaAdd = NULL,.baseFrequency =2450000000,.channelSpacing =1000000,.physicalChannelOffset =0,.channelNumberStart
=0,.channelNumberEnd =20,.maxPower = RAIL_TX_POWER_MAX,// Use this entry when TX power > 1.5dBm.attr =&generated1_entryAttr
  },};
const RAIL_ChannelConfig_t generated1_channelConfig ={.phyConfigBase = generated1,.phyConfigDeltaSubtract = NULL,.configs =
generated1_channels,.length =3};
const uint32_t generated2[]={...};
RAIL_ChannelConfigEntryAttr_t generated2_entryAttr ={...};
const RAIL_ChannelConfigEntry_t generated2_channels[]={{.phyConfigDeltaAdd = NULL,.baseFrequency =2450000000,.channelSpacing
=1000000,.physicalChannelOffset =0,.channelNumberStart =0,.channelNumberEnd =20,.maxPower = RAIL_TX_POWER_MAX,.attr
=&generated2_entryAttr
  },};
const RAIL_ChannelConfig_t generated2_channelConfig ={.phyConfigBase = generated2,.phyConfigDeltaSubtract = NULL,.configs =
generated2_channels,.length =1};
const RAIL_ChannelConfig_t *channelConfigs[]={&generated0_channelConfig,&generated1_channelConfig,&generated2_channelConfig,
  NULL
};// ... in main code ...// Create a unique RAIL handle for each unique channel configuration.
railHandle0 =RAIL_Init(&railCfg0,&RAILCb_RfReady0);
railHandle1 =RAIL_Init(&railCfg1,&RAILCb_RfReady1);
railHandle2 =RAIL_Init(&railCfg2,&RAILCb_RfReady2);// Associate each channel configuration with its corresponding RAIL
handle.RAIL_ConfigChannels(railHandle0, channelConfigs[0]);RAIL_ConfigChannels(railHandle1, channelConfigs[1]);RAIL_ConfigChannels(railHandle2,
channelConfigs[2]);// Use a RAIL handle and channel to access the desired channel configuration entry.RAIL_SetTxPowerDbm(railHandle1,100);//
set 10.0 dBm TX powerRAIL_StartRx(railHandle1,0,&schedInfo);// RX using generated1_channels[2]RAIL_SetTxPowerDbm(railHandle1,0);// set 0 dBm
TX powerRAIL_StartRx(railHandle1,0,&schedInfo);// RX using generated1_channels[1]RAIL_StartRx(railHandle2,0,&schedInfo);// RX using
generated2_channels[0]
```

Public Attributes

RAIL_RadioConfig_t	phyConfigBase Base radio configuration for the corresponding channel configuration entries.
RAIL_RadioConfig_t	phyConfigDeltaSubtract Minimum radio configuration to restore channel entries back to base configuration.
const RAIL_ChannelConfigEntry_t *	configs Pointer to an array of RAIL_ChannelConfigEntry_t entries.

uint32_t length
Number of RAIL_ChannelConfigEntry_t entries.

uint32_t signature
Signature for this structure.

uint32_t xtalFrequencyHz
Crystal Frequency for the channel config.

Public Attribute Documentation

phyConfigBase

> RAIL_RadioConfig_t RAIL_ChannelConfig_t::phyConfigBase

Base radio configuration for the corresponding channel configuration entries.

Definition at line 2314 of file common/rail_types.h

phyConfigDeltaSubtract

> RAIL_RadioConfig_t RAIL_ChannelConfig_t::phyConfigDeltaSubtract

Minimum radio configuration to restore channel entries back to base configuration.

Definition at line 2316 of file common/rail_types.h

configs

> const RAIL_ChannelConfigEntry_t* RAIL_ChannelConfig_t::configs

Pointer to an array of RAIL_ChannelConfigEntry_t entries.

Definition at line 2319 of file common/rail_types.h

length

> uint32_t RAIL_ChannelConfig_t::length

Number of RAIL_ChannelConfigEntry_t entries.

Definition at line 2322 of file common/rail_types.h

signature

> uint32_t RAIL_ChannelConfig_t::signature

Signature for this structure.

Only used on modules.

Definition at line 2323 of file common/rail_types.h

xtalFrequencyHz

```
uint32_t RAIL_ChannelConfig_t::xtalFrequencyHz
```

Crystal Frequency for the channel config.

Definition at line 2324 of file common/rail_types.h

RAIL_ChannelMetadata_t

Container for individual channel metadata.

Public Attributes

uint16_t	channel Channel number.
uint16_t	reserved Word alignment.
uint32_t	frequency Channel frequency, in Hz.

Public Attribute Documentation

channel

```
uint16_t RAIL_ChannelMetadata_t::channel
```

Channel number.

Definition at line 2332 of file common/rail_types.h

reserved

```
uint16_t RAIL_ChannelMetadata_t::reserved
```

Word alignment.

Definition at line 2333 of file common/rail_types.h

frequency

```
uint32_t RAIL_ChannelMetadata_t::frequency
```

Channel frequency, in Hz.

Definition at line 2334 of file common/rail_types.h

RAIL_StackInfoCommon_t

StackInfo fields common to all protocols.

Public Attributes

uint8_t	protocolId Same as RAIL_PtiProtocol_t.
uint8_t	phyId PHY Id depending on the protocol_id value.

Public Attribute Documentation

protocolId

```
uint8_t RAIL_StackInfoCommon_t::protocolId
```

Same as RAIL_PtiProtocol_t.

Definition at line 2342 of file common/rail_types.h

phyId

```
uint8_t RAIL_StackInfoCommon_t::phyId
```

PHY Id depending on the protocol_id value.

Definition at line 2343 of file common/rail_types.h

RAIL_ChannelConfigEntryAttr_t

A channel configuration entry attribute structure.

Items listed are designed to be altered and updated during run-time.

Receive

Receive

APIs related to packet receive.

Modules

RAIL_ScheduleRxConfig_t

RAIL_RxPacketInfo_t

RAIL_RxPacketDetails_t

Address Filtering

Packet Information

Enumerations

enum RAIL_RxOptions_t {

RAIL_RX_OPTION_STORE_CRC_SHIFT = 0
RAIL_RX_OPTION_IGNORE_CRC_ERRORS_SHIFT
RAIL_RX_OPTION_ENABLE_DUALSYNC_SHIFT
RAIL_RX_OPTION_TRACK_ABORTED_FRAMES_SHIFT
RAIL_RX_OPTION_REMOVE_APPENDED_INFO_SHIFT
RAIL_RX_OPTION_ANTENNA0_SHIFT
RAIL_RX_OPTION_ANTENNA1_SHIFT
RAIL_RX_OPTION_DISABLE_FRAME_DETECTION_SHIFT
RAIL_RX_OPTION_CHANNEL_SWITCHING_SHIFT
RAIL_RX_OPTION_FAST_RX2RX_SHIFT
RAIL_RX_OPTION_ENABLE_COLLISION_DETECTION_SHIFT

}

Receive options, in reality a bitmask.

enum RAIL_RxPacketStatus_t {

RAIL_RX_PACKET_NONE = 0
RAIL_RX_PACKET_ABORT_FORMAT
RAIL_RX_PACKET_ABORT_FILTERED
RAIL_RX_PACKET_ABORT_ABORTED
RAIL_RX_PACKET_ABORT_OVERFLOW
RAIL_RX_PACKET_ABORT_CRC_ERROR
RAIL_RX_PACKET_READY_CRC_ERROR
RAIL_RX_PACKET_READY_SUCCESS
RAIL_RX_PACKET_RECEIVING

}

The packet status code associated with a packet received or currently being received.

Typedefs

typedef const void * RAIL_RxPacketHandle_t

A handle used to reference a packet during reception processing.

typedef uint8_t(* RAIL_ConvertLqiCallback_t)(uint8_t lqi, int8_t rssi)
A pointer to a function called before LQI is copied into the RAIL_RxPacketDetails_t structure.

Functions

RAIL_Status_t RAIL_ConfigRxOptions(RAIL_Handle_t railHandle, RAIL_RxOptions_t mask, RAIL_RxOptions_t options)
Configure receive options.

void RAIL_IncludeFrameTypeLength(RAIL_Handle_t railHandle)
Include the code necessary for frame type based length decoding.

void RAILCb_ConfigFrameTypeLength(RAIL_Handle_t railHandle, const RAIL_FrameType_t *frameType)
Handle frame type length.

RAIL_Status_t RAIL_StartRx(RAIL_Handle_t railHandle, uint16_t channel, const RAIL_SchedulerInfo_t *schedulerInfo)
Start the receiver on a specific channel.

RAIL_Status_t RAIL_ScheduleRx(RAIL_Handle_t railHandle, uint16_t channel, const RAIL_ScheduleRxConfig_t *cfg, const RAIL_SchedulerInfo_t *schedulerInfo)
Schedule a receive window for some future time.

RAIL_RxPacketHandle_t RAIL_HoldRxPacket(RAIL_Handle_t railHandle)
Place a temporary hold on this packet's data and information resources within RAIL.

uint16_t RAIL_PeekRxPacket(RAIL_Handle_t railHandle, RAIL_RxPacketHandle_t packetHandle, uint8_t *pDst, uint16_t len, uint16_t offset)
Copy 'len' bytes of packet data starting from 'offset' from the receive FIFO.

RAIL_Status_t RAIL_ReleaseRxPacket(RAIL_Handle_t railHandle, RAIL_RxPacketHandle_t packetHandle)
Release RAIL's internal resources for the packet.

int16_t RAIL_GetRssi(RAIL_Handle_t railHandle, bool wait)
Return the current raw RSSI.

int16_t RAIL_GetRssiAlt(RAIL_Handle_t railHandle, RAIL_Time_t waitTimeout)
Return the current raw RSSI within a definitive time period.

RAIL_Status_t RAIL_StartAverageRssi(RAIL_Handle_t railHandle, uint16_t channel, RAIL_Time_t averagingTimeUs, const RAIL_SchedulerInfo_t *schedulerInfo)
Start the RSSI averaging over a specified time in us.

bool RAIL_IsAverageRssiReady(RAIL_Handle_t railHandle)
Query whether the RSSI averaging is done.

int16_t RAIL_GetAverageRssi(RAIL_Handle_t railHandle)
Get the RSSI averaged over a specified time in us.

RAIL_Status_t RAIL_SetRssiOffset(RAIL_Handle_t railHandle, int8_t rssiOffset)
Set the RSSI offset.

int8_t RAIL_GetRssiOffset(RAIL_Handle_t railHandle)
Get the RSSI offset.

RAIL_Status_t RAIL_SetRssiDetectThreshold(RAIL_Handle_t railHandle, int8_t rssiThresholdDbm)
Set the RSSI detection threshold(in dBm) to trigger RAIL_EVENT_DETECT_RSSI_THRESHOLD.

int8_t RAIL_GetRssiDetectThreshold(RAIL_Handle_t railHandle)
Get the RSSI detection threshold(in dBm).

RAIL_Status_t RAIL_ConvertLqi(RAIL_Handle_t railHandle, RAIL_ConvertLqiCallback_t cb)
Set up a callback function capable of converting a RX packet's LQI value before being consumed by application code.

Macros

#define RAIL_RX_OPTIONS_NONE 0
A value representing no options enabled.

#define RAIL_RX_OPTIONS_DEFAULT RAIL_RX_OPTIONS_NONE
All options are disabled by default.

#define RAIL_RX_OPTION_STORE_CRC (1UL << RAIL_RX_OPTION_STORE_CRC_SHIFT)
An option to configure whether the CRC portion of the packet is included in the packet payload exposed to the app on packet reception.

#define RAIL_RX_OPTION_IGNORE_CRC_ERRORS (1UL << RAIL_RX_OPTION_IGNORE_CRC_ERRORS_SHIFT)
An option to configure whether CRC errors will be ignored.

#define RAIL_RX_OPTION_ENABLE_DUALSYNC (1UL << RAIL_RX_OPTION_ENABLE_DUALSYNC_SHIFT)
An option to control which sync words will be accepted.

#define RAIL_RX_OPTION_TRACK_ABORTED_FRAMES (1UL << RAIL_RX_OPTION_TRACK_ABORTED_FRAMES_SHIFT)
An option to configure whether frames which are aborted during reception should continue to be tracked.

#define RAIL_RX_OPTION_REMOVE_APPENDED_INFO (1UL << RAIL_RX_OPTION_REMOVE_APPENDED_INFO_SHIFT)
An option to suppress capturing the appended information after received frames.

#define RAIL_RX_OPTION_ANTENNA0 (1UL << RAIL_RX_OPTION_ANTENNA0_SHIFT)
An option to select the use of antenna 0 during receive (including Auto-ACK receive).

#define RAIL_RX_OPTION_ANTENNA1 (1UL << RAIL_RX_OPTION_ANTENNA1_SHIFT)
An option to select the use of antenna 1 during receive (including Auto-ACK receive).

#define RAIL_RX_OPTION_ANTENNA_AUTO (RAIL_RX_OPTION_ANTENNA0 | RAIL_RX_OPTION_ANTENNA1)
An option combination to automatically choose an antenna during receive (including Auto-ACK receive).

#define RAIL_RX_OPTION_DISABLE_FRAME_DETECTION (1UL << RAIL_RX_OPTION_DISABLE_FRAME_DETECTION_SHIFT)
An option to disable frame detection.

#define RAIL_RX_OPTION_CHANNEL_SWITCHING (1U << RAIL_RX_OPTION_CHANNEL_SWITCHING_SHIFT)
An option to enable IEEE 802.15.4 RX channel switching.

#define RAIL_RX_OPTION_FAST_RX2RX (1U << RAIL_RX_OPTION_FAST_RX2RX_SHIFT)
An option to enable fast RX2RX state transition.

#define RAIL_RX_OPTION_ENABLE_COLLISION_DETECTION (1U << RAIL_RX_OPTION_ENABLE_COLLISION_DETECTION_SHIFT)
An option to enable collision detection.

#define RAIL_RX_OPTIONS_ALL 0xFFFFFFFFUL
A value representing all possible options.

#define RAIL_RSSI_INVALID_DBM (-128)
The value returned by RAIL for an invalid RSSI, in dBm.

#define RAIL_RSSI_INVALID ((int16_t)(RAIL_RSSI_INVALID_DBM * 4))
The value returned by RAIL for an invalid RSSI: in quarter dBm.

#define RAIL_RSSI_LOWEST ((int16_t)(RAIL_RSSI_INVALID + 1))
The lowest RSSI value returned by RAIL: in quarter dBm.

#define RAIL_RSSI_OFFSET_MAX 35
Maximum absolute value for RSSI offset.

#define RAIL_GET_RSSI_WAIT_WITHOUT_TIMEOUT ((RAIL_Time_t)0xFFFFFFFFU)
A sentinel value to indicate waiting for a valid RSSI without a timeout.

#define RAIL_GET_RSSI_NO_WAIT ((RAIL_Time_t)0U)
A sentinel value to indicate no waiting for a valid RSSI.

#define RAIL_RX_PACKET_HANDLE_INVALID (NULL)
An invalid RX packet handle value.

#define RAIL_RX_PACKET_HANDLE_OLDEST ((RAIL_RxPacketHandle_t) 1)
A special RX packet handle to refer to the oldest unreleased packet.

#define RAIL_RX_PACKET_HANDLE_OLDEST_COMPLETE ((RAIL_RxPacketHandle_t) 2)
A special RX packet handle to refer to the oldest unreleased complete packet.

#define RAIL_RX_PACKET_HANDLE_NEWEST ((RAIL_RxPacketHandle_t) 3)
A special RX packet handle to refer to the newest unreleased packet when in callback context.

Enumeration Documentation

RAIL_RxOptions_t

> RAIL_RxOptions_t

Receive options, in reality a bitmask.

Enumerator	
RAIL_RX_OPTION_STORE_CRC_SHIFT	Shift position of RAIL_RX_OPTION_STORE_CRC bit.
RAIL_RX_OPTION_IGNORE_CRC_ERRORS_SHIFT	Shift position of RAIL_RX_OPTION_IGNORE_CRC_ERRORS bit.
RAIL_RX_OPTION_ENABLE_DUALSYNC_SHIFT	Shift position of RAIL_RX_OPTION_ENABLE_DUALSYNC bit.
RAIL_RX_OPTION_TRACK_ABORTED_FRAMES_SHIFT	Shift position of RAIL_RX_OPTION_TRACK_ABORTED_FRAMES bit.
RAIL_RX_OPTION_REMOVE_APPENDED_INFO_SHIFT	Shift position of RAIL_RX_OPTION_REMOVE_APPENDED_INFO bit.
RAIL_RX_OPTION_ANTENNA0_SHIFT	Shift position of RAIL_RX_OPTION_ANTENNA0 bit.
RAIL_RX_OPTION_ANTENNA1_SHIFT	Shift position of RAIL_RX_OPTION_ANTENNA1 bit.
RAIL_RX_OPTION_DISABLE_FRAME_DETECTION_SHIFT	Shift position of RAIL_RX_OPTION_DISABLE_FRAME_DETECTION bit.
RAIL_RX_OPTION_CHANNEL_SWITCHING_SHIFT	Shift position of RAIL_RX_OPTION_CHANNEL_SWITCHING bit.
RAIL_RX_OPTION_FAST_RX2RX_SHIFT	Shift position of RAIL_RX_OPTION_FAST_RX2RX bit.
RAIL_RX_OPTION_ENABLE_COLLISION_DETECTION_SHIFT	Shift position of RAIL_RX_OPTION_ENABLE_COLLISION_DETECTION bit.

Definition at line 3689 of file common/rail_types.h

RAIL_RxPacketStatus_t

> RAIL_RxPacketStatus_t

The packet status code associated with a packet received or currently being received.

Note

- RECEIVING implies some packet data may be available, but is untrustworthy (not CRC-verified) and might disappear if the packet is rolled back on error. No packet details are yet available.
- In RX RAIL_DataMethod_t::FIFO_MODE, ABORT statuses imply some packet data may be available, but it's incomplete and not trustworthy.

Enumerator	
RAIL_RX_PACKET_NONE	The radio is idle or searching for a packet.
RAIL_RX_PACKET_ABORT_FORMAT	The packet was aborted during filtering because of illegal frame length, CRC or block decoding errors, other RAIL built-in protocol-specific packet content errors, or by the application or multiprotocol scheduler idling the radio with RAIL_IDLE_ABORT or higher.
RAIL_RX_PACKET_ABORT_FILTERED	The packet failed address filtering.
RAIL_RX_PACKET_ABORT_ABORTED	The packet passed any filtering but was aborted by the application or multiprotocol scheduler idling the radio with RAIL_IDLE_ABORT or higher.
RAIL_RX_PACKET_ABORT_OVERFLOW	The packet overflowed the receive buffer.
RAIL_RX_PACKET_ABORT_CRC_ERROR	The packet passed any filtering but subsequently failed CRC check(s) block decoding, or illegal frame length, and was aborted.
RAIL_RX_PACKET_READY_CRC_ERROR	The packet passed any filtering but subsequently failed CRC check(s) with RAIL_RX_OPTION_IGNORE_CRC_ERRORS in effect.
RAIL_RX_PACKET_READY_SUCCESS	The packet was successfully received, passing CRC check(s).
RAIL_RX_PACKET_RECEIVING	A packet is being received and is not yet complete.

Definition at line `3977` of file `common/rail_types.h`

Typedef Documentation

RAIL_RxPacketHandle_t

```
RAIL_RxPacketHandle_t
```

A handle used to reference a packet during reception processing.

There are several sentinel handle values that pertain to certain circumstances: RAIL_RX_PACKET_HANDLE_INVALID, RAIL_RX_PACKET_HANDLE_OLDEST, RAIL_RX_PACKET_HANDLE_OLDEST_COMPLETE and RAIL_RX_PACKET_HANDLE_NEWEST.

Definition at line `4062` of file `common/rail_types.h`

RAIL_ConvertLqiCallback_t

```
RAIL_ConvertLqiCallback_t )(uint8_t lqi, int8_t rssi)
```

A pointer to a function called before LQI is copied into the RAIL_RxPacketDetails_t structure.

Parameters

[in]	lqi	The LQI value obtained by hardware and being readied for application consumption. This LQI value is in integral units ranging from 0 to 255.
[in]	rssi	The RSSI value corresponding to the packet from which the hardware LQI value was obtained. This RSSI value is in integral dBm units.

Returns

- uint8_t The converted LQI value that will be loaded into the RAIL_RxPacketDetails_t structure in preparation for application consumption. This value should likewise be in integral units ranging from 0 to 255.

Definition at line `4275` of file `common/rail_types.h`

Function Documentation

RAIL_ConfigRxOptions

```
RAIL_Status_t RAIL_ConfigRxOptions (RAIL_Handle_t railHandle, RAIL_RxOptions_t mask, RAIL_RxOptions_t options)
```

Configure receive options.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	mask	A bitmask containing which options should be modified.
[in]	options	A bitmask containing desired configuration settings. Bit positions for each option are found in the RAIL_RxOptions_t.

Returns

- Status code indicating success of the function call.

Configure the radio receive flow based on the list of available options. Only the options indicated by the mask parameter will be affected. Pass RAIL_RX_OPTIONS_ALL to set all parameters. The previous settings may affect the current frame if a packet is received during this configuration.

Note

- : On chips where RAIL_IEEE802154_SUPPORTS_RX_CHANNEL_SWITCHING is true, enabling RAIL_RX_OPTION_CHANNEL_SWITCHING without configuring RX channel switching, via RAIL_IEEE802154_ConfigRxChannelSwitching, will return RAIL_STATUS_INVALID_PARAMETER for only this option. Any other RX options (except antenna selection) would still take effect.

Definition at line `3588` of file `common/rail.h`

RAIL_IncludeFrameTypeLength

```
void RAIL_IncludeFrameTypeLength (RAIL_Handle_t railHandle)
```

Include the code necessary for frame type based length decoding.

Parameters

[in]	railHandle	A RAIL instance handle.

This function must be called before RAIL_ConfigChannels to allow configurations using a frame type based length setup. In RAIL 2.x, it is called by default in the RAILCb_ConfigFrameTypeLength API which can be overridden to save code space. In future versions, the user may be required to call this API explicitly.

Definition at line `3603` of file `common/rail.h`

RAILCb_ConfigFrameTypeLength

```
void RAILCb_ConfigFrameTypeLength (RAIL_Handle_t railHandle, const RAIL_FrameType_t *frameType)
```

Handle frame type length.

Parameters

[in]	railHandle	A RAIL instance handle.

[in]	frameType	A frame type configuration structure.

This function is implemented in the radio configuration. Currently, the frame type passed in only handles packet length decoding. If NULL is passed into this function, it clears any currently configured frame type settings. This will either be implemented as an empty function in the radio configuration if it is not needed, to assist in dead code elimination.

Definition at line `3618` of file `common/rail.h`

RAIL_StartRx

> RAIL_Status_t RAIL_StartRx (RAIL_Handle_t railHandle, uint16_t channel, const RAIL_SchedulerInfo_t *schedulerInfo)

Start the receiver on a specific channel.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	channel	The channel to listen on.
[in]	schedulerInfo	Information to allow the radio scheduler to place this receive appropriately. This is only used in multiprotocol version of RAIL and may be set to NULL in all other versions.

Returns

- Status code indicating success of the function call.

This is a non-blocking function. Whenever a packet is received, RAIL_Config_t::eventsCallback will fire with RAIL_EVENT_RX_PACKET_RECEIVED set. If you call this while not idle but with a different channel, any ongoing receive or transmit operation will be aborted.

Definition at line `3636` of file `common/rail.h`

RAIL_ScheduleRx

> RAIL_Status_t RAIL_ScheduleRx (RAIL_Handle_t railHandle, uint16_t channel, const RAIL_ScheduleRxConfig_t *cfg, const RAIL_SchedulerInfo_t *schedulerInfo)

Schedule a receive window for some future time.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	channel	A channel to listen on.
[in]	cfg	The configuration structure to define the receive window.
[in]	schedulerInfo	Information to allow the radio scheduler to place this receive appropriately. This is only used in multiprotocol version of RAIL and may be set to NULL in all other versions.

Returns

- Status code indicating success of the function call.

This API immediately changes the channel and schedules receive to start at the specified time and end at the given end time. If you do not specify an end time, you may call this API later with an end time as long as you set the start time to disabled. You can also terminate the receive operation immediately using the RAIL_Idle() function. Note that relative end times are always relative to the start unless no start time is specified. If changing channels, the channel is changed immediately and will abort any ongoing packet transmission or reception.

Returns an error if a CSMA or LBT transmit is still in progress.

In multiprotocol, ensure that you properly yield the radio after this call. See Yielding the Radio for more details.

Definition at line 3665 of file common/rail.h

RAIL_HoldRxPacket

```
RAIL_RxPacketHandle_t RAIL_HoldRxPacket (RAIL_Handle_t railHandle)
```

Place a temporary hold on this packet's data and information resources within RAIL.

Parameters

[in]	railHandle	A RAIL instance handle.

This function can only be called from within RAIL callback context. It can be used in any RX mode.

Normally, when RAIL issues its callback indicating a packet is ready or aborted, it expects the application's callback to retrieve and copy (or discard) the packet's information and data, and will free up its internal packet data after the callback returns. This function tells RAIL to hold onto those resources after the callback returns in case the application wants to defer processing the packet to a later time, e.g., outside of callback context.

Returns

- The packet handle for the packet associated with the callback, RAIL_RX_PACKET_HANDLE_INVALID if no such packet yet exists or railHandle is not active.

Note

- When using multiprotocol the receive FIFO is reset during protocol switches so any packets held with RAIL_HoldRxPacket() will be lost. It is best to avoid using this in DMP or to at least reset any internal held packet information when the RAIL_EVENT_CONFIG_UNSCHEDULED occurs.

Definition at line 4052 of file common/rail.h

RAIL_PeekRxPacket

```
uint16_t RAIL_PeekRxPacket (RAIL_Handle_t railHandle, RAIL_RxPacketHandle_t packetHandle, uint8_t *pDst, uint16_t len, uint16_t offset)
```

Copy 'len' bytes of packet data starting from 'offset' from the receive FIFO.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	packetHandle	A packet handle as returned from a previous RAIL_GetRxPacketInfo() or RAIL_HoldRxPacket() call, or sentinel values RAIL_RX_PACKET_HANDLE_OLDEST, RAIL_RX_PACKET_HANDLE_OLDEST_COMPLETE or RAIL_RX_PACKET_HANDLE_NEWEST.
[out]	pDst	A pointer to the location where the received bytes will be copied. If NULL, no copying occurs.
[in]	len	A number of packet data bytes to copy.
[in]	offset	A byte offset within remaining packet data from which to copy.

Those bytes remain valid for re-peeking.

Returns

- Number of packet bytes copied.

Note

Peek does not permit peeking beyond the requested packet's available packet data (though there is a small chance it might for a RAIL_RX_PACKET_HANDLE_NEWEST packet at the very end of still being received). Nor can one peek into already-consumed data read by RAIL_ReadRxFifo(). len and offset are relative to the remaining data available in the packet, if any was already consumed by RAIL_ReadRxFifo().

Definition at line 4078 of file common/rail.h

RAIL_ReleaseRxPacket

> RAIL_Status_t RAIL_ReleaseRxPacket (RAIL_Handle_t railHandle, RAIL_RxPacketHandle_t packetHandle)

Release RAIL's internal resources for the packet.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	packetHandle	A packet handle as returned from a previous RAIL_HoldRxPacket() call, or sentinel values RAIL_RX_PACKET_HANDLE_OLDEST, RAIL_RX_PACKET_HANDLE_OLDEST_COMPLETE or RAIL_RX_PACKET_HANDLE_NEWEST. The latter might be used within RAIL callback context to explicitly release the packet associated with the callback early, before it's released automatically by RAIL on callback return (unless explicitly held).

This function must be called for any packet previously held via RAIL_HoldRxPacket(). It may optionally be called within a callback context to release RAIL resources sooner than at callback completion time when not holding the packet. This function can be used in any RX mode.

Returns

- RAIL_STATUS_NO_ERROR if the held packet was released or an appropriate error code otherwise.

Definition at line 4105 of file common/rail.h

RAIL_GetRssi

> int16_t RAIL_GetRssi (RAIL_Handle_t railHandle, bool wait)

Return the current raw RSSI.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	wait	if false returns instant RSSI with no checks.

Returns

- RAIL_RSSI_INVALID if the receiver is disabled and an RSSI value can't be obtained. Otherwise, return the RSSI in quarter dBm, dbm*4.

Gets the current RSSI value. This value represents the current energy of the channel. It can change rapidly and will be low if no RF energy is in the current channel. The function from the value reported to dBm is an offset dependent on the PHY and the PCB layout. Characterize the RSSI received on your hardware and apply an offset in the application to account for board and PHY parameters. When 'wait' is false, the radio needs to be currently in RX and have been in there for a sufficient amount of time for a fresh RSSI value to be read and returned. Otherwise, the RSSI is considered stale and RAIL_RSSI_INVALID is returned instead. When 'wait' is true, if the radio is transitioning to or already in RX, this function will wait for a valid RSSI to be read and return it. Otherwise, if the radio is in or transitions to IDLE or TX, RAIL_RSSI_INVALID will be returned. On low datarate PHYs, this function can take a significantly longer time when wait is true.

In multiprotocol, this function returns RAIL_RSSI_INVALID immediately if railHandle is not the current active RAIL_Handle_t. Additionally, 'wait' should never be set 'true' in multiprotocol as the wait time is not consistent, so scheduling a scheduler

slot cannot be done accurately. Rather if waiting for a valid RSSI is desired, use RAIL_GetRssiAlt instead to apply a bounded time period.

Note

- If RX Antenna Diversity is enabled via RAIL_ConfigRxOptions(), pass true for the wait parameter otherwise it's very likely RAIL_RSSI_INVALID will be returned.
- If RX channel hopping is turned on, do not use this API. Instead, see RAIL_GetChannelHoppingRssi().
- When 'wait' is false, this API is equivalent to RAIL_GetRssiAlt with 'waitTimeout' set to RAIL_GET_RSSI_NO_WAIT. When 'wait' is true, this API is equivalent to RAIL_GetRssiAlt with 'waitTimeout' set to RAIL_GET_RSSI_WAIT_WITHOUT_TIMEOUT. Consider using RAIL_GetRssiAlt if a bounded maximum wait timeout is desired.

Definition at line 4151 of file common/rail.h

RAIL_GetRssiAlt

> int16_t RAIL_GetRssiAlt (RAIL_Handle_t railHandle, RAIL_Time_t waitTimeout)

Return the current raw RSSI within a definitive time period.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	waitTimeout	Sets the maximum time to wait for a valid RSSI. If equal to RAIL_GET_RSSI_NO_WAIT returns instant RSSI with no checks. If equal to RAIL_GET_RSSI_WAIT_WITHOUT_TIMEOUT waits for a valid RSSI with no maximum timeout.

Returns

- RAIL_RSSI_INVALID if the receiver is disabled and an RSSI value can't be obtained. Otherwise, return the RSSI in quarter dBm, dbm*4.

Gets the current RSSI value. This value represents the current energy of the channel. It can change rapidly, and will be low if no RF energy is in the current channel. The function from the value reported to dBm is an offset dependent on the PHY and the PCB layout. Characterize the RSSI received on your hardware and apply an offset in the application to account for board and PHY parameters. If a value of RAIL_GET_RSSI_NO_WAIT is given for waitTimeout, the radio needs to be currently in RX and have been in there for a sufficient amount of time for a fresh RSSI value to be read and returned. Otherwise the RSSI is considered stale and RAIL_RSSI_INVALID is returned instead. For non-zero values of waitTimeout, if the radio is transitioning to or already in RX, this function will wait a maximum time equal to waitTimeout (or indefinitely if waitTimeout is set to RAIL_GET_RSSI_WAIT_WITHOUT_TIMEOUT) for a valid RSSI to be read and return it. Otherwise, if the waitTimeout is reached, or the radio is in or transitions to IDLE or TX, RAIL_RSSI_INVALID will be returned. On low datarate PHYs, this function can take a significantly longer time when waitTimeout is non-zero.

In multiprotocol, this function returns RAIL_RSSI_INVALID immediately if railHandle is not the current active RAIL_Handle_t. Additionally, 'waitTimeout' should never be set to a value other than RAIL_GET_RSSI_NO_WAIT in multiprotocol as the integration between this feature and the radio scheduler has not been implemented.

Note

- If RX Antenna Diversity is enabled via RAIL_ConfigRxOptions(), pass true for the wait parameter otherwise it's very likely RAIL_RSSI_INVALID will be returned.
- If RX Antenna Diversity is enabled via RAIL_ConfigRxOptions(), the RSSI value returned could come from either antenna and vary between antennas.
- If RX channel hopping is turned on, do not use this API. Instead, see RAIL_GetChannelHoppingRssi().

Definition at line 4197 of file common/rail.h

RAIL_StartAverageRssi

RAIL_Status_t RAIL_StartAverageRssi (RAIL_Handle_t railHandle, uint16_t channel, RAIL_Time_t averagingTimeUs, const RAIL_SchedulerInfo_t *schedulerInfo)

Start the RSSI averaging over a specified time in us.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	channel	The physical channel to set.
[in]	averagingTimeUs	Averaging time in microseconds.
[in]	schedulerInfo	Information to allow the radio scheduler to place this operation appropriately. This is only used in multiprotocol version of RAIL and may be set to NULL in all other versions.

Returns

- Status code indicating success of the function call.

Starts a non-blocking hardware-based RSSI averaging mechanism. Only a single instance of RSSI averaging can be run at any time and the radio must be idle to start.

In multiprotocol, this is a scheduled event. It will start when railHandle becomes active. railHandle needs to stay active until the averaging completes. If the averaging is interrupted, calls to RAIL_GetAverageRssi will return RAIL_RSSI_INVALID.

Also in multiprotocol, the user is required to call RAIL_YieldRadio after this event completes (i.e., when RAIL_EVENT_RSSI_AVERAGE_DONE occurs).

Note

- If the radio is idled while RSSI averaging is still in effect, a RAIL_EVENT_RSSI_AVERAGE_DONE event may not occur and RAIL_IsAverageRssiReady may never return true.

Definition at line 4227 of file common/rail.h

RAIL_IsAverageRssiReady

bool RAIL_IsAverageRssiReady (RAIL_Handle_t railHandle)

Query whether the RSSI averaging is done.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Returns true if done and false otherwise.

This function can be used to poll for completion of the RSSI averaging to avoid relying on an interrupt-based callback.

Note

- If the radio is idled while RSSI averaging is still in effect, this function may never return true.

Definition at line 4244 of file common/rail.h

RAIL_GetAverageRssi

int16_t RAIL_GetAverageRssi (RAIL_Handle_t railHandle)

Get the RSSI averaged over a specified time in us.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Return RAIL_RSSI_INVALID if the receiver is disabled an RSSI value can't be obtained. Otherwise, return the RSSI in quarter dBm,dbm*4.

Gets the hardware RSSI average after issuing RAIL_StartAverageRssi. Use after RAIL_StartAverageRssi.

Definition at line `4257` of file `common/rail.h`

RAIL_SetRssiOffset

```
RAIL_Status_t RAIL_SetRssiOffset (RAIL_Handle_t railHandle, int8_t rssiOffset)
```

Set the RSSI offset.

Parameters

[in]	railHandle	a RAIL instance handle.
[in]	rssiOffset	desired offset to be added to the RSSI measurements.

Returns

- Status code indicating success of the function call. RAIL_STATUS_INVALID_CALL if called with chip-specific handle, such as RAIL_EFR32_HANDLE, after RAIL initialization. RAIL_STATUS_INVALID_PARAMETER if the RSSI offset is deemed large enough to cause the RSSI readings to underflow or overflow.

Adds an offset to the RSSI in dBm. This offset affects all functionality that depends on RSSI values, such as CCA functions. Do not modify the offset dynamically during packet reception. This function can only be called while the radio is off, or in the case of multiprotocol, on an inactive protocol.

Note

- : If RAIL has not been initialized, a chip-specific handle, such as RAIL_EFR32_HANDLE, can be used to set a chip level RSSI offset.
- : Setting a large rssiOffset may still cause the RSSI readings to underflow. If that happens, the RSSI value returned by RAIL_GetRssi, RAIL_GetAverageRssi, RAIL_GetChannelHoppingRssi etc. will be RAIL_RSSI_LOWEST
- : During RX Channel Hopping this API will not update the RSSI offset immediately if channel hopping has already been configured. A subsequent call to RAIL_ZWAVE_ConfigRxChannelHopping or RAIL_ConfigRxChannelHopping is required for the new RSSI offset to take effect.

Definition at line `4290` of file `common/rail.h`

RAIL_GetRssiOffset

```
int8_t RAIL_GetRssiOffset (RAIL_Handle_t railHandle)
```

Get the RSSI offset.

Parameters

[in]	railHandle	a RAIL instance handle.

Returns

rssiOffset in dBm corresponding to the current handle.

Note

- : A chip-specific handle, such as RAIL_EFR32_HANDLE, can be used to get the chip level RSSI offset otherwise this will return the RSSI offset value associated with the RAIL instance handle, exclusive of any chip level RSSI offset correction, if any.

Definition at line 4303 of file common/rail.h

RAIL_SetRssiDetectThreshold

```
RAIL_Status_t RAIL_SetRssiDetectThreshold (RAIL_Handle_t railHandle, int8_t rssiThresholdDbm)
```

Set the RSSI detection threshold(in dBm) to trigger RAIL_EVENT_DETECT_RSSI_THRESHOLD.

Parameters

[in]	railHandle	a RAIL instance handle.
[in]	rssiThresholdDbm	desired RSSI threshold(in dBm) over which the event RAIL_EVENT_DETECT_RSSI_THRESHOLD is triggered.

Returns

- Status code indicating success of the function call. Returns RAIL_STATUS_INVALID_STATE in multiprotocol, if the requested RAIL_Handle_t is not active. Returns RAIL_STATUS_INVALID_CALL if called on parts on which this function is not supported.

When in receive, RSSI is sampled and if it exceeds the threshold, RAIL_EVENT_DETECT_RSSI_THRESHOLD is triggered.

Note

- : If the radio is idled or this function is called with rssiThresholdDbm as RAIL_RSSI_INVALID_DBM while RSSI detect is still in effect, a RAIL_EVENT_DETECT_RSSI_THRESHOLD may not occur and the detection is disabled. If the RSSI is already above threshold when this function is called then RAIL_EVENT_DETECT_RSSI_THRESHOLD will occur. Once the RSSI goes over the configured threshold and RAIL_EVENT_DETECT_RSSI_THRESHOLD occurs, this function needs to be called again to reactivate the RSSI threshold detection. This function is only available on series-2 Sub-GHz parts EFR32XG23 and EFR32XG25.

Definition at line 4332 of file common/rail.h

RAIL_GetRssiDetectThreshold

```
int8_t RAIL_GetRssiDetectThreshold (RAIL_Handle_t railHandle)
```

Get the RSSI detection threshold(in dBm).

Parameters

[in]	railHandle	a RAIL instance handle.

Returns

- rssiThreshold (in dBm) corresponding to the current handle.

Note

- : The function returns RAIL_RSSI_INVALID_DBM when RAIL_SetRssiDetectThreshold is not supported or disabled. In multiprotocol, the function returns RAIL_RSSI_INVALID_DBM if railHandle is not active. This function is only available on series-2 Sub-GHz parts EFR32XG23 and EFR32XG25.

Definition at line 4347 of file common/rail.h

RAIL_ConvertLqi

```
RAIL_Status_t RAIL_ConvertLqi (RAIL_Handle_t railHandle, RAIL_ConvertLqiCallback_t cb)
```

Set up a callback function capable of converting a RX packet's LQI value before being consumed by application code.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	cb	A callback of type RAIL_ConvertLqiCallback_t that is called before the RX packet LQI value is loaded into the RAIL_RxPacketDetails_t structure for application consumption.

Returns

- Status code indicating success of the function call.

Definition at line 4359 of file common/rail.h

Macro Definition Documentation

RAIL_RX_OPTIONS_NONE

```
#define RAIL_RX_OPTIONS_NONE
```

Value:

```
0
```

A value representing no options enabled.

Definition at line 3721 of file common/rail_types.h

RAIL_RX_OPTIONS_DEFAULT

```
#define RAIL_RX_OPTIONS_DEFAULT
```

Value:

```
RAIL_RX_OPTIONS_NONE
```

All options are disabled by default.

Definition at line 3723 of file common/rail_types.h

RAIL_RX_OPTION_STORE_CRC

```
#define RAIL_RX_OPTION_STORE_CRC
```

Value:

```
(1UL << RAIL_RX_OPTION_STORE_CRC_SHIFT)
```

An option to configure whether the CRC portion of the packet is included in the packet payload exposed to the app on packet reception.

Defaults to false.

Definition at line 3730 of file common/rail_types.h

RAIL_RX_OPTION_IGNORE_CRC_ERRORS

```
#define RAIL_RX_OPTION_IGNORE_CRC_ERRORS
```

Value:

```
(1UL << RAIL_RX_OPTION_IGNORE_CRC_ERRORS_SHIFT)
```

An option to configure whether CRC errors will be ignored.

If this is set, RX will still be successful, even if the CRC does not pass the check. Defaults to false.

Note

- An expected ACK that fails CRC with this option set will still be considered the expected ACK, terminating the RAIL_AutoAckConfig_t::ackTimeout period.

Definition at line 3740 of file common/rail_types.h

RAIL_RX_OPTION_ENABLE_DUALSYNC

```
#define RAIL_RX_OPTION_ENABLE_DUALSYNC
```

Value:

```
(1UL << RAIL_RX_OPTION_ENABLE_DUALSYNC_SHIFT)
```

An option to control which sync words will be accepted.

Setting it to 0 (default) will cause the receiver to listen for SYNC1 only. Setting it to 1 causes the receiver to listen for either SYNC1 or SYNC2. RX appended info will contain which sync word was detected. Note, this only affects which sync word(s) are received, but not what each of the sync words actually are. This feature may not be available on some combinations of chips, PHYs, and protocols. Use the compile time symbol RAIL_SUPPORTS_DUAL_SYNC_WORDS or the runtime call RAIL_SupportsDualSyncWords() to check whether the platform supports this feature. Also, DUALSYNC may be incompatible with certain radio configurations. In these cases, setting this bit will be ignored. See the data sheet or support team for more details.

Definition at line 3755 of file common/rail_types.h

RAIL_RX_OPTION_TRACK_ABORTED_FRAMES

```
#define RAIL_RX_OPTION_TRACK_ABORTED_FRAMES
```

Value:

```
(1UL << RAIL_RX_OPTION_TRACK_ABORTED_FRAMES_SHIFT)
```

An option to configure whether frames which are aborted during reception should continue to be tracked.

Setting this option allows viewing Packet Trace information for frames which get discarded. Defaults to false.

This option is ignored when doing a RAIL_IDLE_FORCE_SHUTDOWN or RAIL_IDLE_FORCE_SHUTDOWN_CLEAR_FLAGS.

Note

- This option should not be used with coded PHYs since packet data received after the abort will not be decoded properly.

Definition at line 3768 of file common/rail_types.h

RAIL_RX_OPTION_REMOVE_APPENDED_INFO

```
#define RAIL_RX_OPTION_REMOVE_APPENDED_INFO
```

Value:

```
(1UL << RAIL_RX_OPTION_REMOVE_APPENDED_INFO_SHIFT)
```

An option to suppress capturing the appended information after received frames.

Defaults to false. When suppressed, certain RAIL_RxPacketDetails_t details will not be available for received packets whose RAIL_RxPacketStatus_t is among the RAIL_RX_PACKET_READY_ set.

Warnings

- This option should be changed only when the radio is idle and the receive FIFO is empty or has been reset, otherwise RAIL_GetRxPacketInfo() and RAIL_GetRxPacketDetails() may think appended info is packet data or vice-versa.

Definition at line 3782 of file common/rail_types.h

RAIL_RX_OPTION_ANTENNA0

```
#define RAIL_RX_OPTION_ANTENNA0
```

Value:

```
(1UL << RAIL_RX_OPTION_ANTENNA0_SHIFT)
```

An option to select the use of antenna 0 during receive (including Auto-ACK receive).

If no antenna option is selected, the packet will be received on the last antenna used for receive or transmit. Defaults to false. This option is only valid on platforms that support Antenna Control and have been configured via RAIL_ConfigAntenna().

Definition at line 3791 of file common/rail_types.h

RAIL_RX_OPTION_ANTENNA1

```
#define RAIL_RX_OPTION_ANTENNA1
```

Value:

```
(1UL << RAIL_RX_OPTION_ANTENNA1_SHIFT)
```

An option to select the use of antenna 1 during receive (including Auto-ACK receive).

If no antenna option is selected, the packet will be received on the last antenna used for receive or transmit. Defaults to false. This option is only valid on platforms that support Antenna Control and have been configured via

RAIL_ConfigAntenna().

Definition at line 3800 of file common/rail_types.h

RAIL_RX_OPTION_ANTENNA_AUTO

```
#define RAIL_RX_OPTION_ANTENNA_AUTO
```

Value:

```
(RAIL_RX_OPTION_ANTENNA0 | RAIL_RX_OPTION_ANTENNA1)
```

An option combination to automatically choose an antenna during receive (including Auto-ACK receive).

If both antenna 0 and antenna 1 options are set, the radio will dynamically switch between antennas during packet detection and choose the best one for completing the reception. This option is only valid when the antenna diversity field is properly configured via Simplicity Studio. This option is only valid on platforms that support Antenna Control and have been configured via RAIL_ConfigAntenna().

Definition at line 3812 of file common/rail_types.h

RAIL_RX_OPTION_DISABLE_FRAME_DETECTION

```
#define RAIL_RX_OPTION_DISABLE_FRAME_DETECTION
```

Value:

```
(1UL << RAIL_RX_OPTION_DISABLE_FRAME_DETECTION_SHIFT)
```

An option to disable frame detection.

This can be useful for doing energy detection without risking packet reception. Enabling this will abort any frame currently being received in addition to preventing further frames from being received. Defaults to false.

Definition at line 3820 of file common/rail_types.h

RAIL_RX_OPTION_CHANNEL_SWITCHING

```
#define RAIL_RX_OPTION_CHANNEL_SWITCHING
```

Value:

```
(1U << RAIL_RX_OPTION_CHANNEL_SWITCHING_SHIFT)
```

An option to enable IEEE 802.15.4 RX channel switching.

See RAIL_IEEE802154_ConfigRxChannelSwitching() for more information. Defaults to false.

Note

- This option is only supported on specific chips where RAIL_IEEE802154_SUPPORTS_RX_CHANNEL_SWITCHING is true.
- This option overrides RAIL_RX_OPTION_ANTENNA0, RAIL_RX_OPTION_ANTENNA1 and RAIL_RX_OPTION_ANTENNA_AUTO antenna selection options.

Definition at line 3854 of file common/rail_types.h

RAIL_RX_OPTION_FAST_RX2RX

```
#define RAIL_RX_OPTION_FAST_RX2RX
```

Value:

```
(1U << RAIL_RX_OPTION_FAST_RX2RX_SHIFT)
```

An option to enable fast RX2RX state transition.

Once enabled, the sequencer will send the radio to RXSEARCH and get ready to receive the next packet while still processing the previous one. This will reduce RX to RX state transition time but risks impacting receive capability.

Note

- This option is only supported on specific chips where RAIL_SUPPORTS_FAST_RX2RX is true.

Definition at line 3866 of file common/rail_types.h

RAIL_RX_OPTION_ENABLE_COLLISION_DETECTION

```
#define RAIL_RX_OPTION_ENABLE_COLLISION_DETECTION
```

Value:

```
(1U << RAIL_RX_OPTION_ENABLE_COLLISION_DETECTION_SHIFT)
```

An option to enable collision detection.

Once enabled, when a collision with a strong enough packet is detected, the demod will stop the current packet decoding and try to detect the preamble of the incoming packet.

Note

- This option is only supported on specific chips where RAIL_SUPPORTS_COLLISION_DETECTION is true.

Definition at line 3878 of file common/rail_types.h

RAIL_RX_OPTIONS_ALL

```
#define RAIL_RX_OPTIONS_ALL
```

Value:

```
0xFFFFFFFFUL
```

A value representing all possible options.

Definition at line 3881 of file common/rail_types.h

RAIL_RSSI_INVALID_DBM

```
#define RAIL_RSSI_INVALID_DBM
```

Value:

```
(-128)
```

The value returned by RAIL for an invalid RSSI, in dBm.

Definition at line `3884` of file `common/rail_types.h`

RAIL_RSSI_INVALID

```
#define RAIL_RSSI_INVALID
```

Value:

```
((int16_t)(RAIL_RSSI_INVALID_DBM * 4))
```

The value returned by RAIL for an invalid RSSI: in quarter dBm.

Definition at line `3886` of file `common/rail_types.h`

RAIL_RSSI_LOWEST

```
#define RAIL_RSSI_LOWEST
```

Value:

```
((int16_t)(RAIL_RSSI_INVALID + 1))
```

The lowest RSSI value returned by RAIL: in quarter dBm.

Definition at line `3888` of file `common/rail_types.h`

RAIL_RSSI_OFFSET_MAX

```
#define RAIL_RSSI_OFFSET_MAX
```

Value:

```
35
```

Maximum absolute value for RSSI offset.

Definition at line `3891` of file `common/rail_types.h`

RAIL_GET_RSSI_WAIT_WITHOUT_TIMEOUT

```
#define RAIL_GET_RSSI_WAIT_WITHOUT_TIMEOUT
```

Value:

```
((RAIL_Time_t)0xFFFFFFFFU)
```

A sentinel value to indicate waiting for a valid RSSI without a timeout.

Definition at line 3894 of file common/rail_types.h

RAIL_GET_RSSI_NO_WAIT

```
#define RAIL_GET_RSSI_NO_WAIT
```

Value:

```
((RAIL_Time_t)0U)
```

A sentinel value to indicate no waiting for a valid RSSI.

Definition at line 3896 of file common/rail_types.h

RAIL_RX_PACKET_HANDLE_INVALID

```
#define RAIL_RX_PACKET_HANDLE_INVALID
```

Value:

```
(NULL)
```

An invalid RX packet handle value.

Definition at line 4065 of file common/rail_types.h

RAIL_RX_PACKET_HANDLE_OLDEST

```
#define RAIL_RX_PACKET_HANDLE_OLDEST
```

Value:

```
((RAIL_RxPacketHandle_t) 1)
```

A special RX packet handle to refer to the oldest unreleased packet.

This includes the newest unread packet which is possibly incomplete or not yet started. This handle is used implicitly by RAIL_ReadRxFifo().

Definition at line 4072 of file common/rail_types.h

RAIL_RX_PACKET_HANDLE_OLDEST_COMPLETE

```
#define RAIL_RX_PACKET_HANDLE_OLDEST_COMPLETE
```

Value:

```
((RAIL_RxPacketHandle_t) 2)
```

A special RX packet handle to refer to the oldest unreleased complete packet.

This never includes incomplete or unstarted packets. (Using RAIL_RX_PACKET_HANDLE_OLDEST is inappropriate for this purpose because it can refer to an unstarted, incomplete, or unheld packet which are inappropriate to be consumed by the application.)

Definition at line 4080 of file common/rail_types.h

RAIL_RX_PACKET_HANDLE_NEWEST

```
#define RAIL_RX_PACKET_HANDLE_NEWEST
```

Value:

```
((RAIL_RxPacketHandle_t) 3)
```

A special RX packet handle to refer to the newest unreleased packet when in callback context.

For a callback involving a completed receive event, this refers to the packet just completed. For other callback events, this refers to the next packet to be completed, which might be in-progress or might not have even started yet.

Definition at line 4089 of file common/rail_types.h

RAIL_ScheduleRxConfig_t

Configures the scheduled RX algorithm.

Defines the start and end times of the receive window created for a scheduled receive. If either start or end times are disabled, they will be ignored.

Public Attributes

RAIL_Time_t	start The time to start receive.
RAIL_TimeMode_t	startMode How to interpret the time value specified in the start parameter.
RAIL_Time_t	end The time to end receive.
RAIL_TimeMode_t	endMode How to interpret the time value specified in the end parameter.
uint8_t	rxTransitionEndSchedule While in scheduled RX, you can still control the radio state via state transitions.
uint8_t	hardWindowEnd This setting tells RAIL what to do with a packet being received when the window end event occurs.

Public Attribute Documentation

start

> RAIL_Time_t RAIL_ScheduleRxConfig_t::start

The time to start receive.

See startMode for more information about the types of start times that you can specify.

Definition at line 3911 of file common/rail_types.h

startMode

> RAIL_TimeMode_t RAIL_ScheduleRxConfig_t::startMode

How to interpret the time value specified in the start parameter.

See the RAIL_TimeMode_t documentation for more information. Use RAIL_TIME_ABSOLUTE for absolute times, RAIL_TIME_DELAY for times relative to the current time and RAIL_TIME_DISABLED to ignore the start time.

Definition at line 3919 of file common/rail_types.h

end

RAIL_Time_t RAIL_ScheduleRxConfig_t::end

The time to end receive.

See endMode for more information about the types of end times you can specify.

Definition at line 3924 of file common/rail_types.h

endMode

RAIL_TimeMode_t RAIL_ScheduleRxConfig_t::endMode

How to interpret the time value specified in the end parameter.

See the RAIL_TimeMode_t documentation for more information. Note that, in this API, if you specify a RAIL_TIME_DELAY, it is relative to the start time if given and relative to now if none is specified. Also, using RAIL_TIME_DISABLED means that this window will not end unless you explicitly call RAIL_Idle() or add an end event through a future update to this configuration.

Definition at line 3934 of file common/rail_types.h

rxTransitionEndSchedule

uint8_t RAIL_ScheduleRxConfig_t::rxTransitionEndSchedule

While in scheduled RX, you can still control the radio state via state transitions.

This option configures whether a transition to RX goes back to scheduled RX or to the normal RX state. Once in the normal RX state, you will effectively end the scheduled RX window and can continue to receive indefinitely depending on the state transitions. Set to 1 to transition to normal RX and 0 to stay in the scheduled RX.

This setting also influences the posting of RAIL_EVENT_RX_SCHEDULED_RX_END when the scheduled Rx window is implicitly ended by a packet receive (any of the RAIL_EVENTS_RX_COMPLETION events). See that event for details.

Note

- An Rx transition to Idle state will always terminate the scheduled Rx window, regardless of this setting. This can be used to ensure Scheduled RX terminates on the first packet received (or first successful packet if the RX error transition is to Rx while the Rx success transition is to Idle).

Definition at line 3954 of file common/rail_types.h

hardWindowEnd

uint8_t RAIL_ScheduleRxConfig_t::hardWindowEnd

This setting tells RAIL what to do with a packet being received when the window end event occurs.

If set to 0, such a packet will be allowed to complete. Any other setting will cause that packet to be aborted. In either situation, any posting of RAIL_EVENT_RX_SCHEDULED_RX_END is deferred briefly to when the packet's corresponding RAIL_EVENTS_RX_COMPLETION occurs.

Definition at line 3963 of file common/rail_types.h

RAIL_RxPacketInfo_t

Basic information about a packet being received or already completed and awaiting processing, including memory pointers to its data in the circular receive FIFO buffer.

This packet information refers to remaining packet data that has not already been consumed by RAIL_ReadRxFifo(). **Note**

- Because the receive FIFO buffer is circular, a packet might start near the end of the buffer and wrap around to the beginning of the buffer to finish, hence the distinction between the first and last portions. Packets that fit without wrapping only have a first portion (firstPortionBytes == packetBytes and lastPortionData will be NULL).

Public Attributes

RAIL_RxPacketStatus_t	packetStatus The packet status of this packet.
uint16_t	packetBytes The number of packet data bytes available to read in this packet.
uint16_t	firstPortionBytes The number of bytes in the first portion.
uint8_t *	firstPortionData The pointer to the first portion of packet data containing firstPortionBytes number of bytes.
uint8_t *	lastPortionData The pointer to the last portion of a packet, if any; NULL otherwise.
RAIL_AddrFilterMask_t	filterMask A bitmask representing which address filter(s) this packet has passed.

Public Attribute Documentation

packetStatus

```
RAIL_RxPacketStatus_t RAIL_RxPacketInfo_t::packetStatus
```

The packet status of this packet.

Definition at line `4106` of file `common/rail_types.h`

packetBytes

```
uint16_t RAIL_RxPacketInfo_t::packetBytes
```

The number of packet data bytes available to read in this packet.

Definition at line `4107` of file `common/rail_types.h`

firstPortionBytes

> uint16_t RAIL_RxPacketInfo_t::firstPortionBytes

The number of bytes in the first portion.

Definition at line 4109 of file common/rail_types.h

firstPortionData

> uint8_t* RAIL_RxPacketInfo_t::firstPortionData

The pointer to the first portion of packet data containing firstPortionBytes number of bytes.

Definition at line 4110 of file common/rail_types.h

lastPortionData

> uint8_t* RAIL_RxPacketInfo_t::lastPortionData

The pointer to the last portion of a packet, if any; NULL otherwise.

The number of bytes in this portion is packetBytes - firstPortionBytes.

Definition at line 4113 of file common/rail_types.h

filterMask

> RAIL_AddrFilterMask_t RAIL_RxPacketInfo_t::filterMask

A bitmask representing which address filter(s) this packet has passed.

Will be 0 when not filtering or if packet info is retrieved before filtering has completed. It's undefined on platforms lacking RAIL_SUPPORTS_ADDR_FILTER_MASK

Definition at line 4117 of file common/rail_types.h

RAIL_RxPacketDetails_t

Received packet details obtained via RAIL_GetRxPacketDetails() or RAIL_GetRxPacketDetailsAlt().

Note

- Certain details are always available, while others are only available if the RAIL_RxOptions_tRAIL_RX_OPTION_REMOVE_APPENDED_INFO option is not in effect and the received packet's RAIL_RxPacketStatus_t is among the RAIL_RX_PACKET_READY_ set. Each detail's description indicates its availability.

Public Attributes

RAIL_PacketTimeStamp_t	timeReceived The timestamp of the received packet in the RAIL timebase.
bool	crcPassed Indicates whether the CRC passed or failed for the received packet.
bool	isAck Indicate whether the received packet was the expected ACK.
int8_t	rssi RSSI of the received packet in integer dBm.
uint8_t	lqi The link quality indicator of the received packet.
uint8_t	syncWordId For radios and PHY configurations that support multiple sync words, this number is the ID of the sync word that was used for this packet.
uint8_t	subPhyId In configurations where the radio has the option of receiving a given packet in multiple ways, indicates which of the sub-PHY options was used to receive the packet.
uint8_t	antennaId For Antenna Control configurations where the device has multiple antennas, this indicates which antenna received the packet.
uint8_t	channelHoppingChannelIndex When channel hopping is enabled, this field will contain the index of the channel in RAIL_RxChannelHoppingConfig_t::entries on which this packet was received, or a sentinel value.
uint16_t	channel The channel on which the packet was received.

Public Attribute Documentation

timeReceived

```
RAIL_PacketTimeStamp_t RAIL_RxPacketDetails_t::timeReceived
```

The timestamp of the received packet in the RAIL timebase.

When not available it will be RAIL_PACKET_TIME_INVALID.

Definition at line 4144 of file common/rail_types.h

crcPassed

```
bool RAIL_RxPacketDetails_t::crcPassed
```

Indicates whether the CRC passed or failed for the received packet.

It is true for RAIL_RX_PACKET_READY_SUCCESS packets and false for all others.

It is always available.

Definition at line 4152 of file common/rail_types.h

isAck

```
bool RAIL_RxPacketDetails_t::isAck
```

Indicate whether the received packet was the expected ACK.

It is true for the expected ACK and false otherwise.

It is always available.

An expected ACK is defined as a protocol-correct ACK packet successfully-received (RAIL_RX_PACKET_READY_SUCCESS or RAIL_RX_PACKET_READY_CRC_ERROR) and whose sync word was detected within the RAIL_AutoAckConfig_t::ackTimeout period following a transmit which specified RAIL_TX_OPTION_WAIT_FOR_ACK, requested an ACK, and auto-ACK is enabled. When true, the ackTimeout period was terminated so no RAIL_EVENT_RX_ACK_TIMEOUT will be subsequently posted for the transmit.

A "protocol-correct ACK" applies to the 802.15.4 or Z-Wave protocols for which RAIL can discern the frame type and match the ACK's sequence number with that of the transmitted frame. For other protocols, the first packet successfully-received whose sync word was detected within the ackTimeout period is considered the expected ACK; upper layers are responsible for confirming this.

Definition at line 4177 of file common/rail_types.h

rssi

```
int8_t RAIL_RxPacketDetails_t::rssi
```

RSSI of the received packet in integer dBm.

This RSSI measurement is started as soon as the sync word is detected. The duration of the measurement is PHY-specific.

When not available it will be RAIL_RSSI_INVALID_DBM.

Definition at line 4185 of file common/rail_types.h

lqi

```
uint8_t RAIL_RxPacketDetails_t::lqi
```

The link quality indicator of the received packet.

A zero would indicate a very low quality packet while a 255 would indicate a very high quality packet.

When not available it will be 0.

Definition at line 4193 of file common/rail_types.h

syncWordId

```
uint8_t RAIL_RxPacketDetails_t::syncWordId
```

For radios and PHY configurations that support multiple sync words, this number is the ID of the sync word that was used for this packet.

It is always available.

Definition at line 4200 of file common/rail_types.h

subPhyId

```
uint8_t RAIL_RxPacketDetails_t::subPhyId
```

In configurations where the radio has the option of receiving a given packet in multiple ways, indicates which of the sub-PHY options was used to receive the packet.

Most radio configurations do not have this ability and the subPhyId is set to 0.

Currently, this field is used by the BLE Coded PHY, the BLE Simulscan PHY and the SUN OFDM PHYs. In BLE cases, a value of 0 marks a 500 kbps packet, a value of 1 marks a 125 kbps packet, and a value of 2 marks a 1 Mbps packet. Also, see RAIL_BLE_ConfigPhyCoded and RAIL_BLE_ConfigPhySimulscan.

In SUN OFDM cases, the value corresponds to the numerical value of the Modulation and Coding Scheme (MCS) level of the last received packet. The packet bitrate depends on the MCS value, as well as the OFDM option. Packets bitrates for SUN OFDM PHYs can be found in 802.15.4-2020 specification, chapter 20.3, table 20-10. Ex: Packet bitrate for OFDM option 1 MCS0 is 100kb/s and 2400kb/s for MCS6.

In WMBUS cases, when using PHY_wMbus_ModeTC_M2O_100k_frameA with simultaneous RX of T and C modes enabled (RAIL_WMBUS_Config()), the value corresponds to RAIL_WMBUS_Phy_t.

It is always available.

Definition at line 4226 of file common/rail_types.h

antennaId

```
uint8_t RAIL_RxPacketDetails_t::antennaId
```

For Antenna Control configurations where the device has multiple antennas, this indicates which antenna received the packet.

When there is only one antenna, this will be set to the default of 0.

It is always available.

Definition at line 4234 of file common/rail_types.h

channelHoppingChannelIndex

```
uint8_t RAIL_RxPacketDetails_t::channelHoppingChannelIndex
```

When channel hopping is enabled, this field will contain the index of the channel in RAIL_RxChannelHoppingConfig_t::entries on which this packet was received, or a sentinel value.

On EFR32XG1 parts, on which channel hopping is not supported, this value is still part of the structure, but will be a meaningless value.

It is always available.

Definition at line 4244 of file common/rail_types.h

channel

```
uint16_t RAIL_RxPacketDetails_t::channel
```

The channel on which the packet was received.

It is always available.

Note

- It is best to fully process (empty or clear) the receive FIFO before changing channel configurations (RAIL_ConfigChannels() or a built-in configuration) as unprocessed packets' channel could reflect the wrong configuration. On EFR32xG1 only this advice also applies when changing channels for receive or transmit where unprocessed packets' channel could reflect the new channel.

Definition at line 4257 of file common/rail_types.h

Address Filtering

Address Filtering

Configuration APIs for receive packet address filtering.

The address filtering code examines the packet as follows.

Bytes: 0 - 255	0 - 8	0 - 255	0 - 8	Variable
Data0	Field0	Data1	Field1	Data2

In the above structure, anything listed as DataN is an optional section of bytes that RAIL will not process for address filtering. The FieldN segments reference specific sections in the packet that will each be interpreted as an address during address filtering. The application may submit up to four addresses to attempt to match each field segment and each address may have a size of up to 8 bytes. To set up address filtering, first configure the locations and length of the addresses in the packet. Next, configure which combinations of matches in Field0 and Field1 should constitute an address match. Last, enter addresses into tables for each field and enable them. The first two of these are part of the RAIL_AddrConfig_t structure while the second part is configured at runtime using the RAIL_SetAddressFilterAddress() API. A brief description of each configuration is listed below.

The offsets and sizes of the fields are assumed fixed for the RAIL address filter. To set them, specify arrays for these values in the sizes and offsets entries in the RAIL_AddrConfig_t structure. A size of zero indicates that a field is disabled. The start offset for a field is relative to the previous start offset and, if you're using FrameType decoding, the first start offset is relative to the end of the byte containing the frame type.

Configuring which combinations of Field0 and Field1 constitute a match is the most complex portion of the address filter. The easiest way to think about this is with a truth table. If you consider each of the four possible address entries in a field, you can have a match on any one of those or a match for none of them. This is shown in the 5x5 truth table below where Field0 matches are the rows and Field1 matches are the columns.

	No Match	Address 0	Address 1	Address 2	Address 3
No Match	bit0	bit1	bit2	bit3	bit4
Address 0	bit5	bit6	bit7	bit8	bit9
Address 1	bit10	bit11	bit12	bit13	bit14
Address 2	bit15	bit16	bit17	bit18	bit19
Address 3	bit20	bit21	bit22	bit23	bit24

Because this is only 25 bits, it can be represented in one 32-bit integer where 1 indicates a filter pass and 0 indicates a filter fail. This is the matchTable parameter in the configuration structure and is used during filtering. For common simple configurations, two defines are provided with the truth tables as shown below. The first is ADDRCONFIG_MATCH_TABLE_SINGLE_FIELD, which can be used if only using one address field (either field). If using two fields and want to force in the same address entry in each field, use the second define: ADDRCONFIG_MATCH_TABLE_DOUBLE_FIELD. For more complex systems, create a valid custom table.

Note

- Address filtering does not function reliably with PHYs that use a data rate greater than 500 kbps. If this is a requirement, filtering must currently be done by the application.

Modules

RAIL_AddrConfig_t

Typedefs

typedef uint8_t **RAIL_AddrFilterMask_t**
A bitmask representation of which 4 filters passed for each ADDRCONFIG_MAX_ADDRESS_FIELDS when filtering has completed successfully.

Functions

RAIL_Status_t **RAIL_ConfigAddressFilter**(RAIL_Handle_t railHandle, const RAIL_AddrConfig_t *addrConfig)
Configure address filtering.

bool **RAIL_EnableAddressFilter**(RAIL_Handle_t railHandle, bool enable)
Enable address filtering.

bool **RAIL_IsAddressFilterEnabled**(RAIL_Handle_t railHandle)
Return whether address filtering is currently enabled.

void **RAIL_ResetAddressFilter**(RAIL_Handle_t railHandle)
Reset the address filtering configuration.

RAIL_Status_t **RAIL_SetAddressFilterAddress**(RAIL_Handle_t railHandle, uint8_t field, uint8_t index, const uint8_t *value, bool enable)
Set an address for filtering in hardware.

RAIL_Status_t **RAIL_SetAddressFilterAddressMask**(RAIL_Handle_t railHandle, uint8_t field, const uint8_t *bitMask)
Set an address bit mask for filtering in hardware.

RAIL_Status_t **RAIL_EnableAddressFilterAddress**(RAIL_Handle_t railHandle, bool enable, uint8_t field, uint8_t index)
Enable address filtering for the specified address.

Macros

#define **ADDRCONFIG_MATCH_TABLE_SINGLE_FIELD** (0x1FFFFFE)
A default address filtering match table for configurations that use only one address field.

#define **ADDRCONFIG_MATCH_TABLE_DOUBLE_FIELD** (0x1041040)
A default address filtering match table for configurations that use two address fields and want to match the same index in each.

#define **ADDRCONFIG_MAX_ADDRESS_FIELDS** (2)
The maximum number of address fields that can be used by the address filtering logic.

Typedef Documentation

RAIL_AddrFilterMask_t

```
typedef uint8_t RAIL_AddrFilterMask_t
```

A bitmask representation of which 4 filters passed for each ADDRCONFIG_MAX_ADDRESS_FIELDS when filtering has completed successfully.

It's layout is: | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | |-------—+-------—+-------—+-------—+-------—+-------—+-------—+-------| | Second Address Field Nibble | First Address Field Nibble | | Addr 3 | Addr 2 | Addr 1 | Addr 0 | Addr 3 | Addr 2 | Addr 1 | Addr 0 | | match | match | match | match | match | match | match | match | |-------—+-------—+-------—+-------—+-------—+-------—+-------—+-------—|

Note

- This information is valid in RAIL_IEEE802154_Address_t on all platforms, but is only valid in RAIL_RxPacketInfo_t on platforms where RAIL_SUPPORTS_ADDR_FILTER_MASK is true.

Definition at line 3681 of file common/rail_types.h

Function Documentation

RAIL_ConfigAddressFilter

> RAIL_Status_t RAIL_ConfigAddressFilter (RAIL_Handle_t railHandle, const RAIL_AddrConfig_t *addrConfig)

Configure address filtering.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	addrConfig	The configuration structure, which defines how addresses are set up in your packets.

Returns

- Status code indicating success of the function call.

You must call this function to set up address filtering. You may call it multiple times but all previous information is wiped out each time you call and any configured addresses must be reset.

Definition at line 4442 of file common/rail.h

RAIL_EnableAddressFilter

> bool RAIL_EnableAddressFilter (RAIL_Handle_t railHandle, bool enable)

Enable address filtering.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	enable	An argument to indicate whether or not to enable address filtering.

Returns

- True if address filtering was enabled to start with and false otherwise.

Only allow packets through that pass the current address filtering configuration. This does not reset or change the configuration so you can set that up before turning on this feature.

Definition at line 4458 of file common/rail.h

RAIL_IsAddressFilterEnabled

> bool RAIL_IsAddressFilterEnabled (RAIL_Handle_t railHandle)

Return whether address filtering is currently enabled.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- True if address filtering is enabled and false otherwise.

Definition at line 4466 of file common/rail.h

RAIL_ResetAddressFilter

```
void RAIL_ResetAddressFilter (RAIL_Handle_t railHandle)
```

Reset the address filtering configuration.

Parameters

[in]	railHandle	A RAIL instance handle.

Resets all structures related to address filtering. This does not disable address filtering. It leaves the radio in a state where no packets pass filtering.

Definition at line 4477 of file common/rail.h

RAIL_SetAddressFilterAddress

```
RAIL_Status_t RAIL_SetAddressFilterAddress (RAIL_Handle_t railHandle, uint8_t field, uint8_t index, const uint8_t *value,
bool enable)
```

Set an address for filtering in hardware.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	field	Indicates an address field for this address.
[in]	index	Indicates a match entry for this address for a given field.
[in]	value	A pointer to the address data. This must be at least as long as the size specified in RAIL_ConfigAddressFilter(). The first byte, value[0], will be compared to the first byte received over the air for this address field.
[in]	enable	A boolean to indicate whether this address should be enabled immediately.

Returns

- Status code indicating success of the function call.

This function loads the given address into hardware for filtering and starts filtering if you set the enable parameter to true. Otherwise, call RAIL_EnableAddressFilterAddress() to turn it on later.

Definition at line 4498 of file common/rail.h

RAIL_SetAddressFilterAddressMask

```
RAIL_Status_t RAIL_SetAddressFilterAddressMask (RAIL_Handle_t railHandle, uint8_t field, const uint8_t *bitMask)
```

Set an address bit mask for filtering in hardware.

Parameters

[in]	railHandle	A RAIL instance handle.

[in]	field	Indicates an address field for this address bit mask.
[in]	bitMask	A pointer to the address bitmask. This must be at least as long as the size specified in RAIL_ConfigAddressFilter(). The first byte, bitMask[0], will be applied to the first byte received over the air for this address field. Bits set to 1 in the bit mask indicate which bit positions in the incoming packet to compare against the stored addresses during address filtering. Bits set to 0 indicate which bit positions to ignore in the incoming packet during address filtering. This bit mask is applied to all address entries.

Returns

- Status code indicating success of the function call.

This function loads the given address bit mask into hardware for use when address filtering is enabled. All bits in the stored address bit mask are set to 1 during hardware initialization and when either RAIL_ConfigAddressFilter() or RAIL_ResetAddressFilter() are called.

Note

- This feature/API is not supported on the EFR32XG1 family of chips or the EFR32XG21. Use the compile time symbol RAIL_SUPPORTS_ADDR_FILTER_ADDRESS_BIT_MASK or the runtime call RAIL_SupportsAddrFilterAddressBitMask() to check whether the platform supports this feature.

Definition at line 4530 of file common/rail.h

RAIL_EnableAddressFilterAddress

```
RAIL_Status_t RAIL_EnableAddressFilterAddress (RAIL_Handle_t railHandle, bool enable, uint8_t field, uint8_t index)
```

Enable address filtering for the specified address.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	enable	An argument to indicate whether or not to enable address filtering.
[in]	field	Indicates an address for the address.
[in]	index	Indicates a match entry in the given field you want to enable.

Returns

- Status code indicating success of the function call.

Definition at line 4544 of file common/rail.h

Macro Definition Documentation

ADDRCONFIG_MATCH_TABLE_SINGLE_FIELD

```
#define ADDRCONFIG_MATCH_TABLE_SINGLE_FIELD
```

Value:

```
(0x1FFFFFE)
```

A default address filtering match table for configurations that use only one address field.

The truth table for address matching is shown below.

	No Match	Address 0	Address 1	Address 2	Address 3
No Match	0	1	1	1	1
Address 0	1	1	1	1	1
Address 1	1	1	1	1	1
Address 2	1	1	1	1	1
Address 3	1	1	1	1	1

Definition at line 3606 of file common/rail_types.h

ADDRCONFIG_MATCH_TABLE_DOUBLE_FIELD

```
#define ADDRCONFIG_MATCH_TABLE_DOUBLE_FIELD
```

Value:

```
(0×1041040)
```

A default address filtering match table for configurations that use two address fields and want to match the same index in each.

The truth table for address matching is shown below.

	No Match	Address 0	Address 1	Address 2	Address 3
No Match	0	0	0	0	0
Address 0	0	1	0	0	0
Address 1	0	0	1	0	0
Address 2	0	0	0	1	0
Address 3	0	0	0	0	1

Definition at line 3618 of file common/rail_types.h

ADDRCONFIG_MAX_ADDRESS_FIELDS

```
#define ADDRCONFIG_MAX_ADDRESS_FIELDS
```

Value:

```
(2)
```

The maximum number of address fields that can be used by the address filtering logic.

Definition at line 3622 of file common/rail_types.h

RAIL_AddrConfig_t

A structure to configure the address filtering functionality in RAIL.

Public Attributes

uint8_t	offsets A list of the start offsets for each field.
uint8_t	sizes A list of the address field sizes.
uint32_t	matchTable The truth table to determine how the two fields combine to create a match.

Public Attribute Documentation

offsets

```
uint8_t RAIL_AddrConfig_t::offsets[(2)]
```

A list of the start offsets for each field.

These offsets are specified relative to the previous field's end. For the first field, it is relative to either the beginning of the packet or the end of the frame type byte if frame type decoding is enabled. If a field is unused, it's offset should be set to 0.

Definition at line `3637` of file `common/rail_types.h`

sizes

```
uint8_t RAIL_AddrConfig_t::sizes[(2)]
```

A list of the address field sizes.

These sizes are specified in bytes from 0 to 8. If you choose a size of 0, this field is effectively disabled.

Definition at line `3645` of file `common/rail_types.h`

matchTable

```
uint32_t RAIL_AddrConfig_t::matchTable
```

The truth table to determine how the two fields combine to create a match.

For detailed information about how this truth table is formed, see the detailed description of Address Filtering.

For simple predefined configurations use the following defines.

- ADDRCONFIG_MATCH_TABLE_SINGLE_FIELD

- For filtering that only uses a single address field.
- ADDRCONFIG_MATCH_TABLE_DOUBLE_FIELD for two field filtering where you
  - For filtering that uses two address fields in a configurations where you want the following logic `((Field0, Index0) && (Field1, Index0)) || ((Field0, Index1) && (Field1, Index1)) || ...`

Definition at line `3661` of file `common/rail_types.h`

Packet Information

Packet Information

APIs to get information about received packets.

After receiving a packet, RAIL will trigger a RAIL_EVENT_RX_PACKET_RECEIVED event. At that point, there is a variety of information available to the application about the received packet. The following example code assumes that the RAIL_RX_OPTION_REMOVE_APPENDED_INFO is not used, and the application wants as much data about the packet as possible.

```
// Get all information about a received packet.
RAIL_Status_t status;
RAIL_RxPacketInfo_t rxInfo;
RAIL_RxPacketDetails_t rxDetails;
RAIL_RxPacketHandle_t rxHandle
  = RAIL_GetRxPacketInfo(railHandle, RAIL_RX_PACKET_HANDLE_NEWEST, &rxInfo);
assert(rxHandle != RAIL_RX_PACKET_HANDLE_INVALID);
status = RAIL_GetRxPacketDetailsAlt(railHandle, rxHandle, &rxDetails);
assert(status == RAIL_STATUS_NO_ERROR);
if (rxDetails.timeReceived.timePosition == RAIL_PACKET_TIME_INVALID) {
  return; // No timestamp available for this packet
}
// CRC_BYTES only needs to be added when not using RAIL_RX_OPTION_STORE_CRC
rxDetails.timeReceived.totalPacketBytes = rxInfo.packetBytes + CRC_BYTES;
// Choose the function which gives the desired timestamp
status = RAIL_GetRxTimeFrameEndAlt(railHandle, &rxDetails);
assert(status == RAIL_STATUS_NO_ERROR);
// Now all fields of rxInfo and rxDetails have been populated correctly
```

Functions

RAIL_RxPacketHandle_t	RAIL_GetRxPacketInfo(RAIL_Handle_t railHandle, RAIL_RxPacketHandle_t packetHandle, RAIL_RxPacketInfo_t *pPacketInfo) Get basic information about a pending or received packet.
void	RAIL_GetRxIncomingPacketInfo(RAIL_Handle_t railHandle, RAIL_RxPacketInfo_t *pPacketInfo) Get information about the live incoming packet (if any).
void	RAIL_CopyRxPacket(uint8_t *pDest, const RAIL_RxPacketInfo_t *pPacketInfo) Copy a full packet to a user-specified contiguous buffer.
RAIL_Status_t	RAIL_GetRxPacketDetails(RAIL_Handle_t railHandle, RAIL_RxPacketHandle_t packetHandle, RAIL_RxPacketDetails_t *pPacketDetails) Get detailed information about a received packet.
RAIL_Status_t	RAIL_GetRxPacketDetailsAlt(RAIL_Handle_t railHandle, RAIL_RxPacketHandle_t packetHandle, RAIL_RxPacketDetails_t *pPacketDetails) Get detailed information about a received packet.
RAIL_Status_t	RAIL_GetRxTimePreambleStart(RAIL_Handle_t railHandle, uint16_t totalPacketBytes, RAIL_Time_t *pPacketTime) Adjust a RAIL RX timestamp to refer to the start of the preamble.

RAIL_Status_t	RAIL_GetRxTimePreambleStartAlt(RAIL_Handle_t railHandle, RAIL_RxPacketDetails_t *pPacketDetails) Adjust a RAIL RX timestamp to refer to the start of the preamble.
RAIL_Status_t	RAIL_GetRxTimeSyncWordEnd(RAIL_Handle_t railHandle, uint16_t totalPacketBytes, RAIL_Time_t *pPacketTime) Adjust a RAIL RX timestamp to refer to the end of the sync word.
RAIL_Status_t	RAIL_GetRxTimeSyncWordEndAlt(RAIL_Handle_t railHandle, RAIL_RxPacketDetails_t *pPacketDetails) Adjust a RAIL RX timestamp to refer to the end of the sync word.
RAIL_Status_t	RAIL_GetRxTimeFrameEnd(RAIL_Handle_t railHandle, uint16_t totalPacketBytes, RAIL_Time_t *pPacketTime) Adjust a RAIL RX timestamp to refer to the end of frame.
RAIL_Status_t	RAIL_GetRxTimeFrameEndAlt(RAIL_Handle_t railHandle, RAIL_RxPacketDetails_t *pPacketDetails) Adjust a RAIL RX timestamp to refer to the end of frame.

Function Documentation

RAIL_GetRxPacketInfo

> RAIL_RxPacketHandle_t RAIL_GetRxPacketInfo (RAIL_Handle_t railHandle, RAIL_RxPacketHandle_t packetHandle, RAIL_RxPacketInfo_t *pPacketInfo)

Get basic information about a pending or received packet.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	packetHandle	A packet handle for the unreleased packet as returned from a previous call, or sentinel values RAIL_RX_PACKET_HANDLE_OLDEST, RAIL_RX_PACKET_HANDLE_OLDEST_COMPLETE or RAIL_RX_PACKET_HANDLE_NEWEST.
[out]	pPacketInfo	An application-provided pointer to store RAIL_RxPacketInfo_t for the requested packet. Must be non-NULL.

Returns

- The packet handle for the requested packet: if packetHandle was one of the sentinel values, returns the actual packet handle for that packet, otherwise returns packetHandle. It may return RAIL_RX_PACKET_HANDLE_INVALID to indicate an error.

This function can be used in any RX mode. It does not free up any internal resources. If used in RX RAIL_DataMethod_t::FIFO_MODE, the value in RAIL_RxPacketInfo_t::packetBytes will only return the data remaining in the FIFO. Any data read via earlier calls to RAIL_ReadRxFifo() is not included.

Note

- When getting information about an arriving packet that is not yet complete, (i.e., pPacketInfo->packetStatus == RAIL_RX_PACKET_RECEIVING), keep in mind its data is highly suspect because it has not yet passed any CRC integrity checking. Also note that the packet could be aborted, canceled, or fail momentarily, invalidating its data in Packet mode. Furthermore, there is a small chance towards the end of packet reception that the filled-in RAIL_RxPacketInfo_t could include not only packet data received so far, but also some raw radio-appended info detail bytes that RAIL's packet-completion processing will subsequently deal with. It's up to the application to know its packet format well enough to avoid confusing such info as packet data.

Definition at line `3740` of file `common/rail.h`

RAIL_GetRxIncomingPacketInfo

> void RAIL_GetRxIncomingPacketInfo (RAIL_Handle_t railHandle, RAIL_RxPacketInfo_t *pPacketInfo)

Get information about the live incoming packet (if any).

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	pPacketInfo	Application provided pointer to store RAIL_RxPacketInfo_t for the incoming packet.

Differs from RAIL_GetRxPacketInfo() by only returning information about a packet actively being received, something which even the RAIL_RX_PACKET_HANDLE_NEWEST may not represent if there are completed but unprocessed packets in the receive FIFO.

This function can only be called from callback context, e.g., when handling RAIL_EVENT_RX_FILTER_PASSED or RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND. It must not be used with receive RAIL_DataMethod_t::FIFO_MODE if any portion of an incoming packet has already been extracted from the receive FIFO.

Note

- The incomplete data of an arriving packet is highly suspect because it has not yet passed any CRC integrity checking. Also note that the packet could be aborted, canceled, or fail momentarily, invalidating its data in Packet mode. Furthermore, there is a small chance towards the end of packet reception that the filled-in RAIL_RxPacketInfo_t could include not only packet data received so far, but also some raw radio-appended info detail bytes that RAIL's packet-completion processing will subsequently deal with. It's up to the application to know its packet format well enough to avoid confusing such info as packet data.

Definition at line `3773` of file `common/rail.h`

RAIL_CopyRxPacket

```
static void RAIL_CopyRxPacket (uint8_t *pDest, const RAIL_RxPacketInfo_t *pPacketInfo)
```

Copy a full packet to a user-specified contiguous buffer.

Parameters

[out]	pDest	An application-provided pointer to a buffer of at least pPacketInfo->packetBytes in size to store the packet data contiguously. This buffer must never overlay RAIL's receive FIFO buffer. Exactly pPacketInfo->packetBytes of packet data will be written into it.
[in]	pPacketInfo	RAIL_RxPacketInfo_t for the requested packet.

Note

- This is a convenience helper function, which is intended to be expedient. As a result, it does not check the validity of its arguments, so don't pass either as NULL, and don't pass a pDest pointer to a buffer that's too small for the packet's data.
- If only a portion of the packet is needed, use RAIL_PeekRxPacket() instead.

Definition at line `3795` of file `common/rail.h`

RAIL_GetRxPacketDetails

```
RAIL_Status_t RAIL_GetRxPacketDetails (RAIL_Handle_t railHandle, RAIL_RxPacketHandle_t packetHandle,
RAIL_RxPacketDetails_t *pPacketDetails)
```

Get detailed information about a received packet.

Parameters

[in]	railHandle	A RAIL instance handle.

[in]	packetHandle	A packet handle for the unreleased packet as returned from a previous call to RAIL_GetRxPacketInfo() or RAIL_HoldRxPacket(), or sentinel values RAIL_RX_PACKET_HANDLE_OLDEST, RAIL_RX_PACKET_HANDLE_OLDEST_COMPLETE or RAIL_RX_PACKET_HANDLE_NEWEST.
[inout]	pPacketDetails	An application-provided non-NULL pointer to store RAIL_RxPacketDetails_t for the requested packet. For RAIL_RxPacketStatus_t RAIL_RX_PACKET_READY_ packets, the timeReceived fields totalPacketBytes and timePosition must be initialized prior to each call: • totalPacketBytes with the total number of bytes of the received packet for RAIL to use when calculating the specified timestamp. This should account for all bytes received over the air after the Preamble and Sync word(s), including CRC bytes. • timePosition with a RAIL_PacketTimePosition_t value specifying the packet position to put in the timeReceived field on return. This field will also be updated with the actual position corresponding to the timeReceived value filled in.

This function can be used in any RX mode; it does not free up any internal resources.

Returns

- RAIL_STATUS_NO_ERROR if pPacketDetails was filled in, or an appropriate error code otherwise.

Note

- Certain details are always available, while others are only available if the RAIL_RxOptions_tRAIL_RX_OPTION_REMOVE_APPENDED_INFO option is not in effect and the received packet's RAIL_RxPacketStatus_t is among the RAIL_RX_PACKET_READY_ set. See RAIL_RxPacketDetails_t for clarification.
- Consider using RAIL_GetRxPacketDetailsAlt for smaller code size.

Definition at line `3841` of file `common/rail.h`

RAIL_GetRxPacketDetailsAlt

```
RAIL_Status_t RAIL_GetRxPacketDetailsAlt (RAIL_Handle_t railHandle, RAIL_RxPacketHandle_t packetHandle,
RAIL_RxPacketDetails_t *pPacketDetails)
```

Get detailed information about a received packet.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	packetHandle	A packet handle for the unreleased packet as returned from a previous call to RAIL_GetRxPacketInfo() or RAIL_HoldRxPacket(), or sentinel values RAIL_RX_PACKET_HANDLE_OLDESTRAIL_RX_PACKET_HANDLE_OLDEST_COMPLETE or RAIL_RX_PACKET_HANDLE_NEWEST.
[out]	pPacketDetails	An application-provided non-NULL pointer to store RAIL_RxPacketDetails_t for the requested packet. For RAIL_RxPacketStatus_t RAIL_RX_PACKET_READY_ packets, the timeReceived field packetTime will be populated with a timestamp corresponding to a default location in the packet. The timeReceived field timePosition will be populated with a RAIL_PacketTimePosition_t value specifying that default packet location. Call RAIL_GetRxTimePreambleStart, RAIL_GetRxTimeSyncWordEnd, or RAIL_GetRxTimeFrameEnd to adjust that timestamp for different locations in the packet.

This function can be used in any RX mode. It does not free up any internal resources.

Returns

- RAIL_STATUS_NO_ERROR if pPacketDetails was filled in, or an appropriate error code otherwise.

This alternative API allows for smaller code size by deadstripping the timestamp adjustment algorithms which are not in use.

Note

- Certain details are always available, while others are only available if the RAIL_RxOptions_tRAIL_RX_OPTION_REMOVE_APPENDED_INFO option is not in effect and the received packet's RAIL_RxPacketStatus_t is among the RAIL_RX_PACKET_READY_ set. See RAIL_RxPacketDetails_t for clarification.

Definition at line 3878 of file common/rail.h

RAIL_GetRxTimePreambleStart

```
RAIL_Status_t RAIL_GetRxTimePreambleStart (RAIL_Handle_t railHandle, uint16_t totalPacketBytes, RAIL_Time_t *pPacketTime)
```

Adjust a RAIL RX timestamp to refer to the start of the preamble.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	totalPacketBytes	The total number of bytes of the received packet for RAIL to use when calculating the specified timestamp. This should account for all bytes received over the air after the Preamble and Sync word(s), including CRC bytes.
[inout]	pPacketTime	The time that was returned in the RAIL_PacketTimeStamp_t::packetTime field of RAIL_RxPacketDetails_t::timeReceived from a previous call to RAIL_GetRxPacketDetailsAlt for this same packet. After this function, the time at that location will be updated with the time that the preamble for this packet started on air. Must be non-NULL.

Returns

- RAIL_STATUS_NO_ERROR if pPacketTime was successfully calculated, or an appropriate error code otherwise.

Call this API while the given railHandle is active, or it will return an error code of RAIL_STATUS_INVALID_STATE. Note that this API may return incorrect timestamps when sub-phys are in use. Prefer RAIL_GetRxTimePreambleStartAlt in those situations. See RAIL_RxPacketDetails_t::subPhyId for more details.

Definition at line 3905 of file common/rail.h

RAIL_GetRxTimePreambleStartAlt

```
RAIL_Status_t RAIL_GetRxTimePreambleStartAlt (RAIL_Handle_t railHandle, RAIL_RxPacketDetails_t *pPacketDetails)
```

Adjust a RAIL RX timestamp to refer to the start of the preamble.

Parameters

[in]	railHandle	A RAIL instance handle.
[inout]	pPacketDetails	The non-NULL details that were returned from a previous call to RAIL_GetRxPacketDetailsAlt for this same packet. The application must update the timeReceived field totalPacketBytes to be the total number of bytes of the received packet for RAIL to use when calculating the specified timestamp. This should account for all bytes received over the air after the Preamble and Sync word(s), including CRC bytes. After this function, the timeReceived field packetTime will be updated with the time that the preamble for this packet started on air.

Returns

- RAIL_STATUS_NO_ERROR if the packet time was successfully calculated, or an appropriate error code otherwise.

Call this API while the given railHandle is active, or it will return an error code of RAIL_STATUS_INVALID_STATE.

Definition at line 3927 of file common/rail.h

RAIL_GetRxTimeSyncWordEnd

> RAIL_Status_t RAIL_GetRxTimeSyncWordEnd (RAIL_Handle_t railHandle, uint16_t totalPacketBytes, RAIL_Time_t *pPacketTime)

Adjust a RAIL RX timestamp to refer to the end of the sync word.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	totalPacketBytes	The total number of bytes of the received packet for RAIL to use when calculating the specified timestamp. This should account for all bytes received over the air after the Preamble and Sync word(s), including CRC bytes.
[inout]	pPacketTime	The time that was returned in the RAIL_PacketTimeStamp_t::packetTime field of RAIL_RxPacketDetails_t::timeReceived from a previous call to RAIL_GetRxPacketDetailsAlt for this same packet. After this function, the time at that location will be updated with the time that the sync word for this packet finished on air. Must be non-NULL.

Returns

- RAIL_STATUS_NO_ERROR if pPacketTime was successfully calculated, or an appropriate error code otherwise.

Call this API while the given railHandle is active, or it will return an error code of RAIL_STATUS_INVALID_STATE. Note that this API may return incorrect timestamps when sub-phys are in use. Prefer RAIL_GetRxTimePreambleStartAlt in those situations. See RAIL_RxPacketDetails_t::subPhyId for more details.

Definition at line 3953 of file common/rail.h

RAIL_GetRxTimeSyncWordEndAlt

> RAIL_Status_t RAIL_GetRxTimeSyncWordEndAlt (RAIL_Handle_t railHandle, RAIL_RxPacketDetails_t *pPacketDetails)

Adjust a RAIL RX timestamp to refer to the end of the sync word.

Parameters

[in]	railHandle	A RAIL instance handle.
[inout]	pPacketDetails	The non-NULL details that were returned from a previous call to RAIL_GetRxPacketDetailsAlt for this same packet. The application must update the timeReceived field totalPacketBytes to be the total number of bytes of the received packet for RAIL to use when calculating the specified timestamp. This should account for all bytes received over the air after the Preamble and Sync word(s), including CRC bytes. After this function, the timeReceived field packetTime will be updated with the time that the sync word for this packet finished on air.

Returns

- RAIL_STATUS_NO_ERROR if the packet time was successfully calculated, or an appropriate error code otherwise.

Call this API while the given railHandle is active, or it will return an error code of RAIL_STATUS_INVALID_STATE.

Definition at line 3975 of file common/rail.h

RAIL_GetRxTimeFrameEnd

> RAIL_Status_t RAIL_GetRxTimeFrameEnd (RAIL_Handle_t railHandle, uint16_t totalPacketBytes, RAIL_Time_t *pPacketTime)

Adjust a RAIL RX timestamp to refer to the end of frame.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	totalPacketBytes	The total number of bytes of the received packet for RAIL to use when calculating the specified timestamp. This should account for all bytes received over the air after the Preamble and Sync word(s), including CRC bytes.
[inout]	pPacketTime	The time that was returned in the RAIL_PacketTimeStamp_t::packetTime field of RAIL_RxPacketDetails_t::timeReceived from a previous call to RAIL_GetRxPacketDetailsAlt for this same packet. After this function, the time at that location will be updated with the time that this packet finished on air. Must be non-NULL.

Returns

- RAIL_STATUS_NO_ERROR if pPacketTime was successfully calculated, or an appropriate error code otherwise.

Call this API while the given railHandle is active, or it will return an error code of RAIL_STATUS_INVALID_STATE. Note that this API may return incorrect timestamps when sub-phys are in use. Prefer RAIL_GetRxTimePreambleStartAlt in those situations. See RAIL_RxPacketDetails_t::subPhyId for more details.

Definition at line `4001` of file `common/rail.h`

RAIL_GetRxTimeFrameEndAlt

```
RAIL_Status_t RAIL_GetRxTimeFrameEndAlt (RAIL_Handle_t railHandle, RAIL_RxPacketDetails_t *pPacketDetails)
```

Adjust a RAIL RX timestamp to refer to the end of frame.

Parameters

[in]	railHandle	A RAIL instance handle.
[inout]	pPacketDetails	The non-NULL details that were returned from a previous call to RAIL_GetRxPacketDetailsAlt for this same packet. The application must update the timeReceived field totalPacketBytes to be the total number of bytes of the received packet for RAIL to use when calculating the specified timestamp. This should account for all bytes received over the air after the Preamble and Sync word(s), including CRC bytes. After this function, the timeReceived field packetTime will be updated with the time that the packet finished on air.

Returns

- RAIL_STATUS_NO_ERROR if the packet time was successfully calculated, or an appropriate error code otherwise.

Call this API while the given railHandle is active, or it will return an error code of RAIL_STATUS_INVALID_STATE.

Definition at line `4023` of file `common/rail.h`

Retiming

Retiming

EFR32-specific retiming capability.

The EFR product families have many digital and analog modules that can run in parallel with a radio. These combinations can cause interference and degradation on the radio RX sensitivity. Retiming can modify the clocking of the digital modules to reduce the interference.

Enumerations

enum RAIL_RetimeOptions_t {

RAIL_RETIME_OPTION_HFXO_SHIFT = 0
RAIL_RETIME_OPTION_HFRCO_SHIFT
RAIL_RETIME_OPTION_DCDC_SHIFT
RAIL_RETIME_OPTION_LCD_SHIFT

}

Retiming options bit shifts.

Functions

RAIL_Status_t RAIL_ConfigRetimeOptions(RAIL_Handle_t railHandle, RAIL_RetimeOptions_t mask, RAIL_RetimeOptions_t options)

Configure retiming options.

RAIL_Status_t RAIL_GetRetimeOptions(RAIL_Handle_t railHandle, RAIL_RetimeOptions_t *pOptions)

Get the currently configured retiming option.

RAIL_Status_t RAIL_ChangedDcdc(void)

Indicate that the DCDC peripheral bus clock enable has changed allowing RAIL to react accordingly.

Macros

#define RAIL_RETIME_OPTION_HFXO (1U << RAIL_RETIME_OPTION_HFXO_SHIFT)

An option to configure HFXO retiming.

#define RAIL_RETIME_OPTION_HFRCO (1U << RAIL_RETIME_OPTION_HFRCO_SHIFT)

An option to configure HFRCO retiming.

#define RAIL_RETIME_OPTION_DCDC (1U << RAIL_RETIME_OPTION_DCDC_SHIFT)

An option to configure DCDC retiming.

#define RAIL_RETIME_OPTION_LCD (1U << RAIL_RETIME_OPTION_LCD_SHIFT)

An option to configure LCD retiming.

#define RAIL_RETIME_OPTIONS_NONE 0x0U

A value representing no retiming options.

#define RAIL_RETIME_OPTIONS_ALL 0xFFU

A value representing all retiming options.

Enumeration Documentation

RAIL_RetimeOptions_t

> RAIL_RetimeOptions_t

Retiming options bit shifts.

Enumerator	
RAIL_RETIME_OPTION_HFXO_SHIFT	Shift position of RAIL_RETIME_OPTION_HFXO bit.
RAIL_RETIME_OPTION_HFRCO_SHIFT	Shift position of RAIL_RETIME_OPTION_HFRCO bit.
RAIL_RETIME_OPTION_DCDC_SHIFT	Shift position of RAIL_RETIME_OPTION_DCDC bit.
RAIL_RETIME_OPTION_LCD_SHIFT	Shift position of RAIL_RETIME_OPTION_LCD bit.

Definition at line `5725` of file `common/rail_types.h`

Function Documentation

RAIL_ConfigRetimeOptions

> RAIL_Status_t RAIL_ConfigRetimeOptions (RAIL_Handle_t railHandle, RAIL_RetimeOptions_t mask, RAIL_RetimeOptions_t options)

Configure retiming options.

Parameters

[in]	railHandle	A handle of RAIL instance.
[in]	mask	A bitmask containing which options should be modified.
[in]	options	A bitmask containing desired configuration settings. Bit positions for each option are found in the RAIL_RetimeOptions_t.

Returns

- Status code indicating success of the function call.

Definition at line `5778` of file `common/rail_types.h`

RAIL_GetRetimeOptions

> RAIL_Status_t RAIL_GetRetimeOptions (RAIL_Handle_t railHandle, RAIL_RetimeOptions_t *pOptions)

Get the currently configured retiming option.

Parameters

[in]	railHandle	A handle of RAIL instance.
[out]	pOptions	A pointer to configured retiming options bitmask indicating which are enabled.

Returns

- Status code indicating success of the function call.

Definition at line `5790` of file `common/rail_types.h`

RAIL_ChangedDcdc

```
RAIL_Status_t RAIL_ChangedDcdc (void)
```

Indicate that the DCDC peripheral bus clock enable has changed allowing RAIL to react accordingly.

Parameters

N/A		

Note

- This should be called after DCDC has been enabled or disabled.

Returns

- Status code indicating success of the function call.

Definition at line 5801 of file common/rail_types.h

Macro Definition Documentation

RAIL_RETIME_OPTION_HFXO

```
#define RAIL_RETIME_OPTION_HFXO
```

Value:

```
(1U << RAIL_RETIME_OPTION_HFXO_SHIFT)
```

An option to configure HFXO retiming.

Definition at line 5740 of file common/rail_types.h

RAIL_RETIME_OPTION_HFRCO

```
#define RAIL_RETIME_OPTION_HFRCO
```

Value:

```
(1U << RAIL_RETIME_OPTION_HFRCO_SHIFT)
```

An option to configure HFRCO retiming.

Definition at line 5746 of file common/rail_types.h

RAIL_RETIME_OPTION_DCDC

```
#define RAIL_RETIME_OPTION_DCDC
```

Value:

```
(1U << RAIL_RETIME_OPTION_DCDC_SHIFT)
```

An option to configure DCDC retiming.

Ignored on platforms that lack DCDC.

Definition at line 5753 of file common/rail_types.h

RAIL_RETIME_OPTION_LCD

```
#define RAIL_RETIME_OPTION_LCD
```

Value:

```
(1U << RAIL_RETIME_OPTION_LCD_SHIFT)
```

An option to configure LCD retiming.

Ignored on platforms that lack LCD.

Definition at line 5760 of file common/rail_types.h

RAIL_RETIME_OPTIONS_NONE

```
#define RAIL_RETIME_OPTIONS_NONE
```

Value:

```
0x0U
```

A value representing no retiming options.

Definition at line 5764 of file common/rail_types.h

RAIL_RETIME_OPTIONS_ALL

```
#define RAIL_RETIME_OPTIONS_ALL
```

Value:

```
0xFFU
```

A value representing all retiming options.

Definition at line 5767 of file common/rail_types.h

Sleep

Sleep

These APIs help when putting the system to an EM2/EM3/EM4 sleep states where the high frequency clock is disabled.

The RAIL library has its own timebase and the ability to schedule operations into the future. When going to any power mode that disables the HF clock used for the radio (EM2/EM3/EM4), it is important that this timebase is synchronized to a running LFCLK and the chip is set to wake up before the next scheduled event. If RAIL has not been configured to use the power manager, RAIL_Sleep and RAIL_Wake must be called for performing this synchronization. If RAIL has been configured to use the power manager, RAIL_InitPowerManager, it will automatically perform timer synchronization based on the selected RAIL_TimerSyncConfig_t. Calls to RAIL_Sleep and RAIL_Wake are unsupported in such a scenario.

Following example code snippets demonstrate synchronizing the timebase with and without timer synchronization:

Sleep with timer synchronization:

When sleeping with timer synchronization, you must first get the required LFCLK up and running and leave it running across sleep so that the high frequency clock that drives the RAIL time base can be synchronized to it. The RAIL_Sleep() API will also set up a wake event on the timer to wake up wakeupTime before the next timer event so that it can run successfully. See the EFR32 sections on Low-Frequency Clocks and RAIL Timer Synchronization for more setup details.

This is useful when maintaining packet timestamps across sleep or use the scheduled RX/TX APIs while sleeping in between. It does take more time and code to do the synchronization. If your application does not need this, it should be avoided.

Example (without Power Manager):

```
#include <rail.h>
#include <rail_types.h>

extern RAIL_Handle_t railHandle;
// Wakeup time for your crystal/board/chip combination
extern uint32_t wakeupTime;

void main(void) {
  RAIL_Status_t status;
  bool shouldSleep = false;

  // This function depends on your board/chip but it must enable the LFCLK
  // you intend to use for RTCC sync before we configure sleep as that function
  // will attempt to auto detect the clock.
  BoardSetupLFCLK()

  // Configure sleep for timer synchronization
  status = RAIL_ConfigSleep(railHandle, RAIL_SLEEP_CONFIG_TIMERSYNC_ENABLED);
  assert(status == RAIL_STATUS_NO_ERROR);

  // Application main loop
  while(1) {
    // ... do normal app stuff and set shouldSleep to true when we want to
    // sleep
    if (shouldSleep) {
      bool sleepAllowed = false;

      // Go critical to assess sleep decisions
      CORE_ENTER_CRITICAL();
      if (RAIL_Sleep(wakeupTime, &sleepAllowed) != RAIL_STATUS_NO_ERROR) {
        printf("Error trying to go to sleep!");
        CORE_EXIT_CRITICAL();
        continue;
      }
      if (sleepAllowed) {
        // Go to sleep
      }
      // Wakeup and sync the RAIL timebase back up
      RAIL_Wake(0);
      CORE_EXIT_CRITICAL();
    }
  }
}
```

Example (with Power Manager):

```
#include <rail.h>
#include <rail_types.h>
#include <sl_power_manager.h>

extern RAIL_Handle_t railHandle;

void main(void) {
  RAIL_Status_t status;
  bool shouldSleep = false;

  // This function depends on your board/chip but it must enable the LFCLK
  // you intend to use for RTCC sync before we configure sleep as that function
  // will attempt to auto detect the clock.
  BoardSetupLFCLK();
  // Configure sleep for timer synchronization
  status = RAIL_ConfigSleep(railHandle, RAIL_SLEEP_CONFIG_TIMERSYNC_ENABLED);
  assert(status == RAIL_STATUS_NO_ERROR);
  // Initialize application-level power manager service
  sl_power_manager_init();
  // Initialize RAIL library's use of the power manager
  RAIL_InitPowerManager();

  // Application main loop
  while(1) {
    // ... do normal app stuff and set shouldSleep to true when we want to
    // sleep
    if (shouldSleep) {
      // Let the CPU go to sleep if the system allows it.
      sl_power_manager_sleep();
    }
  }
}
```

RAIL APIs such as, RAIL_StartScheduledTx, RAIL_ScheduleRx, RAIL_SetTimer, RAIL_SetMultiTimer can be used to schedule periodic wakeups to perform a scheduled operation. The call to sl_power_manager_sleep() in the main loop ensures that the device sleeps until the scheduled operation is due. Upon completion, each instantaneous or scheduled RX/TX operation will indicate radio busy to the power manager to allow the application to service the RAIL event and perform subsequent operations before going to sleep. Therefore, it is important that the application idle the radio by either calling RAIL_Idle or RAIL_YieldRadio. If the radio transitions to RX after an RX or TX operation, always call RAIL_Idle in order transition to a lower sleep state. If the radio transitions to idle after an RX or TX operation, RAIL_YieldRadio should suffice in indicating to the power manager that the radio is no longer busy and the device can sleep.

The following example shows scheduling periodic TX on getting a TX completion event:

```
void RAILCb_Event(RAIL_Handle_t railHandle, RAIL_Events_t events) {
  // Omitting other event handlers
  if (events & RAIL_EVENTS_TX_COMPLETION) {
    // Schedule the next TX.
    RAIL_ScheduleTxConfig_t config = {
      .when = (RAIL_Time_t)parameters->startTime,
      .mode = (RAIL_TimeMode_t)parameters->startTimeMode
    };
    (void)RAIL_StartScheduledTx(radio.handle, channel, 0, &config, NULL);
  }
}
```

Note

- The above code assumes that RAIL automatic state transitions after TX are idle. Set RAIL_SetTxTransitions to ensure the right state transitions. Radio must be idle for the device to enter EM2 or lower energy mode.
- When using the power manager, usage of RAIL_YieldRadio in single protocol RAIL is similar to its usage in multiprotocol RAIL. See Yielding the Radio for more details.

- Back to back scheduled operations do not require an explicit call to RAIL_YieldRadio if the radio transitions to idle.

Sleep without timer synchronization:

When sleeping without timer synchronization, you are free to enable only the LFCLKs and wake sources required by the application. RAIL will not attempt to configure any wake events and may miss anything that occurs over sleep.

This is useful when your application does not care about packet timestamps or scheduling operations accurately over sleep.

Example (without Power Manager):

```
#include <rail.h>
#include <rail_types.h>

extern RAIL_Handle_t railHandle;

void main(void) {
  RAIL_Status_t status;
  bool shouldSleep = false;

  // Configure sleep for timer synchronization
  status = RAIL_ConfigSleep(railHandle, RAIL_SLEEP_CONFIG_TIMERSYNC_DISABLED);
  assert(status == RAIL_STATUS_NO_ERROR);

  // Application main loop
  while(1) {
    // ... do normal app stuff and set shouldSleep to true when we want to
    // sleep
    if (shouldSleep) {
      bool sleepAllowed = false;
      uint32_t sleepTime = 0;

      // Go critical to assess sleep decisions
      CORE_ENTER_CRITICAL();
      if (RAIL_Sleep(0, &sleepAllowed) != RAIL_STATUS_NO_ERROR) {
        printf("Error trying to go to sleep!");
        CORE_EXIT_CRITICAL();
        continue;
      }
      if (sleepAllowed) {
        // Go to sleep and optionally update sleepTime to the correct value
        // in microseconds
      }
      // Wakeup and sync the RAIL timebase back up
      RAIL_Wake(sleepTime);
      CORE_EXIT_CRITICAL();
    }
  }
}
```

Example (with Power Manager):

```
#include <rail.h>
#include <rail_types.h>
#include <sl_power_manager.h>

extern RAIL_Handle_t railHandle;

void main(void) {
  RAIL_Status_t status;
  bool shouldSleep = false;

  // This function depends on your board/chip but it must enable the LFCLK
  // you intend to use for RTCC sync before we configure sleep as that function
  // will attempt to auto detect the clock.
  BoardSetupLFCLK();
  // Configure sleep for timer synchronization
  status = RAIL_ConfigSleep(railHandle, RAIL_SLEEP_CONFIG_TIMERSYNC_DISABLED);
  assert(status == RAIL_STATUS_NO_ERROR);
  // Initialize application-level power manager service
  sl_power_manager_init();
  // Initialize RAIL library's use of the power manager
  RAIL_InitPowerManager();

  // Application main loop
  while(1) {
    // ... do normal app stuff and set shouldSleep to true when we want to
    // sleep
    if (shouldSleep) {
      // Let the CPU go to sleep if the system allows it.
      sl_power_manager_sleep();
    }
  }
}
```

Modules

RAIL_TimerSyncConfig_t

Enumerations

enum RAIL_SleepConfig_t {

RAIL_SLEEP_CONFIG_TIMERSYNC_DISABLED
RAIL_SLEEP_CONFIG_TIMERSYNC_ENABLED

}

The configuration.

Functions

void RAILCb_ConfigSleepTimerSync(RAIL_TimerSyncConfig_t *timerSyncConfig)

Configure RAIL timer synchronization.

RAIL_Status_t RAIL_ConfigSleep(RAIL_Handle_t railHandle, RAIL_SleepConfig_t sleepConfig)

Initialize RAIL timer synchronization.

RAIL_Status_t RAIL_ConfigSleepAlt(RAIL_Handle_t railHandle, RAIL_TimerSyncConfig_t *syncConfig)

Initialize RAIL timer synchronization.

RAIL_Status_t RAIL_Sleep(uint16_t wakeupProcessTime, bool *deepSleepAllowed)

Stop the RAIL timer and prepare RAIL for sleep.

RAIL_Status_t RAIL_Wake(RAIL_Time_t elapsedTime)
Wake RAIL from sleep and restart the RAIL timer.

RAIL_Status_t RAIL_InitPowerManager(void)
Initialize RAIL Power Manager.

RAIL_Status_t RAIL_DeinitPowerManager(void)
Stop the RAIL Power Manager.

Macros

#define RAIL_TIMER_SYNC_PRS_CHANNEL_DEFAULT (7U)
Default PRS channel to use when configuring sleep.

#define RAIL_TIMER_SYNC_RTCC_CHANNEL_DEFAULT (0U)
Default RTCC channel to use when configuring sleep.

#define RAIL_TIMER_SYNC_DEFAULT undefined
Default timer synchronization configuration.

Enumeration Documentation

RAIL_SleepConfig_t

> RAIL_SleepConfig_t

The configuration.

Enumerator	
RAIL_SLEEP_CONFIG_TIMERSYNC_DISABLED	Disable timer sync before and after sleep.
RAIL_SLEEP_CONFIG_TIMERSYNC_ENABLED	Enable timer sync before and after sleep.

Definition at line 424 of file common/rail_types.h

Function Documentation

RAILCb_ConfigSleepTimerSync

> void RAILCb_ConfigSleepTimerSync (RAIL_TimerSyncConfig_t *timerSyncConfig)

Configure RAIL timer synchronization.

Parameters

[inout]	timerSyncConfig	A pointer to the RAIL_TimerSyncConfig_t structure containing the configuration parameters for timer sync. The RAIL_TimerSyncConfig_t::sleep field is ignored in this call.

This function is optional to implement.

This function is called during RAIL_ConfigSleep to allow an application to configure the PRS and RTCC channels used for timer sync to values other than their defaults. The default channels are populated in timerSyncConfig and can be overwritten by the application. If this function is not implemented by the application, a default implementation from within the RAIL library will be used that simply maintains the default channel values in timerSyncConfig.

If an unsupported channel is selected by the application, RAIL_ConfigSleep will return RAIL_STATUS_INVALID_PARAMETER.

```
void RAILCb_ConfigSleepTimerSync(RAIL_TimerSyncConfig_t *timerSyncConfig)
{
  timerSyncConfig->prsChannel = MY_TIMERSYNC_PRS_CHANNEL;
  timerSyncConfig->rtccChannel = MY_TIMERSYNC_RTCC_CHANNEL;
}
```

Definition at line 1398 of file common/rail.h

RAIL_ConfigSleep

> RAIL_Status_t RAIL_ConfigSleep (RAIL_Handle_t railHandle, RAIL_SleepConfig_t sleepConfig)

Initialize RAIL timer synchronization.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	sleepConfig	A sleep configuration.

Returns

- Status code indicating success of the function call.

Definition at line 1408 of file common/rail.h

RAIL_ConfigSleepAlt

> RAIL_Status_t RAIL_ConfigSleepAlt (RAIL_Handle_t railHandle, RAIL_TimerSyncConfig_t *syncConfig)

Initialize RAIL timer synchronization.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	syncConfig	A pointer to the timer synchronization configuration.

The default structure used to enable timer synchronization across sleep is RAIL_TIMER_SYNC_DEFAULT.

Returns

- Status code indicating success of the function call.

Definition at line 1422 of file common/rail.h

RAIL_Sleep

> RAIL_Status_t RAIL_Sleep (uint16_t wakeupProcessTime, bool *deepSleepAllowed)

Stop the RAIL timer and prepare RAIL for sleep.

Parameters

[in]	wakeupProcessTime	Time in microseconds that the application and hardware need to recover from sleep state.
[out]	deepSleepAllowed	true - system can go to deep sleep. false - system should not go to deep sleep. Deep sleep should be blocked in this case.

Returns

- Status code indicating success of the function call.

Warnings

- The active RAIL configuration must be idle to enable sleep.

Note

- This API must not be called if RAIL Power Manager is initialized.

Definition at line 1441 of file common/rail.h

RAIL_Wake

```
RAIL_Status_t RAIL_Wake (RAIL_Time_t elapsedTime)
```

Wake RAIL from sleep and restart the RAIL timer.

Parameters

[in]	elapsedTime	Add the sleep duration to the RAIL timer before restarting the RAIL timer.

Returns

- Status code indicating success of the function call.

If the timer sync was enabled by RAIL_ConfigSleep, synchronize the RAIL timer using an alternate timer. Otherwise, add elapsedTime to the RAIL timer.

Note

- This API must not be called if RAIL Power Manager is initialized.

Definition at line 1457 of file common/rail.h

RAIL_InitPowerManager

```
RAIL_Status_t RAIL_InitPowerManager (void)
```

Initialize RAIL Power Manager.

Parameters

N/A		

Returns

- Status code indicating success of the function call.

Note

- Call this function only when the application is built and initialized with Power Manager plugin. RAIL will perform timer synchronization, upon transitioning from EM2 or lower to EM1 or higher energy mode or vice-versa, in the Power Manager EM transition callback. Since EM transition callbacks are not called in a deterministic order, it is suggested to not call any RAIL time dependent APIs in an EM transition callback.

Definition at line 1472 of file common/rail.h

RAIL_DeinitPowerManager

```
RAIL_Status_t RAIL_DeinitPowerManager (void)
```

Stop the RAIL Power Manager.

Parameters

N/A		

Returns

- Status code indicating success of the function call.

Note

- The active RAIL configuration must be idle to disable radio power manager and there should be no outstanding requirements by radio power manager.

Definition at line `1483` of file `common/rail.h`

Macro Definition Documentation

RAIL_TIMER_SYNC_PRS_CHANNEL_DEFAULT

```
#define RAIL_TIMER_SYNC_PRS_CHANNEL_DEFAULT
```

Value:

```
(7U)
```

Default PRS channel to use when configuring sleep.

Definition at line `325` of file `chip/efr32/efr32xg1x/rail_chip_specific.h`

RAIL_TIMER_SYNC_RTCC_CHANNEL_DEFAULT

```
#define RAIL_TIMER_SYNC_RTCC_CHANNEL_DEFAULT
```

Value:

```
(0U)
```

Default RTCC channel to use when configuring sleep.

Definition at line `332` of file `chip/efr32/efr32xg1x/rail_chip_specific.h`

RAIL_TIMER_SYNC_DEFAULT

```
#define RAIL_TIMER_SYNC_DEFAULT
```

Value:

```
0   {     \
0     RAIL_TIMER_SYNC_PRS_CHANNEL_DEFAULT, \
0     RAIL_TIMER_SYNC_RTCC_CHANNEL_DEFAULT, \
0     RAIL_SLEEP_CONFIG_TIMERSYNC_ENABLED, \
0   }
```

Default timer synchronization configuration.

Definition at line 336 of file chip/efr32/efr32xg1x/rail_chip_specific.h

RAIL_TimerSyncConfig_t

Channel values used to perform timer sync before and after sleep.

The default value of this structure is provided in the RAIL_TIMER_SYNC_DEFAULT macro.

Public Attributes

uint8_t	prsChannel PRS Channel used for timer sync operations.
uint8_t	rtccChannel RTCC Channel used for timer sync operations.
RAIL_SleepConfig_t	sleep Whether to sync the timer before and after sleeping.

Public Attribute Documentation

prsChannel

```
uint8_t RAIL_TimerSyncConfig_t::prsChannel
```

PRS Channel used for timer sync operations.

Definition at line 446 of file common/rail_types.h

rtccChannel

```
uint8_t RAIL_TimerSyncConfig_t::rtccChannel
```

RTCC Channel used for timer sync operations.

Definition at line 450 of file common/rail_types.h

sleep

```
RAIL_SleepConfig_t RAIL_TimerSyncConfig_t::sleep
```

Whether to sync the timer before and after sleeping.

Definition at line 454 of file common/rail_types.h

State Transitions

State Transitions

Modules

RAIL_StateTiming_t

RAIL_StateTransitions_t

EFR32

Enumerations

enum RAIL_RadioState_t {

```
RAIL_RF_STATE_INACTIVE = 0u
RAIL_RF_STATE_ACTIVE = (1u << 0)
RAIL_RF_STATE_RX = (1u << 1)
RAIL_RF_STATE_TX = (1u << 2)
RAIL_RF_STATE_IDLE = (RAIL_RF_STATE_ACTIVE)
RAIL_RF_STATE_RX_ACTIVE = (RAIL_RF_STATE_RX | RAIL_RF_STATE_ACTIVE)
RAIL_RF_STATE_TX_ACTIVE = (RAIL_RF_STATE_TX | RAIL_RF_STATE_ACTIVE)
```

}

The state of the radio.

enum RAIL_RadioStateDetail_t {

```
RAIL_RF_STATE_DETAIL_IDLE_STATE_SHIFT = 0u
RAIL_RF_STATE_DETAIL_RX_STATE_SHIFT
RAIL_RF_STATE_DETAIL_TX_STATE_SHIFT
RAIL_RF_STATE_DETAIL_TRANSITION_SHIFT
RAIL_RF_STATE_DETAIL_ACTIVE_SHIFT
RAIL_RF_STATE_DETAIL_NO_FRAMES_SHIFT
RAIL_RF_STATE_DETAIL_LBT_SHIFT
```

}

The detailed state of the radio.

enum RAIL_IdleMode_t {

```
RAIL_IDLE
RAIL_IDLE_ABORT
RAIL_IDLE_FORCE_SHUTDOWN
RAIL_IDLE_FORCE_SHUTDOWN_CLEAR_FLAGS
```

}

An enumeration for the different types of supported idle modes.

Typedefs

typedef uint32_t RAIL_TransitionTime_t

Suitable type for the supported transition time range.

Functions

RAIL_Status_t	RAIL_SetRxTransitions(RAIL_Handle_t railHandle, const RAIL_StateTransitions_t *transitions) Configure RAIL automatic state transitions after RX.
RAIL_Status_t	RAIL_GetRxTransitions(RAIL_Handle_t railHandle, RAIL_StateTransitions_t *transitions) Get the current RAIL automatic state transitions after RX.
RAIL_Status_t	RAIL_SetTxTransitions(RAIL_Handle_t railHandle, const RAIL_StateTransitions_t *transitions) Configure RAIL automatic state transitions after TX.
RAIL_Status_t	RAIL_GetTxTransitions(RAIL_Handle_t railHandle, RAIL_StateTransitions_t *transitions) Get the current RAIL automatic state transitions after TX.
RAIL_Status_t	RAIL_SetNextTxRepeat(RAIL_Handle_t railHandle, const RAIL_TxRepeatConfig_t *repeatConfig) Set up automatic repeated transmits after the next transmit.
uint16_t	RAIL_GetTxPacketsRemaining(RAIL_Handle_t railHandle) Get the number of transmits remaining in a repeat operation.
RAIL_Status_t	RAIL_SetStateTiming(RAIL_Handle_t railHandle, RAIL_StateTiming_t *timings) Configure RAIL automatic state transition timing.
void	RAIL_Idle(RAIL_Handle_t railHandle, RAIL_IdleMode_t mode, bool wait) Place the radio into an idle state.
RAIL_RadioState_t	RAIL_GetRadioState(RAIL_Handle_t railHandle) Get the current radio state.
RAIL_RadioStateDetail_t	RAIL_GetRadioStateDetail(RAIL_Handle_t railHandle) Get the detailed current radio state.
RAIL_Status_t	RAIL_EnableCacheSynthCal(RAIL_Handle_t railHandle, bool enable) Enable/disable caching of synth calibration value.

Macros

#define	RAIL_TRANSITION_TIME_KEEP ((RAIL_TransitionTime_t) -1) A value to use in RAIL_StateTiming_t fields when calling RAIL_SetStateTiming() to keep that timing parameter at it current setting.
#define	RAIL_RF_STATE_DETAIL_INACTIVE (0U) Radio is inactive.
#define	RAIL_RF_STATE_DETAIL_IDLE_STATE (1U << RAIL_RF_STATE_DETAIL_IDLE_STATE_SHIFT) Radio is in or headed to the idle state.
#define	RAIL_RF_STATE_DETAIL_RX_STATE (1U << RAIL_RF_STATE_DETAIL_RX_STATE_SHIFT) Radio is in or headed to the receive state.
#define	RAIL_RF_STATE_DETAIL_TX_STATE (1U << RAIL_RF_STATE_DETAIL_TX_STATE_SHIFT) Radio is in or headed to the transmit state.
#define	RAIL_RF_STATE_DETAIL_TRANSITION (1U << RAIL_RF_STATE_DETAIL_TRANSITION_SHIFT) Radio is headed to the idle, receive, or transmit state.
#define	RAIL_RF_STATE_DETAIL_ACTIVE (1U << RAIL_RF_STATE_DETAIL_ACTIVE_SHIFT) Radio is actively transmitting or receiving.
#define	RAIL_RF_STATE_DETAIL_NO_FRAMES (1U << RAIL_RF_STATE_DETAIL_NO_FRAMES_SHIFT) Radio has frame detect disabled.

#define RAIL_RF_STATE_DETAIL_LBT (1U << RAIL_RF_STATE_DETAIL_LBT_SHIFT)
LBT/CSMA operation is currently ongoing.

#define RAIL_RF_STATE_DETAIL_CORE_STATE_MASK undefined
Mask for core radio state bits.

Enumeration Documentation

RAIL_RadioState_t

RAIL_RadioState_t

The state of the radio.

Enumerator	
RAIL_RF_STATE_INACTIVE	Radio is inactive.
RAIL_RF_STATE_ACTIVE	Radio is either idle or, in combination with the RX and TX states, receiving or transmitting a frame.
RAIL_RF_STATE_RX	Radio is in receive.
RAIL_RF_STATE_TX	Radio is in transmit.
RAIL_RF_STATE_IDLE	Radio is idle.
RAIL_RF_STATE_RX_ACTIVE	Radio is actively receiving a frame.
RAIL_RF_STATE_TX_ACTIVE	Radio is actively transmitting a frame.

Definition at line 2652 of file common/rail_types.h

RAIL_RadioStateDetail_t

RAIL_RadioStateDetail_t

The detailed state of the radio.

The three radio state bits RAIL_RF_STATE_DETAIL_IDLE_STATE, RAIL_RF_STATE_DETAIL_RX_STATE, and RAIL_RF_STATE_DETAIL_TX_STATE comprise a set of mutually exclusive core radio states. Only one (or none) of these bits can be set at a time. Otherwise, the value is invalid.

The precise meaning of each of these three core bits, when set, depends on the value of the two bits RAIL_RF_STATE_DETAIL_TRANSITION and RAIL_RF_STATE_DETAIL_ACTIVE. When RAIL_RF_STATE_DETAIL_TRANSITION is set, the radio is transitioning into the core radio state corresponding to the set state bit. When it is clear, the radio is already in the core radio state that corresponds to the set state bit. When RAIL_RF_STATE_DETAIL_ACTIVE is set, the radio is actively transmitting or receiving. When it is clear, the radio is not actively transmitting or receiving. This bit will always be clear when RAIL_RF_STATE_DETAIL_IDLE_STATE is set, and will always be set when RAIL_RF_STATE_DETAIL_TX_STATE is set. Otherwise, the value is invalid.

The bit RAIL_RF_STATE_DETAIL_NO_FRAMES is set if the radio is currently operating with frame detection disabled, and clear otherwise. The bit RAIL_RF_STATE_DETAIL_LBT_SHIFT is set if an LBT/CSMA operation (e.g., performing CCA) is currently ongoing, and clear otherwise.

Enumerator	
RAIL_RF_STATE_DETAIL_IDLE_STATE_SHIFT	Shift position of RAIL_RF_STATE_DETAIL_IDLE_STATE bit.
RAIL_RF_STATE_DETAIL_RX_STATE_SHIFT	Shift position of RAIL_RF_STATE_DETAIL_RX_STATE bit.
RAIL_RF_STATE_DETAIL_TX_STATE_SHIFT	Shift position of RAIL_RF_STATE_DETAIL_TX_STATE bit.
RAIL_RF_STATE_DETAIL_TRANSITION_SHIFT	Shift position of RAIL_RF_STATE_DETAIL_TRANSITION bit.

RAIL_RF_STATE_DETAIL_ACTIVE_SHIFT	Shift position of RAIL_RF_STATE_DETAIL_ACTIVE bit.
RAIL_RF_STATE_DETAIL_NO_FRAMES_SHIFT	Shift position of RAIL_RF_STATE_DETAIL_NO_FRAMES bit.
RAIL_RF_STATE_DETAIL_LBT_SHIFT	Shift position of RAIL_RF_STATE_DETAIL_LBT bit.

Definition at line 2770 of file common/rail_types.h

RAIL_IdleMode_t

> RAIL_IdleMode_t

An enumeration for the different types of supported idle modes.

These vary how quickly and destructively they put the radio into idle.

Enumerator	
RAIL_IDLE	Idle the radio by turning off receive and canceling any future scheduled receive or transmit operations.
RAIL_IDLE_ABORT	Idle the radio by turning off receive and any scheduled events.
RAIL_IDLE_FORCE_SHUTDOWN	Force the radio into a shutdown mode by stopping whatever state is in progress.
RAIL_IDLE_FORCE_SHUTDOWN_CLEAR_FLAGS	Similar to the RAIL_IDLE_FORCE_SHUTDOWN command, however, it will also clear any pending RAIL events related to receive and transmit.

Definition at line 2813 of file common/rail_types.h

Typedef Documentation

RAIL_TransitionTime_t

> RAIL_TransitionTime_t

Suitable type for the supported transition time range.

Refer to platform-specific RAIL_MINIMUM_TRANSITION_US and RAIL_MAXIMUM_TRANSITION_US for the valid range of this type.

Definition at line 2604 of file common/rail_types.h

Function Documentation

RAIL_SetRxTransitions

> RAIL_Status_t RAIL_SetRxTransitions (RAIL_Handle_t railHandle, const RAIL_StateTransitions_t *transitions)

Configure RAIL automatic state transitions after RX.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	transitions	The state transitions to apply after reception.

Returns

Status code indicating success of the function call.

This function fails if unsupported transitions are passed in or if the radio is currently in the RX state. Success can transition to TX, RX, or IDLE, while error can transition to RX or IDLE. The timings of state transitions from the RX state are not guaranteed when packets are longer than 16 seconds on-air.

Definition at line `2194` of file `common/rail.h`

RAIL_GetRxTransitions

```
RAIL_Status_t RAIL_GetRxTransitions (RAIL_Handle_t railHandle, RAIL_StateTransitions_t *transitions)
```

Get the current RAIL automatic state transitions after RX.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	transitions	The state transitions that apply after receive.

Returns

- Status code indicating a success of the function call.

Retrieves the current state transitions after RX and stores them in the transitions argument.

Definition at line `2207` of file `common/rail.h`

RAIL_SetTxTransitions

```
RAIL_Status_t RAIL_SetTxTransitions (RAIL_Handle_t railHandle, const RAIL_StateTransitions_t *transitions)
```

Configure RAIL automatic state transitions after TX.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	transitions	The state transitions to apply after transmission.

Returns

- Status code indicating a success of the function call.

This function fails if unsupported transitions are passed in or if the radio is currently in the TX state. Success and error can each transition to RX or IDLE. For the ability to run repeated transmits, see RAIL_SetNextTxRepeat.

Definition at line `2222` of file `common/rail.h`

RAIL_GetTxTransitions

```
RAIL_Status_t RAIL_GetTxTransitions (RAIL_Handle_t railHandle, RAIL_StateTransitions_t *transitions)
```

Get the current RAIL automatic state transitions after TX.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	transitions	The state transitions that apply after transmission.

Returns

- Status code indicating a success of the function call.

Retrieves the current state transitions after TX and stores them in the transitions argument.

Definition at line 2235 of file common/rail.h

RAIL_SetNextTxRepeat

> RAIL_Status_t RAIL_SetNextTxRepeat (RAIL_Handle_t railHandle, const RAIL_TxRepeatConfig_t *repeatConfig)

Set up automatic repeated transmits after the next transmit.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	repeatConfig	The configuration structure for repeated transmits.

Returns

- Status code indicating a success of the function call.

Repeated transmits will occur after an application-initiated transmit caused by calling one of the Packet Transmit APIs. The repetition will only occur after the first application-initiated transmit after this function is called. Future repeated transmits must be requested by calling this function again.

Each repeated transmit that occurs will have full Packet Trace (PTI) information, and will receive events such as RAIL_EVENT_TX_PACKET_SENT as normal.

If a TX error occurs during the repetition, the process will abort and the TX error transition from RAIL_SetTxTransitions will be used. If the repetition completes successfully, then the TX success transition from RAIL_SetTxTransitions will be used.

Use RAIL_GetTxPacketsRemaining() if need to know how many transmit completion events are expected before the repeating sequence is done, or how many were not performed due to a transmit error.

Any call to RAIL_Idle or RAIL_StopTx will clear the pending repeated transmits. The state will also be cleared by another call to this function. A DMP switch will clear this state only if the initial transmit triggering the repeated transmits has started.

One can change the repeated transmit configuration by re-calling this function with new parameters as long as that occurs prior to calling a Packet Transmit API. Passing a RAIL_TxRepeatConfig_t::iterations count of 0 will prevent the next transmit from repeating.

The application is responsible for populating the transmit data to be used by the repeated transmits via RAIL_SetTxFifo or RAIL_WriteTxFifo. Data will be transmitted from the Transmit FIFO. If the Transmit FIFO does not have sufficient data to transmit, a TX error will be caused and a RAIL_EVENT_TX_UNDERFLOW will occur. In order to avoid an underflow, the application should queue data to be transmitted as early as possible. Consider using RAIL_TX_OPTION_RESEND if the same packet data is to be repeated: then the Transmit FIFO only needs to be set/written once.

Do not call this function after starting a transmit operation via a Packet Transmit API call or before processing the final transmit completion event of a prior transmit. This function will fail to (re)configure the repetition if a transmit of any kind is ongoing, including during the time between an initial transmit and the end of a previously-configured repetition.

Note

- This feature/API is not supported on the EFR32XG1 family of chips. Use the compile time symbol RAIL_SUPPORTS_TX_TO_TX or the runtime call RAIL_SupportsTxToTx() to check whether the platform supports this feature.

Definition at line 2296 of file common/rail.h

RAIL_GetTxPacketsRemaining

```
uint16_t RAIL_GetTxPacketsRemaining (RAIL_Handle_t railHandle)
```

Get the number of transmits remaining in a repeat operation.

Parameters

[in]	railHandle	A RAIL instance handle.

Must only be called from within event callback context when handling one of the RAIL_EVENTS_TX_COMPLETION events.

Returns

- transmits remaining as described below.

If the TX completion event is RAIL_EVENT_TX_PACKET_SENT the returned value indicates how many more such events are expected before the repeat transmit operation is done. Due to interrupt latency and timing, this may be an overcount if greater than 0 but is guaranteed to be accurate when 0.

If the TX completion event is an error, the returned value indicates the number of requested transmits that were not performed. For RAIL_EVENT_TX_ABORTED and RAIL_EVENT_TX_UNDERFLOW the count does not include the failing transmit itself. For the other errors where a transmit never started or was blocked, the count would include the failing transmit, which may be one higher than the configured RAIL_TxRepeatConfig_t::iterations if it was the original transmit that was blocked.

If an infinite repeat was configured, this will return RAIL_TX_REPEAT_INFINITE_ITERATIONS.

Definition at line `2325` of file `common/rail.h`

RAIL_SetStateTiming

```
RAIL_Status_t RAIL_SetStateTiming (RAIL_Handle_t railHandle, RAIL_StateTiming_t *timings)
```

Configure RAIL automatic state transition timing.

Parameters

[in]	railHandle	A RAIL instance handle.
[inout]	timings	The timings used to configure the RAIL state machine. This structure is overwritten with the actual times that were set, if an input timing is invalid.

Returns

- Status code indicating a success of the function call.

The timings given are close to the actual transition time. However, a still uncharacterized software overhead occurs. Also, timings are not always adhered to when using an automatic transition after an error, due to the cleanup required to recover from the error.

Definition at line `2341` of file `common/rail.h`

RAIL_Idle

```
void RAIL_Idle (RAIL_Handle_t railHandle, RAIL_IdleMode_t mode, bool wait)
```

Place the radio into an idle state.

Parameters

[in]	railHandle	A RAIL instance handle.

[in]	mode	The method for shutting down the radio.
[in]	wait	Whether this function should wait for the radio to reach idle before returning.

This function is used to remove the radio from TX and RX states. How these states are left is defined by the mode parameter.

In multiprotocol, this API will also cause the radio to be yielded so that other tasks can be run. See Yielding the Radio for more details.

Definition at line 2358 of file common/rail.h

RAIL_GetRadioState

```
RAIL_RadioState_t RAIL_GetRadioState (RAIL_Handle_t railHandle)
```

Get the current radio state.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- An enumeration for the current radio state.

Returns the state of the radio as a bitmask containing: RAIL_RF_STATE_IDLE, RAIL_RF_STATE_RX, RAIL_RF_STATE_TX, and RAIL_RF_STATE_ACTIVE. RAIL_RF_STATE_IDLE, RAIL_RF_STATE_RX, and RAIL_RF_STATE_TX bits are mutually exclusive. The radio can transition through intermediate states, which are not reported but are instead considered part of the state most closely associated. For example, when the radio is warming up or shutting down the transmitter or receiver, this function returns RAIL_RF_STATE_TX or RAIL_RF_STATE_RX, respectively. When transitioning directly from RX to TX or vice-versa, this function returns the earlier state.

Note

- For a more detailed radio state, see RAIL_GetRadioStateDetail

Definition at line 2382 of file common/rail.h

RAIL_GetRadioStateDetail

```
RAIL_RadioStateDetail_t RAIL_GetRadioStateDetail (RAIL_Handle_t railHandle)
```

Get the detailed current radio state.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- An enumeration for the current detailed radio state.

Returns the state of the radio as a bitmask. The three core radio states IDLE, RX, and TX are represented by mutually exclusive bits RAIL_RF_STATE_DETAIL_IDLE_STATE, RAIL_RF_STATE_DETAIL_RX_STATE, and RAIL_RF_STATE_DETAIL_TX_STATE respectively. If the radio is transitioning between these three states, the returned bitmask will have RAIL_RF_STATE_DETAIL_TRANSITION set along with a bit corresponding to the destination core radio state. If, while in the receive state, the radio is actively receiving a packet, RAIL_RF_STATE_DETAIL_ACTIVE will be set; otherwise, this bit will be clear. If frame detection is disabled, RAIL_RF_STATE_DETAIL_NO_FRAMES in the returned state bitmask will be set; otherwise, this bit will be clear. If the radio is performing an LBT/CSMA operation (e.g., a backoff period) RAIL_RF_STATE_DETAIL_LBT in the returned state bitmask will be set; otherwise, this bit will be clear.

For the most part, the more detailed radio states returned by this API correspond to radio states returned by RAIL_GetRadioState as follows:

RAIL_RadioStateDetail_tRAIL_RadioState_t RAIL_RF_STATE_DETAIL_INACTIVE RAIL_RF_STATE_INACTIVE RAIL_RF_STATE_DETAIL_IDLE_STATE | RAIL_STATE_DETAIL_TRANSITION If RX overflow or leaving RX unforced: RAIL_RF_STATE_RX Else if leaving TX unforced: RAIL_RF_STATE_TX Else: RAIL_RF_STATE_IDLE RAIL_RF_STATE_DETAIL_IDLE_STATE RAIL_RF_STATE_IDLE RAIL_RF_STATE_DETAIL_IDLE_STATE | RAIL_STATE_DETAIL_LBT RAIL_RF_STATE_TX RAIL_RF_STATE_DETAIL_RX_STATE | RAIL_STATE_DETAIL_TRANSITION If leaving TX: RAIL_RF_STATE_TX Else: RAIL_RF_STATE_RX RAIL_RF_STATE_DETAIL_RX_STATE | RAIL_RF_STATE_DETAIL_TRANSITION | RAIL_RF_STATE_DETAIL_NO_FRAMES If leaving TX: RAIL_RF_STATE_TX Else: RAIL_RF_STATE_RX RAIL_RF_STATE_DETAIL_RX_STATE RAIL_RF_STATE_RX RAIL_RF_STATE_DETAIL_RX_STATE | RAIL_RF_STATE_DETAIL_NO_FRAMES RAIL_RF_STATE_RX RAIL_RF_STATE_DETAIL_RX_STATE | RAIL_RF_STATE_DETAIL_LBT RAIL_RF_STATE_RX RAIL_RF_STATE_DETAIL_RX_STATE | RAIL_RF_STATE_DETAIL_NO_FRAMES | RAIL_RF_STATE_DETAIL_LBT RAIL_RF_STATE_RX RAIL_RF_STATE_DETAIL_RX_STATE | RAIL_RF_STATE_DETAIL_ACTIVE RAIL_RF_STATE_RX_ACTIVE RAIL_RF_STATE_DETAIL_TX_STATE | RAIL_RF_STATE_TRANSITION If leaving RX: RAIL_RF_STATE_RX Else: RAIL_RF_STATE_TX RAIL_RF_STATE_DETAIL_TX_STATE | RAIL_RF_STATE_ACTIVE RAIL_RF_STATE_TX_ACTIVE

Definition at line 2448 of file common/rail.h

RAIL_EnableCacheSynthCal

```
RAIL_Status_t RAIL_EnableCacheSynthCal (RAIL_Handle_t railHandle, bool enable)
```

Enable/disable caching of synth calibration value.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	enable	A booloean to enable or disable caching of synth calibration.

Returns

- Status code indicating success of the function call.

Once enabled, the sequencer will start caching synth calibration values for channels and apply them instead of performing calibration on every state transition and channel change. This will increase the transition time for the first time calibration is performed. Subsequent state transitions will be faster. The cache size is 2. RAIL_IEEE802154_SUPPORTS_RX_CHANNEL_SWITCHING internally uses this feature and there is no need to enable/disable it. This function returns RAIL_STATUS_INVALID_STATE if we try to disable it while RAIL_IEEE802154_SUPPORTS_RX_CHANNEL_SWITCHING is enabled.

Note

- This function will improve the minimum timings that can be achieved in RAIL_StateTiming_t::idleToRx, RAIL_StateTiming_t::idleToTx, RAIL_StateTiming_t::rxToTx, RAIL_StateTiming_t::txToRx and RAIL_StateTiming_t::txToTx. A call to RAIL_SetStateTiming() is needed to achieve lower transition times.
- On a protocol switch the cache is cleared, so it is not suitable for applications where a protocol switch happens frequently, like with Dynamic Multiprotocol.

Definition at line 2477 of file common/rail.h

Macro Definition Documentation

RAIL_TRANSITION_TIME_KEEP

```
#define RAIL_TRANSITION_TIME_KEEP
```

Value:

```
((RAIL_TransitionTime_t) -1)
```

A value to use in RAIL_StateTiming_t fields when calling RAIL_SetStateTiming() to keep that timing parameter at it current setting.

Definition at line 2612 of file common/rail_types.h

RAIL_RF_STATE_DETAIL_INACTIVE

```
#define RAIL_RF_STATE_DETAIL_INACTIVE
```

Value:

```
(0U)
```

Radio is inactive.

Definition at line 2788 of file common/rail_types.h

RAIL_RF_STATE_DETAIL_IDLE_STATE

```
#define RAIL_RF_STATE_DETAIL_IDLE_STATE
```

Value:

```
(1U << RAIL_RF_STATE_DETAIL_IDLE_STATE_SHIFT)
```

Radio is in or headed to the idle state.

Definition at line 2790 of file common/rail_types.h

RAIL_RF_STATE_DETAIL_RX_STATE

```
#define RAIL_RF_STATE_DETAIL_RX_STATE
```

Value:

```
(1U << RAIL_RF_STATE_DETAIL_RX_STATE_SHIFT)
```

Radio is in or headed to the receive state.

Definition at line 2792 of file common/rail_types.h

RAIL_RF_STATE_DETAIL_TX_STATE

```
#define RAIL_RF_STATE_DETAIL_TX_STATE
```

Value:

```
(1U << RAIL_RF_STATE_DETAIL_TX_STATE_SHIFT)
```

Radio is in or headed to the transmit state.

Definition at line 2794 of file common/rail_types.h

RAIL_RF_STATE_DETAIL_TRANSITION

```
#define RAIL_RF_STATE_DETAIL_TRANSITION
```

Value:

```
(1U << RAIL_RF_STATE_DETAIL_TRANSITION_SHIFT)
```

Radio is headed to the idle, receive, or transmit state.

Definition at line 2796 of file common/rail_types.h

RAIL_RF_STATE_DETAIL_ACTIVE

```
#define RAIL_RF_STATE_DETAIL_ACTIVE
```

Value:

```
(1U << RAIL_RF_STATE_DETAIL_ACTIVE_SHIFT)
```

Radio is actively transmitting or receiving.

Definition at line 2798 of file common/rail_types.h

RAIL_RF_STATE_DETAIL_NO_FRAMES

```
#define RAIL_RF_STATE_DETAIL_NO_FRAMES
```

Value:

```
(1U << RAIL_RF_STATE_DETAIL_NO_FRAMES_SHIFT)
```

Radio has frame detect disabled.

Definition at line 2800 of file common/rail_types.h

RAIL_RF_STATE_DETAIL_LBT

```
#define RAIL_RF_STATE_DETAIL_LBT
```

Value:

```
(1U << RAIL_RF_STATE_DETAIL_LBT_SHIFT)
```

LBT/CSMA operation is currently ongoing.

Definition at line 2802 of file common/rail_types.h

RAIL_RF_STATE_DETAIL_CORE_STATE_MASK

```
#define RAIL_RF_STATE_DETAIL_CORE_STATE_MASK
```

Value:

```
                        (RAIL_RF_STATE_DETAIL_IDLE_STATE \
                        | RAIL_RF_STATE_DETAIL_RX_STATE \
                        | RAIL_RF_STATE_DETAIL_TX_STATE)
```

Mask for core radio state bits.

Definition at line `2804` of file `common/rail_types.h`

RAIL_StateTiming_t

A timing configuration structure for the RAIL State Machine.

Configure the timings of the radio state transitions for common situations. All of the listed timings are in microseconds. Transitions from an active radio state to idle are not configurable, and will always happen as fast as possible. No timing value can exceed platform-specific RAIL_MAXIMUM_TRANSITION_US. Use RAIL_TRANSITION_TIME_KEEP to keep an existing setting.

For idleToRx, idleToTx, rxToTx, txToRx, and txToTx a value of 0 for the transition time means that the specified transition should happen as fast as possible, even if the timing cannot be as consistent. Otherwise, the timing value cannot be below the platform-specific RAIL_MINIMUM_TRANSITION_US.

For idleToTx, rxToTx, and txToTx setting a longer RAIL_TxPowerConfig_t::rampTime may result in a larger minimum value.

For rxSearchTimeout and txToRxSearchTimeout, there is no minimum value. A value of 0 disables the feature, functioning as an infinite timeout.

Public Attributes

RAIL_TransitionTime_t	idleToRx Transition time from IDLE to RX.
RAIL_TransitionTime_t	txToRx Transition time from TX to RX.
RAIL_TransitionTime_t	idleToTx Transition time from IDLE to TX.
RAIL_TransitionTime_t	rxToTx Transition time from RX packet to TX.
RAIL_TransitionTime_t	rxSearchTimeout Length of time the radio will search for a packet when coming from idle or RX.
RAIL_TransitionTime_t	txToRxSearchTimeout Length of time the radio will search for a packet when coming from TX.
RAIL_TransitionTime_t	txToTx Transition time from TX packet to TX.

Public Attribute Documentation

idleToRx

```
RAIL_TransitionTime_t RAIL_StateTiming_t::idleToRx
```

Transition time from IDLE to RX.

Definition at line 2637 of file common/rail_types.h

txToRx

> RAIL_TransitionTime_t RAIL_StateTiming_t::txToRx

Transition time from TX to RX.

Definition at line `2638` of file `common/rail_types.h`

idleToTx

> RAIL_TransitionTime_t RAIL_StateTiming_t::idleToTx

Transition time from IDLE to TX.

Definition at line `2639` of file `common/rail_types.h`

rxToTx

> RAIL_TransitionTime_t RAIL_StateTiming_t::rxToTx

Transition time from RX packet to TX.

Definition at line `2640` of file `common/rail_types.h`

rxSearchTimeout

> RAIL_TransitionTime_t RAIL_StateTiming_t::rxSearchTimeout

Length of time the radio will search for a packet when coming from idle or RX.

Definition at line `2641` of file `common/rail_types.h`

txToRxSearchTimeout

> RAIL_TransitionTime_t RAIL_StateTiming_t::txToRxSearchTimeout

Length of time the radio will search for a packet when coming from TX.

Definition at line `2643` of file `common/rail_types.h`

txToTx

> RAIL_TransitionTime_t RAIL_StateTiming_t::txToTx

Transition time from TX packet to TX.

Definition at line `2645` of file `common/rail_types.h`

RAIL_StateTransitions_t

Used to specify radio states to transition to on success or failure.

Public Attributes

RAIL_RadioState_t	success Indicate the state the radio should return to after a successful action.
RAIL_RadioState_t	error Indicate the state the radio should return to after an error.

Public Attribute Documentation

success

```
RAIL_RadioState_t RAIL_StateTransitions_t::success
```

Indicate the state the radio should return to after a successful action.

Definition at line 2737 of file common/rail_types.h

error

```
RAIL_RadioState_t RAIL_StateTransitions_t::error
```

Indicate the state the radio should return to after an error.

Definition at line 2741 of file common/rail_types.h

EFR32

EFR32

Macros

#define RAIL_MINIMUM_TRANSITION_US (100U)
The minimum value for a consistent RAIL transition.

#define RAIL_MAXIMUM_TRANSITION_US (1000000U)
The maximum value for a consistent RAIL transition.

Macro Definition Documentation

RAIL_MINIMUM_TRANSITION_US

```
#define RAIL_MINIMUM_TRANSITION_US
```

Value:

```
(100U)
```

The minimum value for a consistent RAIL transition.

Note

- Transitions may need to be slower than this when using longer RAIL_TxPowerConfig_t::rampTime values

Definition at line 356 of file chip/efr32/efr32xg1x/rail_chip_specific.h

RAIL_MAXIMUM_TRANSITION_US

```
#define RAIL_MAXIMUM_TRANSITION_US
```

Value:

```
(1000000U)
```

The maximum value for a consistent RAIL transition.

(_SILICON_LABS_32B_SERIES_1_CONFIG == 1)

Definition at line 365 of file chip/efr32/efr32xg1x/rail_chip_specific.h

System Timing

System Timing

Functionality related to the RAIL timer and general system time.

These functions can be used to get information about the current system time or to manipulate the RAIL timer.

The system time returned by RAIL_GetTime() is in the same timebase that is used throughout RAIL. Any callbacks or structures that provide a timestamp, such as RAIL_RxPacketDetails_t::timeReceived, will use the same timebase as will any APIs that accept an absolute time for scheduling their action. Throughout the documentation, the timebase is referred to as the RAIL timebase. The timebase is currently a value in microseconds from RAIL_Init() time, which means that it will wrap every 1.19 hours. (`(2^32 - 1) / (3600 sec/hr * 1000000 us/sec)`).

The provided timer is hardware-backed and interrupt-driven. It can be used for timing any event in the system, but is especially helpful for timing protocol-based state machines and other systems that interact with the radio. To avoid processing the expiration in interrupt context, leave the cb parameter passed to RAIL_SetTimer() as NULL and poll for expiration with the RAIL_IsTimerExpired() function. See below for an example of the interrupt driven method of interacting with the timer.

```
void timerCb(RAIL_Handle_t cbArg)
{
  // Timer callback action
}

void main(void)
{
  // Initialize RAIL ...

  // Set up a timer for 1 ms from now
  RAIL_SetTimer(railHandle, 1000, RAIL_TIME_RELATIVE, &timerCb);

  // Run main loop
  while(1);
}
```

If multiple software timers are needed to be run off of the one available hardware timer, enable a software multiplexing layer within RAIL using the RAIL_ConfigMultiTimer() function. This will allow you to set up as many timers as you want using the RAIL_*MultiTimer() functions. See the example below for using the multitimer functionality.

```
// Declare timer structures in global space or somewhere that will exist
// until the callback has fired
RAIL_MultiTimer_t tmr1, tmr2;

void timerCb(RAIL_MultiTimer_t *tmr,
             RAIL_Time_t expectedTimeOfEvent,
             void *cbArg)
{
  if (tmr == tmr1) {
    // Timer 1 action
  } else {
    // Timer 2 action
  }
}

void main(void)
{
  // Initialize RAIL ...

  RAIL_ConfigMultiTimer(true);

  // Set up one timer for 1 ms from now and one at time 2000000 in the RAIL
  // timebase
  RAIL_SetMultiTimer(&tmr1, 1000, RAIL_TIME_RELATIVE, &timerCb, NULL);
  RAIL_SetMultiTimer(&tmr2, 2000000, RAIL_TIME_ABSOLUTE, &timerCb, NULL);

  // Run main loop
  while(1);
}
```

Modules

RAIL_MultiTimer_t

RAIL_PacketTimeStamp_t

Enumerations

enum RAIL_TimeMode_t {

RAIL_TIME_ABSOLUTE
RAIL_TIME_DELAY
RAIL_TIME_DISABLED

}

Specify a time offset in RAIL APIs.

enum RAIL_PacketTimePosition_t {

RAIL_PACKET_TIME_INVALID = 0
RAIL_PACKET_TIME_DEFAULT = 1
RAIL_PACKET_TIME_AT_PREAMBLE_START = 2
RAIL_PACKET_TIME_AT_PREAMBLE_START_USED_TOTAL = 3
RAIL_PACKET_TIME_AT_SYNC_END = 4
RAIL_PACKET_TIME_AT_SYNC_END_USED_TOTAL = 5
RAIL_PACKET_TIME_AT_PACKET_END = 6
RAIL_PACKET_TIME_AT_PACKET_END_USED_TOTAL = 7
RAIL_PACKET_TIME_COUNT

}

The available packet timestamp position choices.

Typedefs

typedef uint32_t RAIL_Time_t
Time in microseconds.

typedef void(* RAIL_TimerCallback_t)(RAIL_Handle_t cbArg)
A pointer to the callback called when the RAIL timer expires.

typedef void(* RAIL_MultiTimerCallback_t)(struct RAIL_MultiTimer *tmr, RAIL_Time_t expectedTimeOfEvent, void *cbArg)
Callback fired when timer expires.

Functions

RAIL_Time_t RAIL_GetTime(void)
Get the current RAIL time.

RAIL_Status_t RAIL_SetTime(RAIL_Time_t time)
Set the current RAIL time.

RAIL_Status_t RAIL_DelayUs(RAIL_Time_t microseconds)
Blocking delay routine for a specified number of microseconds.

RAIL_Status_t RAIL_SetTimer(RAIL_Handle_t railHandle, RAIL_Time_t time, RAIL_TimeMode_t mode, RAIL_TimerCallback_t cb)
Schedule a timer to expire using the RAIL timebase.

RAIL_Time_t RAIL_GetTimer(RAIL_Handle_t railHandle)
Return the absolute time that the RAIL timer was configured to expire.

void RAIL_CancelTimer(RAIL_Handle_t railHandle)
Stop the currently scheduled RAIL timer.

bool RAIL_IsTimerExpired(RAIL_Handle_t railHandle)
Check whether the RAIL timer has expired.

bool RAIL_IsTimerRunning(RAIL_Handle_t railHandle)
Check whether the RAIL timer is currently running.

bool RAIL_ConfigMultiTimer(bool enable)
Configure the RAIL software timer feature.

RAIL_Status_t RAIL_SetMultiTimer(RAIL_MultiTimer_t *tmr, RAIL_Time_t expirationTime, RAIL_TimeMode_t expirationMode, RAIL_MultiTimerCallback_t callback, void *cbArg)
Start a multitimer instance.

bool RAIL_CancelMultiTimer(RAIL_MultiTimer_t *tmr)
Stop the currently scheduled RAIL multitimer.

bool RAIL_IsMultiTimerRunning(RAIL_MultiTimer_t *tmr)
Check if a given timer is running.

bool RAIL_IsMultiTimerExpired(RAIL_MultiTimer_t *tmr)
Check if a given timer has expired.

RAIL_Time_t RAIL_GetMultiTimer(RAIL_MultiTimer_t *tmr, RAIL_TimeMode_t timeMode)
Get time left before a given timer instance expires.

Enumeration Documentation

RAIL_TimeMode_t

> RAIL_TimeMode_t

Specify a time offset in RAIL APIs.

Different APIs use the same constants and may provide more specifics about how they're used but the general use for each is described below.

Enumerator	
RAIL_TIME_ABSOLUTE	The time specified is an exact time in the RAIL timebase.
RAIL_TIME_DELAY	The time specified is relative to the current time.
RAIL_TIME_DISABLED	The specified time is invalid and should be ignored.

Definition at line 228 of file common/rail_types.h

RAIL_PacketTimePosition_t

```
RAIL_PacketTimePosition_t
```

The available packet timestamp position choices.

Enumerator	
RAIL_PACKET_TIME_INVALID	Indicate that a timestamp is not to be or was not provided.
RAIL_PACKET_TIME_DEFAULT	Request the choice most expedient for RAIL to calculate, which may depend on the radio and/or its configuration.
RAIL_PACKET_TIME_AT_PREAMBLE_START	Request the timestamp corresponding to the first preamble bit sent or received.
RAIL_PACKET_TIME_AT_PREAMBLE_START_USED_TOTAL	Request the timestamp corresponding to the first preamble bit sent or received.
RAIL_PACKET_TIME_AT_SYNC_END	Request the timestamp corresponding to right after its last SYNC word bit has been sent or received.
RAIL_PACKET_TIME_AT_SYNC_END_USED_TOTAL	Request the timestamp corresponding to right after its last SYNC word bit has been sent or received.
RAIL_PACKET_TIME_AT_PACKET_END	Request the timestamp corresponding to right after its last bit has been sent or received.
RAIL_PACKET_TIME_AT_PACKET_END_USED_TOTAL	Request the timestamp corresponding to right after its last bit has been sent or received.
RAIL_PACKET_TIME_COUNT	A count of the choices in this enumeration.

Definition at line 302 of file common/rail_types.h

Typedef Documentation

RAIL_Time_t

```
RAIL_Time_t
```

Time in microseconds.

Definition at line 197 of file common/rail_types.h

RAIL_TimerCallback_t

```
typedef void(* RAIL_TimerCallback_t) (RAIL_Handle_t cbArg) )(RAIL_Handle_t cbArg)
```

A pointer to the callback called when the RAIL timer expires.

Parameters

[in]	cbArg	The argument passed to the callback.

Definition at line `219` of file `common/rail_types.h`

RAIL_MultiTimerCallback_t

```
RAIL_MultiTimerCallback_t )(struct RAIL_MultiTimer *tmr, RAIL_Time_t expectedTimeOfEvent, void *cbArg)
```

Callback fired when timer expires.

Parameters

[in]	tmr	A pointer to an expired timer.
[in]	expectedTimeOfEvent	An absolute time event fired.
[in]	cbArg	A user-supplied callback argument.

Definition at line `276` of file `common/rail_types.h`

Function Documentation

RAIL_GetTime

```
RAIL_Time_t RAIL_GetTime (void)
```

Get the current RAIL time.

Parameters

N/A		

Returns

- Returns the RAIL timebase in microseconds. Note that this wraps after about 1.19 hours since it's stored in a 32 bit value.

Returns the current time in the RAIL timebase (microseconds). It can be used to compare with packet timestamps or to schedule transmits.

Definition at line `904` of file `common/rail.h`

RAIL_SetTime

```
RAIL_Status_t RAIL_SetTime (RAIL_Time_t time)
```

Set the current RAIL time.

Parameters

[in]	time	Set the RAIL timebase to this value in microseconds.

Warnings

- Use this API only for testing purposes or in very limited circumstances during RAIL Timer Synchronization. Undefined behavior can result by calling it in multiprotocol or when the radio is not idle or timed events are active. Applications using

RAIL_GetTime() may not be designed for discontinuous changes to the RAIL time base.

Returns

- Status code indicating the success of the function call.

Sets the current time in the RAIL timebase in microseconds.

Definition at line 921 of file common/rail.h

RAIL_DelayUs

```
RAIL_Status_t RAIL_DelayUs (RAIL_Time_t microseconds)
```

Blocking delay routine for a specified number of microseconds.

Parameters

[in]	microseconds	Delay duration in microseconds.

Returns

- Status code indicating success of the function call.

Use this RAIL API only for short blocking delays because it has less overhead than calling RAIL_GetTime() in a loop. **Note**

- Passing large delay values may give unpredictable results or trigger the Watchdog reset. Also, this function will start the clocks required for the RAIL timebase if they are not running, except between RAIL_Sleep() and RAIL_Wake() where the timer must remain stopped. Interrupts are not disabled during the delay, so the delay may be longer if an interrupt extends beyond the delay duration.

Definition at line 940 of file common/rail.h

RAIL_SetTimer

```
RAIL_Status_t RAIL_SetTimer (RAIL_Handle_t railHandle, RAIL_Time_t time, RAIL_TimeMode_t mode, RAIL_TimerCallback_t cb)
```

Schedule a timer to expire using the RAIL timebase.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	time	The timer's expiration time in the RAIL timebase.
[in]	mode	Indicates whether the time argument is an absolute RAIL time or relative to the current RAIL time. Specifying mode RAIL_TIME_DISABLED is the same as calling RAIL_CancelTimer().
[in]	cb	The callback for RAIL to call when the timer expires.

Returns

- RAIL_STATUS_NO_ERROR on success and RAIL_STATUS_INVALID_PARAMETER if the timer can't be scheduled.

Configures a timer to expire after a period in the RAIL timebase. This timer can be used to implement low-level protocol features.

Warnings

- Attempting to schedule the timer when it is still running from a previous request is bad practice, unless the cb callback is identical to that used in the previous request, in which case the timer is rescheduled to the new time. Note that if the original

timer expires as it is being rescheduled, the callback may or may not occur. It is generally good practice to cancel a running timer before rescheduling it to minimize ambiguity.

Definition at line `965` of file `common/rail.h`

RAIL_GetTimer

```
RAIL_Time_t RAIL_GetTimer (RAIL_Handle_t railHandle)
```

Return the absolute time that the RAIL timer was configured to expire.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- The absolute time that this timer was set to expire.

Provides the absolute time regardless of the RAIL_TimeMode_t that was passed into RAIL_SetTimer. Note that the time might be in the past if the timer has already expired. The return value is undefined if the timer was never set.

Definition at line `981` of file `common/rail.h`

RAIL_CancelTimer

```
void RAIL_CancelTimer (RAIL_Handle_t railHandle)
```

Stop the currently scheduled RAIL timer.

Parameters

[in]	railHandle	A RAIL instance handle. Cancels the timer. If this function is called before the timer expires, the cb callback specified in the earlier RAIL_SetTimer() call will never be called.

Definition at line `991` of file `common/rail.h`

RAIL_IsTimerExpired

```
bool RAIL_IsTimerExpired (RAIL_Handle_t railHandle)
```

Check whether the RAIL timer has expired.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- True if the previously scheduled timer has expired and false otherwise.

Polling with this function is an alternative to the callback.

Definition at line `1002` of file `common/rail.h`

RAIL_IsTimerRunning

bool RAIL_IsTimerRunning (RAIL_Handle_t railHandle)

Check whether the RAIL timer is currently running.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Returns true if the timer is running and false if the timer has expired or was never set.

Definition at line `1011` of file `common/rail.h`

RAIL_ConfigMultiTimer

bool RAIL_ConfigMultiTimer (bool enable)

Configure the RAIL software timer feature.

Parameters

[in]	enable	Enables/disables the RAIL multitimer.

Returns

- True if the multitimer was successfully enabled/disabled, false otherwise.

Turning this on will add a software timer layer above the physical RAIL timer so that the user can have as many timers as desired. It is not necessary to call this function if the MultiTimer APIs are not used.

Note

- This function must be called before calling RAIL_SetMultiTimer. This function is a no-op on multiprotocol as this layer is already used under the hood. Do not call this function while the RAIL timer is running. Call RAIL_IsTimerRunning before enabling/disabling the multitimer. If the multitimer is not needed, do not call this function to allow the multitimer code to be dead stripped. If the multitimer is enabled for use, the multitimer and timer APIs can both be used. However, no timer can be in use while this function is being called.

Definition at line `1034` of file `common/rail.h`

RAIL_SetMultiTimer

RAIL_Status_t RAIL_SetMultiTimer (RAIL_MultiTimer_t *tmr, RAIL_Time_t expirationTime, RAIL_TimeMode_t expirationMode, RAIL_MultiTimerCallback_t callback, void *cbArg)

Start a multitimer instance.

Parameters

[inout]	tmr	A pointer to the timer instance to start.
[in]	expirationTime	A time when the timer is set to expire.
[in]	expirationMode	Select mode of expirationTime. See RAIL_TimeMode_t.
[in]	callback	A function to call on timer expiry. See RAIL_MultiTimerCallback_t. NULL is a legal value.
[in]	cbArg	An extra callback function parameter for the user application.

Note

- It is legal to start an already running timer. If this is done, the timer will first be stopped before the new configuration is applied. If expirationTime is 0, the callback is called immediately.

Returns

- RAIL_STATUS_NO_ERROR on success.RAIL_STATUS_INVALID_PARAMETER if tmr has an illegal value or if timeout is in the past.

Definition at line 1058 of file common/rail.h

RAIL_CancelMultiTimer

```
bool RAIL_CancelMultiTimer (RAIL_MultiTimer_t *tmr)
```

Stop the currently scheduled RAIL multitimer.

Parameters

[inout]	tmr	A RAIL timer instance handle.

Returns

- true if the timer was successfully canceled. false if the timer was not running.

Cancels the timer. If this function is called before the timer expires, the cb callback specified in the earlier RAIL_SetTimer() call will never be called.

Definition at line 1077 of file common/rail.h

RAIL_IsMultiTimerRunning

```
bool RAIL_IsMultiTimerRunning (RAIL_MultiTimer_t *tmr)
```

Check if a given timer is running.

Parameters

[in]	tmr	A pointer to the timer structure to query.

Returns

- true if the timer is running. false if the timer is not running.

Definition at line 1088 of file common/rail.h

RAIL_IsMultiTimerExpired

```
bool RAIL_IsMultiTimerExpired (RAIL_MultiTimer_t *tmr)
```

Check if a given timer has expired.

Parameters

[in]	tmr	A pointer to the timer structure to query.

Returns

- true if the timer is expired. false if the timer is running.

Definition at line 1099 of file common/rail.h

RAIL_GetMultiTimer

> RAIL_Time_t RAIL_GetMultiTimer (RAIL_MultiTimer_t *tmr, RAIL_TimeMode_t timeMode)

Get time left before a given timer instance expires.

Parameters

[in]	tmr	A pointer to the timer structure to query.
[in]	timeMode	Indicates how the function provides the time remaining. By choosing RAIL_TimeMode_t::RAIL_TIME_ABSOLUTE, the function returns the absolute expiration time, and by choosing RAIL_TimeMode_t::RAIL_TIME_DELAY, the function returns the amount of time remaining before the timer's expiration.

Returns

- Time left expressed in RAIL's time units. 0 if the soft timer is not running or has already expired.

Definition at line 1116 of file common/rail.h

RAIL_MultiTimer_t

RAIL timer state structure.

This structure is filled out and maintained internally only. The user/application should not alter any elements of this structure.

Public Attributes

RAIL_Time_t	absOffset Absolute time before the next event.
RAIL_Time_t	relPeriodic Relative, periodic time between events; 0 = timer is oneshot.
RAIL_MultiTimerCallback_t	callback A user callback.
void *	cbArg A user callback argument.
struct RAIL_MultiTimer *	next A pointer to the next soft timer structure.
uint8_t	priority A priority of the callback; 0 = highest priority; 255 = lowest.
bool	isRunning Indicates the timer is currently running.
bool	doCallback Indicates the callback needs to run.

Public Attribute Documentation

absOffset

> RAIL_Time_t RAIL_MultiTimer_t::absOffset

Absolute time before the next event.

Definition at line `288` of file `common/rail_types.h`

relPeriodic

> RAIL_Time_t RAIL_MultiTimer_t::relPeriodic

Relative, periodic time between events; 0 = timer is oneshot.

Definition at line `289` of file `common/rail_types.h`

callback

RAIL_MultiTimerCallback_t RAIL_MultiTimer_t::callback

A user callback.

Definition at line 290 of file common/rail_types.h

cbArg

void* RAIL_MultiTimer_t::cbArg

A user callback argument.

Definition at line 291 of file common/rail_types.h

next

struct RAIL_MultiTimer* RAIL_MultiTimer_t::next

A pointer to the next soft timer structure.

Definition at line 292 of file common/rail_types.h

priority

uint8_t RAIL_MultiTimer_t::priority

A priority of the callback; 0 = highest priority; 255 = lowest.

Definition at line 293 of file common/rail_types.h

isRunning

bool RAIL_MultiTimer_t::isRunning

Indicates the timer is currently running.

Definition at line 294 of file common/rail_types.h

doCallback

bool RAIL_MultiTimer_t::doCallback

Indicates the callback needs to run.

Definition at line 295 of file common/rail_types.h

RAIL_PacketTimeStamp_t

Information for calculating and representing a packet timestamp.

Public Attributes

RAIL_Time_t	packetTime Timestamp of the packet in the RAIL timebase.
uint16_t	totalPacketBytes A value specifying the total length in bytes of the packet used when calculating the packetTime requested by the timePosition field.
RAIL_PacketTimePosition_t	timePosition A RAIL_PacketTimePosition_t value specifying the packet position to return in the packetTime field.
RAIL_Time_t	packetDurationUs In RX for EFR32xG25 only : A value specifying the on-air duration of the data packet, starting with the first bit of the PHR (i.e.

Public Attribute Documentation

packetTime

```
RAIL_Time_t RAIL_PacketTimeStamp_t::packetTime
```

Timestamp of the packet in the RAIL timebase.

Definition at line 377 of file common/rail_types.h

totalPacketBytes

```
uint16_t RAIL_PacketTimeStamp_t::totalPacketBytes
```

A value specifying the total length in bytes of the packet used when calculating the packetTime requested by the timePosition field.

This should account for all bytes sent over the air after the Preamble and Sync word(s) including CRC bytes.

Definition at line 384 of file common/rail_types.h

timePosition

```
RAIL_PacketTimePosition_t RAIL_PacketTimeStamp_t::timePosition
```

A RAIL_PacketTimePosition_t value specifying the packet position to return in the packetTime field.

If this is RAIL_PACKET_TIME_DEFAULT, this field will be updated with the actual position corresponding to the packetTime value filled in by a call using this structure.

Definition at line 392 of file common/rail_types.h

packetDurationUs

> RAIL_Time_t RAIL_PacketTimeStamp_t::packetDurationUs

In RX for EFR32xG25 only : A value specifying the on-air duration of the data packet, starting with the first bit of the PHR (i.e.

end of sync word). Preamble and sync word duration are hence excluded.

In Tx for all EFR32 Series 2 except EFR32xG21 : A value specifying the on-air duration of the data packet, starting at the preamble (i.e. includes preamble, sync word, PHR, payload and FCS). This value can be use to compute duty cycles.

At the present time, this field is set to zero for all EFR32 Series 1 and EFR32xG21, and also for transmission of auto-ack.

Definition at line 407 of file common/rail_types.h

TX Channel Hopping

Modules

RAIL_TxChannelHoppingConfigEntry_t

RAIL_TxChannelHoppingConfig_t

Macros

#define RAIL_CHANNEL_HOPPING_BUFFER_SIZE_PER_CHANNEL_WORST_CASE (65U)
The worst-case platform-agnostic static amount of memory needed per channel for channel hopping, measured in 32 bit words, regardless of the size of radio configuration structures.

Macro Definition Documentation

RAIL_CHANNEL_HOPPING_BUFFER_SIZE_PER_CHANNEL_WORST_CASE

```
#define RAIL_CHANNEL_HOPPING_BUFFER_SIZE_PER_CHANNEL_WORST_CASE
```

Value:

```
(65U)
```

The worst-case platform-agnostic static amount of memory needed per channel for channel hopping, measured in 32 bit words, regardless of the size of radio configuration structures.

Definition at line 2937 of file common/rail_types.h

RAIL_TxChannelHoppingConfigEntry_t

Structure that represents one of the channels that is part of a RAIL_TxChannelHoppingConfig_t sequence of channels used in channel hopping.

Public Attributes

uint16_t	channel The channel number to be used for this entry in the channel hopping sequence.
uint8_t	reserved Pad bytes reserved for future use and currently ignored.
uint32_t	delay Idle time in microseconds to wait before transmitting on the channel indicated by this entry.

Public Attribute Documentation

channel

```
uint16_t RAIL_TxChannelHoppingConfigEntry_t::channel
```

The channel number to be used for this entry in the channel hopping sequence.

If this is an invalid channel for the current PHY, the call to RAIL_SetNextTxRepeat() will fail.

Definition at line 2871 of file common/rail_types.h

reserved

```
uint8_t RAIL_TxChannelHoppingConfigEntry_t::reserved[2]
```

Pad bytes reserved for future use and currently ignored.

Definition at line 2875 of file common/rail_types.h

delay

```
uint32_t RAIL_TxChannelHoppingConfigEntry_t::delay
```

Idle time in microseconds to wait before transmitting on the channel indicated by this entry.

Definition at line 2880 of file common/rail_types.h

RAIL_TxChannelHoppingConfig_t

Wrapper struct that will contain the sequence of RAIL_TxChannelHoppingConfigEntry_t that represents the channel sequence to use during TX Channel Hopping.

Public Attributes

uint32_t *	buffer Pointer to contiguous global read-write memory that will be used by RAIL to store channel hopping information throughout its operation.
uint16_t	bufferLength This parameter must be set to the length of the buffer array, in 32 bit words.
uint8_t	numberOfChannels The number of channels in the channel hopping sequence, which is the number of elements in the array that entries points to.
uint8_t	reserved Pad byte reserved for future use and currently ignored.
RAIL_TxChannelHoppingConfigEntry_t *	entries A pointer to the first element of an array of RAIL_TxChannelHoppingConfigEntry_t that represents the channels used during channel hopping.

Public Attribute Documentation

buffer

```
uint32_t* RAIL_TxChannelHoppingConfig_t::buffer
```

Pointer to contiguous global read-write memory that will be used by RAIL to store channel hopping information throughout its operation.

It need not be initialized and applications should never write data anywhere in this buffer.

Note

- the size of this buffer must be at least as large as 3 + RAIL_CHANNEL_HOPPING_BUFFER_SIZE_PER_CHANNEL * numberOfChannels, plus the sum of the sizes of the radioConfigDeltaAdd's of the required channels, plus the size of the radioConfigDeltaSubtract. In the case that one channel appears two or more times in your channel sequence (e.g., 1, 2, 3, 2), you must account for the radio configuration size that number of times (i.e., need to count channel 2's radio configuration size twice for the given example). The buffer is for internal use to the library.

Definition at line `2907` of file `common/rail_types.h`

bufferLength

```
uint16_t RAIL_TxChannelHoppingConfig_t::bufferLength
```

This parameter must be set to the length of the buffer array, in 32 bit words.

This way, during configuration, the software can confirm it's writing within the bounds of the buffer. The configuration API will return an error or trigger RAIL_ASSERT_CHANNEL_HOPPING_BUFFER_TOO_SHORT if bufferLength is insufficient.

Definition at line 2915 of file common/rail_types.h

numberOfChannels

```
uint8_t RAIL_TxChannelHoppingConfig_t::numberOfChannels
```

The number of channels in the channel hopping sequence, which is the number of elements in the array that entries points to.

Definition at line 2920 of file common/rail_types.h

reserved

```
uint8_t RAIL_TxChannelHoppingConfig_t::reserved
```

Pad byte reserved for future use and currently ignored.

Definition at line 2924 of file common/rail_types.h

entries

```
RAIL_TxChannelHoppingConfigEntry_t* RAIL_TxChannelHoppingConfig_t::entries
```

A pointer to the first element of an array of RAIL_TxChannelHoppingConfigEntry_t that represents the channels used during channel hopping.

The length of this array must be numberOfChannels.

Definition at line 2931 of file common/rail_types.h

Thermal Protection

Modules

RAIL_ChipTempConfig_t

RAIL_ChipTempMetrics_t

Functions

RAIL_Status_t RAIL_ConfigThermalProtection(RAIL_Handle_t genericRailHandle, const RAIL_ChipTempConfig_t *chipTempConfig)
Enable or disable the thermal protection if RAIL_SUPPORTS_THERMAL_PROTECTION is defined and update the temperature threshold and cool down hysteresis preventing or allowing transmissions.

RAIL_Status_t RAIL_GetThermalProtection(RAIL_Handle_t genericRailHandle, RAIL_ChipTempConfig_t *chipTempConfig)
Get the current thermal configuration parameter and status.

Macros

#define RAIL_CHIP_TEMP_THRESHOLD_MAX (398U)
Maximum junction temperature in Kelvin.

#define RAIL_CHIP_TEMP_COOLDOWN_DEFAULT (7U)
Default number of Kelvin degrees below threshold needed to allow transmissions.

#define RAIL_CHIP_TEMP_MEASURE_COUNT (3U)
Number of temperature values provided for the chip thermal protection.

Function Documentation

RAIL_ConfigThermalProtection

> RAIL_Status_t RAIL_ConfigThermalProtection (RAIL_Handle_t genericRailHandle, const RAIL_ChipTempConfig_t *chipTempConfig)

Enable or disable the thermal protection if RAIL_SUPPORTS_THERMAL_PROTECTION is defined and update the temperature threshold and cool down hysteresis preventing or allowing transmissions.

Parameters

[in]	genericRailHandle	A generic RAIL instance handle.
[in]	chipTempConfig	A pointer to the struct RAIL_ChipTempConfig_t that contains the configuration to be applied.

When the temperature threshold minus a precise number of degrees specified by the cool down hysteresis parameter is exceeded, any future transmits are blocked until the temperature decreases below that limit. Besides, if the temperature threshold is exceeded, any active transmit is aborted.

By default the threshold is set to RAIL_CHIP_TEMP_THRESHOLD_MAX and the cool down hysteresis is set to RAIL_CHIP_TEMP_COOLDOWN_DEFAULT.

Returns

- Status code indicating the result of the function call. Returns RAIL_STATUS_INVALID_PARAMETER if enable field from RAIL_ChipTempConfig_t is set to false when an EFF is present on the board.

Note

- The thermal protection is automatically enabled when an EFF is present on the board. There is no use in calling this API in this case.

Definition at line 6358 of file common/rail.h

RAIL_GetThermalProtection

```
RAIL_Status_t RAIL_GetThermalProtection (RAIL_Handle_t genericRailHandle, RAIL_ChipTempConfig_t *chipTempConfig)
```

Get the current thermal configuration parameter and status.

Parameters

[in]	genericRailHandle	A generic RAIL instance handle.
[inout]	chipTempConfig	A pointer to the struct RAIL_ChipTempConfig_t that will contain the current configuration.

Returns

- Status code indicating the result of the function call.

Definition at line 6369 of file common/rail.h

Macro Definition Documentation

RAIL_CHIP_TEMP_THRESHOLD_MAX

```
#define RAIL_CHIP_TEMP_THRESHOLD_MAX
```

Value:

```
(398U)
```

Maximum junction temperature in Kelvin.

A margin is subtracted before using it when RAIL_SUPPORTS_THERMAL_PROTECTION is enabled.

Definition at line 5667 of file common/rail_types.h

RAIL_CHIP_TEMP_COOLDOWN_DEFAULT

```
#define RAIL_CHIP_TEMP_COOLDOWN_DEFAULT
```

Value:

```
(7U)
```

Default number of Kelvin degrees below threshold needed to allow transmissions.

Definition at line 5672 of file common/rail_types.h

RAIL_CHIP_TEMP_MEASURE_COUNT

```
#define RAIL_CHIP_TEMP_MEASURE_COUNT
```

Value:

```
(3U)
```

Number of temperature values provided for the chip thermal protection.

Definition at line 5688 of file common/rail_types.h

RAIL_ChipTempConfig_t

Configuration parameters for thermal protection.

Public Attributes

bool	enable Indicates whether the protection is enabled.
uint8_t	coolDownK Mandatory temperature cool down when the threshold is exceeded, in degrees Kelvin.
uint16_t	thresholdK Temperature above which transmit is blocked, in degrees Kelvin.

Public Attribute Documentation

enable

```
bool RAIL_ChipTempConfig_t::enable
```

Indicates whether the protection is enabled.

Definition at line `5680` of file `common/rail_types.h`

coolDownK

```
uint8_t RAIL_ChipTempConfig_t::coolDownK
```

Mandatory temperature cool down when the threshold is exceeded, in degrees Kelvin.

Definition at line `5681` of file `common/rail_types.h`

thresholdK

```
uint16_t RAIL_ChipTempConfig_t::thresholdK
```

Temperature above which transmit is blocked, in degrees Kelvin.

Definition at line `5683` of file `common/rail_types.h`

RAIL_ChipTempMetrics_t

Data used for thermal protection.

Public Attributes

uint16_t	tempK Store chip temperature for metrics.
uint16_t	minTempK Minimum temperature recorded.
uint16_t	maxTempK Maximum temperature recorded.
bool	resetPending Indicates if data should be reset.
uint8_t	reservedChipTemp Reserved for future use.

Public Attribute Documentation

tempK

```
uint16_t RAIL_ChipTempMetrics_t::tempK
```

Store chip temperature for metrics.

Definition at line `5697` of file `common/rail_types.h`

minTempK

```
uint16_t RAIL_ChipTempMetrics_t::minTempK
```

Minimum temperature recorded.

Definition at line `5698` of file `common/rail_types.h`

maxTempK

```
uint16_t RAIL_ChipTempMetrics_t::maxTempK
```

Maximum temperature recorded.

Definition at line `5699` of file `common/rail_types.h`

resetPending

bool RAIL_ChipTempMetrics_t::resetPending

Indicates if data should be reset.

Definition at line 5700 of file common/rail_types.h

reservedChipTemp

uint8_t RAIL_ChipTempMetrics_t::reservedChipTemp

Reserved for future use.

Definition at line 5701 of file common/rail_types.h

Transmit

Transmit

APIs related to transmitting data packets.

Modules

RAIL_TxPacketDetails_t

RAIL_ScheduleTxConfig_t

RAIL_CsmaConfig_t

RAIL_LbtConfig_t

RAIL_SyncWordConfig_t

RAIL_TxRepeatConfig_t

Packet Transmit

Power Amplifier (PA)

Enumerations

enum RAIL_StopMode_t {

RAIL_STOP_MODE_ACTIVE_SHIFT = 0
RAIL_STOP_MODE_PENDING_SHIFT = 1

}

Stop radio operation options bit mask.

enum RAIL_TxOptions_t {

RAIL_TX_OPTION_WAIT_FOR_ACK_SHIFT = 0
RAIL_TX_OPTION_REMOVE_CRC_SHIFT
RAIL_TX_OPTION_SYNC_WORD_ID_SHIFT
RAIL_TX_OPTION_ANTENNA0_SHIFT
RAIL_TX_OPTION_ANTENNA1_SHIFT
RAIL_TX_OPTION_ALT_PREAMBLE_LEN_SHIFT
RAIL_TX_OPTION_CCA_PEAK_RSSI_SHIFT
RAIL_TX_OPTION_CCA_ONLY_SHIFT
RAIL_TX_OPTION_RESEND_SHIFT
RAIL_TX_OPTION_CONCURRENT_PHY_ID_SHIFT
RAIL_TX_OPTIONS_COUNT

}

Transmit options, in reality a bitmask.

enum RAIL_ScheduledTxDuringRx_t {

RAIL_SCHEDULED_TX_DURING_RX_POSTPONE_TX
RAIL_SCHEDULED_TX_DURING_RX_ABORT_TX

}

Enumerates the possible outcomes of what will occur if a scheduled TX ends up firing during RX.

enum RAIL_TxRepeatOptions_t {

```
RAIL_TX_REPEAT_OPTION_HOP_SHIFT = 0
RAIL_TX_REPEAT_OPTION_START_TO_START_SHIFT = 1
```

}

Transmit repeat options, in reality a bitmask.

Functions

RAIL_Status_t RAIL_StopTx(RAIL_Handle_t railHandle, RAIL_StopMode_t mode)

Stop an active or pending transmit.

RAIL_Status_t RAIL_SetCcaThreshold(RAIL_Handle_t railHandle, int8_t ccaThresholdDbm)

Set the CCA threshold in dBm.

RAIL_Status_t RAIL_GetTxPacketDetails(RAIL_Handle_t railHandle, RAIL_TxPacketDetails_t *pPacketDetails)

Get detailed information about the last packet transmitted.

RAIL_Status_t RAIL_GetTxPacketDetailsAlt(RAIL_Handle_t railHandle, bool isAck, RAIL_Time_t *pPacketTime)

Get detailed information about the last packet transmitted.

RAIL_Status_t RAIL_GetTxPacketDetailsAlt2(RAIL_Handle_t railHandle, RAIL_TxPacketDetails_t *pPacketDetails)

Get detailed information about the last packet transmitted.

RAIL_Status_t RAIL_GetTxTimePreambleStart(RAIL_Handle_t railHandle, uint16_t totalPacketBytes, RAIL_Time_t *pPacketTime)

Adjust a RAIL TX completion timestamp to refer to the start of the preamble.

RAIL_Status_t RAIL_GetTxTimePreambleStartAlt(RAIL_Handle_t railHandle, RAIL_TxPacketDetails_t *pPacketDetails)

Adjust a RAIL TX completion timestamp to refer to the start of the preamble.

RAIL_Status_t RAIL_GetTxTimeSyncWordEnd(RAIL_Handle_t railHandle, uint16_t totalPacketBytes, RAIL_Time_t *pPacketTime)

Adjust a RAIL TX timestamp to refer to the end of the sync word.

RAIL_Status_t RAIL_GetTxTimeSyncWordEndAlt(RAIL_Handle_t railHandle, RAIL_TxPacketDetails_t *pPacketDetails)

Adjust a RAIL TX timestamp to refer to the end of the sync word.

RAIL_Status_t RAIL_GetTxTimeFrameEnd(RAIL_Handle_t railHandle, uint16_t totalPacketBytes, RAIL_Time_t *pPacketTime)

Adjust a RAIL TX timestamp to refer to the end of frame.

RAIL_Status_t RAIL_GetTxTimeFrameEndAlt(RAIL_Handle_t railHandle, RAIL_TxPacketDetails_t *pPacketDetails)

Adjust a RAIL TX timestamp to refer to the end of frame.

void RAIL_EnableTxHoldOff(RAIL_Handle_t railHandle, bool enable)

Prevent the radio from starting a transmit.

bool RAIL_IsTxHoldOffEnabled(RAIL_Handle_t railHandle)

Check whether or not TX hold off is enabled.

RAIL_Status_t RAIL_SetTxAltPreambleLength(RAIL_Handle_t railHandle, uint16_t length)

Set an alternate transmitter preamble length.

Macros

#define RAIL_STOP_MODES_NONE (0U)

Do not stop any radio operations.

#define RAIL_STOP_MODE_ACTIVE (1U << RAIL_STOP_MODE_ACTIVE_SHIFT)

Stop active radio operations only.

#define RAIL_STOP_MODE_PENDING (1U << RAIL_STOP_MODE_PENDING_SHIFT)
Stop pending radio operations.

#define RAIL_STOP_MODES_ALL (0xFFU)
Stop all radio operations.

#define RAIL_TX_OPTIONS_NONE 0UL
A value representing no options enabled.

#define RAIL_TX_OPTIONS_DEFAULT RAIL_TX_OPTIONS_NONE
All options disabled by default.

#define RAIL_TX_OPTION_WAIT_FOR_ACK (1UL << RAIL_TX_OPTION_WAIT_FOR_ACK_SHIFT)
An option to configure whether or not the TXing node will listen for an ACK.

#define RAIL_TX_OPTION_REMOVE_CRC (1UL << RAIL_TX_OPTION_REMOVE_CRC_SHIFT)
An option to remove CRC bytes from TX packets.

#define RAIL_TX_OPTION_SYNC_WORD_ID (1UL << RAIL_TX_OPTION_SYNC_WORD_ID_SHIFT)
An option to select which sync word is used during the transmission.

#define RAIL_TX_OPTION_ANTENNA0 (1UL << RAIL_TX_OPTION_ANTENNA0_SHIFT)
An option to select antenna 0 for transmission.

#define RAIL_TX_OPTION_ANTENNA1 (1UL << RAIL_TX_OPTION_ANTENNA1_SHIFT)
An option to select antenna 1 for transmission.

#define RAIL_TX_OPTION_ALT_PREAMBLE_LEN (1UL << RAIL_TX_OPTION_ALT_PREAMBLE_LEN_SHIFT)
An option to use the alternate preamble length established by RAIL_SetTxAltPreambleLength() for the transmission.

#define RAIL_TX_OPTION_CCA_PEAK_RSSI (1UL << RAIL_TX_OPTION_CCA_PEAK_RSSI_SHIFT)
An option to use peak rather than average RSSI energy detected during CSMA's RAIL_CsmaConfig_t::ccaDuration or LBT's RAIL_LbtConfig_t::lbtDuration to determine whether the channel is clear or busy.

#define RAIL_TX_OPTION_CCA_ONLY (1UL << RAIL_TX_OPTION_CCA_ONLY_SHIFT)
An option to only perform the CCA (CSMA/LBT) operation but **not** automatically transmit if the channel is clear.

#define RAIL_TX_OPTION_RESEND (1UL << RAIL_TX_OPTION_RESEND_SHIFT)
An option to resend packet at the beginning of the Transmit FIFO.

#define RAIL_TX_OPTION_CONCURRENT_PHY_ID (1UL << RAIL_TX_OPTION_CONCURRENT_PHY_ID_SHIFT)
An option to specify which PHY is used to transmit in the case of concurrent mode.

#define RAIL_TX_OPTIONS_ALL 0xFFFFFFFFUL
A value representing all possible options.

#define RAIL_MAX_LBT_TRIES (15U)
The maximum number of LBT/CSMA retries supported.

#define RAIL_MAX_CSMA_EXPONENT (8U)
The maximum power-of-2 exponent for CSMA backoffs.

#define RAIL_CSMA_CONFIG_802_15_4_2003_2p4_GHz_OQPSK_CSMA undefined
RAIL_CsmaConfig_t initializer configuring CSMA per IEEE 802.15.4-2003 on 2.4 GHz OSPSK, commonly used by ZigBee.

#define RAIL_CSMA_CONFIG_SINGLE_CCA undefined
RAIL_CsmaConfig_t initializer configuring a single CCA prior to TX.

#define RAIL_LBT_CONFIG_ETSI_EN_300_220_1_V2_4_1 undefined
RAIL_LbtConfig_t initializer configuring LBT per ETSI 300 220-1 V2.4.1 for a typical Sub-GHz band.

#define RAIL_LBT_CONFIG_ETSI_EN_300_220_1_V3_1_0 undefined
RAIL_LbtConfig_t initializer configuring LBT per ETSI 300 220-1 V3.1.0 for a typical Sub-GHz band.

#define RAIL_TX_REPEAT_OPTIONS_NONE 0U
A value representing no repeat options enabled.

#define RAIL_TX_REPEAT_OPTIONS_DEFAULT RAIL_TX_REPEAT_OPTIONS_NONE
All repeat options disabled by default.

#define RAIL_TX_REPEAT_OPTION_HOP (1U << RAIL_TX_REPEAT_OPTION_HOP_SHIFT)
An option to configure whether or not to channel-hop before each repeated transmit.

#define RAIL_TX_REPEAT_OPTION_START_TO_START (1 << RAIL_TX_REPEAT_OPTION_START_TO_START_SHIFT)
An option to configure the delay between transmissions to be from start to start instead of end to start.

#define RAIL_TX_REPEAT_INFINITE_ITERATIONS (0xFFFFU)
RAIL_TxRepeatConfig_t::iterations initializer configuring infinite repeated transmissions.

Enumeration Documentation

RAIL_StopMode_t

> RAIL_StopMode_t

Stop radio operation options bit mask.

Enumerator	
RAIL_STOP_MODE_ACTIVE_SHIFT	Shift position of RAIL_STOP_MODE_ACTIVE bit.
RAIL_STOP_MODE_PENDING_SHIFT	Shift position of RAIL_STOP_MODE_PENDING bit.

Definition at line 2953 of file common/rail_types.h

RAIL_TxOptions_t

> RAIL_TxOptions_t

Transmit options, in reality a bitmask.

Enumerator	
RAIL_TX_OPTION_WAIT_FOR_ACK_SHIFT	Shift position of RAIL_TX_OPTION_WAIT_FOR_ACK bit.
RAIL_TX_OPTION_REMOVE_CRC_SHIFT	Shift position of RAIL_TX_OPTION_REMOVE_CRC bit.
RAIL_TX_OPTION_SYNC_WORD_ID_SHIFT	Shift position of RAIL_TX_OPTION_SYNC_WORD_ID bit.
RAIL_TX_OPTION_ANTENNA0_SHIFT	Shift position of RAIL_TX_OPTION_ANTENNA0 bit.
RAIL_TX_OPTION_ANTENNA1_SHIFT	Shift position of RAIL_TX_OPTION_ANTENNA1 bit.
RAIL_TX_OPTION_ALT_PREAMBLE_LEN_SHIFT	Shift position of RAIL_TX_OPTION_ALT_PREAMBLE_LEN bit.
RAIL_TX_OPTION_CCA_PEAK_RSSI_SHIFT	Shift position of RAIL_TX_OPTION_CCA_PEAK_RSSI bit.
RAIL_TX_OPTION_CCA_ONLY_SHIFT	Shift position of RAIL_TX_OPTION_CCA_ONLY bit.
RAIL_TX_OPTION_RESEND_SHIFT	Shift position of RAIL_TX_OPTION_RESEND bit.
RAIL_TX_OPTION_CONCURRENT_PHY_ID_SHIFT	Shift position of RAIL_TX_OPTION_CONCURRENT_PHY_ID bit.
RAIL_TX_OPTIONS_COUNT	A count of the choices in this enumeration.

Definition at line 2973 of file common/rail_types.h

RAIL_ScheduledTxDuringRx_t

RAIL_ScheduledTxDuringRx_t

Enumerates the possible outcomes of what will occur if a scheduled TX ends up firing during RX.

Because RX and TX can't happen at the same time, it is up to the user how the TX should be handled. This enumeration is passed into RAIL_StartScheduledTx() as part of RAIL_ScheduleTxConfig_t.

Enumerator	
RAIL_SCHEDULED_TX_DURING_RX_POSTPONE_TX	The scheduled TX will be postponed until RX completes and then sent.
RAIL_SCHEDULED_TX_DURING_RX_ABORT_TX	The scheduled TX will be aborted and a RAIL_EVENT_TX_BLOCKED event will fire.

Definition at line 3141 of file common/rail_types.h

RAIL_TxRepeatOptions_t

RAIL_TxRepeatOptions_t

Transmit repeat options, in reality a bitmask.

Enumerator	
RAIL_TX_REPEAT_OPTION_HOP_SHIFT	Shift position of RAIL_TX_REPEAT_OPTION_HOP bit.
RAIL_TX_REPEAT_OPTION_START_TO_START_SHIFT	Shift position of the RAIL_TX_REPEAT_OPTION_START_TO_START bit.

Definition at line 3511 of file common/rail_types.h

Function Documentation

RAIL_StopTx

RAIL_Status_t RAIL_StopTx (RAIL_Handle_t railHandle, RAIL_StopMode_t mode)

Stop an active or pending transmit.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	mode	Configure the types of transmits to stop.

Returns

- RAIL_STATUS_NO_ERROR if the transmit was successfully stopped or RAIL_STATUS_INVALID_STATE if there is no transmit operation to stop.

Note

- When mode includes RAIL_STOP_MODE_ACTIVE, this can also stop an active auto-ACK transmit. When an active transmit is stopped, RAIL_EVENT_TX_ABORTED or RAIL_EVENT_TXACK_ABORTED should occur. When mode includes RAIL_STOP_MODE_PENDING this can also stop a RAIL_TX_OPTION_CCA_ONLY transmit operation. When a pending transmit is stopped, RAIL_EVENT_TX_BLOCKED should occur.

Definition at line 3241 of file common/rail.h

RAIL_SetCcaThreshold

RAIL_Status_t RAIL_SetCcaThreshold (RAIL_Handle_t railHandle, int8_t ccaThresholdDbm)

Set the CCA threshold in dBm.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	ccaThresholdDbm	The CCA threshold in dBm.

Returns

- Status code indicating success of the function call.

Unlike RAIL_StartCcaCsmaTx() or RAIL_StartCcaLbtTx(), which can cause a transmit, this function only modifies the CCA threshold. A possible use case for this function involves setting the CCA threshold to invalid RSSI of -128 which blocks transmission by preventing clear channel assessments from succeeding.

Definition at line 3256 of file common/rail.h

RAIL_GetTxPacketDetails

RAIL_Status_t RAIL_GetTxPacketDetails (RAIL_Handle_t railHandle, RAIL_TxPacketDetails_t *pPacketDetails)

Get detailed information about the last packet transmitted.

Parameters

[in]	railHandle	A RAIL instance handle.
[inout]	pPacketDetails	An application-provided pointer to store RAIL_TxPacketDetails_t corresponding to the transmit event. The isAck and timeSent fields totalPacketBytes and timePosition must be initialized prior to each call: • isAck true to obtain details about the most recent ACK transmit, false to obtain details about the most recent app-initiated transmit. • totalPacketBytes with the total number of bytes of the transmitted packet for RAIL to use when calculating the specified timestamp. This should account for all bytes sent over the air after the Preamble and Sync word(s), including CRC bytes. • timePosition with a RAIL_PacketTimePosition_t value specifying the packet position to put in the timeSent field on return. This field will also be updated with the actual position corresponding to the timeSent value filled in.

Returns

- RAIL_STATUS_NO_ERROR if pPacketDetails was filled in, or an appropriate error code otherwise.

Note

- Consider using RAIL_GetTxPacketDetailsAlt2 for smaller code size.

This function can only be called from callback context for either RAIL_EVENT_TX_PACKET_SENT or RAIL_EVENT_TXACK_PACKET_SENT events.

Definition at line 3286 of file common/rail.h

RAIL_GetTxPacketDetailsAlt

RAIL_Status_t RAIL_GetTxPacketDetailsAlt (RAIL_Handle_t railHandle, bool isAck, RAIL_Time_t *pPacketTime)

Get detailed information about the last packet transmitted.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	isAck	True to obtain details about the most recent ACK transmit. False to obtain details about the most recent app-initiated transmit.
[out]	pPacketTime	An application-provided non-NULL pointer to store a RAIL_Time_t corresponding to the transmit event. This will be populated with a timestamp corresponding to an arbitrary location in the packet. Call RAIL_GetTxTimePreambleStart, RAIL_GetTxTimeSyncWordEnd, or RAIL_GetTxTimeFrameEnd to adjust the timestamp for different locations in the packet.

Returns

- RAIL_STATUS_NO_ERROR if pPacketTime was filled in, or an appropriate error code otherwise.

Note

- Consider using RAIL_GetTxPacketDetailsAlt2 to pass in a RAIL_PacketTimeStamp_t structure instead of a RAIL_Time_t structure, particularly when RAIL_PacketTimePosition_t information is needed or useful.

This function can only be called from callback context for either RAIL_EVENT_TX_PACKET_SENT or RAIL_EVENT_TXACK_PACKET_SENT events.

Definition at line `3313` of file `common/rail.h`

RAIL_GetTxPacketDetailsAlt2

> RAIL_Status_t RAIL_GetTxPacketDetailsAlt2 (RAIL_Handle_t railHandle, RAIL_TxPacketDetails_t *pPacketDetails)

Get detailed information about the last packet transmitted.

Parameters

[in]	railHandle	A RAIL instance handle.
[inout]	pPacketDetails	An application-provided pointer to store RAIL_TxPacketDetails_t corresponding to the transmit event. The isAck must be initialized prior to each call: • isAck true to obtain details about the most recent ACK transmit, false to obtain details about the most recent app-initiated transmit. The timeSent field packetTime will be populated with a timestamp corresponding to a default location in the packet. The timeSent field timePosition will be populated with a RAIL_PacketTimePosition_t value specifying that default packet location. Call RAIL_GetTxTimePreambleStartAlt, RAIL_GetTxTimeSyncWordEndAlt, or RAIL_GetTxTimeFrameEndAlt to adjust the timestamp for different locations in the packet.

Returns

- RAIL_STATUS_NO_ERROR if pPacketDetails was filled in, or an appropriate error code otherwise.

This function can only be called from callback context for either RAIL_EVENT_TX_PACKET_SENT or RAIL_EVENT_TXACK_PACKET_SENT events.

Definition at line `3339` of file `common/rail.h`

RAIL_GetTxTimePreambleStart

RAIL_Status_t RAIL_GetTxTimePreambleStart (RAIL_Handle_t railHandle, uint16_t totalPacketBytes, RAIL_Time_t *pPacketTime)

Adjust a RAIL TX completion timestamp to refer to the start of the preamble.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	totalPacketBytes	The total number of bytes of the transmitted packet for RAIL to use when adjusting the provided timestamp. This should account for all bytes transmitted over the air after the Preamble and Sync word(s), including CRC bytes. Pass RAIL_TX_STARTED_BYTES to retrieve the start-of-normal-TX timestamp (see below).
[inout]	pPacketTime	This points to the RAIL_Time_t returned from a previous call to RAIL_GetTxPacketDetailsAlt for this same packet. That time will be updated with the time that the preamble for this packet started on air. Must be non-NULL.

Also used to retrieve the RAIL_EVENT_TX_STARTED timestamp.

Returns

- RAIL_STATUS_NO_ERROR if pPacketTime was successfully determined or an appropriate error code otherwise.

When used for timestamp adjustment, call this function in the same transmit-complete event-handling context as RAIL_GetTxPacketDetailsAlt() is called.

This function may be called when handling the RAIL_EVENT_TX_STARTED event to retrieve that event's start-of-normal-TX timestamp. (ACK transmits currently have no equivalent event or associated timestamp.) In this case, totalPacketBytes must be RAIL_TX_STARTED_BYTES, and pPacketTime is an output-only parameter filled in with that time (so no need to initialize it beforehand by calling RAIL_GetTxPacketDetailsAlt()).

Definition at line `3374` of file `common/rail.h`

RAIL_GetTxTimePreambleStartAlt

RAIL_Status_t RAIL_GetTxTimePreambleStartAlt (RAIL_Handle_t railHandle, RAIL_TxPacketDetails_t *pPacketDetails)

Adjust a RAIL TX completion timestamp to refer to the start of the preamble.

Parameters

[in]	railHandle	A RAIL instance handle.
[inout]	pPacketDetails	The non-NULL details that were returned from a previous call to RAIL_GetTxPacketDetailsAlt2 for this same packet. The application must update the timeSent field totalPacketBytes to be the total number of bytes of the sent packet for RAIL to use when calculating the specified timestamp. This should account for all bytes transmitted over the air after the Preamble and Sync word(s), including CRC bytes. Pass RAIL_TX_STARTED_BYTES to retrieve the start-of-normal-TX timestamp (see below). After this function, the timeSent field packetTime will be updated with the time that the preamble for this packet started on air.

Also used to retrieve the RAIL_EVENT_TX_STARTED timestamp.

Returns

- RAIL_STATUS_NO_ERROR if the packet time was successfully calculated, or an appropriate error code otherwise.

When used for timestamp adjustment, call this function in the same transmit-complete event-handling context as RAIL_GetTxPacketDetailsAlt2() is called.

This function may be called when handling the RAIL_EVENT_TX_STARTED event to retrieve that event's start-of-normal-TX timestamp. (ACK transmits currently have no equivalent event or associated timestamp.) In this case, the timeSent field totalPacketBytes must be RAIL_TX_STARTED_BYTES, and the timeSent field packetTime is an output-only parameter filled in with that time (so no need to initialize it beforehand by calling RAIL_GetTxPacketDetailsAlt2()).

Definition at line 3409 of file common/rail.h

RAIL_GetTxTimeSyncWordEnd

```
RAIL_Status_t RAIL_GetTxTimeSyncWordEnd (RAIL_Handle_t railHandle, uint16_t totalPacketBytes, RAIL_Time_t *pPacketTime)
```

Adjust a RAIL TX timestamp to refer to the end of the sync word.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	totalPacketBytes	The total number of bytes of the transmitted packet for RAIL to use when calculating the specified timestamp. This should account for all bytes transmitted over the air after the Preamble and Sync word(s), including CRC bytes.
[inout]	pPacketTime	The time that was returned in a RAIL_Time_t from a previous call to RAIL_GetTxPacketDetailsAlt for this same packet. After this function, the time at that location will be updated with the time that the sync word for this packet finished on air. Must be non-NULL.

Returns

- RAIL_STATUS_NO_ERROR if pPacketTime was successfully calculated, or an appropriate error code otherwise.

Call the timestamp adjustment function in the same transmit-complete event-handling context as RAIL_GetTxPacketDetailsAlt() is called.

Definition at line 3432 of file common/rail.h

RAIL_GetTxTimeSyncWordEndAlt

```
RAIL_Status_t RAIL_GetTxTimeSyncWordEndAlt (RAIL_Handle_t railHandle, RAIL_TxPacketDetails_t *pPacketDetails)
```

Adjust a RAIL TX timestamp to refer to the end of the sync word.

Parameters

[in]	railHandle	A RAIL instance handle.
[inout]	pPacketDetails	The non-NULL details that were returned from a previous call to RAIL_GetTxPacketDetailsAlt2 for this same packet. The application must update the timeSent field totalPacketBytes to be the total number of bytes of the sent packet for RAIL to use when calculating the specified timestamp. This should account for all bytes transmitted over the air after the Preamble and Sync word(s), including CRC bytes. Pass RAIL_TX_STARTED_BYTES to retrieve the start-of-normal-TX timestamp (see below). After this function, the timeSent field packetTime will be updated with the time that the sync word for this packet finished on air. Must be non-NULL.

Returns

- RAIL_STATUS_NO_ERROR if the packet time was successfully calculated, or an appropriate error code otherwise.

Call the timestamp adjustment function in the same transmit-complete event-handling context as RAIL_GetTxPacketDetailsAlt2() is called.

Definition at line 3457 of file common/rail.h

RAIL_GetTxTimeFrameEnd

```
RAIL_Status_t RAIL_GetTxTimeFrameEnd (RAIL_Handle_t railHandle, uint16_t totalPacketBytes, RAIL_Time_t *pPacketTime)
```

Adjust a RAIL TX timestamp to refer to the end of frame.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	totalPacketBytes	The total number of bytes of the transmitted packet for RAIL to use when calculating the specified timestamp. This should account for all bytes transmitted over the air after the Preamble and Sync word(s), including CRC bytes.
[inout]	pPacketTime	The time that was returned in a RAIL_Time_t from a previous call to RAIL_GetTxPacketDetailsAlt for this same packet. After this function, the time at that location will be updated with the time that this packet finished on air. Must be non-NULL.

Returns

- RAIL_STATUS_NO_ERROR if pPacketTime was successfully calculated, or an appropriate error code otherwise.

Call the timestamp adjustment function in the same transmit-complete event-handling context as RAIL_GetTxPacketDetailsAlt() is called.

Definition at line 3480 of file common/rail.h

RAIL_GetTxTimeFrameEndAlt

```
RAIL_Status_t RAIL_GetTxTimeFrameEndAlt (RAIL_Handle_t railHandle, RAIL_TxPacketDetails_t *pPacketDetails)
```

Adjust a RAIL TX timestamp to refer to the end of frame.

Parameters

[in]	railHandle	A RAIL instance handle.
[inout]	pPacketDetails	The non-NULL details that were returned from a previous call to RAIL_GetTxPacketDetailsAlt2 for this same packet. The application must update the timeSent field totalPacketBytes to be the total number of bytes of the sent packet for RAIL to use when calculating the specified timestamp. This should account for all bytes transmitted over the air after the Preamble and Sync word(s), including CRC bytes. Pass RAIL_TX_STARTED_BYTES to retrieve the start-of-normal-TX timestamp (see below). After this function, the timeSent field packetTime will be updated with the time that this packet finished on air. Must be non-NULL.

Returns

- RAIL_STATUS_NO_ERROR if the packet time was successfully calculated, or an appropriate error code otherwise.

Call the timestamp adjustment function in the same transmit-complete event-handling context as RAIL_GetTxPacketDetailsAlt2() is called.

Definition at line 3506 of file common/rail.h

RAIL_EnableTxHoldOff

```
void RAIL_EnableTxHoldOff (RAIL_Handle_t railHandle, bool enable)
```

Prevent the radio from starting a transmit.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	enable	Enable/Disable TX hold off.

Enable TX hold off to prevent the radio from starting any transmits. Disable TX hold off to allow the radio to transmit again. Attempting to transmit with the TX hold off enabled will result in RAIL_EVENT_TX_BLOCKED and/or RAIL_EVENT_TXACK_BLOCKED events.

Note

- This function does not affect a transmit that has already started. To stop an already-started transmission, use RAIL_Idle() with RAIL_IDLE_ABORT.

Definition at line 3525 of file common/rail.h

RAIL_IsTxHoldOffEnabled

> bool RAIL_IsTxHoldOffEnabled (RAIL_Handle_t railHandle)

Check whether or not TX hold off is enabled.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- Returns true if TX hold off is enabled, false otherwise.

TX hold off can be enabled/disabled using RAIL_EnableTxHoldOff. Attempting to transmit with the TX hold off enabled will block the transmission and result in RAIL_EVENT_TX_BLOCKED and/or RAIL_EVENT_TXACK_BLOCKED events.

Definition at line 3538 of file common/rail.h

RAIL_SetTxAltPreambleLength

> RAIL_Status_t RAIL_SetTxAltPreambleLength (RAIL_Handle_t railHandle, uint16_t length)

Set an alternate transmitter preamble length.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	length	The desired preamble length, in bits.

Returns

- Status code indicating success of the function call.

To cause a transmission to use this alternate preamble length, specify RAIL_TX_OPTION_ALT_PREAMBLE_LEN in the txOptions parameter passed to the respective RAIL transmit API.

Note

- Attempting to set a preamble length of 0xFFFF bits will result in RAIL_STATUS_INVALID_PARAMETER.

Definition at line 3554 of file common/rail.h

Macro Definition Documentation

RAIL_STOP_MODES_NONE

```
#define RAIL_STOP_MODES_NONE
```

Value:

```
(0U)
```

Do not stop any radio operations.

Definition at line 2961 of file common/rail_types.h

RAIL_STOP_MODE_ACTIVE

```
#define RAIL_STOP_MODE_ACTIVE
```

Value:

```
(1U << RAIL_STOP_MODE_ACTIVE_SHIFT)
```

Stop active radio operations only.

Definition at line 2963 of file common/rail_types.h

RAIL_STOP_MODE_PENDING

```
#define RAIL_STOP_MODE_PENDING
```

Value:

```
(1U << RAIL_STOP_MODE_PENDING_SHIFT)
```

Stop pending radio operations.

Definition at line 2965 of file common/rail_types.h

RAIL_STOP_MODES_ALL

```
#define RAIL_STOP_MODES_ALL
```

Value:

```
(0xFFU)
```

Stop all radio operations.

Definition at line 2967 of file common/rail_types.h

RAIL_TX_OPTIONS_NONE

```
#define RAIL_TX_OPTIONS_NONE
```

Value:

```
0UL
```

A value representing no options enabled.

Definition at line 2999 of file common/rail_types.h

RAIL_TX_OPTIONS_DEFAULT

```
#define RAIL_TX_OPTIONS_DEFAULT
```

Value:

```
RAIL_TX_OPTIONS_NONE
```

All options disabled by default.

This is the fastest TX option to apply.

Definition at line 3001 of file common/rail_types.h

RAIL_TX_OPTION_WAIT_FOR_ACK

```
#define RAIL_TX_OPTION_WAIT_FOR_ACK
```

Value:

```
(1UL << RAIL_TX_OPTION_WAIT_FOR_ACK_SHIFT)
```

An option to configure whether or not the TXing node will listen for an ACK.

If this is false, the isAck flag in RAIL_RxPacketDetails_t of a received packet will always be false.

Definition at line 3007 of file common/rail_types.h

RAIL_TX_OPTION_REMOVE_CRC

```
#define RAIL_TX_OPTION_REMOVE_CRC
```

Value:

```
(1UL << RAIL_TX_OPTION_REMOVE_CRC_SHIFT)
```

An option to remove CRC bytes from TX packets.

To receive packets when the sender has this option set true, set RAIL_RX_OPTION_IGNORE_CRC_ERRORS on the receive side.

Definition at line 3013 of file common/rail_types.h

RAIL_TX_OPTION_SYNC_WORD_ID

```
#define RAIL_TX_OPTION_SYNC_WORD_ID
```

Value:

```
(1UL << RAIL_TX_OPTION_SYNC_WORD_ID_SHIFT)
```

An option to select which sync word is used during the transmission.

When two sync words are configured by the PHY or RAIL_ConfigSyncWords() enabling this option selects SYNC2 rather than SYNC1 for the upcoming transmit.

This option should not be used when only one sync word has been configured.

Note

- There are a few special radio configurations (e.g. BLE Viterbi) that do not support transmitting different sync words.

Definition at line `3024` of file `common/rail_types.h`

RAIL_TX_OPTION_ANTENNA0

```
#define RAIL_TX_OPTION_ANTENNA0
```

Value:

```
(1UL << RAIL_TX_OPTION_ANTENNA0_SHIFT)
```

An option to select antenna 0 for transmission.

If the antenna selection option is not set or if both antenna options are set, then the transmit will occur on either antenna depending on the last receive or transmit selection. This option is only valid on platforms that support Antenna Control and have been configured via RAIL_ConfigAntenna().

Note

- These TX antenna options do not control the antenna used for Auto-ACK transmissions, which always occur on the same antenna used to receive the packet being acknowledged.

Definition at line `3036` of file `common/rail_types.h`

RAIL_TX_OPTION_ANTENNA1

```
#define RAIL_TX_OPTION_ANTENNA1
```

Value:

```
(1UL << RAIL_TX_OPTION_ANTENNA1_SHIFT)
```

An option to select antenna 1 for transmission.

If the antenna selection option is not set or if both antenna options are set, then the transmit will occur on either antenna depending on the last receive or transmit selection. This option is only valid on platforms that support Antenna Control and have been configured via RAIL_ConfigAntenna().

Note

- These TX antenna options do not control the antenna used for Auto-ACK transmissions, which always occur on the same antenna used to receive the packet being acknowledged.

Definition at line 3048 of file common/rail_types.h

RAIL_TX_OPTION_ALT_PREAMBLE_LEN

```
#define RAIL_TX_OPTION_ALT_PREAMBLE_LEN
```

Value:

```
(1UL << RAIL_TX_OPTION_ALT_PREAMBLE_LEN_SHIFT)
```

An option to use the alternate preamble length established by RAIL_SetTxAltPreambleLength() for the transmission.

When not set, the PHY configuration's preamble length is used.

Definition at line 3054 of file common/rail_types.h

RAIL_TX_OPTION_CCA_PEAK_RSSI

```
#define RAIL_TX_OPTION_CCA_PEAK_RSSI
```

Value:

```
(1UL << RAIL_TX_OPTION_CCA_PEAK_RSSI_SHIFT)
```

An option to use peak rather than average RSSI energy detected during CSMA's RAIL_CsmaConfig_t::ccaDuration or LBT's RAIL_LbtConfig_t::lbtDuration to determine whether the channel is clear or busy.

This option is only valid when calling one of the CCA transmit routines: RAIL_StartCcaCsmaTx, RAIL_StartCcaLbtTx, RAIL_StartScheduledCcaCsmaTx, or RAIL_StartScheduledCcaLbtTx.

Note

- This option does nothing on platforms like EFR32XG1 that lack support for capturing peak RSSI energy.

Definition at line 3066 of file common/rail_types.h

RAIL_TX_OPTION_CCA_ONLY

```
#define RAIL_TX_OPTION_CCA_ONLY
```

Value:

```
(1UL << RAIL_TX_OPTION_CCA_ONLY_SHIFT)
```

An option to only perform the CCA (CSMA/LBT) operation but **not** automatically transmit if the channel is clear.

This option is only valid when calling one of the CCA transmit routines: RAIL_StartCcaCsmaTx, RAIL_StartCcaLbtTx, RAIL_StartScheduledCcaCsmaTx, or RAIL_StartScheduledCcaLbtTx.

Application can then use the RAIL_EVENT_TX_CHANNEL_CLEAR to initiate transmit manually, e.g., giving it the opportunity to adjust outgoing packet data before the packet goes out.

Note

- Configured state transitions to Rx or Idle are suspended during this CSMA/LBT operation. If packet reception occurs, the radio will return to the state it was in just prior to the CSMA/LBT operation when that reception (including any AutoACK response) is complete.

Definition at line 3083 of file common/rail_types.h

RAIL_TX_OPTION_RESEND

```
#define RAIL_TX_OPTION_RESEND
```

Value:

```
(1UL << RAIL_TX_OPTION_RESEND_SHIFT)
```

An option to resend packet at the beginning of the Transmit FIFO.

The packet to be resent must have been previously provided by RAIL_SetTxFifo() or RAIL_WriteTxFifo() passing true for the latter's reset parameter. It works by setting the transmit FIFO's read offset to the beginning of the FIFO while leaving its write offset intact. For this to work, RAIL_DataConfig_t::txMethod must be RAIL_DataMethod_t::PACKET_MODE (i.e., the packet can't exceed the Transmit FIFO's size), otherwise undefined behavior will result.

This option can also be used with RAIL_SetNextTxRepeat() to cause the repeated packet(s) to all be the same as the first.

Definition at line 3100 of file common/rail_types.h

RAIL_TX_OPTION_CONCURRENT_PHY_ID

```
#define RAIL_TX_OPTION_CONCURRENT_PHY_ID
```

Value:

```
(1UL << RAIL_TX_OPTION_CONCURRENT_PHY_ID_SHIFT)
```

An option to specify which PHY is used to transmit in the case of concurrent mode.

Concurrent mode is only allowed on EFR32xG25 for some predefined combinations of Wi-SUN PHYs. When set/unset, the alternate/base PHY is used to transmit.

Definition at line 3107 of file common/rail_types.h

RAIL_TX_OPTIONS_ALL

```
#define RAIL_TX_OPTIONS_ALL
```

Value:

```
0xFFFFFFFFUL
```

A value representing all possible options.

Definition at line 3110 of file common/rail_types.h

RAIL_MAX_LBT_TRIES

```
#define RAIL_MAX_LBT_TRIES
```

Value:

```
(15U)
```

The maximum number of LBT/CSMA retries supported.

Definition at line `3187` of file `common/rail_types.h`

RAIL_MAX_CSMA_EXPONENT

```
#define RAIL_MAX_CSMA_EXPONENT
```

Value:

```
(8U)
```

The maximum power-of-2 exponent for CSMA backoffs.

Definition at line `3193` of file `common/rail_types.h`

RAIL_CSMA_CONFIG_802_15_4_2003_2p4_GHz_OQPSK_CSMA

```
#define RAIL_CSMA_CONFIG_802_15_4_2003_2p4_GHz_OQPSK_CSMA
```

Value:

```
0   {                \
0   /* CSMA per 802.15.4-2003 on 2.4 GHz OSPSK, commonly used by ZigBee    */\
0   /* csmaMinBoExp */ 3,  /* 2^3-1 for 0..7 backoffs on 1st try        */\
0   /* csmaMaxBoExp */ 5,  /* 2^5-1 for 0..31 backoffs on 3rd+ tries    */\
0   /* csmaTries    */ 5,  /* 5 tries overall (4 re-tries)             */\
0   /* ccaThreshold */ -75, /* 10 dB above sensitivity                  */\
0   /* ccaBackoff   */ 320, /* 20 symbols at 16 us/symbol               */\
0   /* ccaDuration  */ 128, /* 8 symbols at 16 us/symbol                */\
0   /* csmaTimeout  */ 0,  /* No timeout                               */\
0 }
```

RAIL_CsmaConfig_t initializer configuring CSMA per IEEE 802.15.4-2003 on 2.4 GHz OSPSK, commonly used by ZigBee.

Definition at line `3336` of file `common/rail_types.h`

RAIL_CSMA_CONFIG_SINGLE_CCA

```
#define RAIL_CSMA_CONFIG_SINGLE_CCA
```

Value:

```
0   {                           \
0   /* Perform a single CCA after 'fixed' delay                    */\
0   /* csmaMinBoExp */ 0,  /* Used for fixed backoff               */\
0   /* csmaMaxBoExp */ 0,  /* Used for fixed backoff               */\
0   /* csmaTries    */ 1,  /* Single try                           */\
0   /* ccaThreshold */ -75, /* Override if not desired choice      */\
0   /* ccaBackoff   */ 0,  /* No backoff (override with fixed value)  */\
0   /* ccaDuration  */ 128, /* Override if not desired length     */\
0   /* csmaTimeout  */ 0,  /* no timeout                          */\
```

```
}
```

RAIL_CsmaConfig_t initializer configuring a single CCA prior to TX.

It can be used to as a basis for implementing other channel access schemes with custom backoff delays. Users can override ccaBackoff with a fixed delay on each use.

Definition at line 3354 of file common/rail_types.h

RAIL_LBT_CONFIG_ETSI_EN_300_220_1_V2_4_1

```
#define RAIL_LBT_CONFIG_ETSI_EN_300_220_1_V2_4_1
```

Value:

```
  {                   \
  /* LBT per ETSI 300 220-1 V2.4.1                          */  \
  /* LBT time = random backoff of 0-5 ms in .5 ms increments plus 5 ms fixed */ \
  /* lbtMinBoRand */ 0,   /*                                */  \
  /* lbtMaxBoRand */ 10,  /*                                */  \
  /* lbtTries    */ RAIL_MAX_LBT_TRIES, /* the maximum supported      */  \
  /* lbtThreshold */ -87, /*                                */  \
  /* lbtBackoff  */ 500, /* 0.5 ms                          */  \
  /* lbtDuration */ 5000, /* 5 ms                           */  \
  /* lbtTimeout  */ 0,   /* No timeout (recommend user override)     */  \
}
```

RAIL_LbtConfig_t initializer configuring LBT per ETSI 300 220-1 V2.4.1 for a typical Sub-GHz band.

To be practical, users should override lbtTries and/or lbtTimeout so channel access failure will be reported in a reasonable time frame rather than the unbounded time frame ETSI defined.

Definition at line 3456 of file common/rail_types.h

RAIL_LBT_CONFIG_ETSI_EN_300_220_1_V3_1_0

```
#define RAIL_LBT_CONFIG_ETSI_EN_300_220_1_V3_1_0
```

Value:

```
  {                   \
  /* LBT per ETSI 300 220-1 V3.1.0                          */ \
  /* LBT time = random backoff of 160-4960 us in 160 us increments      */ \
  /* lbtMinBoRand */ 1,   /*                                */ \
  /* lbtMaxBoRand */ 31,  /* app-chosen; 31*lbtBackoff = 4960 us      */ \
  /* lbtTries    */ RAIL_MAX_LBT_TRIES, /* the maximum supported      */ \
  /* lbtThreshold */ -85,  /* 15 dB above Rx sensitivity per Table 45    */ \
  /* lbtBackoff  */ 160,  /* 160 us per Table 48 Minimum CCA interval   */ \
  /* lbtDuration */ 160,  /* 160 us per Table 48 Minimum deferral period */ \
  /* lbtTimeout  */ 0,   /* No timeout (recommend user override)     */ \
}
```

RAIL_LbtConfig_t initializer configuring LBT per ETSI 300 220-1 V3.1.0 for a typical Sub-GHz band.

To be practical, users should override lbtTries and/or lbtTimeout so channel access failure will be reported in a reasonable time frame rather than the unbounded time frame ETSI defined.

Definition at line 3475 of file common/rail_types.h

RAIL_TX_REPEAT_OPTIONS_NONE

```
#define RAIL_TX_REPEAT_OPTIONS_NONE
```

Value:

```
0U
```

A value representing no repeat options enabled.

Definition at line 3519 of file common/rail_types.h

RAIL_TX_REPEAT_OPTIONS_DEFAULT

```
#define RAIL_TX_REPEAT_OPTIONS_DEFAULT
```

Value:

```
RAIL_TX_REPEAT_OPTIONS_NONE
```

All repeat options disabled by default.

Definition at line 3521 of file common/rail_types.h

RAIL_TX_REPEAT_OPTION_HOP

```
#define RAIL_TX_REPEAT_OPTION_HOP
```

Value:

```
(1U << RAIL_TX_REPEAT_OPTION_HOP_SHIFT)
```

An option to configure whether or not to channel-hop before each repeated transmit.

Definition at line 3526 of file common/rail_types.h

RAIL_TX_REPEAT_OPTION_START_TO_START

```
#define RAIL_TX_REPEAT_OPTION_START_TO_START
```

Value:

```
(1 << RAIL_TX_REPEAT_OPTION_START_TO_START_SHIFT)
```

An option to configure the delay between transmissions to be from start to start instead of end to start.

Delay must be long enough to cover the prior transmit's time.

Definition at line 3532 of file common/rail_types.h

RAIL_TX_REPEAT_INFINITE_ITERATIONS

```
#define RAIL_TX_REPEAT_INFINITE_ITERATIONS
```

Value:

(0xFFFFU)

RAIL_TxRepeatConfig_t::iterations initializer configuring infinite repeated transmissions.

Definition at line 3578 of file common/rail_types.h

RAIL_TxPacketDetails_t

Detailed information requested about the packet that was just, or is currently being, transmitted.

Public Attributes

RAIL_PacketTimeStamp_t	timeSent The timestamp of the transmitted packet in the RAIL timebase, filled in by RAIL_GetTxPacketDetails().
bool	isAck Indicate whether the transmitted packet was an automatic ACK.

Public Attribute Documentation

timeSent

```
RAIL_PacketTimeStamp_t RAIL_TxPacketDetails_t::timeSent
```

The timestamp of the transmitted packet in the RAIL timebase, filled in by RAIL_GetTxPacketDetails().

Definition at line 3122 of file common/rail_types.h

isAck

```
bool RAIL_TxPacketDetails_t::isAck
```

Indicate whether the transmitted packet was an automatic ACK.

In a generic sense, an automatic ACK is defined as a packet sent in response to a received ACK-requesting frame when auto-ACK is enabled. In a protocol specific sense this definition may be more or less restrictive to match the specification and you should refer to that protocol's documentation.

Definition at line 3130 of file common/rail_types.h

RAIL_ScheduleTxConfig_t

A configuration structure for a scheduled transmit.

Public Attributes

RAIL_Time_t	when The time when to transmit this packet.
RAIL_TimeMode_t	mode The type of delay.
RAIL_ScheduledTxDuringRx_t	txDuringRx Indicate which action to take with a scheduled TX if it occurs during RX.

Public Attribute Documentation

when

```
RAIL_Time_t RAIL_ScheduleTxConfig_t::when
```

The time when to transmit this packet.

The exact interpretation of this value depends on the mode specified below.

Definition at line 3168 of file common/rail_types.h

mode

```
RAIL_TimeMode_t RAIL_ScheduleTxConfig_t::mode
```

The type of delay.

See the RAIL_TimeMode_t documentation for more information. Be sure to use RAIL_TIME_ABSOLUTE delays for time-critical protocols.

Definition at line 3174 of file common/rail_types.h

txDuringRx

```
RAIL_ScheduledTxDuringRx_t RAIL_ScheduleTxConfig_t::txDuringRx
```

Indicate which action to take with a scheduled TX if it occurs during RX.

See RAIL_ScheduledTxDuringRx_t structure for more information on potential options.

Definition at line 3180 of file common/rail_types.h

RAIL_CsmaConfig_t

A configuration structure for the CSMA transmit algorithm.

One of RAIL's schemes for polite spectrum access is an implementation of a Carrier-Sense Multiple Access (CSMA) algorithm based on IEEE 802.15.4 (unslotted). In pseudo-code it works like this, showing relevant event notifications:

```
// Return true to transmit packet, false to not transmit packet.
bool performCsma(const RAIL_CsmaConfig_t *csmaConfig)
{
  bool isFixedBackoff = ((csmaConfig->csmaMinBoExp == 0)
                         && (csmaConfig->csmaMaxBoExp == 0));
  int backoffExp = csmaConfig->csmaMinBoExp; // Initial backoff exponent
  int backoffMultiplier = 1; // Assume fixed backoff
  int try;

  // Special-case tries == 0 to transmit immediately without backoff+CCA
  if (csmaConfig->csmaTries == 0) {
    return true;
  }

  // Start overall timeout if specified:
  if (csmaConfig->csmaTimeout > 0) {
    StartAbortTimer(csmaConfig->csmaTimeout, RAIL_EVENT_TX_CHANNEL_BUSY);
    // If timeout occurs, abort and signal the indicated event.
  }

  for (try = 0; try < csmaConfig->csmaTries; try++) {
    if (try > 0) {
      signalEvent(RAIL_EVENT_TX_CCA_RETRY);
    }
    // Determine the backoff multipler for this try:
    if (isFixedBackoff) {
      // backoffMultiplier already set to 1 for fixed backoff
    } else {
      // Start with the backoff exponent for this try:
      if (try > 0) {
        backoffExp++;
        if (backoffExp > csmaConfig->csmaMaxBoExp) {
          backoffExp = csmaConfig->csmaMaxBoExp;
        }
      }
      // Pick random multiplier between 0 and 2^backoffExp - 1 inclusive:
      backoffMultiplier = pickRandomInteger(0, (1 << backoffExp) - 1);
    }
    // Perform the backoff:
    delayMicroseconds(backoffMultiplier * csmaConfig->ccaBackoff);
    // Perform the Clear-Channel Assessment (CCA):
    // Channel is considered busy if radio is actively receiving or
    // transmitting, or the average energy detected across duration
    // is above the threshold.
    signalEvent(RAIL_EVENT_TX_START_CCA);
    if (performCca(csmaConfig->ccaDuration, csmaConfig->ccaThreshold)) {
      // CCA (and CSMA) success: Transmit after RX-to-TX turnaround
      StopAbortTimer();
      signalEvent(RAIL_EVENT_TX_CHANNEL_CLEAR);
      return true;
    } else {
      // CCA failed: loop to try again, or exit if out of tries
    }
  }
  // Overall CSMA failure: Don't transmit
  StopAbortTimer();
  signalEvent(RAIL_EVENT_TX_CHANNEL_BUSY);
  return false;
}
```

Public Attributes

uint8_t	csmaMinBoExp The minimum (starting) exponent for CSMA random backoff (2^exp - 1).
uint8_t	csmaMaxBoExp The maximum exponent for CSMA random backoff (2^exp - 1).
uint8_t	csmaTries The number of backoff-then-CCA iterations that can fail before reporting RAIL_EVENT_TX_CHANNEL_BUSY.
int8_t	ccaThreshold The CCA RSSI threshold, in dBm, above which the channel is considered 'busy'.
uint16_t	ccaBackoff The backoff unit period in RAIL's microsecond time base.
uint16_t	ccaDuration The minimum desired CCA check duration in microseconds.
RAIL_Time_t	csmaTimeout An overall timeout, in RAIL's microsecond time base, for the operation.

Public Attribute Documentation

csmaMinBoExp

```
uint8_t RAIL_CsmaConfig_t::csmaMinBoExp
```

The minimum (starting) exponent for CSMA random backoff (2^exp - 1).

It can range from 0 to RAIL_MAX_CSMA_EXPONENT.

Warnings

- On EFR32, due to a hardware limitation, this can only be 0 if csmaMaxBoExp is also 0 specifying a non-random fixed backoff. RAIL_STATUS_INVALID_PARAMETER will result otherwise. If you really want CSMA's first iteration to have no backoff prior to CCA, with subsequent iterations having random backoff as the exponent is increased, you must do a fixed backoff of 0 operation first (csmaMinBoExp = 0, csmaMaxBoExp = 0, ccaBackoff = 0, csmaTries = 1), and if that fails (RAIL_EVENT_TX_CHANNEL_BUSY), follow up with a random backoff operation starting at csmaMinBoExp = 1 for the remaining iterations.

Definition at line 3281 of file common/rail_types.h

csmaMaxBoExp

```
uint8_t RAIL_CsmaConfig_t::csmaMaxBoExp
```

The maximum exponent for CSMA random backoff (2^exp - 1).

It can range from 0 to RAIL_MAX_CSMA_EXPONENT and must be greater than or equal to csmaMinBoExp. If both exponents are 0, a non-random fixed backoff of ccaBackoff duration results.

Definition at line 3289 of file common/rail_types.h

csmaTries

> uint8_t RAIL_CsmaConfig_t::csmaTries

The number of backoff-then-CCA iterations that can fail before reporting RAIL_EVENT_TX_CHANNEL_BUSY.

Typically ranges from 1 to RAIL_MAX_LBT_TRIES; higher values are disallowed. A value 0 always transmits immediately without performing CSMA, similar to calling RAIL_StartTx().

Definition at line 3297 of file common/rail_types.h

ccaThreshold

> int8_t RAIL_CsmaConfig_t::ccaThreshold

The CCA RSSI threshold, in dBm, above which the channel is considered 'busy'.

Definition at line 3302 of file common/rail_types.h

ccaBackoff

> uint16_t RAIL_CsmaConfig_t::ccaBackoff

The backoff unit period in RAIL's microsecond time base.

It is multiplied by the random backoff exponential controlled by csmaMinBoExp and csmaMaxBoExp to determine the overall backoff period. For random backoffs, any value above 511 microseconds will be truncated. For fixed backoffs it can go up to 65535 microseconds.

Definition at line 3310 of file common/rail_types.h

ccaDuration

> uint16_t RAIL_CsmaConfig_t::ccaDuration

The minimum desired CCA check duration in microseconds.

The RSSI is averaged over this duration by default but can be set to use the peak RSSI, on supported platforms, using the RAIL_TX_OPTION_CCA_PEAK_RSSI option.

Note

- Depending on the radio configuration, due to hardware constraints, the actual duration may be longer. Also, if the requested duration is too large for the radio to accommodate, RAIL_StartCcaCsmaTx() will fail returning RAIL_STATUS_INVALID_PARAMETER.

Definition at line 3321 of file common/rail_types.h

csmaTimeout

> RAIL_Time_t RAIL_CsmaConfig_t::csmaTimeout

An overall timeout, in RAIL's microsecond time base, for the operation.

If the transmission doesn't start before this timeout expires, the transmission will fail with RAIL_EVENT_TX_CHANNEL_BUSY. A value 0 means no timeout is imposed.

Definition at line 3328 of file common/rail_types.h

RAIL_LbtConfig_t

A configuration structure for the LBT transmit algorithm.

One of RAIL's schemes for polite spectrum access is an implementation of a Listen-Before-Talk (LBT) algorithm, loosely based on ETSI 300 220-1. Currently, however, it is constrained by the EFR32's CSMA-oriented hardware so is turned into an equivalent RAIL_CsmaConfig_t configuration and passed to the CSMA engine:

```
if (lbtMaxBoRand == lbtMinBoRand) {
  // Fixed backoff
  csmaMinBoExp = csmaMaxBoExp = 0;
  if (lbtMinBoRand == 0) {
    ccaBackoff = lbtBackoff;
  } else {
    ccaBackoff = lbtMinBoRand * lbtBackoff;
  }
  ccaDuration = lbtDuration;
} else {
  // Random backoff: map to random range 0 .. (lbtMaxBoRand - lbtMinBoRand)
  csmaMinBoExp = csmaMaxBoExp = ceiling(log2(lbtMaxBoRand - lbtMinBoRand));
  ccaBackoff = round((lbtBackoff * (lbtMaxBoRand - lbtMinBoRand))
                     / (1 << csmaMinBoExp));
  ccaDuration = lbtDuration + (lbtMinBoRand * lbtBackoff);
}
csmaTries    = lbtTries;
ccaThreshold = lbtThreshold;
csmaTimeout  = lbtTimeout;
```

Public Attributes

uint8_t	lbtMinBoRand The minimum backoff random multiplier.
uint8_t	lbtMaxBoRand The maximum backoff random multiplier.
uint8_t	lbtTries The number of LBT iterations that can fail before reporting RAIL_EVENT_TX_CHANNEL_BUSY.
int8_t	lbtThreshold The LBT RSSI threshold, in dBm, above which the channel is considered 'busy'.
uint16_t	lbtBackoff The backoff unit period, in RAIL's microsecond time base.
uint16_t	lbtDuration The minimum desired LBT check duration in microseconds.
RAIL_Time_t	lbtTimeout An overall timeout, in RAIL's microsecond time base, for the operation.

Public Attribute Documentation

lbtMinBoRand

> uint8_t RAIL_LbtConfig_t::lbtMinBoRand

The minimum backoff random multiplier.

Definition at line 3400 of file common/rail_types.h

lbtMaxBoRand

> uint8_t RAIL_LbtConfig_t::lbtMaxBoRand

The maximum backoff random multiplier.

It must be greater than or equal to lbtMinBoRand. If both backoff multipliers are identical, a non-random fixed backoff of lbtBackoff times the multiplier (minimum 1) duration results.

Definition at line 3407 of file common/rail_types.h

lbtTries

> uint8_t RAIL_LbtConfig_t::lbtTries

The number of LBT iterations that can fail before reporting RAIL_EVENT_TX_CHANNEL_BUSY.

Typically ranges from 1 to RAIL_MAX_LBT_TRIES; higher values are disallowed. A value 0 always transmits immediately without performing LBT, similar to calling RAIL_StartTx().

Definition at line 3415 of file common/rail_types.h

lbtThreshold

> int8_t RAIL_LbtConfig_t::lbtThreshold

The LBT RSSI threshold, in dBm, above which the channel is considered 'busy'.

Definition at line 3420 of file common/rail_types.h

lbtBackoff

> uint16_t RAIL_LbtConfig_t::lbtBackoff

The backoff unit period, in RAIL's microsecond time base.

It is multiplied by the random backoff multiplier controlled by lbtMinBoRand and lbtMaxBoRand to determine the overall backoff period. For random backoffs, any value above 511 microseconds will be truncated. For fixed backoffs, it can go up to 65535 microseconds.

Definition at line 3428 of file common/rail_types.h

lbtDuration

> uint16_t RAIL_LbtConfig_t::lbtDuration

The minimum desired LBT check duration in microseconds.

Note

- Depending on the radio configuration, due to hardware constraints, the actual duration may be longer. Also, if the requested duration is too large for the radio to accommodate, RAIL_StartCcaLbtTx() will fail returning RAIL_STATUS_INVALID_PARAMETER.

Definition at line 3437 of file common/rail_types.h

lbtTimeout

> RAIL_Time_t RAIL_LbtConfig_t::lbtTimeout

An overall timeout, in RAIL's microsecond time base, for the operation.

If the transmission doesn't start before this timeout expires, the transmission will fail with RAIL_EVENT_TX_CHANNEL_BUSY. This is important for limiting LBT due to LBT's unbounded requirement that if the channel is busy, the next try must wait for the channel to clear. A value 0 means no timeout is imposed.

Definition at line 3446 of file common/rail_types.h

RAIL_SyncWordConfig_t

RAIL sync words and length configuration.

Public Attributes

uint8_t	syncWordBits Sync word length in bits, between 2 and 32, inclusive.
uint32_t	syncWord1 Sync Word1.
uint32_t	syncWord2 Sync Word2.

Public Attribute Documentation

syncWordBits

```
uint8_t RAIL_SyncWordConfig_t::syncWordBits
```

Sync word length in bits, between 2 and 32, inclusive.

Definition at line `3494` of file `common/rail_types.h`

syncWord1

```
uint32_t RAIL_SyncWordConfig_t::syncWord1
```

Sync Word1.

Note

- Only syncWordBits number of LS bits are used, which are sent or received on air LSB first.

Definition at line `3499` of file `common/rail_types.h`

syncWord2

```
uint32_t RAIL_SyncWordConfig_t::syncWord2
```

Sync Word2.

Note

- Only syncWordBits number of LS bits are used, which are sent or received on air LSB first.

Definition at line `3504` of file `common/rail_types.h`

RAIL_TxRepeatConfig_t

A configuration structure for repeated transmits.

Note

- The PA will always be ramped down and up in between transmits so there will always be some minimum delay between transmits depending on the ramp time configuration.

Public Attributes

uint16_t	iterations The number of repeated transmits to run.
RAIL_TxRepeatOptions_t	repeatOptions Repeat option(s) to apply.
RAIL_TransitionTime_t	delay When RAIL_TX_REPEAT_OPTION_HOP is not set, specifies the delay time between each repeated transmit.
RAIL_TxChannelHoppingConfig_t	channelHopping When RAIL_TX_REPEAT_OPTION_HOP is set, this specifies the channel hopping configuration to use when hopping between repeated transmits.
union RAIL_TxRepeatConfig_t::@0	delayOrHop Per-repeat delay or hopping configuration, depending on repeatOptions.

Public Attribute Documentation

iterations

```
uint16_t RAIL_TxRepeatConfig_t::iterations
```

The number of repeated transmits to run.

A total of (iterations + 1) transmits will go on-air in the absence of errors.

Definition at line 3545 of file common/rail_types.h

repeatOptions

```
RAIL_TxRepeatOptions_t RAIL_TxRepeatConfig_t::repeatOptions
```

Repeat option(s) to apply.

Definition at line 3549 of file common/rail_types.h

delay

RAIL_TransitionTime_t RAIL_TxRepeatConfig_t::delay

When RAIL_TX_REPEAT_OPTION_HOP is not set, specifies the delay time between each repeated transmit.

Specify RAIL_TRANSITION_TIME_KEEP to use the current RAIL_StateTiming_t::txToTx transition time setting. When using RAIL_TX_REPEAT_OPTION_START_TO_START the delay must be long enough to cover the prior transmit's time.

Definition at line 3562 of file common/rail_types.h

channelHopping

RAIL_TxChannelHoppingConfig_t RAIL_TxRepeatConfig_t::channelHopping

When RAIL_TX_REPEAT_OPTION_HOP is set, this specifies the channel hopping configuration to use when hopping between repeated transmits.

Per-hop delays are configured within each RAIL_TxChannelHoppingConfigEntry_t::delay rather than this union's delay field. When using RAIL_TX_REPEAT_OPTION_START_TO_START the hop delay must be long enough to cover the prior transmit's time.

Definition at line 3572 of file common/rail_types.h

delayOrHop

union RAIL_TxRepeatConfig_t::@0 RAIL_TxRepeatConfig_t::delayOrHop

Per-repeat delay or hopping configuration, depending on repeatOptions.

Definition at line 3573 of file common/rail_types.h

Packet Transmit

Packet Transmit

APIs which initiate a packet transmission in RAIL.

When using any of these functions, the data to be transmitted must have been previously written to the Transmit FIFO via RAIL_SetTxFifo() and/or RAIL_WriteTxFifo().

Functions

RAIL_Status_t	RAIL_StartTx(RAIL_Handle_t railHandle, uint16_t channel, RAIL_TxOptions_t options, const RAIL_SchedulerInfo_t *schedulerInfo) Start a transmit.
RAIL_Status_t	RAIL_StartScheduledTx(RAIL_Handle_t railHandle, uint16_t channel, RAIL_TxOptions_t options, const RAIL_ScheduleTxConfig_t *config, const RAIL_SchedulerInfo_t *schedulerInfo) Schedule sending a packet.
RAIL_Status_t	RAIL_StartCcaCsmaTx(RAIL_Handle_t railHandle, uint16_t channel, RAIL_TxOptions_t options, const RAIL_CsmaConfig_t *csmaConfig, const RAIL_SchedulerInfo_t *schedulerInfo) Start a transmit using CSMA.
RAIL_Status_t	RAIL_StartCcaLbtTx(RAIL_Handle_t railHandle, uint16_t channel, RAIL_TxOptions_t options, const RAIL_LbtConfig_t *lbtConfig, const RAIL_SchedulerInfo_t *schedulerInfo) Start a transmit using LBT.
RAIL_Status_t	RAIL_StartScheduledCcaCsmaTx(RAIL_Handle_t railHandle, uint16_t channel, RAIL_TxOptions_t options, const RAIL_ScheduleTxConfig_t *scheduleTxConfig, const RAIL_CsmaConfig_t *csmaConfig, const RAIL_SchedulerInfo_t *schedulerInfo) Schedule a transmit using CSMA.
RAIL_Status_t	RAIL_StartScheduledCcaLbtTx(RAIL_Handle_t railHandle, uint16_t channel, RAIL_TxOptions_t options, const RAIL_ScheduleTxConfig_t *scheduleTxConfig, const RAIL_LbtConfig_t *lbtConfig, const RAIL_SchedulerInfo_t *schedulerInfo) Schedule a transmit using LBT.

Function Documentation

RAIL_StartTx

> RAIL_Status_t RAIL_StartTx (RAIL_Handle_t railHandle, uint16_t channel, RAIL_TxOptions_t options, const RAIL_SchedulerInfo_t *schedulerInfo)

Start a transmit.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	channel	Define the channel to transmit on.
[in]	options	TX options to be applied to this transmit only.
[in]	schedulerInfo	Information to allow the radio scheduler to place this transmit appropriately. This is only used in multiprotocol version of RAIL and may be set to NULL in all other versions.

Returns

- Status code indicating success of the function call. If successfully initiated, transmit completion or failure will be reported by a later RAIL_Config_t::eventsCallback with the appropriate RAIL_Events_t.

The transmit process will begin immediately or as soon as a packet being received has finished. The data to be transmitted must have been previously established via RAIL_SetTxFifo() and/or RAIL_WriteTxFifo().

Returns an error if a previous transmit is still in progress. If changing channels, any ongoing packet reception is aborted.

In multiprotocol, ensure that the radio is properly yielded after this operation completes. See Yielding the Radio for more details.

Definition at line 2978 of file common/rail.h

RAIL_StartScheduledTx

> RAIL_Status_t RAIL_StartScheduledTx (RAIL_Handle_t railHandle, uint16_t channel, RAIL_TxOptions_t options, const RAIL_ScheduleTxConfig_t *config, const RAIL_SchedulerInfo_t *schedulerInfo)

Schedule sending a packet.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	channel	Define the channel to transmit on.
[in]	options	TX options to be applied to this transmit only.
[in]	config	A pointer to the RAIL_ScheduleTxConfig_t structure containing when the transmit should occur.
[in]	schedulerInfo	Information to allow the radio scheduler to place this transmit appropriately. This is only used in multiprotocol version of RAIL and may be set to NULL in all other versions.

Returns

- Status code indicating success of the function call. If successfully initiated, a transmit completion or failure will be reported by a later RAIL_Config_t::eventsCallback with the appropriate RAIL_Events_t.

The transmit process will begin at the scheduled time. The data to be transmitted must have been previously established via RAIL_SetTxFifo() and/or RAIL_WriteTxFifo(). The time (in microseconds) and whether that time is absolute or relative is specified using the RAIL_ScheduleTxConfig_t structure. What to do if a scheduled transmit fires in the middle of receiving a packet is also specified in this structure.

Returns an error if a previous transmit is still in progress. If changing channels, the channel is changed immediately and will abort any ongoing packet reception.

In multiprotocol, ensure that the radio is properly yielded after this operation completes. See Yielding the Radio for more details.

Definition at line 3013 of file common/rail.h

RAIL_StartCcaCsmaTx

> RAIL_Status_t RAIL_StartCcaCsmaTx (RAIL_Handle_t railHandle, uint16_t channel, RAIL_TxOptions_t options, const RAIL_CsmaConfig_t *csmaConfig, const RAIL_SchedulerInfo_t *schedulerInfo)

Start a transmit using CSMA.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	channel	Define the channel to transmit on.

[in]	options	TX options to be applied to this transmit only.
[in]	csmaConfig	A pointer to the RAIL_CsmaConfig_t structure describing the CSMA parameters to use for this transmit. In multiprotocol this must point to global or heap storage that remains valid after the API returns until the transmit is actually started.
[in]	schedulerInfo	Information to allow the radio scheduler to place this transmit appropriately. This is only used in multiprotocol version of RAIL and may be set to NULL in all other versions.

Returns

- Status code indicating success of the function call. If successfully initiated, a transmit completion or failure will be reported by a later RAIL_Config_t::eventsCallback with the appropriate RAIL_Events_t.

Perform the Carrier Sense Multiple Access (CSMA) algorithm, and if the channel is deemed clear (RSSI below the specified threshold), it will commence transmission. The data to be transmitted must have been previously established via RAIL_SetTxFifo() and/or RAIL_WriteTxFifo(). Packets can be received during CSMA backoff periods if receive is active throughout the CSMA process. This will happen either by starting the CSMA process while receive is already active, or if the csmaBackoff time in the RAIL_CsmaConfig_t is less than the idleToRx time (set by RAIL_SetStateTiming()). If the csmaBackoff time is greater than the idleToRx time, receive will only be active during CSMA's clear channel assessments.

If the CSMA algorithm deems the channel busy, the RAIL_Config_t::eventsCallback occurs with RAIL_EVENT_TX_CHANNEL_BUSY, and the contents of the transmit FIFO remain intact.

Returns an error if a previous transmit is still in progress. If changing channels, the channel is changed immediately and any ongoing packet reception is aborted.

Returns an error if a scheduled RX is still in progress.

In multiprotocol, ensure that the radio is properly yielded after this operation completes. See Yielding the Radio for more details.

Definition at line `3061` of file `common/rail.h`

RAIL_StartCcaLbtTx

```
RAIL_Status_t RAIL_StartCcaLbtTx (RAIL_Handle_t railHandle, uint16_t channel, RAIL_TxOptions_t options, const RAIL_LbtConfig_t *lbtConfig, const RAIL_SchedulerInfo_t *schedulerInfo)
```

Start a transmit using LBT.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	channel	Define the channel to transmit on.
[in]	options	TX options to be applied to this transmit only.
[in]	lbtConfig	A pointer to the RAIL_LbtConfig_t structure describing the LBT parameters to use for this transmit. In multiprotocol this must point to global or heap storage that remains valid after the API returns until the transmit is actually started.
[in]	schedulerInfo	Information to allow the radio scheduler to place this transmit appropriately. This is only used in multiprotocol version of RAIL and may be set to NULL in all other versions.

Returns

- Status code indicating success of the function call. If successfully initiated, a transmit completion or failure will be reported by a later RAIL_Config_t::eventsCallback with the appropriate RAIL_Events_t.

Performs the Listen Before Talk (LBT) algorithm, and if the channel is deemed clear (RSSI below the specified threshold), it will commence transmission. The data to be transmitted must have been previously established via RAIL_SetTxFifo() and/or RAIL_WriteTxFifo(). Packets can be received during LBT backoff periods if receive is active throughout the LBT process. This will happen either by starting the LBT process while receive is already active, or if the lbtBackoff time in the

RAIL_LbtConfig_t is less than the idleToRx time (set by RAIL_SetStateTiming()). If the lbtBackoff time is greater than the idleToRx time, receive will only be active during LBT's clear channel assessments.

If the LBT algorithm deems the channel busy, the RAIL_Config_t::eventsCallback occurs with RAIL_EVENT_TX_CHANNEL_BUSY, and the contents of the transmit FIFO remain intact.

Returns an error if a previous transmit is still in progress. If changing channels, the channel is changed immediately and any ongoing packet reception is aborted.

Returns an error if a scheduled RX is still in progress.

In multiprotocol, ensure that the radio is properly yielded after this operation completes. See Yielding the Radio for more details.

Definition at line `3109` of file `common/rail.h`

RAIL_StartScheduledCcaCsmaTx

```
RAIL_Status_t RAIL_StartScheduledCcaCsmaTx (RAIL_Handle_t railHandle, uint16_t channel, RAIL_TxOptions_t options,
const RAIL_ScheduleTxConfig_t *scheduleTxConfig, const RAIL_CsmaConfig_t *csmaConfig, const RAIL_SchedulerInfo_t
*schedulerInfo)
```

Schedule a transmit using CSMA.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	channel	Define the channel to transmit on.
[in]	options	TX options to be applied to this transmit only.
[in]	scheduleTxConfig	A pointer to the RAIL_ScheduleTxConfig_t structure describing the CSMA parameters to use for this transmit.
[in]	csmaConfig	A pointer to the RAIL_CsmaConfig_t structure describing the CSMA parameters to use for this transmit. In multiprotocol this must point to global or heap storage that remains valid after the API returns until the transmit is actually started.
[in]	schedulerInfo	Information to allow the radio scheduler to place this transmit appropriately. This is only used in multiprotocol version of RAIL and may be set to NULL in all other versions.

Returns

- Status code indicating success of the function call. If successfully initiated, a transmit completion or failure will be reported by a later RAIL_Config_t::eventsCallback with the appropriate RAIL_Events_t.

Internally, the RAIL library needs a PRS channel for this feature. It will allocate an available PRS channel to use and hold onto that channel for future use. If no PRS channel is available, the function returns with RAIL_STATUS_INVALID_CALL.

Perform the Carrier Sense Multiple Access (CSMA) algorithm at the scheduled time, and if the channel is deemed clear (RSSI below the specified threshold), it will commence transmission. The data to be transmitted must have been previously established via RAIL_SetTxFifo() and/or RAIL_WriteTxFifo(). Packets can be received during CSMA backoff periods if receive is active throughout the CSMA process. This will happen either by starting the CSMA process while receive is already active, or if the csmaBackoff time in the RAIL_CsmaConfig_t is less than the idleToRx time (set by RAIL_SetStateTiming()). If the csmaBackoff time is greater than the idleToRx time, receive will only be active during CSMA's clear channel assessments.

If the CSMA algorithm deems the channel busy, the RAIL_Config_t::eventsCallback occurs with RAIL_EVENT_TX_CHANNEL_BUSY, and the contents of the transmit FIFO remain intact.

Returns an error if a previous transmit is still in progress. If changing channels, the channel is changed immediately and any ongoing packet reception is aborted.

In multiprotocol, ensure that the radio is properly yielded after this operation completes. See Yielding the Radio for more details.

Definition at line 3162 of file common/rail.h

RAIL_StartScheduledCcaLbtTx

```
RAIL_Status_t RAIL_StartScheduledCcaLbtTx (RAIL_Handle_t railHandle, uint16_t channel, RAIL_TxOptions_t options, const RAIL_ScheduleTxConfig_t *scheduleTxConfig, const RAIL_LbtConfig_t *lbtConfig, const RAIL_SchedulerInfo_t *schedulerInfo)
```

Schedule a transmit using LBT.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	channel	Define the channel to transmit on.
[in]	options	TX options to be applied to this transmit only.
[in]	scheduleTxConfig	A pointer to the RAIL_ScheduleTxConfig_t structure describing the CSMA parameters to use for this transmit.
[in]	lbtConfig	A pointer to the RAIL_LbtConfig_t structure describing the LBT parameters to use for this transmit. In multiprotocol this must point to global or heap storage that remains valid after the API returns until the transmit is actually started.
[in]	schedulerInfo	Information to allow the radio scheduler to place this transmit appropriately. This is only used in multiprotocol version of RAIL and may be set to NULL in all other versions.

Returns

- Status code indicating success of the function call. If successfully initiated, a transmit completion or failure will be reported by a later RAIL_Config_t::eventsCallback with the appropriate RAIL_Events_t.

Internally, the RAIL library needs a PRS channel for this feature. It will allocate an available PRS channel to use and hold onto that channel for future use. If no PRS channel is available, the function returns with RAIL_STATUS_INVALID_CALL.

Performs the Listen Before Talk (LBT) algorithm at the scheduled time, and if the channel is deemed clear (RSSI below the specified threshold), it will commence transmission. The data to be transmitted must have been previously established via RAIL_SetTxFifo() and/or RAIL_WriteTxFifo(). Packets can be received during LBT backoff periods if receive is active throughout the LBT process. This will happen either by starting the LBT process while receive is already active, or if the lbtBackoff time in the RAIL_LbtConfig_t is less than the idleToRx time (set by RAIL_SetStateTiming()). If the lbtBackoff time is greater than the idleToRx time, receive will only be active during LBT's clear channel assessments.

If the LBT algorithm deems the channel busy, the RAIL_Config_t::eventsCallback occurs with RAIL_EVENT_TX_CHANNEL_BUSY, and the contents of the transmit FIFO remain intact.

Returns an error if a previous transmit is still in progress. If changing channels, the channel is changed immediately and any ongoing packet reception is aborted.

In multiprotocol, ensure that the radio is properly yielded after this operation completes. See Yielding the Radio for more details.

Definition at line 3216 of file common/rail.h

Power Amplifier (PA)

Power Amplifier (PA)

APIs for interacting with one of the on chip PAs.

These APIs let you configure the on-chip PA to get the appropriate output power.

These are the function types: 1) Configuration functions: These functions set and get configuration for the PA. In this case, "configuration" refers to a) indicating which PA to use, b) the voltage supplied by your board to the PA, and c) the ramp time over which to ramp the PA up to its full power. 2) Power-setting functions: These functions consume the actual values written to the PA registers and write them appropriately. These values are referred to as "(raw) power levels". The range of acceptable values for these functions depends on which PA is currently active. The higher the power level set, the higher the dBm power output by the chip. However, the mapping between dBm and these power levels can vary greatly between modules/boards. 3) Conversion functions: These functions convert between the "power levels" discussed previously and the dBm values output by the chip. Continue reading for more information about unit conversion.

The accuracy of the chip output power is application-specific. For some protocols or channels, the protocol itself or legal limitations require applications to know exactly what power they're transmitting at, in dBm. Other applications do not have these restrictions, and users determine power level(s) that fit their criteria for the trade-off between radio range and power savings, regardless of what dBm power that maps to.

RAIL_ConvertRawToDbm and RAIL_ConvertDbmToRaw, which convert between the dBm power and the raw power levels, provide a solution that fits all these applications. The levels of customization are outlined below: 1) No customization needed: for a given dBm value, the result of RAIL_ConvertDbmToRaw provides an appropriate raw power level that, when written to the registers via RAIL_SetPowerLevel, causes the chip to output at that dBm power. In this case, no action is needed by the user, the WEAK versions of the conversion functions can be used and the default include paths in pa_conversions_efr32.h can be used. 2) The mapping of power level to dBm is not ideal, but the level of precision is sufficient: In pa_conversions_efr32.c, the WEAK versions of the conversion functions work by using 8-segment piecewise linear curves to convert between dBm and power levels for PA's with hundreds of power levels and simple mapping tables for use with PA's with only a few levels. If this method is sufficiently precise, but the mapping between power levels and dBm is incorrect, copy pa_curves_efr32.h into a new file, updating the segments to form a better fit (_DCDC_CURVES or _VBAT_CURVES defines) and then add the RAIL_PA_CURVES define to your build with the path to the new file. 3) A different level of precision is needed and the fit is bad: If the piecewise-linear line segment fit is not appropriate for your solution, the functions in pa_conversions_efr32.c can be totally rewritten, as long as RAIL_ConvertDbmToRaw and RAIL_ConvertRawToDbm have the same signatures. It is completely acceptable to re-write these in a way that makes the pa_curves_efr32.h and pa_curve_types_efr32.h files referenced in pa_conversions_efr32.h unnecessary. Those files are needed solely for the provided conversion methods. 4) dBm values are not necessary: If the application does not require dBm values at all, overwrite RAIL_ConvertDbmToRaw and RAIL_ConvertRawToDbm with smaller functions (i.e., return 0 or whatever was input). These functions are called from within the RAIL library, so they can never be deadstripped, but making them as small as possible is the best way to reduce code size. From there, call RAIL_SetTxPower, without converting from a dBm value. To stop the library from coercing the power based on channels, overwrite RAIL_ConvertRawToDbm to always return 0 and overwrite RAIL_ConvertDbmToRaw to always return 255.

The following is example code that shows how to initialize your PA

```
#include "pa_conversions_efr32.h"

// A helper macro to declare all the curve structures used by the provided
// conversion functions.
RAIL_DECLARE_TX_POWER_VBAT_CURVES(piecewiseSegments, curvesSg, curves24Hp, curves24Lp);

// Puts the variables declared above into the appropriate structure.
RAIL_TxPowerCurvesConfig_t txPowerCurvesConfig = { curves24Hp, curvesSg, curves24Lp, piecewiseSegments };

// Saves those curves
// to be referenced when the conversion functions are called.
RAIL_InitTxPowerCurves(&txPowerCurvesConfig);

// Declares the structure used to configure the PA.
RAIL_TxPowerConfig_t txPowerConfig = { RAIL_TX_POWER_MODE_2P4_HP, 3300, 10 };

// Initializes the PA. Here, it is assumed that 'railHandle' is a valid RAIL_Handle_t
// that has already been initialized.
RAIL_ConfigTxPower(railHandle, &txPowerConfig);

// Picks a dBm power to use: 100 deci-dBm = 10 dBm. See docs on RAIL_TxPower_t.
RAIL_TxPower_t power = 100;

// Gets the config written by RAIL_ConfigTxPower to confirm what was actually set.
RAIL_GetTxPowerConfig(railHandle, &txPowerConfig);

// RAIL_ConvertDbmToRaw is the default weak version,
// or the customer version, if overwritten.
RAIL_TxPowerLevel_t powerLevel = RAIL_ConvertDbmToRaw(railHandle,
                                      txPowerConfig.mode,
                                      power);

// Writes the result of the conversion to the PA power registers in terms
// of raw power levels.
RAIL_SetTxPower(railHandle, powerLevel);
```

Note

- All lines following "RAIL_TxPower_t power = 100;" can be replaced with the provided utility function, RAIL_SetTxPowerDbm. However, the full example here was provided for clarity. See the documentation on RAIL_SetTxPowerDbm for more details.

Modules

RAIL_TxPowerConfig_t

RAIL_PaAutoModeConfigEntry_t

EFR32

Enumerations

enum RAIL_TxPowerMode_t {

RAIL_TX_POWER_MODE_2P4GIG_HP = 0U
RAIL_TX_POWER_MODE_2P4_HP = RAIL_TX_POWER_MODE_2P4GIG_HP
RAIL_TX_POWER_MODE_2P4GIG_MP = 1U
RAIL_TX_POWER_MODE_2P4_MP = RAIL_TX_POWER_MODE_2P4GIG_MP
RAIL_TX_POWER_MODE_2P4GIG_LP = 2U
RAIL_TX_POWER_MODE_2P4_LP = RAIL_TX_POWER_MODE_2P4GIG_LP
RAIL_TX_POWER_MODE_2P4GIG_LLP = 3U
RAIL_TX_POWER_MODE_2P4GIG_HIGHEST = 4U
RAIL_TX_POWER_MODE_2P4_HIGHEST = RAIL_TX_POWER_MODE_2P4GIG_HIGHEST
RAIL_TX_POWER_MODE_SUBGIG_POWERSETTING_TABLE = 5U
RAIL_TX_POWER_MODE_SUBGIG_HP = 6U
RAIL_TX_POWER_MODE_SUBGIG = RAIL_TX_POWER_MODE_SUBGIG_HP
RAIL_TX_POWER_MODE_SUBGIG_MP = 7U
RAIL_TX_POWER_MODE_SUBGIG_LP = 8U
RAIL_TX_POWER_MODE_SUBGIG_LLP = 9U
RAIL_TX_POWER_MODE_SUBGIG_HIGHEST = 10U
RAIL_TX_POWER_MODE_OFDM_PA_POWERSETTING_TABLE = 11U
RAIL_TX_POWER_MODE_OFDM_PA = RAIL_TX_POWER_MODE_OFDM_PA_POWERSETTING_TABLE
RAIL_TX_POWER_MODE_NONE

}

An enumeration of the EFR32 power modes.

enum RAIL_PaBand_t {

RAIL_PA_BAND_2P4GIG
RAIL_PA_BAND_SUBGIG
RAIL_PA_BAND_COUNT

}

Enum used to specify the band for a PA.

Typedefs

typedef int16_t RAIL_TxPower_t

The transmit power in deci-dBm units (e.g., 4.5 dBm -> 45 deci-dBm).

typedef uint8_t RAIL_TxPowerLevel_t

Raw power levels used directly by the RAIL_Get/SetTxPower API where a higher numerical value corresponds to a higher output power.

typedef uint32_t RAIL_PaPowerSetting_t

PA power setting used directly by the RAIL_GetPaPowerSetting() and RAIL_SetPaPowerSetting() APIs which is decoded to the actual hardware register value(s).

Variables

const RAIL_PaAutoModeConfigEntry_t * RAIL_PaAutoModeConfig

The actual PA auto mode configuration structure used by the auto mode plugin to control output power.

Functions

RAIL_Status_t RAIL_ConfigTxPower(RAIL_Handle_t railHandle, const RAIL_TxPowerConfig_t *config)

Initialize TX power settings.

RAIL_Status_t RAIL_GetTxPowerConfig(RAIL_Handle_t railHandle, RAIL_TxPowerConfig_t *config)

Get the TX power settings currently used in the amplifier.

RAIL_Status_t	RAIL_SetTxPower(RAIL_Handle_t railHandle, RAIL_TxPowerLevel_t powerLevel) Set the TX power in units of raw units (see rail_chip_specific.h for value ranges).
RAIL_TxPowerLevel_t	RAIL_GetTxPower(RAIL_Handle_t railHandle) Return the current power setting of the PA.
RAIL_TxPower_t	RAIL_ConvertRawToDbm(RAIL_Handle_t railHandle, RAIL_TxPowerMode_t mode, RAIL_TxPowerLevel_t powerLevel) Convert raw values written to registers to decibel value (in units of deci-dBm).
RAIL_TxPowerLevel_t	RAIL_ConvertDbmToRaw(RAIL_Handle_t railHandle, RAIL_TxPowerMode_t mode, RAIL_TxPower_t power) Convert the desired decibel value (in units of deci-dBm) to raw integer values used by the TX amplifier registers.
void	RAIL_VerifyTxPowerCurves(const struct RAIL_TxPowerCurvesConfigAlt *config) Verify the TX Power Curves on modules.
RAIL_Status_t	RAIL_SetTxPowerDbm(RAIL_Handle_t railHandle, RAIL_TxPower_t power) Set the TX power in terms of deci-dBm instead of raw power level.
RAIL_TxPower_t	RAIL_GetTxPowerDbm(RAIL_Handle_t railHandle) Get the TX power in terms of deci-dBm instead of raw power level.
const RAIL_PaPowerSetting_t *	RAIL_GetPowerSettingTable(RAIL_Handle_t railHandle, RAIL_TxPowerMode_t mode, RAIL_TxPower_t *minPower, RAIL_TxPower_t *maxPower, RAIL_TxPowerLevel_t *step) Get the TX PA power setting table and related values.
RAIL_Status_t	RAIL_SetPaPowerSetting(RAIL_Handle_t railHandle, RAIL_PaPowerSetting_t paPowerSetting, RAIL_TxPower_t minPowerDbm, RAIL_TxPower_t maxPowerDbm, RAIL_TxPower_t currentPowerDbm) Set the TX PA power setting used to configure the PA hardware for the PA output power determined by RAIL_SetTxPowerDbm().
RAIL_PaPowerSetting_t	RAIL_GetPaPowerSetting(RAIL_Handle_t railHandle) Get the TX PA power setting, which is used to configure power configurations when the dBm to paPowerSetting mapping table mode is used.
RAIL_Status_t	RAIL_EnablePaAutoMode(RAIL_Handle_t railHandle, bool enable) Enable automatic switching between PAs internally to the RAIL library.
bool	RAIL_IsPaAutoModeEnabled(RAIL_Handle_t railHandle) Query status of PA Auto Mode.
RAIL_Status_t	RAILCb_PaAutoModeDecision(RAIL_Handle_t railHandle, RAIL_TxPower_t *power, RAIL_TxPowerMode_t *mode, const RAIL_ChannelConfigEntry_t *chCfgEntry) Callback that decides which PA and power level should be used while in PA auto mode.
RAIL_Status_t	RAIL_ConfigPaAutoEntry(RAIL_Handle_t railHandle, const RAIL_PaAutoModeConfigEntry_t *paAutoModeEntry) Configure the PA auto mode entries.

Macros

#define	RAIL_TX_POWER_MAX ((RAIL_TxPower_t)0×7FFF) The maximum valid value for a RAIL_TxPower_t.
#define	RAIL_TX_POWER_MIN ((RAIL_TxPower_t)0×8000) The minimum valid value for a RAIL_TxPower_t.
#define	RAIL_TX_POWER_CURVE_DEFAULT_MAX ((RAIL_TxPower_t)200) The maximum power in deci-dBm the curve supports.
#define	RAIL_TX_POWER_CURVE_DEFAULT_INCREMENT ((RAIL_TxPower_t)40) The increment step in deci-dBm for calculating power level.

#define RAIL_TX_POWER_VOLTAGE_SCALING_FACTOR 1000
mV are used for all TX power voltage values.

#define RAIL_TX_POWER_DBM_SCALING_FACTOR 10
deci-dBm are used for all TX power dBm values.

#define RAIL_TX_POWER_LEVEL_INVALID (255U)
Invalid RAIL_TxPowerLevel_t value returned when an error occurs with RAIL_GetTxPower.

#define RAIL_TX_POWER_LEVEL_MAX (254U)
Sentinel value that can be passed to RAIL_SetTxPower to set the highest power level available on the current PA, regardless of which one is selected.

#define RAIL_TX_PA_POWER_SETTING_UNSUPPORTED (0U)
Returned by RAIL_GetPaPowerSetting when the device does not support the dBm to power setting mapping table.

#define RAIL_TX_POWER_MODE_NAMES undefined
The names of the TX power modes.

#define RAIL_POWER_MODE_IS_ANY_EFF (x)
Convenience macro for any EFF power mode.

#define RAIL_POWER_MODE_IS_DBM_POWERSETTING_MAPPING_TABLE_OFDM (x)
Convenience macro for any OFDM mapping table mode.

#define RAIL_POWER_MODE_IS_DBM_POWERSETTING_MAPPING_TABLE_SUBGIG (x)
Convenience macro for any Sub-GHz mapping table mode.

#define RAIL_POWER_MODE_IS_ANY_DBM_POWERSETTING_MAPPING_TABLE (x)
Convenience macro for any mapping table mode.

#define RAIL_POWER_MODE_IS_ANY_OFDM (x)
Convenience macro for any OFDM mode.

Enumeration Documentation

RAIL_TxPowerMode_t

> RAIL_TxPowerMode_t

An enumeration of the EFR32 power modes.

The power modes on the EFR32 correspond to the different on-chip PAs that are available. For more information about the power and performance characteristics of a given amplifier, see the data sheet.

Enumerator	
RAIL_TX_POWER_MODE_2P4GIG_HP	High-power 2.4 GHz amplifier EFR32xG1X: up to 20 dBm, raw values: 0-252 EFR32xG21: up to 20 dBm, raw values: 1-180 EFR32xG22: up to 6 dBm, raw values: 1-128 EFR32xG24: up to 20 dBm, raw values: 0-180, or up to 10 dBm, raw values: 0-90 EFR32xG26: same as EFR32xG24 EFR32xG27: up to 6 dBm, raw values: 1-128 EFR32xG28: up to 10 dBm, raw values: 0-240 Not supported on other platforms.
RAIL_TX_POWER_MODE_2P4_HP	
RAIL_TX_POWER_MODE_2P4GIG_MP	Mid-power 2.4 GHz amplifier EFR32xG21: up to 10 dBm, raw values: 1-90 Not supported on other platforms.
RAIL_TX_POWER_MODE_2P4_MP	
RAIL_TX_POWER_MODE_2P4GIG_LP	Low-power 2.4 GHz amplifier EFR32xG1x: up to 0 dBm, raw values: 1-7 EFR32xG21: up to 0 dBm, raw values: 1-64 EFR32xG22: up to 0 dBm, raw values: 1-16 EFR32xG24: up to 0 dBm, raw values: 1-16 EFR32xG26: same as EFR32xG24 EFR32xG27: up to 0 dBm, raw values: 1-16 Not supported on other platforms.

RAIL_TX_POWER_MODE_2P4_LP	
RAIL_TX_POWER_MODE_2P4GIG_LLP	Low-Low-power 2.4 GHz amplifier Not currently supported on any EFR32 platform.
RAIL_TX_POWER_MODE_2P4GIG_HIGHEST	Select the highest 2.4 GHz power PA available on the current chip.
RAIL_TX_POWER_MODE_2P4_HIGHEST	
RAIL_TX_POWER_MODE_SUBGIG_POWERSETTING_TABLE	PA for all Sub-GHz dBm values in range, using RAIL_PaPowerSetting_t table.
RAIL_TX_POWER_MODE_SUBGIG_HP	High-power Sub-GHz amplifier (Class D mode) EFR32xG1x: up to 20 dBm, raw values: 0-248 Also supported on FR32xG23 and EFR32xG28.
RAIL_TX_POWER_MODE_SUBGIG	
RAIL_TX_POWER_MODE_SUBGIG_MP	Mid-power Sub-GHz amplifier Supported only on EFR32xG23 and EFR32xG28.
RAIL_TX_POWER_MODE_SUBGIG_LP	Low-power Sub-GHz amplifier Supported only on EFR32xG23 and EFR32xG28.
RAIL_TX_POWER_MODE_SUBGIG_LLP	Low-Low-power Sub-GHz amplifier Supported only on EFR32xG23 and EFR32xG28.
RAIL_TX_POWER_MODE_SUBGIG_HIGHEST	Select the highest Sub-GHz power PA available on the current chip.
RAIL_TX_POWER_MODE_OFDM_PA_POWERSETTING_TABLE	PA for all OFDM Sub-GHz dBm values in range, using RAIL_PaPowerSetting_t table.
RAIL_TX_POWER_MODE_OFDM_PA	
RAIL_TX_POWER_MODE_NONE	Invalid amplifier Selection.

Definition at line 1754 of file common/rail_types.h

RAIL_PaBand_t

```
RAIL_PaBand_t
```

Enum used to specify the band for a PA.

Enumerator	
RAIL_PA_BAND_2P4GIG	Indicates a 2.4GHz band PA.
RAIL_PA_BAND_SUBGIG	Indicates a Sub-GHz band PA.
RAIL_PA_BAND_COUNT	A count of the choices in this enumeration.

Definition at line 55 of file plugin/pa-auto-mode/pa_auto_mode.h

Typedef Documentation

RAIL_TxPower_t

```
typedef int16_t RAIL_TxPower_t
```

The transmit power in deci-dBm units (e.g., 4.5 dBm -> 45 deci-dBm).

These values are used by the conversion functions to convert a RAIL_TxPowerLevel_t to deci-dBm for the application consumption. On EFR32, they can range from RAIL_TX_POWER_MIN to RAIL_TX_POWER_MAX.

Definition at line 1688 of file common/rail_types.h

RAIL_TxPowerLevel_t

```
typedef uint8_t RAIL_TxPowerLevel_t
```

Raw power levels used directly by the RAIL_Get/SetTxPower API where a higher numerical value corresponds to a higher output power.

These are referred to as 'raw (values/units)'. On EFR32, they can range from one of RAIL_TX_POWER_LEVEL_2P4_LP_MIN, RAIL_TX_POWER_LEVEL_2P4_HP_MIN, or RAIL_TX_POWER_LEVEL_SUBGIG_HP_MIN to one of RAIL_TX_POWER_LEVEL_2P4_LP_MAX, RAIL_TX_POWER_LEVEL_2P4_HP_MAX, and RAIL_TX_POWER_LEVEL_SUBGIG_HP_MAX, respectively, depending on the selected RAIL_TxPowerMode_t.

Definition at line 1718 of file common/rail_types.h

RAIL_PaPowerSetting_t

```
typedef uint32_t RAIL_PaPowerSetting_t
```

PA power setting used directly by the RAIL_GetPaPowerSetting() and RAIL_SetPaPowerSetting() APIs which is decoded to the actual hardware register value(s).

Definition at line 1738 of file common/rail_types.h

Variable Documentation

RAIL_PaAutoModeConfig

```
const RAIL_PaAutoModeConfigEntry_t* RAIL_PaAutoModeConfig
```

The actual PA auto mode configuration structure used by the auto mode plugin to control output power.

Definition at line 91 of file plugin/pa-auto-mode/pa_auto_mode.h

Function Documentation

RAIL_ConfigTxPower

```
RAIL_Status_t RAIL_ConfigTxPower (RAIL_Handle_t railHandle, const RAIL_TxPowerConfig_t *config)
```

Initialize TX power settings.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	config	An instance which contains desired initial settings for the TX amplifier.

Returns

- RAIL_Status_t indicating success or an error.

These settings include the selection between the multiple TX amplifiers, voltage supplied to the TX power amplifier, and ramp times. This must be called before any transmit occurs or RAIL_SetTxPower is called. While this function should always

be called during initialization, it can also be called any time if these settings need to change to adapt to a different application/protocol. This API also resets TX power to RAIL_TX_POWER_LEVEL_INVALID, so RAIL_SetTxPower must be called afterwards.

At times, certain combinations of configurations cannot be achieved. This API attempts to get as close as possible to the requested settings. The following "RAIL_Get..." API can be used to determine what values were set. A change in RAIL_TxPowerConfig_t::rampTime may affect the minimum timings that can be achieved in RAIL_StateTiming_t::idleToTx and RAIL_StateTiming_t::rxToTx. Call RAIL_SetStateTiming() again to check whether these times have changed.

Definition at line 2640 of file common/rail.h

RAIL_GetTxPowerConfig

> RAIL_Status_t RAIL_GetTxPowerConfig (RAIL_Handle_t railHandle, RAIL_TxPowerConfig_t *config)

Get the TX power settings currently used in the amplifier.

Parameters

[in]	railHandle	A RAIL instance handle.
[out]	config	A pointer to memory allocated to hold the current TxPower configuration structure. A NULL configuration will produce undefined behavior.

Returns

- RAIL status variable indicating whether or not the get was successful.

Note that this API does not return the current TX power, which is separately managed by the RAIL_GetTxPower / RAIL_SetTxPower APIs. Use this API to determine which values were set as a result of RAIL_ConfigTxPower.

Definition at line 2658 of file common/rail.h

RAIL_SetTxPower

> RAIL_Status_t RAIL_SetTxPower (RAIL_Handle_t railHandle, RAIL_TxPowerLevel_t powerLevel)

Set the TX power in units of raw units (see rail_chip_specific.h for value ranges).

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	powerLevel	Power in chip-specific RAIL_TxPowerLevel_t units.

Returns

- RAIL_Status_t indicating success or an error.

To convert between decibels and the integer values that the registers take, call RAIL_ConvertDbmToRaw. A weak version of this function, which works well with our boards is provided. However, customers using a custom board need to characterize chip operation on that board and override the function to convert appropriately from the desired dB values to raw integer values.

Depending on the configuration used in RAIL_ConfigTxPower, not all power levels are achievable. This API will get as close as possible to the desired power without exceeding it, and calling RAIL_GetTxPower is the only way to know the exact value written.

Calling this function before configuring the PA (i.e., before a successful call to RAIL_ConfigTxPower) will return an error.

Definition at line 2684 of file common/rail.h

RAIL_GetTxPower

```
RAIL_TxPowerLevel_t RAIL_GetTxPower (RAIL_Handle_t railHandle)
```

Return the current power setting of the PA.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- The chip-specific RAIL_TxPowerLevel_t value of the current transmit power.

This API returns the raw value that was set by RAIL_SetTxPower. A weak version of RAIL_ConvertRawToDbm that works with our boards to convert the raw values into actual output dBm values is provided. However, customers using a custom board need to re-characterize the relationship between raw and decibel values and rewrite the provided function.

Calling this function before configuring the PA (i.e., before a successful call to RAIL_ConfigTxPower) will return an error (RAIL_TX_POWER_LEVEL_INVALID).

Definition at line `2705` of file `common/rail.h`

RAIL_ConvertRawToDbm

```
RAIL_TxPower_t RAIL_ConvertRawToDbm (RAIL_Handle_t railHandle, RAIL_TxPowerMode_t mode, RAIL_TxPowerLevel_t powerLevel)
```

Convert raw values written to registers to decibel value (in units of deci-dBm).

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	mode	PA mode for which to convert.
[in]	powerLevel	A raw amplifier register value to be converted to deci-dBm.

Returns

- raw amplifier values converted to units of deci-dBm.

A weak version of this function is provided that is tuned to provide accurate values for our boards. For a custom board, the relationship between what is written to the TX amplifier and the actual output power should be re-characterized and implemented in an overriding version of RAIL_ConvertRawToDbm. For minimum code size and best speed, use only raw values with the TxPower API and override this function with a smaller function. In the weak version provided with the RAIL library, railHandle is only used to indicate to the user from where the function was called, so it is OK to use either a real protocol handle, or one of the chip-specific ones, such as RAIL_EFR32_HANDLE.

Although the definitions of this function may change, the signature must be as declared here.

Definition at line `2731` of file `common/rail.h`

RAIL_ConvertDbmToRaw

```
RAIL_TxPowerLevel_t RAIL_ConvertDbmToRaw (RAIL_Handle_t railHandle, RAIL_TxPowerMode_t mode, RAIL_TxPower_t power)
```

Convert the desired decibel value (in units of deci-dBm) to raw integer values used by the TX amplifier registers.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	mode	PA mode for which to do the conversion.
[in]	power	Desired dBm values in units of deci-dBm.

Returns

- deci-dBm value converted to a raw integer value that can be used directly with RAIL_SetTxPower.

A weak version of this function is provided that is tuned to provide accurate values for our boards. For a custom board, the relationship between what is written to the TX amplifier and the actual output power should be characterized and implemented in an overriding version of RAIL_ConvertDbmToRaw. For minimum code size and best speed use only raw values with the TxPower API and override this function with a smaller function. In the weak version provided with the RAIL library, railHandle is only used to indicate to the user from where the function was called, so it is OK to use either a real protocol handle, or one of the chip-specific ones, such as RAIL_EFR32_HANDLE.

Although the definitions of this function may change, the signature must be as declared here.

Note

- This function is called from within the RAIL library for comparison between channel limitations and current power. It will throw an assert if you haven't called RAIL_InitTxPowerCurves which initializes the mappings between raw power levels and actual dBm powers. To avoid the assert, ensure that the maxPower of all channel configuration entries is RAIL_TX_POWER_MAX or above, or override this function to always return 255.

Definition at line `2767` of file `common/rail.h`

RAIL_VerifyTxPowerCurves

> void RAIL_VerifyTxPowerCurves (const struct RAIL_TxPowerCurvesConfigAlt *config)

Verify the TX Power Curves on modules.

Parameters

[in]	config	TX Power Curves to use on this module.

This function only needs to be called when using a module and has no effect otherwise. Transmit will not work before this function is called.

Definition at line `2778` of file `common/rail.h`

RAIL_SetTxPowerDbm

> RAIL_Status_t RAIL_SetTxPowerDbm (RAIL_Handle_t railHandle, RAIL_TxPower_t power)

Set the TX power in terms of deci-dBm instead of raw power level.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	power	A desired deci-dBm power to be set.

Returns

- RAIL Status variable indicate whether setting the power was successful.

This is a utility function for user convenience. Normally, to set TX power in dBm, do the following:

```
RAIL_TxPower_t power = 100; // 100 deci-dBm, 10 dBm
RAIL_TxPowerConfig_t txPowerConfig;
RAIL_GetTxPowerConfig(railHandle, &txPowerConfig);
// RAIL_ConvertDbmToRaw will be the weak version provided by Silicon Labs
// by default, or the customer version, if overwritten.
RAIL_TxPowerLevel_t powerLevel = RAIL_ConvertDbmToRaw(railHandle,
                                  txPowerConfig.mode,
                                  power);
RAIL_SetTxPower(railHandle, powerLevel);
```

This function wraps all those calls in a single function with power passed in as a parameter.

Definition at line 2805 of file common/rail.h

RAIL_GetTxPowerDbm

> RAIL_TxPower_t RAIL_GetTxPowerDbm (RAIL_Handle_t railHandle)

Get the TX power in terms of deci-dBm instead of raw power level.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- The current output power in deci-dBm.

This is a utility function for user convenience. Normally, to get TX power in dBm, do the following:

```
RAIL_TxPowerLevel_t powerLevel = RAIL_GetTxPower(railHandle);
RAIL_TxPowerConfig_t txPowerConfig;
RAIL_GetTxPowerConfig(railHandle, &txPowerConfig);
// RAIL_ConvertRawToDbm will be the weak version provided by Silicon Labs
// by default, or the customer version, if overwritten.
RAIL_TxPower_t power = RAIL_ConvertRawToDbm(railHandle,
                                txPowerConfig.mode,
                                powerLevel);
return power;
```

This function wraps all those calls in a single function with power returned as the result.

Definition at line 2831 of file common/rail.h

RAIL_GetPowerSettingTable

> const RAIL_PaPowerSetting_t * RAIL_GetPowerSettingTable (RAIL_Handle_t railHandle, RAIL_TxPowerMode_t mode, RAIL_TxPower_t *minPower, RAIL_TxPower_t *maxPower, RAIL_TxPowerLevel_t *step)

Get the TX PA power setting table and related values.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	mode	PA mode for which to get the powersetting table
[out]	minPower	A pointer to a RAIL_TxPower_t
[out]	maxPower	A pointer to a RAIL_TxPower_t

[out]	step	In deci-dBm increments. A pointer to a RAIL_TxPowerLevel_t

Returns

- Power setting table start address. When NULL is returned all out params above won't be set.

The number of entries in the table can be calculated based on the minPower, maxPower, and step parameters. For example, for minPower = 115 (11.5 dBm), maxPower = 300 (30 dBm), and step = 1, the number of entries in table would be 186

Definition at line 2848 of file common/rail.h

RAIL_SetPaPowerSetting

> RAIL_Status_t RAIL_SetPaPowerSetting (RAIL_Handle_t railHandle, RAIL_PaPowerSetting_t paPowerSetting, RAIL_TxPower_t minPowerDbm, RAIL_TxPower_t maxPowerDbm, RAIL_TxPower_t currentPowerDbm)

Set the TX PA power setting used to configure the PA hardware for the PA output power determined by RAIL_SetTxPowerDbm().

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	paPowerSetting	The desired PA power setting.
[in]	minPowerDbm	The minimum power in dBm that the PA can output.
[in]	maxPowerDbm	The maximum power in dBm that the PA can output.
[in]	currentPowerDbm	The corresponding output power in dBm for this power setting.

Returns

- RAIL Status variable indicate whether setting the PA power setting was successful.

Definition at line 2864 of file common/rail.h

RAIL_GetPaPowerSetting

> RAIL_PaPowerSetting_t RAIL_GetPaPowerSetting (RAIL_Handle_t railHandle)

Get the TX PA power setting, which is used to configure power configurations when the dBm to paPowerSetting mapping table mode is used.

Parameters

[in]	railHandle	A RAIL instance handle.

Returns

- The current PA power setting.

Definition at line 2877 of file common/rail.h

RAIL_EnablePaAutoMode

> RAIL_Status_t RAIL_EnablePaAutoMode (RAIL_Handle_t railHandle, bool enable)

Enable automatic switching between PAs internally to the RAIL library.

Parameters

[in]	railHandle	A real (not generic) RAIL instance handle.
[in]	enable	Enable or disable PA Auto Mode.

While PA Automode is enabled, the PA will be chosen and set automatically whenever RAIL_SetTxPowerDbm is called or whenever powers are coerced automatically, internally to the RAIL library during a channel change. While PA Auto Mode is enabled, users cannot call RAIL_ConfigTxPower or RAIL_SetTxPower. When entering auto mode, RAIL_SetTxPowerDbm must be called to specify the desired power. When leaving auto mode, RAIL_ConfigTxPower as well as one of RAIL_SetTxPower or RAIL_SetTxPowerDbm must be called to re-specify the desired PA and power level combination.

Note

- : Power conversion curves must be initialized before calling this function. That is, RAIL_ConvertDbmToRaw and RAIL_ConvertRawToDbm most both be able to operate properly to ensure that PA Auto Mode functions correctly. See the PA Conversions plugin or AN1127 for more details.

Returns

- Status parameter indicating success of function call.

Definition at line `2900` of file `common/rail.h`

RAIL_IsPaAutoModeEnabled

```
bool RAIL_IsPaAutoModeEnabled (RAIL_Handle_t railHandle)
```

Query status of PA Auto Mode.

Parameters

[in]	railHandle	A real (not generic) RAIL instance handle on which to query PA Auto Mode status.

Returns

- Indicator of whether Auto Mode is enabled (true) or not (false).

Definition at line `2909` of file `common/rail.h`

RAILCb_PaAutoModeDecision

```
RAIL_Status_t RAILCb_PaAutoModeDecision (RAIL_Handle_t railHandle, RAIL_TxPower_t *power, RAIL_TxPowerMode_t *mode, const RAIL_ChannelConfigEntry_t *chCfgEntry)
```

Callback that decides which PA and power level should be used while in PA auto mode.

Parameters

[in]	railHandle	A RAIL instance handle.
[inout]	power	Pointer to the dBm output power (in deci-dBm, 10*dBm) being requested. The value this points to when the function returns will be applied to the radio.
[out]	mode	Pointer to the RAIL_TxPowerMode_t to be used to achieve the requested power. The value this points to when the function returns will be applied to the radio.
[in]	chCfgEntry	Pointer to a RAIL_ChannelConfigEntry_t. While switching channels, it will be the entry RAIL is switch **to**, during a call to RAIL_SetTxPowerDbm, it will be the entry RAIL is **already on**. Can be NULL if a channel configuration was not set or no valid channels are present.

Returns

-

Return status indicating result of function call. If this is anything except RAIL_STATUS_NO_ERROR, neither PA's nor their powers will be configured automatically.

Whatever values mode and powerLevel point to when this function return will be applied to the PA hardware and used for transmits. **Note**

- The mode and power level provided by this function depends on the RAIL_PaAutoModeConfig provided for the chip. The RAIL_PaAutoModeConfig definition for a chip should tend to all the bands supported by the chip and cover the full range of power to find a valid entry for requested power for a specific band.

Definition at line `2939` of file `common/rail.h`

RAIL_ConfigPaAutoEntry

```
RAIL_Status_t RAIL_ConfigPaAutoEntry (RAIL_Handle_t railHandle, const RAIL_PaAutoModeConfigEntry_t
*paAutoModeEntry)
```

Configure the PA auto mode entries.

Parameters

[in]	railHandle	A RAIL instance handle.
[in]	paAutoModeEntry	Entries used to configure PA auto mode decision points.

Returns

- Status parameter indicating success of function call.

Definition at line `100` of file `plugin/pa-auto-mode/pa_auto_mode.h`

Macro Definition Documentation

RAIL_TX_POWER_MAX

```
#define RAIL_TX_POWER_MAX
```

Value:

```
((RAIL_TxPower_t)0x7FFF)
```

The maximum valid value for a RAIL_TxPower_t.

Definition at line `1691` of file `common/rail_types.h`

RAIL_TX_POWER_MIN

```
#define RAIL_TX_POWER_MIN
```

Value:

```
((RAIL_TxPower_t)0x8000)
```

The minimum valid value for a RAIL_TxPower_t.

Definition at line `1693` of file `common/rail_types.h`

RAIL_TX_POWER_CURVE_DEFAULT_MAX

```
#define RAIL_TX_POWER_CURVE_DEFAULT_MAX
```

Value:

```
((RAIL_TxPower_t)200)
```

The maximum power in deci-dBm the curve supports.

Definition at line 1696 of file common/rail_types.h

RAIL_TX_POWER_CURVE_DEFAULT_INCREMENT

```
#define RAIL_TX_POWER_CURVE_DEFAULT_INCREMENT
```

Value:

```
((RAIL_TxPower_t)40)
```

The increment step in deci-dBm for calculating power level.

Definition at line 1698 of file common/rail_types.h

RAIL_TX_POWER_VOLTAGE_SCALING_FACTOR

```
#define RAIL_TX_POWER_VOLTAGE_SCALING_FACTOR
```

Value:

```
1000
```

mV are used for all TX power voltage values.

TX power voltages take and return voltages multiplied by this factor.

Definition at line 1702 of file common/rail_types.h

RAIL_TX_POWER_DBM_SCALING_FACTOR

```
#define RAIL_TX_POWER_DBM_SCALING_FACTOR
```

Value:

```
10
```

deci-dBm are used for all TX power dBm values.

All dBm inputs to TX power functions take dBm power times this factor.

Definition at line 1706 of file common/rail_types.h

RAIL_TX_POWER_LEVEL_INVALID

```
#define RAIL_TX_POWER_LEVEL_INVALID
```

Value:

```
(255U)
```

Invalid RAIL_TxPowerLevel_t value returned when an error occurs with RAIL_GetTxPower.

Definition at line 1724 of file common/rail_types.h

RAIL_TX_POWER_LEVEL_MAX

```
#define RAIL_TX_POWER_LEVEL_MAX
```

Value:

```
(254U)
```

Sentinel value that can be passed to RAIL_SetTxPower to set the highest power level available on the current PA, regardless of which one is selected.

Definition at line 1731 of file common/rail_types.h

RAIL_TX_PA_POWER_SETTING_UNSUPPORTED

```
#define RAIL_TX_PA_POWER_SETTING_UNSUPPORTED
```

Value:

```
(0U)
```

Returned by RAIL_GetPaPowerSetting when the device does not support the dBm to power setting mapping table.

Definition at line 1744 of file common/rail_types.h

RAIL_TX_POWER_MODE_NAMES

```
#define RAIL_TX_POWER_MODE_NAMES
```

Value:

```
{                                       \
  "RAIL_TX_POWER_MODE_2P4GIG_HP",               \
  "RAIL_TX_POWER_MODE_2P4GIG_MP",               \
  "RAIL_TX_POWER_MODE_2P4GIG_LP",               \
  "RAIL_TX_POWER_MODE_2P4GIG_LLP",              \
  "RAIL_TX_POWER_MODE_2P4GIG_HIGHEST",          \
  "RAIL_TX_POWER_MODE_SUBGIG_POWERSETTING_TABLE",    \
  "RAIL_TX_POWER_MODE_SUBGIG_HP",               \
  "RAIL_TX_POWER_MODE_SUBGIG_MP",               \
  "RAIL_TX_POWER_MODE_SUBGIG_LP",               \
  "RAIL_TX_POWER_MODE_SUBGIG_LLP",              \
  "RAIL_TX_POWER_MODE_SUBGIG_HIGHEST",          \
  "RAIL_TX_POWER_MODE_OFDM_PA_POWERSETTING_TABLE",    \
  "RAIL_TX_POWER_MODE_SUBGIG_EFF_POWERSETTING_TABLE", \
  "RAIL_TX_POWER_MODE_OFDM_PA_EFF_POWERSETTING_TABLE", \
  "RAIL_TX_POWER_MODE_NONE"                     \
}
```

The names of the TX power modes.

A list of the names for the TX power modes on EFR32 parts. This macro is useful for test applications and debugging output.

Definition at line 1892 of file common/rail_types.h

RAIL_POWER_MODE_IS_ANY_EFF

```
#define RAIL_POWER_MODE_IS_ANY_EFF
```

Value:

```
(((x) == RAIL_TX_POWER_MODE_OFDM_PA_EFF_POWERSETTING_TABLE) \
 || ((x) == RAIL_TX_POWER_MODE_SUBGIG_EFF_POWERSETTING_TABLE))
```

Convenience macro for any EFF power mode.

Definition at line 1928 of file common/rail_types.h

RAIL_POWER_MODE_IS_DBM_POWERSETTING_MAPPING_TABLE_OFDM

```
#define RAIL_POWER_MODE_IS_DBM_POWERSETTING_MAPPING_TABLE_OFDM
```

Value:

```
(((x) == RAIL_TX_POWER_MODE_OFDM_PA_POWERSETTING_TABLE)      \
 || ((x) == RAIL_TX_POWER_MODE_OFDM_PA_EFF_POWERSETTING_TABLE))
```

Convenience macro for any OFDM mapping table mode.

Definition at line 1932 of file common/rail_types.h

RAIL_POWER_MODE_IS_DBM_POWERSETTING_MAPPING_TABLE_SUBGIG

```
#define RAIL_POWER_MODE_IS_DBM_POWERSETTING_MAPPING_TABLE_SUBGIG
```

Value:

```
(((x) == RAIL_TX_POWER_MODE_SUBGIG_EFF_POWERSETTING_TABLE)      \
 || ((x) == RAIL_TX_POWER_MODE_SUBGIG_POWERSETTING_TABLE))
```

Convenience macro for any Sub-GHz mapping table mode.

Definition at line 1936 of file common/rail_types.h

RAIL_POWER_MODE_IS_ANY_DBM_POWERSETTING_MAPPING_TABLE

```
#define RAIL_POWER_MODE_IS_ANY_DBM_POWERSETTING_MAPPING_TABLE
```

Value:

```
(((x) == RAIL_TX_POWER_MODE_OFDM_PA_POWERSETTING_TABLE)       \
 || ((x) == RAIL_TX_POWER_MODE_OFDM_PA_EFF_POWERSETTING_TABLE) \
 || ((x) == RAIL_TX_POWER_MODE_SUBGIG_POWERSETTING_TABLE)     \
 || ((x) == RAIL_TX_POWER_MODE_SUBGIG_EFF_POWERSETTING_TABLE))
```

Convenience macro for any mapping table mode.

Definition at line 1940 of file common/rail_types.h

RAIL_POWER_MODE_IS_ANY_OFDM

```
#define RAIL_POWER_MODE_IS_ANY_OFDM
```

Value:

```
(x)
```

Convenience macro for any OFDM mode.

Definition at line 1946 of file common/rail_types.h

RAIL_TxPowerConfig_t

A structure containing values used to initialize the power amplifiers.

Public Attributes

RAIL_TxPowerMode_t	mode TX power mode.
uint16_t	voltage Power amplifier supply voltage in mV, generally: DCDC supply ~ 1800 mV (1.8 V) Battery supply ~ 3300 mV (3.3 V)
uint16_t	rampTime The amount of time to spend ramping for TX in uS.

Public Attribute Documentation

mode

```
RAIL_TxPowerMode_t RAIL_TxPowerConfig_t::mode
```

TX power mode.

Definition at line 1917 of file common/rail_types.h

voltage

```
uint16_t RAIL_TxPowerConfig_t::voltage
```

Power amplifier supply voltage in mV, generally: DCDC supply ~ 1800 mV (1.8 V) Battery supply ~ 3300 mV (3.3 V)

Definition at line 1922 of file common/rail_types.h

rampTime

```
uint16_t RAIL_TxPowerConfig_t::rampTime
```

The amount of time to spend ramping for TX in uS.

Definition at line 1924 of file common/rail_types.h

RAIL_PaAutoModeConfigEntry_t

Struct to ease specification of appropriate ranges within which a PA should be used.

Public Attributes

RAIL_TxPower_t	min The minimum (inclusive) deci-dBm power to use with this entry.
RAIL_TxPower_t	max The maximum (inclusive) deci-dBm power to use with this entry.
RAIL_TxPowerMode_t	mode The PA that this range of powers applies to.
RAIL_PaBand_t	band The RF band that this PA works with.

Public Attribute Documentation

min

```
RAIL_TxPower_t RAIL_PaAutoModeConfigEntry_t::min
```

The minimum (inclusive) deci-dBm power to use with this entry.

Definition at line `78` of file `plugin/pa-auto-mode/pa_auto_mode.h`

max

```
RAIL_TxPower_t RAIL_PaAutoModeConfigEntry_t::max
```

The maximum (inclusive) deci-dBm power to use with this entry.

Definition at line `80` of file `plugin/pa-auto-mode/pa_auto_mode.h`

mode

```
RAIL_TxPowerMode_t RAIL_PaAutoModeConfigEntry_t::mode
```

The PA that this range of powers applies to.

Definition at line `82` of file `plugin/pa-auto-mode/pa_auto_mode.h`

band

RAIL_PaBand_t RAIL_PaAutoModeConfigEntry_t::band

The RF band that this PA works with.

Definition at line 84 of file plugin/pa-auto-mode/pa_auto_mode.h

EFR32

EFR32

Types specific to the EFR32 for dealing with the on-chip PAs.

Macros

#define RAIL_TX_POWER_LEVEL_2P4_LP_MAX (7U)
The maximum valid value for the RAIL_TxPowerLevel_t when in RAIL_TX_POWER_MODE_2P4_LP mode.

#define RAIL_TX_POWER_LEVEL_2P4_HP_MAX (252U)
The maximum valid value for the RAIL_TxPowerLevel_t when in RAIL_TX_POWER_MODE_2P4_HP mode.

#define RAIL_TX_POWER_LEVEL_SUBGIG_HP_MAX (248U)
The maximum valid value for the RAIL_TxPowerLevel_t when in RAIL_TX_POWER_MODE_SUBGIG mode.

#define RAIL_TX_POWER_LEVEL_2P4_LP_MIN (1U)
The minimum valid value for the RAIL_TxPowerLevel_t when in RAIL_TX_POWER_MODE_2P4_LP mode.

#define RAIL_TX_POWER_LEVEL_2P4_HP_MIN (0U)
The minimum valid value for the RAIL_TxPowerLevel_t when in RAIL_TX_POWER_MODE_2P4_HP mode.

#define RAIL_TX_POWER_LEVEL_SUBGIG_HP_MIN (0U)
The minimum valid value for the RAIL_TxPowerLevel_t when in RAIL_TX_POWER_MODE_SUBGIG mode.

#define RAIL_TX_POWER_LEVEL_LP_MAX RAIL_TX_POWER_LEVEL_2P4_LP_MAX
Backwards compatibility define.

#define RAIL_TX_POWER_LEVEL_HP_MAX RAIL_TX_POWER_LEVEL_2P4_HP_MAX
Backwards compatibility define.

#define RAIL_TX_POWER_LEVEL_SUBGIG_MAX RAIL_TX_POWER_LEVEL_SUBGIG_HP_MAX
Backwards compatibility define.

#define RAIL_TX_POWER_LEVEL_LP_MIN RAIL_TX_POWER_LEVEL_2P4_LP_MIN
Backwards compatibility define.

#define RAIL_TX_POWER_LEVEL_HP_MIN RAIL_TX_POWER_LEVEL_2P4_HP_MIN
Backwards compatibility define.

#define RAIL_TX_POWER_LEVEL_SUBGIG_MIN RAIL_TX_POWER_LEVEL_SUBGIG_HP_MIN
Backwards compatibility define.

#define RAIL_NUM_PA (3U)
The number of PA's on this chip.

Macro Definition Documentation

RAIL_TX_POWER_LEVEL_2P4_LP_MAX

```
#define RAIL_TX_POWER_LEVEL_2P4_LP_MAX
```

Value:

```
(7U)
```

The maximum valid value for the RAIL_TxPowerLevel_t when in RAIL_TX_POWER_MODE_2P4_LP mode.

Definition at line 241 of file chip/efr32/efr32xg1x/rail_chip_specific.h

RAIL_TX_POWER_LEVEL_2P4_HP_MAX

```
#define RAIL_TX_POWER_LEVEL_2P4_HP_MAX
```

Value:

```
(252U)
```

The maximum valid value for the RAIL_TxPowerLevel_t when in RAIL_TX_POWER_MODE_2P4_HP mode.

Definition at line 246 of file chip/efr32/efr32xg1x/rail_chip_specific.h

RAIL_TX_POWER_LEVEL_SUBGIG_HP_MAX

```
#define RAIL_TX_POWER_LEVEL_SUBGIG_HP_MAX
```

Value:

```
(248U)
```

The maximum valid value for the RAIL_TxPowerLevel_t when in RAIL_TX_POWER_MODE_SUBGIG mode.

Definition at line 251 of file chip/efr32/efr32xg1x/rail_chip_specific.h

RAIL_TX_POWER_LEVEL_2P4_LP_MIN

```
#define RAIL_TX_POWER_LEVEL_2P4_LP_MIN
```

Value:

```
(1U)
```

The minimum valid value for the RAIL_TxPowerLevel_t when in RAIL_TX_POWER_MODE_2P4_LP mode.

Definition at line 256 of file chip/efr32/efr32xg1x/rail_chip_specific.h

RAIL_TX_POWER_LEVEL_2P4_HP_MIN

```
#define RAIL_TX_POWER_LEVEL_2P4_HP_MIN
```

Value:

```
(0U)
```

The minimum valid value for the RAIL_TxPowerLevel_t when in RAIL_TX_POWER_MODE_2P4_HP mode.

Definition at line 261 of file chip/efr32/efr32xg1x/rail_chip_specific.h

RAIL_TX_POWER_LEVEL_SUBGIG_HP_MIN

```
#define RAIL_TX_POWER_LEVEL_SUBGIG_HP_MIN
```

Value:

```
(0U)
```

The minimum valid value for the RAIL_TxPowerLevel_t when in RAIL_TX_POWER_MODE_SUBGIG mode.

Definition at line 266 of file chip/efr32/efr32xg1x/rail_chip_specific.h

RAIL_TX_POWER_LEVEL_LP_MAX

```
#define RAIL_TX_POWER_LEVEL_LP_MAX
```

Value:

```
RAIL_TX_POWER_LEVEL_2P4_LP_MAX
```

Backwards compatibility define.

Definition at line 269 of file chip/efr32/efr32xg1x/rail_chip_specific.h

RAIL_TX_POWER_LEVEL_HP_MAX

```
#define RAIL_TX_POWER_LEVEL_HP_MAX
```

Value:

```
RAIL_TX_POWER_LEVEL_2P4_HP_MAX
```

Backwards compatibility define.

Definition at line 271 of file chip/efr32/efr32xg1x/rail_chip_specific.h

RAIL_TX_POWER_LEVEL_SUBGIG_MAX

```
#define RAIL_TX_POWER_LEVEL_SUBGIG_MAX
```

Value:

```
RAIL_TX_POWER_LEVEL_SUBGIG_HP_MAX
```

Backwards compatibility define.

Definition at line 273 of file chip/efr32/efr32xg1x/rail_chip_specific.h

RAIL_TX_POWER_LEVEL_LP_MIN

```
#define RAIL_TX_POWER_LEVEL_LP_MIN
```

Value:

```
RAIL_TX_POWER_LEVEL_2P4_LP_MIN
```

Backwards compatibility define.

Definition at line 275 of file chip/efr32/efr32xg1x/rail_chip_specific.h

RAIL_TX_POWER_LEVEL_HP_MIN

```
#define RAIL_TX_POWER_LEVEL_HP_MIN
```

Value:

```
RAIL_TX_POWER_LEVEL_2P4_HP_MIN
```

Backwards compatibility define.

Definition at line 277 of file chip/efr32/efr32xg1x/rail_chip_specific.h

RAIL_TX_POWER_LEVEL_SUBGIG_MIN

```
#define RAIL_TX_POWER_LEVEL_SUBGIG_MIN
```

Value:

```
RAIL_TX_POWER_LEVEL_SUBGIG_HP_MIN
```

Backwards compatibility define.

Definition at line 279 of file chip/efr32/efr32xg1x/rail_chip_specific.h

RAIL_NUM_PA

```
#define RAIL_NUM_PA
```

Value:

```
(3U)
```

The number of PA's on this chip.

(Including Virtual PAs)

Definition at line 284 of file chip/efr32/efr32xg1x/rail_chip_specific.h

Components

Components

Software Components

These are some of the RAIL software components available for application use.

Libraries

RAIL library software components provide the ability to choose support for single or multiple protocols. Current components include:

- RAIL Library, Single Protocol
- RAIL Library, Multiprotocol

Utilities

Radio utility software components provide optional source code that aid with radio setup and functionality. Current components include:

- Angle of Arrival/Angle of Departure (AoX)
- Antenna Diversity
- Built-in PHYs across HFXO frequencies
- Callbacks
- Coexistence
- Direct Memory Access (DMA)
- Energy Friendly Front End Module (EFF)
- Front End Module (FEM)
- Initialization
- Packet Trace Interface (PTI)
- Power Amplifier (PA)
- Protocol
- Recommended
- RF Path
- RF Path Switch Utility
- Received Signal Strength Indicator (RSSI)
- Radio Sequencer Image Selection Utility
- Software Modem (SFM) sequencer image selection Utility
- Thermistor Utility

RAILtest Application

RAILtest application software components are specific to the RAILtest application. Current components include:

- RAILtest Application Core
- RAILtest Application Graphics

Modules

RAIL Library

Angle of Arrival/Departure (AoX) Utility

- Antenna Diversity Utility
- Built-in PHYs for alternate HFXO frequencies
- Callbacks Utility
- Coexistence Utility
- Direct Memory Access (DMA) Utility
- Energy Friendly Front End Module (EFF) Utility
- Front End Module (FEM) Utility
- Initialization Utility
- Packet Trace Interface (PTI) Utility
- Power Amplifier (PA) Utility
- Protocol Utility
- Recommended Utility
- RF Path Utility
- Received Signal Strength Indicator (RSSI) Utility
- RAILtest Application Core
- RAILtest Application Graphics
- Thermistor Utility
- RF Path Switch Utility
- Radio Sequencer Image Selection Utility
- Software Modem (SFM) sequencer image selection Utility

RAIL Library

RAIL Library

RAIL Multiprotocol Library

The RAIL multiprotocol library allows multiple protocols to be initialized, set up, and managed at the library level. Dynamic Multiprotocol (DMP) requires the use of the RAIL multiprotocol library. Additional information can be found here: RAIL Multiprotocol.

RAIL Single Protocol Library

The RAIL single protocol library reduces flash and memory footprints compared to the RAIL multiprotocol library. The RAIL_SchedulerInfo_t input parameter, used by select RAIL library API functions, is always ignored in the RAIL single protocol library. The single protocol library can be used for Switched Multiprotocol (SMP) because the RAIL library is initialized and used for one wireless protocol at a time.

Angle of Arrival/Departure (AoX) Utility

Angle of Arrival/Departure (AoX) Utility

RAIL Angle of Arrival/Departure (AoX) Utility

This optional software component can be enabled to include default functionality related to AoX configuration. AoX allows the Bluetooth stack to estimate the relative angle of another radio based on samples from multiple antennas.

The RAIL AoX Utility is only necessary if 2 or more antennas are configured. If fewer than 2 antennas are configured, CTE(constant tone extension) data can be transmitted or received on the default antenna.

Configuration Options

The following configuration options can be changed:

- Number of AoX antenna pins(0-6)

The following hardware options can be changed:

- GPIO pin for each antenna

Antenna Diversity Utility

Antenna Diversity Utility

RAIL Antenna Diversity Utility

This optional software component can be enabled to include default functionality related to antenna diversity configuration.

When using a Silicon Labs-developed board, the configuration options for this software component are set up automatically based on the selected board. When using a custom board, manually configure the configuration options.

Configuration Options

The following configuration options can be changed:

- Antenna Diversity:
  - RX Antenna Diversity Mode: Choose the mode to use for receive antenna diversity. The possible modes are disabled, enabled, Antenna 0 only, or Antenna 1 only.
  - TX Antenna Diversity Mode: Choose the mode to use for transmit antenna diversity. The possible modes are disabled, enabled, Antenna 0 only, or Antenna 1 only.
  - Enable/disable Runtime PHY Selection: This option only works when using the Silicon Labs Zigbee or Open Thread stacks. If enabled it will allow turning on and off the antenna diversity version of the IEEE 802.15.4 PHY at runtime by these stacks.

The following hardware options can be changed:

- Configure the antenna diversity transmit select GPIO (ANT0 signal).
- Configure the antenna diversity inverse transmit select GPIO (ANT1 signal).

Built-in PHYs for alternate HFXO frequencies

Built-in PHYs for alternate HFXO frequencies

RAIL Utility for built-in PHYs

This optional software component can be enabled to override the built-in PHYs used to support standards-based protocols based on the HFXO frequency configured as part of the device_init_hfxo component.

Currently the following combinations of chips and frequencies are supported:

- EFR32XG24: 38.4 MHz, 39 MHz and 40 MHz

On unsupported chips, this component is safe to include, and will have no effect.

Callbacks Utility

Callbacks Utility

RAIL Callbacks Utility

This optional software component can be enabled to include default functionality related to RAIL callback configuration.

This utility provides basic callbacks that can be made available to RAIL. If a callback is not registered for use, it will be compiled out. The provided callbacks include:

- RAILCb_AssertFailed
  - This callback is called when a RAIL library-level assertion occurs. This callback occurs regardless of the other configuration options enabled in this software component.
  - This callback overrides a weak function in the RAIL library and as such cannot be compiled out. Any assert within the RAIL library will cause this callback to run and provide information as to the cause.
  - The weak function `sl_rail_util_on_assert_failed` is provided for application use and will be called when RAILCb_AssertFailed is called.
  - This callback override can be configured to be excluded, in which case the application is responsible for creating its own RAILCb_AssertFailed override if so desired.
- `sli_rail_util_on_rf_ready`
  - This callback is called when the call to RAIL_Init completes.
  - The `Initialize RAIL` configuration option must be enabled, and this callback must be passed as a parameter to RAIL_Init to enable its use.
  - The weak function `sl_rail_util_on_rf_ready` is provided for application use and will be called when `sli_rail_util_on_rf_ready` is called.
- `sli_rail_util_on_channel_config_change`
  - This callback is called when an automatic switch from one channel configuration entry to another is performed internal to the RAIL library.
  - The `Initialize RAIL` and `Protocol Configuration` configuration options must be enabled, and this callback must be passed as a parameter to RAIL_ConfigChannels to enable its use.
  - The weak function `sl_rail_util_on_channel_config_change` is provided for application use and will be called when `sli_rail_util_on_channel_config_change` is called.
- `sli_rail_util_on_event`
  - This callback is called when a configured radio event occurs. The radio events capable of causing this callback are identified in RAIL_Events_t.
  - The `Initialize RAIL` and `Radio Event Configuration` configuration options must be enabled, and this callback must be passed as a parameter to RAIL_Init to enable its use.
  - The weak function `sl_rail_util_on_event` is provided for application use and will be called when `sli_rail_util_on_event` is called.

Configuration Options

The following configuration options can be changed:

- Include/exclude RAILCb_AssertFailed callback override.

Coexistence Utility

Coexistence Utility

RAIL Coexistence Utility

This optional software component can be enabled to include default functionality related to the coexistence configuration.

See the following documents for in-depth explanations about how to configure and use the available coexistence features:

- AN1128: Bluetooth® Coexistence with Wi-Fi®
- AN1017: Zigbee® and Silicon Labs® Thread Coexistence with Wi-Fi®
- UG350: Silicon Labs Coexistence Development Kit (SLWSTK-COEXBP)

Direct Memory Access (DMA) Utility

Direct Memory Access (DMA) Utility

RAIL DMA Utility

This optional software component can be enabled to include default functionality related to DMA configuration.

Configuration Options

The following configuration options can be changed:

- Enable/disable a DMA channel for use by RAIL. Allocating a DMA channel to the RAIL library allows for faster radio initialization.
  - Allow for a DMA channel to be selected automatically.
  - If not automatically allocating a DMA channel, identify a specific channel to allocate.

Energy Friendly Front End Module (EFF) Utility

Energy Friendly Front End Module (EFF) Utility

RAIL EFF Utility

This optional software component can be enabled to include default functionality related to EFF configuration.

The EFF is a high-performance, transmit/receive (T/R) front end module (FEM) for sub-GHz EFR32 devices. The device transmit chain features a power amplifier and a switched attenuator to extend the dynamic range of the device. The device receive chain features a low-noise amplifier (LNA), a voltage peak detector to determine presence of blockers and an LNA position that can be selected to be either before or after the external filter to optimize system noise floor in the absence of blockers.

When using a Silicon Labs-developed board with an EFF, the configuration options for this software component are set up automatically based on the selected board. When using a custom board, manually configure the configuration options.

Configuration Options

The following configuration options can be changed:

- Enable/disable Rural LNA Mode.
- Enable/disable Urban LNA Mode.
- Enable/disable Bypass LNA Mode.
- Configure the maximum continuous transfer power.
- Configure the maximum transmit duty cycle.

The following hardware pins must be enabled to use the EFF and are fixed to specific GPIO pins on your chip that cannot be changed:

- The CTRL0 GPIO pin.
- The CTRL1 GPIO pin.
- The CTRL2 GPIO pin.
- The CTRL3 GPIO pin.
- The SENSE GPIO pin.

The following hardware options can be set based on your design:

- Configure the TEST GPIO pin.
- Specify the part number of the attached EFF device.

Front End Module (FEM) Utility

Front End Module (FEM) Utility

Radio FEM Utility

Enabling this optional software component includes default functionality related to FEM configuration. It also configures the GPIOs together with the Peripheral Reflex System (PRS) signals to control the FEM.

Note that this is a Radio Utility, instead of a RAIL Utility, because this code is independent from the RAIL library.

Configuration Options

The following options can be configured:

- Enable RX Mode: Enables the output of RX based on the selected port and pin.
- Enable TX Mode: Enables the output of TX based on the selected port and pin.
- Enable Bypass Mode: Enables communication that bypasses the PA (Power Amplifier) and the LNA (Low Noise Amplifier) on the FEM.
- Enable TX High Power Mode: Enables high power transmit mode on the FEM, if disabled low power transmit will be used.

Typically, the signal routing is as follows:

- SL_FEM_UTIL_BYPASS: Bypass control (FEM pin **CPS**).
- SL_FEM_UTIL_RX: RX control (FEM pin **CRX**). Active based on the radio activity (whenever the LNA is enabled).
- SL_FEM_UTIL_SLEEP: Sleep control (FEM pin **CSD**). Active based on the radio activity (whenever the PA or LNA are enabled).
- SL_FEM_UTIL_TX: TX control (FEM pin **CTX**). Active based on the radio activity (whenever the PA is enabled).
- SL_FEM_UTIL_TX_HIGH_POWER: Tx power mode control (FEM pin **CHL**).

Refer to the FEM datasheet for the required settings as not all configuration options are applicable to every module.

Initialization Utility

Initialization Utility

RAIL Initialization Utility

This optional software component can be enabled to include default functionality related to RAIL initialization configuration.

Note: Multiple instances of this component can be enabled in application code. This is particularly useful for multiprotocol scenarios.

Configuration Options

The following configuration options can be changed:

- Enable/disable RAIL initialization.
  - Enable/disable Bluetooth BLE support. If not using Bluetooth, this option can be disabled to reduce memory usage, but if left enabled, no difference in performance occurs.
  - Enable/disable dynamic multiprotocol (DMP) support. This option allocates memory for use by the radio scheduler incorporated into the multiprotocol RAIL library. This should be left disabled for single protocol RAIL library use, which includes switched multiprotocol (SMP) support.
  - Enable/disable initialization complete callback.
- Enable/disable channel configuration.
  - Configure a proprietary channel configuration on boot. This specifically involves the use of the `channelConfigs` array in rail_config.c/h.
  - If using a proprietary channel configuration, specify the channel index for use (i.e., index `x` of array `channelConfigs[x]`).
- Enable/disable calibration configuration.
  - Enable/disable temperature-dependent calibrations (e.g., voltage controlled oscillator calibration).
  - Enable/disable one-time calibrations (e.g., image rejection calibration).
- Enable/disable auto transition configuration.
  - Configure TX transitions on Success and Failure.
  - Configure RX transitions on Success and Failure.
- Enable/disable radio callback event configuration.
  - Configure RX events.
  - Configure TX events.
  - Configure protocol-specific events.
  - Configure dynamic multiprotocol-specific events.
  - Configure maintenance events. Note: Some protocol-specific events share the same event flag with other protocols; enabling one will enable them all. See rail_types.h::RAIL_Event_t.

Packet Trace Interface (PTI) Utility

Packet Trace Interface (PTI) Utility

RAIL PTI Utility

This optional software component can be enabled to configure the PTI interface. The PTI interface allows the user to collect receive and transmit packet data in real time over a dedicated serial interface for debugging network activity.

When using a Silicon Labs radio board, the configuration options for this software component are set up automatically to match the pin mapping on that board. When using a custom board these options can be set any way that makes sense to the user. Keep in mind that the Silicon Labs WSTK is able to capture and parse this data but currently requires UART mode and defaults to a 1.6Mbps data rate.

Configuration Options

The following configuration options can be changed:

- PTI mode
  - RAIL_PTI_MODE_UART : In this mode two pins will be used. One for normal 8 bit UART data (no parity, one stop bit, LSB first) and one for a framing signal to indicate when a packet begins and ends.
  - RAIL_PTI_MODE_SPI : In SPI mode three pins will be used. One for data, one for the clock, and one for the framing signal.
  - RAIL_PTI_MODE_UART_ONEWIRE : In this mode one pin will be used for 9 bit UART data (no parity, one stop bit, LSB first) where the 9th bit is 1 for control bytes and 0 for normal data. This along with knowledge of the control bytes can be used to parse the data stream.
  - RAIL_PTI_MODE_DISABLED : Turn off the peripheral.
- PTI baudrate
  - The PTI buadrate is created using an 8 bit integer divisor on the radio clock (HFXO) frequency. This limits the selectable range of baud rate values but also means that as the frequency increases the achievable baud rates will be spaced further apart. For this reason SPI mode is preferred at high data rates. Valid ranges for different crystal frequencies are shown below:
    - 38.4 MHz HFXO: Baud rates in the range [149.4kbps, 19.2Mbps]
    - 39.0 MHz HFXO: Baud rates in the range [151.8kbps, 19.5Mbps]

The following hardware options can be changed:

- Configure the DOUT GPIO pin. DOUT is needed for all PTI modes (UART, UART Onewire, SPI) and contains the actual serial data.
- Configure the DFRAME GPIO pin. DFRAME is needed for UART and SPI modes only and is used to frame the start and end of packets. This is not used for UART Onewire mode.
- Configure the DCLK GPIO pin. DCLK is needed for SPI mode only.

Power Amplifier (PA) Utility

Power Amplifier (PA) Utility

RAIL PA Utility

This optional software component can be enabled to include default functionality related to PA configuration.

When using a Silicon Labs-developed board, the configuration options for this software component are set up automatically based on the selected board. When using a custom board, manually configure the configuration options.

Configuration Options

The following configuration options can be changed:

- PA Configuration
  - Initial PA power
  - PA ramp time
    - PA ramp time is a hardware approximation aimed at being as close to the desired ramp time as possible without going over the specified time.
  - Milli-volts provided to the PA supply pin (PA_VDD)
  - 2.4 GHz PA selection (for applicable devices)
  - Sub-1 GHz PA selection (for applicable devices)
- PA Curve Configuration
  - Header file containing custom PA curves
  - Header file containing PA curve type types
- PA Calibration Configuration
  - Apply PA Calibration Factory Offset
    - PA calibration offset ensures that PA power remains consistent chip-to-chip. These adjustments occur automatically when the PA is in use, and the application is not notified when these adjustments occur.

See these documents for in-depth explanations of how to configure and use the available PA features:

- AN1127: Power Amplifier Power Conversion Functions in RAIL 2.x

Protocol Utility

Protocol Utility

RAIL Protocol Utility

This optional software component can be enabled to include default functionality related to protocol configuration.

This RAIL utility permits the configuration of protocol-specific settings. Each of these protocols can be configured:

- Bluetooth LE
- IEEE 802.15.4, 2.4 GHz
- IEEE 802.16.5, GB868 (sub 1-GHz)
- Z-Wave

Configuration Options

The following configuration options can be changed:

- Timing configurations common to every protocol
  - Transition Times
    - Transition time (microseconds) from idle to RX
    - Transition time (microseconds) from TX to RX
    - Transition time (microseconds) from idle to TX
    - Transition time (microseconds) from RX to TX
  - RX Search Timeouts
    - Enable RX Search timeout after idle
    - Enable RX Search timeout after TX
- IEEE 802.15.4, 2.4 GHz-specific configurations
  - Node Configurations
    - Enable PAN Coordinator
    - Enable Promiscuous Mode
    - Enable default Frame Pending bit value for outgoing ACKs in response to Data Request Command
  - Receivable Frame Types
    - Beacon Frames
    - Data Frames
    - ACK Frames
    - Command Frames
  - Auto ACKs
    - RX ACK timeout duration (microseconds)
    - Radio state transition after attempting to receive ACK
    - Radio state transition after transmitting ACK
- IEEE 802.16.5, GB868 (sub 1-GHz)-specific configurations
  - The same protocol-specific configurations exist for this protocol as exist for IEEE 802.15.4, 2.4 GHz, above.
- Z-Wave-specific configurations
  - Node Configurations
    - Enable Promiscuous Mode
    - Accept Beam Frames
    - Filter Packets Based on Node ID

Recommended Utility

Recommended Utility

RAIL Recommended Utility

This optional software component can be enabled to include a recommended collection of other optional RAIL utilities.

This RAIL utility attempts to identify those RAIL utilities most commonly used by applications. If a list of utilities different from what is auto-selected here is sought, this component must be removed from the project with the custom list manually being added to the project.

Common RAIL utilities recommended for general use include:

- Initialization
- Callbacks
- Protocol
- Power Amplifier (PA)
- Packet Trace Interface (PTI)
- Received Signal Strength Indicator (RSSI)
- Direct Memory Access (DMA)
  - This primarily accelerates radio initialization, so it is currently included only for dynamic multiprotocol (DMP) scenarios.

RF Path Utility

RF Path Utility

RAIL RF Path Utility

This optional software component can be enabled to include default functionality related to device-internal and device-external RF path configuration.

When using a Silicon Labs-developed board, the configuration options for this software component are set up automatically based on the selected board. When using a custom board, manually configure the configuration options.

Configuration Options

The following configuration options can be changed:

- Device-internal (EFR32XG21 only)
  - Switch between 2 RF paths (i.e., device pins).

Received Signal Strength Indicator (RSSI) Utility

Received Signal Strength Indicator (RSSI) Utility

RAIL RSSI Utility

This optional software component can be enabled to include default functionality related to RSSI configuration.

To make RSSI values more consistent among families of chips, this software offset can be automatically applied.

Configuration Options

The following configuration options can be changed:

- Software RSSI offset value

RAILtest Application Core

RAILtest Application Core

RAILtest Core

This software component contains the core functionality of RAILtest and must be enabled for RAILtest to operate.

Configuration Options

The following configuration options can be changed:

- Enable/disable external radio configuration file support (e.g., rail_config.c/h).
- Configure default radio configuration for use.
- Alter application name output over CLI on boot. Default is `RAILtest`.
- Alter transfer period between consecutively transmitted packets.
- Alter maximum packet length. This is the size of the TX buffer in the RAILtest application, which contains the packet eventually to be loaded into the RAIL library's TX FIFO for transmission.
- Alter transmit buffer size. This is the size of the TX FIFO provided to the RAIL library.
- Alter receive buffer size. This is the size of the RX FIFO provided to the RAIL library.
- Alter the duration that distinguishes between short and long button presses.

The following hardware options can be changed:

- Configure the Packet Error Rate (PER) GPIO pin. By default, this RAILtest output GPIO pin is not configured for use. For additional information regarding its use in RAILtest, see the section **Packet Error Rate Testing** in the document **UG409: RAILtest User's Guide**.

Various RAILtest-specific buffer dependencies exist. For example, to transmit and receive a 4096 byte packet (i.e., the largest packet size supported in TX and RX packet modes), the buffers must be configured as follows:

- Note: TX and RX buffer sizes provided to the RAIL library must comply with the sizing constraints identified in the section **Receive and Transmit FIFO Buffers** of EFR32.
- TX device
  - SL_RAIL_TEST_MAX_PACKET_LENGTH = 4096
    - This is the size of the TX buffer used by the RAILtest application to contain the packet that will eventually be loaded into the library's TX FIFO.
    - Up to 4096 bytes may be transmitted if the receiving device is not collecting the RX packet's appended info (e.g., no CRC status, no RSSI value, no timestamp).
    - Up to 4090 bytes may be transmitted if the receiving device is collecting the RX packet's appended info (e.g., CRC status, RSSI value, timestamp).
  - SL_RAIL_TEST_TX_BUFFER_SIZE = 4096 // must be power of 2
    - This is the size of the TX FIFO made available to the RAIL library.
- RX device
  - BUFFER_POOL_ALLOCATOR_BUFFER_SIZE_MAX = 4144
    - This is the size of the buffer used by the RAILtest application for CLI output purposes. See the `Memory Manager` software component to configure this value.
    - 4144 = receive 4096 byte packet + 0 bytes appended info (i.e., no CRC status, no RSSI value, no timestamp) + 48 bytes extra info for printf (i.e., sizeof(RailEvent_t))
    - 4144 = receive 4090 byte packet + 6 bytes appended info (i.e., CRC status, RSSI value, timestamp) + 48 bytes extra info for printf (i.e., sizeof(RailEvent_t))
  - SL_RAIL_TEST_RX_BUFFER_SIZE = 4096 // must be power of 2
    - This is the size of the RX FIFO made available to the RAIL library.

RAILtest Application Graphics

RAILtest Application Graphics

RAILtest Graphics

This software component can optionally be enabled for RAILtest. If enabled, the following information is displayed on the LCD:

- Number of transmitted packets
- Number of received packets
- Channel number
- Application name

When using a Silicon Labs-developed board, the hardware GPIO configuration is set up automatically for this software component based on the selected board. When using a custom board, manually configure the hardware pins. This can be done from within the `Memory LCD` software component. The `Memory LCD` component determines which LCD displays are compatible with the `RAILtest, Graphics` component. (The `Memory LCD` component is a dependency of the `RAILtest, Graphics` component and is included automatically in a project when the `RAILtest, Graphics` component is enabled.)

Configuration Options

The following configuration options can be changed:

- Alter application name displayed on boot. Default is `RAILtest` .

Thermistor Utility

Thermistor Utility

RAIL Thermistor Utility

This optional software component can be enabled to include advanced functionalities related to temperature on target having RAIL_SUPPORTS_EXTERNAL_THERMISTOR bit set.

The thermistor component allows the user to enabled multiple features related to temperature such as:

- High Frequency Crystal Oscillator (HFXO) compensation
- Thermal protection

Note that thermistor can be used independantly of this component using already exisiting API such as:

- RAIL_StartThermistorMeasurement().
- RAIL_GetThermistorImpedance().

This component offers a default configuration and conversion functions.

When using a Silicon Labs-developed board with a thermistor, the configuration options for this software component are set up automatically and the conversion functions are based on the selected board. When using a custom board, user must manually configure the configuration options. If the thermistor is custom as well, user must manually override the conversion functions.

Configuration Options

The following hardware options can be changed:

- Configure the THERMISTOR GPIO pin.
- Configure the THERMISTOR GPIO port.

The following conversion functions can be changed:

- Impedance to temperature conversion (RAIL_ConvertThermistorImpedance()).
- Temperature to PPM deviation conversion (RAIL_ComputeHFXOPPMError()).

RF Path Switch Utility

RF Path Switch Utility

RAIL RF Path Switch Utility

This optional software component can be enabled to control a GPIO toggled switch based on RFPATH selection and radio state. An example usage of this component is toggling a switch on RFPATH0 to ground when RFPATH1 is in use to avoid spurious harmonics.

When using a Silicon Labs-developed board, the configuration options for this software component are set up automatically based on the selected board. When using a custom board, manually configure the configuration options.

Configuration Options

The following configuration macros are available for modification:

- RF Path Switch:
  - SL_RAIL_UTIL_RF_PATH_SWITCH_CONTROL_PORT: GPIO port for rfpath selection signal output.
  - SL_RAIL_UTIL_RF_PATH_SWITCH_CONTROL_PIN: GPIO pin for rfpath selection signal output.
  - SL_RAIL_UTIL_RF_PATH_SWITCH_INVERTED_CONTROL_PORT: GPIO port for logical NOT rfpath selection signal output.
  - SL_RAIL_UTIL_RF_PATH_SWITCH_INVERTED_CONTROL_PIN: GPIO pin for logical NOT rfpath selection signal output.
  - SL_RAIL_UTIL_RF_PATH_SWITCH_RADIO_ACTIVE_PORT: GPIO port for radio active signal output.
  - SL_RAIL_UTIL_RF_PATH_SWITCH_RADIO_ACTIVE_PIN: GPIO pin for radio active signal output.
  - SL_RAIL_UTIL_RF_PATH_SWITCH_RADIO_ACTIVE_MODE: Logical AND radio active signal output with CONTROL or INVERTED_CONTROL signal outputs if set to SL_RAIL_UTIL_RF_PATH_SWITCH_RADIO_ACTIVE_COMBINE.

Radio Sequencer Image Selection Utility

Radio Sequencer Image Selection Utility

RAIL Utility for Selecting a Radio Sequencer Image

This optional software component allows for the selection of a specific radio sequencer image on platforms that support multiple images. The selection can be done automatically based on properties of the OPN, or manually based on the application needs of the user.

Currently, only EFR32XG24 supports multiple sequencer images. The two images available for that platform are for the 10 dBm PA and 20 dBm PA variants. This component will select the correct image automatically based on the OPN. In the future, other sequencer images may be introduced on other platforms that allow the user to choose an application specific custom sequencer image.

On platforms that support only one default sequencer image, this component is safe to include, and will have no effect.

Configuration Options

The following configuration options can be changed:

- Enable/disable run-time selection of the sequencer image.

Software Modem (SFM) sequencer image selection Utility

Software Modem (SFM) sequencer image selection Utility

RAIL Utility for selecting SFM sequencer image

This optional software component allows for the selection of a specific SFM sequencer image on EFR32XG25 platform. Currently, available images are intended for the support of following sets of modulations:

- SUN FSK + SUN OFDM + SUN OQPSK
- SUN OFDM + SUN OQPSK
- SUN OFDM
- NONE

EFR32

EFR32

While RAIL attempts to be chip-agnostic, certain hardware specifics can't be overlooked. Where possible, missing features will be simulated in software, but performance characteristics may vary. This section covers EFR32-specific content including hardware-specific configurations or calibrations.

Clocks

High-Frequency Clocks

The EFR32 has a configurable clock tree with a number of different prescalars. One thing to keep in mind, however, is that the radio must always run off of an HFXO at 38.4 MHz. This means that you must switch your HFCLK source to the HFXO before calling RAIL_Init() and make sure to leave the CMU->HFPRESC set to 1 to prevent dividing down the radio clock. You are free to scale the peripheral (CMU->HFPERPRESC) or Cortex (CMU->HFCOREPRESC) clocks using their prescalars, which are further down the clock tree. Keep in mind that this may introduce extra wait states when interacting with the radio and slow down some operations. On EFR32xG22, we limit going to EM1P sleep mode when an 80MHz HRFRCO PLL system clock is selected. Going to EM1P sleep may cause a noticeable clock drift which would impact radio timing and tuning.

If using the DPLL on EFR32 Series-2 platforms, it is best to configure cmuDPLLLockMode_Phase where the frequency error will tend to 0. Using cmuDPLLLockMode_Freq should be avoided because it can cause issues with applications requiring very precise frequencies like proprietary protocols or Bluetooth.

Low-Frequency Clocks

RAIL can use a low-frequency clock on the EFR32 to synchronize its time base across EM2 sleep. This only happens if you call RAIL_ConfigSleep with RAIL_SleepConfig_t::RAIL_SLEEP_CONFIG_TIMERSYNC_ENABLED. On the EFR32xG1 and EFR32xG12 this uses the RTCC's clock which is the LFE clock with a possible additional prescalar. The slower the clock source, the longer it will take to synchronize because the process requires several ticks of the RTCC clock. EFR32xG13 and EFR32xG14 have a separate timer to help with the synchronization, but either the LFXO or LFRCO are still used as a clock. Note that to use this internal timer the cmuClock_HFLE clock must always be enabled. When calling RAIL_ConfigSleep(), the code will first try to set up the timer with the LFXO if it's started, then fall back to the LFRCO, and finally assert if no LF clocks are running.

Receive and Transmit FIFO Buffers

The RAIL library for the EFR32 optionally allocates an internal buffer to store receive data contiguously. Its required alignment is 32-bit (4-byte); applications can use RAIL_FIFO_ALIGNMENT or the RAIL_FIFO_ALIGNMENT_TYPE to ensure conformance.

The buffer is set to 512 bytes at build time by default, but can be changed by calling RAIL_SetRxFifo, for example in an implementation of RAILCb_SetupRxFifo. The receive FIFO's size must be a power of 2 between 64 and 4096 bytes for hardware compatibility.

The chosen buffer size limits the maximum size of receive packets in packet mode and determines the size of the receive FIFO in FIFO mode. Because each receive packet has several bytes of overhead, you can only receive up to one (buffer size - overhead) byte packet without switching to FIFO mode. In FIFO mode, you must read out packet data as you approach this limit and store it off to construct the full packet later. This overhead is currently 8 bytes on all EFR32 platforms except EFR32xG1 where it is 6 bytes. Note that this overhead may increase or decrease in future releases as the functionality is changed though large jumps are not expected in either direction.

Supplementing the receive FIFO used for packet data, the RAIL library also allocates an internal packet metadata FIFO capable of holding up to 16 packets' worth of metadata. An entry in this FIFO is made on every packet completion (successful or not) signaled to the application. If the application calls RAIL_HoldRxPacket() that entry is held along with any

of its packet data in the receive FIFO. When either the receive FIFO or internal packet metadata FIFO fills or overflows, RAIL_EVENT_RX_FIFO_FULL or RAIL_EVENT_RX_FIFO_OVERFLOW occur at packet completion time, respectively.

The transmit FIFO must be set at runtime with RAIL_SetTxFifo(). Like the receive FIFO the transmit FIFO's required alignment is 32-bit (4-byte); applications can use RAIL_FIFO_ALIGNMENT or the RAIL_FIFO_ALIGNMENT_TYPE to ensure conformance.

The transmit FIFO's size has the same restrictions as the receive FIFO. The size must be a power of 2 from 64 to 4096 for hardware compatibility. Note that there is no difference between packet and FIFO mode on the transmit side. A packet may either be loaded all at once or in pieces using the RAIL_WriteTxFifo() function. You may also use the RAIL_SetTxFifoThreshold() function and the RAIL_EVENT_TX_FIFO_ALMOST_EMPTY event to load data as there is space available in the FIFO during transmission.

RAIL also provides an internal transmit AutoAck FIFO whose size is fixed at RAIL_AUTOACK_MAX_LENGTH bytes. Applications can write to it via RAIL_WriteAutoAckFifo(). Since RAIL does not permit the queuing of multiple transmits, there is no need for any internal transmit metadata FIFO like the one which exists on the receive side.

Data Reception Sources

When receiving data, you can configure hardware to provide data from three different hardware sources.

First, you can configure the hardware to provide a packet of information. This configuration uses the built-in demodulator and frame controller. Use RAIL_RxDataSource_t::RX_PACKET_DATA to enable this receive source.

Second, you can configure hardware to provide data directly from the demodulator. In this mode, hardware demodulation is used, but the user is responsible for implementing frame controller functionality. In other words, preamble detection, sync word detection, CRC validation, and so on must be performed in software by the application. All data returned is represented as 8-bit 2's-compliment values. Use RAIL_RxDataSource_t::RX_DEMOD_DATA to enable the receive source.

Third, you can configure hardware to provide data directly from the I and Q ADCs. In this mode, the user is responsible for implementing demodulator and frame controller functionality. The receive signal hardware has a 19-bit dynamic range. The user can select whether to return the upper 16 bits of the 19-bit value (RAIL_RxDataSource_t::RX_IQDATA_FILTMSB) or the lower 16 bits (RAIL_RxDataSource_t::RX_IQDATA_FILTLSB). All data returned is represented as 16-bit 2s-complement values. The I and Q values are put into the buffer in an alternating fashion where the first 16-bit value is the first I ADC value and the second 16-bit value is the first Q ADC value, and so on.

Interrupt Vectors

The RAIL library for EFR32 implements all of the interrupt vectors for radio peripherals, which is required for the RAIL library to function correctly. RAIL does not, however, set the priorities of these interrupt vectors except for FRC_PRI_IRQHandler on EFR32xG12 which runs at the highest priority. The others must be handled by your application using the CMSIS NVIC_SetPriority() API or direct manipulation of the NVIC. Below is the full list of interrupts used by the radio. You **must** run them all except FRC_PRI_IRQHandler at the same priority which is what all Silicon Labs RAIL applications do by default. You are free to choose what that priority is based on the requirements of your application (e.g., putting them below FreeRTOS OS atomic for access to OS functions in RAIL callbacks). Keep in mind that putting the radio interrupts at too low a priority can cause missed packets and other radio performance issues. A restriction on EFR32xG12 and newer is that you must not disable radio interrupts for longer than a quarter of the RAIL timebase (2^32/4 microseconds or 18 minutes) to ensure proper timekeeping. See EFR32xG1x_Interrupts or EFR32xG2x_Interrupts for more information.

Chip-Specific Initialization

EFR32 Hardware Initialization

EFR32 includes EMLIB and EMDRV to create a basic HAL layer. A lot of this initialization code is completely up to you, but there are a couple of requirements when building a RAIL app. Specifically, the radio will only work if you are running off a high-precision crystal oscillator. Since some APIs will assume this is running, make sure to initialize and switch to the crystal before calling any radio APIs.

For the Wireless Starter Kit (WSTK), you can use the crystal configuration in the HAL configuration header file for your specific kit. Example code for this is shown below. If you have a custom hardware layout, you may want to create your own HFXOInit structure to account for things such as your specific CTUNE value.

```
#include "bsp.h" // Contains WSTK versions of the HFXO init structure

void efrStartup(void)
{
 CMU_HFXOInit_TypeDef hfxoInit = CMU_HFXOINIT_WSTK_DEFAULT;

 // Initialize the HFXO using the settings from the WSTK bspconfig.h
 // Note: This configures things like the capacitive tuning CTUNE variable
 //   which can vary based on your hardware design.
 CMU_HFXOInit(&hfxoInit);

 // Switch HFCLK to HFXO and disable HFRCO
 CMU_ClockSelectSet(cmuClock_HF, cmuSelect_HFXO);
 CMU_OscillatorEnable(cmuOsc_HFRCO, false, false);
}
```

Radio-Specific Initialization

Currently, RAIL_ConfigTxPower() and RAIL_ConfigPti() initialization functions depend on the board configuration and must be manually called at startup. You **MUST** initialize the power amplifier (PA) to transmit. You may initialize the packet trace interface (PTI) for use in debugging.

Power Amplifier (PA) Initialization

High Power 2.4 GHz PA, Low Power 2.4 GHz PA, and Sub GHz PA PAs are available for the EFR32xG1 family of chips. The specific set of PAs you have available and the supported power levels for those PAs are determined by your part number. See the appropriate data sheet for more details.

Each PA supports a raw power level value of type RAIL_TxPowerLevel_t where a numerically smaller number causes less power to be output from the chip than a higher number. Keep in mind that these values may be capped at the upper and lower ends and do not necessarily map linearly to output power in dBm. To map these values to and from dBm values, the RAIL library uses the RAIL_ConvertRawToDbm() and RAIL_ConvertDbmToRaw() APIs. By default, these use whatever piecewise linear power curves were passed into RAIL_InitTxPowerCurves() to do the conversion. On Silicon Labs boards, these curves are included with the board header files. For custom boards, this must be measured using test equipment to take into account the differences in the RF path that may exist. In addition, for even more custom use cases, the conversion functions can be strongly defined in the customer application and provide whatever type of mapping is found to be most effective. Below is an example of using the standard APIs to initialize all PAs on a dual band chip and switch to the 2.4 GHz high-power PA.

```
#include "rail.h"
#include "rail_chip_specific.h"
#include "plugin/pa-conversion/pa_conversion_efr32.h"

RAIL_DECLARE_TX_POWER_VBAT_CURVES(piecewiseSegments, curvesSg, curves24Hp, curves24Lp);

// Must be called with a valid RAIL_Handle_t returned by RAIL_Init()
void initPa(RAIL_Handle_t myRailHandle)
{
  // Initialize the RAIL Tx power curves for all PAs on this chip
  RAIL_TxPowerCurvesConfig_t txPowerCurvesConfig = {
    curves24Hp,
    curvesSg,
    curves24Lp,
    piecewiseSegments
  };
  if (RAIL_InitTxPowerCurves(&txPowerCurvesConfig) != RAIL_STATUS_NO_ERROR) {
    // Could not initialize transmit power curves so something is configured
    // wrong. Please fix and rebuild.
    while(1);
  }

  // Switch to the 2.4GHz HP PA powered off the 1.8V DCDC connection
  RAIL_TxPowerConfig_t railTxPowerConfig = {
    RAIL_TX_POWER_MODE_2P4_HP, // 2.4GHz HP Power Amplifier mode
    1800,                      // 1.8V vPA voltage for DCDC connection
    10                         // Desired ramp time in us
  };
  if (RAIL_ConfigTxPower(myRailHandle, &railTxPowerConfig)
      != RAIL_STATUS_NO_ERROR) {
    // Error: The requested PA could not be selected. Fix your configuration
    // and try again.
    while(1);
  }

  // Set the output power to the maximum supported by this chip
  RAIL_SetTxPower(myRailHandle, 255);
}
```

Packet Trace Interface (PTI) Initialization

Packet trace on the EFR32 provides a mechanism for viewing transmitted and received radio packets for network sniffing or debugging. It can also be captured by a WSTK and sent to Simplicity Studio for viewing data in its Network Analyzer tool.

To use this functionality, configure the pins to use for this output and, optionally, how to format data. Note that the WSTK **requires** the following output format:

```
Mode: 8 bit UART mode
Baudrate: 1.6 Mbps
Framing Signal: Enabled
```

Introducing changes is currently unsupported by the WSTK. Other output formats are currently unsupported.

To choose the pins, look in the applicable data sheet and select a valid route location for the FRC_DOUT and FRC_DFRAME signals. These map to PTI.DATA and PTI.FRAME on the WSTK respectively. Once you've found pins that work for your hardware, configure the RouteLocation, port, and pin fields in the PTI initialization structure. When using the WSTK for example, initialize as follows:

```
#include "rail.h"
#include "rail_chip_specific.h"

void initPti(void)
{
 RAIL_PtiConfig_t ptiConfig =  {
  RAIL_PTI_MODE_UART,    // Only supported output mode for the WSTK
  1600000,               // Choose 1.6 Mbps for the WSTK
  6,                  // WSTK uses location 6 for DOUT
  gpioPortB,             // FRC_DOUT#6 is PB12
  12,                 // FRC_DOUT#6 is PB12
  6,                  // UNUSED IN UART MODE
  gpioPortB,             // UNUSED IN UART MODE
  11,                 // UNUSED IN UART MODE
  6,                  // WSTK uses location 6 for DFRAME
  gpioPortB,             // FRC_DOUT#6 is PB13
  13,                 // FRC_DOUT#6 is PB13
 };

 // Initialize the Packet Trace Interface (PTI) for the EFR32
 RAIL_ConfigPti(RAIL_EFR32_HANDLE, &ptiConfig);

 // Enable Packet Trace (PTI)
 RAIL_EnablePti(RAIL_EFR32_HANDLE, true);
}
```

Other Radio GPIO Functions

Various useful signals related to the radio can be output on a GPIO using the Peripheral Reflex System (PRS). PRS is an advanced system where you can route signals to channels and then output those channels to a number of configurable locations. For more information, see the PRS chapter in the reference manual.

A list of some of the most interesting PRS signals related to the radio is shown below along with how to enable them in the PRS. The definition for these signals for a given chip can be found in the release at `platform/Device/SiliconLabs/<chipFamily>/Include/<chipFamily>_prs_signals.h` where `<chipFamily>` would be the beginning of the part number (EFR32MG12P for example).

Signal	Summary
PRS_RAC_ACTIVE	Radio enabled
PRS_RAC_TX	Transmit mode enabled
PRS_RAC_RX	Receive mode enabled
PRS_RAC_LNAEN	LNA enabled for RX
PRS_RAC_PAEN	PA enabled for TX
PRS_MODEM_FRAMEDET	Frame detected
PRS_MODEM_PREDET	Preamble detected
PRS_MODEM_TIMDET	Timing detected
PRS_MODEM_FRAMESENT	Frame sent
PRS_MODEM_SYNCSENT	Sync word sent
PRS_MODEM_PRESENT	Preamble sent

The example below shows how to configure a PRS channel to output RAC_RX on a GPIO. In this example, the WSTK is used with the BRD4153A radio board. The code below will put RAC_RX on PRS Channel 0 and output PRS Channel 0 on pin PC10 which is wired to WSTK_P12 and EXP_HEADER15 on the WSTK.

```
#include "em_cmu.h"
#include "em_prs.h"
#include "em_gpio.h"
```

```
#include "em_prs.h"
#include "em_gpio.h"
#include "em_device.h"

void enableDebugGpios(void){// Turn on the PRS and GPIO clocks to access their
registers.CMU_ClockEnable(cmuClock_PRS,true);CMU_ClockEnable(cmuClock_GPIO,true);// Configure PC10 as an
output.GPIO_PinModeSet(gpioPortC,10, gpioModePushPull,0);// Configure PRS Channel 0 to output RAC_RX.PRS_SourceSignalSet(0,((PRS_RAC_RX
& _PRS_CH_CTRL_SOURCESEL_MASK)>> _PRS_CH_CTRL_SOURCESEL_SHIFT),((PRS_RAC_RX & _PRS_CH_CTRL_SIGSEL_MASK)>>
_PRS_CH_CTRL_SIGSEL_SHIFT),
                prsEdgeOff);// Configure PRS Channel 0 to use output location 12 (PC10 - see data sheet).
 PRS->ROUTELOC0 &=~_PRS_ROUTELOC0_CH0LOC_MASK;
 PRS->ROUTELOC0 |= PRS_ROUTELOC0_CH0LOC_LOC12;// Enable PRS Channel 0.
 PRS->ROUTEPEN |= PRS_ROUTEPEN_CH0PEN;}
```

Required Dependencies

Most of the RAIL library is self-contained, however, there are some dependencies on external functions, such as functions from the C standard library, EMLIB, and CMSIS. Below is a complete list of these dependencies. Note that changing the implementation of any of these functions while maintaining their functionality could still impact RAIL operation.

Group	Functions
CMSIS	SystemHFXOClockGet() SystemHFClockGet() SystemLFRCOClockGet() SystemLFXOClockGet() SystemULFRCOClockGet()
EMLIB (em_cmu)	CMU_ClockEnable CMU_ClockSelectGet CMU_ClockSelectSet CMU_OscillatorEnable
EMLIB (em_gpio)	GPIO_PinModeSet()
EMLIB (em_system)	SYSTEM_ChipRevisionGet()
EMLIB (em_core)	CORE_EnterCritical() CORE_ExitCritical() CORE_EnterAtomic() CORE_ExitAtomic()
EMLIB (em_emu)	EMU_DCDCLnRcoBandSet()
stdlib	memcpy() memset()

Peripherals Consumed by RAIL

Certain functionality in RAIL requires some of the chip's peripherals. The specifics of these requirements are enumerated below.

RAIL Timer Synchronization

If you call RAIL_ConfigSleep with RAIL_SleepConfig_t::RAIL_SLEEP_CONFIG_TIMERSYNC_ENABLED to keep the clock synchronized across sleep RAIL must use the PRS and RTCC.

- On all platforms, PRS channel 7 is used to perform the synchronization. This channel is only initialized one time when calling the RAIL_ConfigSleep function so you must not change it after that or the synchronization may fail.
- On the EFR32xG1, EFR32xG12, and EFR32xG21 platforms, the timer synchronization also uses RTCC channel 0. This channel should not be used by the application after a call to RAIL_Sleep and before the corresponding RAIL_Wake call. The application is also responsible for enabling this top level interrupt if they want it to serve as a wake source and implementing an interrupt handler that clears the RTCC_IFC_CC0 flag in the RTCC->IF register any time it is pended to prevent the chip from getting stuck in the handler. RAIL does not need to be tied into the interrupt handler in any other way.
- On the EFR32xG13 and EFR32xG14 platforms you must also ensure that the CMU's cmuClock_HFLE is enabled or the internal timer used for sleep will not be able to power up. This happens by default on the other platforms when configuring the RTCC but since this is not required on these chips it must be done separately.

DCDC

If you enable or disable the DCDC peripheral, you must follow it by calling RAIL_ChangedDcdc. This API indicates that the DCDC peripheral bus clock has changed allowing RAIL to react accordingly.

Entropy Generation

EFR32 supports true entropy collection using the radio. It is able to collect 1 bit per radio clock cycle while in receive mode. This means that the receiver must be enabled when collecting entropy. If you attempt to transmit or otherwise delay entry into receive, data collection will take longer. You can still receive packets while collecting entropy. This means that requesting random numbers longer than the length of your preamble and sync word may trigger a packet reception. This may be fixed in a future revision but for now it ensures a proper packet reception during a random data collection.

Due to the nature of random data collection, either the number of bytes or zero is returned after the full amount requested. Zero is returned if the radio is uninitialized and cannot be enabled for collection.

RAIL Timebase

EFR32 uses a dedicated radio timer to create the RAIL timebase and perform scheduled transmit and receive operations. This timer ticks at roughly every 2 us on the EFR32xG1 platform and 0.5 us on the EFR32xG12 and newer. The exact tick value depends on the high-frequency crystal in use so there may be a small rounding error. All internal APIs account for this error, so over long periods of time it is generally unnoticeable. Note that RAIL uses the hardware tick rate to create a 1 us time base that is exposed to the user. This is all that should matter for most users and the information here is included to explain the underlying implementation.

Radio Calibration

EFR32 supports image rejection (IR) calibrations, VCO temperature calibration, and HFXO compensation. You may choose to enable some or all depending on your use case. Using IRCAL and VCO_TEMPCAL is recommended for optimal performance of the radio.

Image Rejection Calibration (IRCAL)

This calibration should be run each time your PHY configuration changes. It is based on the modulation scheme, frequency band, and other radio settings. RAIL will request this by calling RAIL_Config_t::eventsCallback with the RAIL_EVENT_CAL_NEEDED bit set and the RAIL_CAL_ONETIME_IRCAL bit set in RAIL_GetPendingCal().

Using a proper value for this will improve sensitivity by several dBm, so it's highly recommended.

It can take on the order of 700 ms to complete this calibration. You may want to save off a known good value for this calibration and load it each time you switch PHYs to save time.

This calibration should be initialized before calling RAIL_ConfigChannels() to ensure that it is properly configured by the time the first channel is set up. The initialization involves enabling the algorithm and passing in parameters to configure the algorithm for the specific PHY generated by the EFR32 Radio Configurator.

You cannot use the radio while this calibration is being performed or you may generate an incorrect calibration. Application code should ensure that the radio remains in the idle state during this calibration.

This calibration is only meaningful for Zigbee, BLE, and sub GHz radio configurations. For all others you should still enable the calibration but the algorithm will apply a safe, default, calibration value.

Image Rejection Calibration for Transmit (TXIRCAL)

This calibration is only meaningful for sub GHz radio configurations using SUN-OFDM PHYs and it should be run only once per multi-PHY configuration. For all other configurations this calibration won't run. RAIL will request this by calling RAIL_Config_t::eventsCallback with the RAIL_EVENT_CAL_NEEDED bit set and the RAIL_CAL_ONETIME_IRCAL bit set in RAIL_GetPendingCal().

This feature is automatically run after IRCAL, when a SUN-OFDM PHY is configured. If a SUN-OFDM PHY exists in the multi-PHYs radio configuration, it is advised to first load this OFDM PHY and then call RAIL_Calibrate(), so that both the IRCAL and TXIRCAL are run immediately. Otherwise the calibration will be requested the first time an OFDM PHY will be loaded.

Like IRCAL, using a proper value for this will improve sensitivity by several dBm, so it's highly recommended.

It can take on the order of 150 ms to complete this calibration. You may want to save off a known good value for this calibration and load it each time you switch multi-PHY configuration to save time.

You cannot use the radio while this calibration is being performed or you may generate an incorrect calibration. Application code should ensure that the radio remains in the idle state during this calibration.

VCO Temperature Calibration (VCO_TEMPCAL)

When staying in receive for a very long time and the temperature changes causing the radio to drift off-frequency, RAIL will request this calibration via RAIL_Config_t::eventsCallback with the RAIL_EVENT_CAL_NEEDED bit set and the RAIL_CAL_TEMP_VCO bit set in RAIL_CalPendingGet().

This calibration is automatically run every time receive is entered. If the application, by its nature, frequently re-enters receive mode, this calibration may not need to be enabled.

On EFR, the application will get this event when the absolute temperature crosses 0C degrees as well as when the temperature delta from the last calibration increases or decreases by 70C.

It is always recommended to enable and handle this calibration since it doesn't add much overhead and is much safer.

HFXO Temperature Compensation (HFXO_TEMPCOMP)

This optional feature requires plugin Thermistor Utility. More details about the compensation are available in the plugin page.

When the themperature varies too much, HFXO frequency drifts, resulting in signal degradation. The compensation corrects the effects of this deviation in the radio, and is done in 2 steps (2 events).

The first event is RAIL_EVENT_CAL_NEEDED with the RAIL_CAL_TEMP_HFXO bit set in RAIL_GetPendingCal(). This signals the need to call RAIL_StartThermistorMeasurement().

The second event occurs at the end of this measurement triggering RAIL_EVENT_THERMISTOR_DONE with the RAIL_CAL_COMPENSATE_HFXO bit set in RAIL_GetPendingCal(). This signals the need to idle the radio and call RAIL_CalibrateHFXO().

When enabled, RAIL_EVENT_CAL_NEEDED event is automatically set when the number of degrees set in either RAIL_HFXOCompensationConfig_t::deltaNominal or RAIL_HFXOCompensationConfig_t::deltaCritical are exceeded.

State Transition Timing

EFR32 allows automatic transitions from receive to transmit to precisely time transmitted packets after packet reception. During this process, internal calculations are performed which require the received packet's duration, from sync word to end of CRC, to be less than 32 ms. If received packets in a given protocol have an on-air time greater than 32 ms, the automatic transition from receive to transmit may not be used with that protocol because of incorrect timing of that transition.

Radio State Verification

RAIL includes a radio state verification feature capable of verifying radio register contents. There are multiple ways of configuring and running the verification process. All radio state verification should occur when the radio is idle. If not idle, the number of radio registers different from their reference values varies at any given time.

Default vs. Custom Radio Configuration

When verification occurs, a reference value is compared with a radio register's contents. The reference value is provided through one of two ways.

The radio configuration (originally passed into RAIL_ConfigChannels()) is used as a reference with which to compare radio register values. When the default radio configuration is used for verification, only those addresses flagged as being verifiable by the radio configurator will be checked. Bit 27 of an encoded address indicates whether that address is verifiable or not. For example, encoded address 0x08020004 is flagged as being verifiable and will be verified when RAIL_Verify() is called, but the encoded address 0x00020008 will not be verified.

A custom radio configuration can be provided to the verification API (passed into RAIL_ConfigVerification()). When a custom radio configuration is provided, all addresses in the provided radio configuration will be checked, regardless of whether or not their encoded addresses are flagged as being verifiable.

Approval Callback

When verification occurs, no register value differences are allowed unless an application-level approval callback is defined.

Without an approval callback, any difference will be flagged as data corruption, with RAIL_Verify() returning a status of RAIL_STATUS_INVALID_STATE.

With an approval callback, the application can scrutinize all differences and deem them acceptable or not. By providing an approval callback, individual register fields, instead of the entire register contents, can be compared against reference values at the application level.

Restarting

Verification can be restarted if necessary.

When running verification from the beginning to completion (by specifying a sufficiently long test duration or by specifying a duration of RAIL_VERIFY_DURATION_MAX), the verification process is indifferent to the restart parameter.

When running verification for a duration that does not permit verification to run to completion, the RAIL_Verify() function will return with a status of RAIL_STATUS_SUSPENDED. This indicates test success for those values already verified, but sufficient time was not permitted to verify all values. In this scenario, the initial run of RAIL_Verify() is indifferent to the value of the restart input, but subsequent runs of RAIL_Verify() should set the restart input to false so that verification will resume where it left off on the previous call. To record where a previous verification left off, the RAIL_VerifyConfig_t structure must be declared by the application and provided to RAIL. These structure contents should only be altered by RAIL.

Configuration and Running Options

With the ability to specify default radio configuration vs. custom radio configuration, approval callback vs. no approval callback, and run to completion vs. run for a specific time, there are multiple ways in which to run RAIL's radio state verification feature. The example below shows running verification in each of these modes.

```
#include "rail.h"
#include "rail_types.h"
#include "rail_config.h"

RAIL_VerifyConfig_t configVerify;

const uint32_t customRadioConfig[] = {
  0x00010048UL, 0x00000000UL,
  0x000400A0UL, 0x00004CFFUL,
    /* 00A4 */ 0x00000000UL,
    /* 00A8 */ 0x00004DFFUL,
    /* 00AC */ 0x00000000UL,
  0x00012000UL, 0x00000744UL,
  ...
  0xFFFFFFFFUL,
};

bool RAILCb_VerificationApproval(uint32_t address,
                                 uint32_t expectedValue,
                                 uint32_t actualValue)
{
  responsePrint("verifyRadioCb",
                "address:0x%08x,expectedValue:0x%08x,actualValue:0x%08x",
                address, expectedValue, actualValue);
  return true OR false; // true = change approved; false = change unapproved
}

main ()
{
  RAIL_Status_t result;
  ...
  // Initialize the radio.
  railHandle = RAIL_Init(..., ...);
  ...
  // Associate a radio config with this RAIL handle.
  RAIL_ConfigChannels(railHandle, channelConfigs[0], ...);
  ...
  // Idle the radio before verifying registers.
  RAIL_Idle(railHandle, RAIL_IDLE, true);

  uint8_t verifyConfigOption = 1; // 1,2,3,4
  uint8_t verifyRunOption    = 1; // 1,2

  // Configure the radio state verification in one of 4 ways:
  if (1 == verifyConfigOption) {
    // Config Option 1: Use default radio config with no approval callback.
    RAIL_ConfigVerification(railHandle,
                            &configVerify,
                            NULL,   // use channelConfigs[0] for verification
                            NULL);
  } else if (2 == verifyConfigOption) {
    // Config Option 2: Use default radio config with approval callback.
    RAIL_ConfigVerification(railHandle,
                            &configVerify,
                            NULL,   // use channelConfigs[0] for verification
                            RAILCb_VerificationApproval);
  } else if (3 == verifyConfigOption) {
    // Config Option 3: Use custom radio config with no approval callback.
    RAIL_ConfigVerification(railHandle,
                            &configVerify,
                            &customRadioConfig[0],
                            NULL);
  } else {
    // Config Option 4: Use custom radio config with approval callback.
    RAIL_ConfigVerification(railHandle,
                            &configVerify,
                            &customRadioConfig[0],
```

```
                    RAILCb_VerificationApproval);}elseif(3== verifyConfigOption){// Config Option 3: Use custom radio config with no approval
callback.RAIL_ConfigVerification(railHandle,&configVerify,&customRadioConfig[0],
                    NULL);}else{// Config Option 4: Use custom radio config with approval
callback.RAIL_ConfigVerification(railHandle,&configVerify,&customRadioConfig[0],
                    RAILCb_VerificationApproval);}// Run the radio state verification in one of 2 ways:if(1== verifyRunOption){// Run Option 1: Run
verification once to completion.
  result =RAIL_Verify(railHandle,&configVerify,
                  RAIL_VERIFY_DURATION_MAX,true);// run from the beginning of verification}else{// Run Option 2: Run verification for a specific time
before returning.
  result =RAIL_Verify(railHandle,&configVerify,100,// run for 100 microseconds before returningtrue);// run from the beginning of
verificationwhile(RAIL_STATUS_SUSPENDED == result){responsePrint("verifyRadio","verification suspended - run again to continue");
    result =RAIL_Verify(railHandle,&configVerify,100,// run for 100 us before returningfalse);// run from where verification left off}}

 switch (result){
  case RAIL_STATUS_NO_ERROR:{responsePrint("verifyRadio","success, done");break;}
  case RAIL_STATUS_INVALID_PARAMETER:{responsePrint("verifyRadio","invalid input parameter");break;}
  case RAIL_STATUS_INVALID_STATE:
  default:{responsePrint("verifyRadio","data corruption");}}...}
```

RFSENSE

The RAIL library for EFR32 supports RF Sensing for EFR32 Series 1 devices and EFR32xG22. On EFR32xG22, enabling RFSENSE will enable the ULFRCO clock, if not already enabled.

Energy Detection Mode

This mode is supported by all EFR32 Series 1 devices and EFR32xG22, where the chip senses the presence of RF Energy and triggers an event if that energy is continuously present for certain duration of time. On Series 1 devices, rounding is done to the nearest choice for requested period (in microseconds). RAIL_StartRfSense() will return the actual senseTime used, which may be different than the requested time due to hardware limitations. Also, on Series 1, RFSENSE functionality is only guaranteed from 0 to 85 degrees Celsius and should be disabled outside of this temperature range. On EFR32xG22 device, the RFSENSE Energy Duration ranges from 1 ms - 128 ms as a power of 2. Based on the user's RFSENSE duration (senseTime) input, the duration is checked against the actual period and the next higher period. If the senseTime is greater than or equal to ((actualPeriod + nextHigherPeriod) / 2), the next higher duration is chosen. Both Series 1 and EFR32xG22 support setting the energy detection threshold for high sensitivity (low noise environment) and low sensitivity (high noise environment) by selecting appropriate RF Band via RAIL_RfSenseBand_t.

Selective Mode

Only supported on EFR32xG22, the Selective mode uses 0.5 kbps Manchester encoded OOK detection which allows the chip to wake up upon detecting particular preamble and sync word pattern sent using OOK. The transmitting node must be configured to use the OOK based RFSENSE PHY for waking up EFR32xG22, via RAIL_ConfigRfSenseSelectiveOokWakeupPhy, followed by setting the transmit FIFO to include the Preamble Byte and the sync word (1 byte - 4 bytes), using RAIL_SetRfSenseSelectiveOokWakeupPayload. Note that the RAIL_ConfigRfSenseSelectiveOokWakeupPhy() API is only supported on 2.4GHz chips and does not work on EFR32xG21 devices.

RAIL Multiprotocol

RAIL Multiprotocol

Overview

As of version 2.1, the RAIL library supports dynamic multiprotocol. In dynamic multiprotocol, an application can allocate multiple RAIL instances and configure them independently. Some radio operations are then arbitrated by a radio scheduler, which is internal to RAIL, to allow the different instances to coexist. The scheduler attempts to allow as many of these operations to run as possible by moving them around within the bounds allowed by the protocol.

Radio Scheduler

The radio scheduler is critical for multiprotocol operation. Its job is to arbitrate all radio operations, switch radio configurations, and ensure that radio operations run at the time requested. Not every RAIL function is a radio operation considered by the scheduler. Simple state update functions, such as RAIL_SetTxPower(), will directly change the hardware if active or update the state cache if inactive to have that change take affect the next time this protocol is loaded. The following RAIL functions are handled by the scheduler:

- RAIL_StartRx()
- RAIL_ScheduleRx()
- RAIL_StartTx()
- RAIL_StartScheduledTx()
- RAIL_StartCcaCsmaTx()
- RAIL_StartCcaLbtTx()
- RAIL_StartAverageRssi()
- RAIL_StartTxStream()

To arbitrate the radio operations, the scheduler uses startTime, priority, slipTime, and transactionTime. startTime comes implicitly from the RAIL API that started the transaction. For example, calling RAIL_StartTx() will use a start time of right now while calling RAIL_StartScheduledTx() takes an explicit start time as a parameter. The other three parameters are passed to all RAIL functions that require the scheduler in the RAIL_SchedulerInfo_t structure. The scheduler attempts to run each task at its desired startTime, but may also choose to run it up until startTime + slipTime. If it cannot run the task within that window, it will trigger a RAIL_Config_t::eventsCallback with the RAIL_EVENT_SCHEDULER_STATUS event set once that window has passed.

The scheduler is preemptive and will always ensure that a higher-priority task runs over a lower-priority task, but it does not guarantee ordering of these tasks. For example, it could choose to run a short lower-priority task with a small slipTime before a higher-priority task with a long slipTime to reduce dropped operations. Once an operation is started by the radio scheduler, it will either run to completion and trigger a normal RAIL event callback to indicate it is done or it will be aborted by a higher-priority task and ended with a RAIL_EVENT_SCHEDULER_STATUS event. Ultimately, the application is responsible for terminating most operations with a call to RAIL_YieldRadio(). This call cleans up internal state and allows the scheduler to switch to a lower priority operation.

Note

- For the RAIL_SchedulerInfo_t::priority parameter, 0 is the highest priority while 255 is the lowest.

Scheduler Operations

The APIs mentioned above can be classified as finite operations, infinite operations, and debug operations.

Finite Operations

Finite operations include the following APIs:

- RAIL_ScheduleRx()
- RAIL_StartTx()
- RAIL_StartScheduledTx()
- RAIL_StartCcaCsmaTx()
- RAIL_StartCcaLbtTx()
- RAIL_StartAverageRssi()

Finite operations make up the majority of the radio scheduler calls. A finite operation has a defined start time and either a defined or estimated end time. Each RAIL handle may only have one of these types of operations queued up at any time. The upper level application code is responsible for waiting until the previous operation has completed before starting a new one like in the single protocol RAIL library. If a new task is issued while a previous task is in progress, or if a higher-priority task interrupts an operation before the user has yielded, it will trigger the RAIL_EVENT_SCHEDULER_STATUS event with an appropriate RAIL_SchedulerStatus_t set and this operation will not be scheduled again.

Once a finite operation has begun, the scheduler will use the same priority until a new finite event is provided or a yield occurs. The scheduler will ensure that unnecessary switches do not occur if this next event is in the near future at the same or a higher priority. For more information see Yielding the Radio.

Infinite Operations

Infinite operations include the following APIs:

- RAIL_StartRx()

An infinite or background operation is treated by the scheduler as a radio state change. It still has a priority associated with it and will be started whenever it can be, but it will never fail. Another difference is that if this event is interrupted by a higher-priority task, it will be restarted whenever that task finishes where a finite task would be aborted.

Each RAIL handle may have only one infinite task but it may have both an infinite and finite or debug task at the same time. The one interesting aspect of this is that, if a finite task starts for this RAIL handle, the state will be updated to reflect what the infinite task has requested for that finite task's time period even if the infinite task is lower priority than another task in the system. For example, if you have one handle with a RAIL_StartRx() of priority 5 and another handle with a RAIL_StartRx() priority of 6 and a RAIL_StartTx() priority of 3, the scheduler will choose the second handle's finite transmit task. While running this transmit task though, the receiver will be set to on so that, as soon as this transmit completes, the radio will transition from transmit into receive. It will remain in receive until the finite operation yields at which time the first handle's receive will take precedence.

This implementation is intended to get the radio to stay in receive for a very long time on one of the RAIL handles. It is generally only advisable to do this on one radio configuration at a time and for this to be the lowest-priority task. For well defined receive events, such as looking for an ACK scheduled receive, windows or state transitions will generally work better.

Note

- A known bug exists with turning on an infinite task during a finite task. For now, you must always configure the infinite task first to ensure it takes effect.
- If using state transitions that can lead from receive to idle, the user must tell the scheduler that this happened with a call to RAIL_Idle(). This restriction will hopefully be removed in future versions.

Debug Operations

Debug operations include the following APIs:

- RAIL_StartTxStream()

Debug operations are somewhat special because they don't allow the user to pass a RAIL_SchedulerInfo_t structure to them. Instead, they implicitly provide a start time of now and use the highest possible priority. This means that they will have the highest priority and interrupt most other protocol operations to switch into the test mode. Nothing can preempt them

once started because the scheduler does not allow tasks of equal priority to preempt. These operations can be stooped by calling RAIL_StopTxStream(), RAIL_YieldRadio(), or RAIL_Idle().

The modal nature of these debug operations is intentional to prevent multiple tone or stream operations from happening concurrently. It is expected that the user will manage these test modes from a higher level to prevent the different protocols from using tone or stream at the same time.

Yielding the Radio

Yielding is used by finite radio scheduler operations. Once an operation is started, that operation will run until explicitly yielded by the upper layer or interrupted by a higher-priority task. A yield is forced by a call to RAIL_YieldRadio() or RAIL_Idle(). It is important that the application yield the radio as soon as it's done to allow lower-priority tasks to run.

Yielding is not handled by RAIL since terminating an operation can be application-specific. While RAIL may know that an individual transmit or receive operation is completed, there are sometimes other reasons to prevent a switch or to run a follow on operation. For example, if you are using ACKing you may want to wait for the RAIL_EVENT_TXACK_PACKET_SENT after a receive packet instead of yielding right after the receive. This also allows you to chain together many protocol-specific operations without the scheduler constantly context switching. If you do schedule a follow on operation that is far in the future, the scheduler will also be smart enough to treat this as a yield and slot in other lower-priority operations.

This does mean that you must carefully look through the Events in RAIL and subscribe to any that can impact your state machine. Specifically, things like transmit or receive success conditions, transmit and receive error conditions, and scheduler events (RAIL_EVENT_SCHEDULER_STATUS, RAIL_EVENT_CONFIG_UNSCHEDULED, and RAIL_EVENT_CONFIG_SCHEDULED) are important to keep track of. These would likely be needed in any upper layer to properly update the internal state machine as well so they're likely already in the implementation.

Building a Multiprotocol Application

The RAIL API is the same whether the targeted application is using the multiprotocol version or not. This was done intentionally to make switching between the two versions simple. From a build perspective, choose the desired RAIL library binary file to link into the application. For multiprotocol applications, the file name is librail_multiprotocol_CHIP_COMPILER_release.a while for other applications the file name is librail_CHIP_COMPILER_release.a.

Optionally, non multiprotocol applications can save memory by not allocating the RAIL_Config_t::scheduler state structure and passing NULL for all RAIL_SchedulerInfo_t pointers in APIs.

Starting in RAIL 2.12, the RAIL multiprotocol library internally provides two statically allocated RAM state buffers supporting two protocols, and both RAILSched_Config_t and RAIL_StateBuffer_t have been reduced to minimally-sized dummy structures solely for backwards compatibility. RAIL_Config_t structures no longer need to be in RAM or globals but can be allocated on the stack just for RAIL_Init(). In addition, RAIL_Config_t::protocol no longer needs to be provided for BLE (can always be NULL). If your multiprotocol application needs more than two protocols, additional state buffers for them must be provided by calling RAIL_AddStateBuffer3() or RAIL_AddStateBuffer4() prior to calling RAIL_Init() for these protocols. If more than four protocols are needed, contact Silicon Labs for advice.

Example

Below is a simple example showing how to initialize two RAIL protocols and have one stay in infinite receive while the other one periodically transmits a packet at a higher priority.

```
#include <stdint.h>
#include <stdbool.h>
#include "rail.h"
#include "rail_config.h" // Generated by radio calculator

static RAIL_Handle_t gRailHandle1 = NULL;
static RAIL_Handle_t gRailHandle2 = NULL;

// Boolean to track when a packet has been sent.
static bool packetSendComplete = true;

// Creates transmit FIFOs for each PHY.
#define TX_FIFO_SIZE 128
static uint8_t txFifo1[TX_FIFO_SIZE];
static uint8_t txFifo2[TX_FIFO_SIZE];

static void radioConfigChangedHandler(RAIL_Handle_t railHandle,
                                      const RAIL_ChannelConfigEntry_t *entry)
{
  bool isSubgig = (entry->baseFrequency < 1000000000UL);

  // ... handle radio configuration change, e.g., select the desired PA possibly
  // using isSubgig to handle multiple configurations.
  RAIL_ConfigTxPower(railHandle, &railTxPowerConfig);

  // Reapply the Tx power after changing the PA above.
  RAIL_SetTxPowerDbm(railHandle, txPower);
}

static void radioEventHandler(RAIL_Handle_t railHandle,
                              RAIL_Events_t events)
{
  // Note that two different callbacks above could be used,
  // but this example only uses one which is split based on the handle.
  if (railHandle == gRailHandle1) {
    // Handle events for protocol 1.
    // NOTE: Since in infinite receive, the radio does not have to be yielded
    // here as below. If this were changed, the radio would have to be yielded here.
  } else if(railHandle == gRailHandle2) {
    // Handle any packet completion event (success or failure) and set us up
    // to send another packet.
    if (events & (RAIL_EVENT_TX_PACKET_SENT
                  | RAIL_EVENT_TX_ABORTED
                  | RAIL_EVENT_TX_UNDERFLOW
                  | RAIL_EVENT_SCHEDULER_STATUS)) {
      packetSendComplete = true;
      RAIL_YieldRadio(railHandle);
    }
  }
}

// Initializes the two PHY configurations for the radio.
void radioInitialize(void)
{
  // Creates each RAIL handle with their own configuration structures.
  RAIL_Config_t railCfg1 = {
    .eventsCallback = &radioEventHandler,
  };
  RAIL_Config_t railCfg2 = {
    .eventsCallback = &radioEventHandler,
  };
  gRailHandle1 = RAIL_Init(&railCfg1, NULL);
  gRailHandle2 = RAIL_Init(&railCfg2, NULL);

  // Sets up transmit FIFOs.
  RAIL_SetTxFifo(gRailHandle1, txFifo1, 0, TX_FIFO_SIZE);
  RAIL_SetTxFifo(gRailHandle2, txFifo2, 0, TX_FIFO_SIZE);
```

```
// Configures radio according to the generated radio settings.RAIL_ConfigChannels(gRailHandle1,
channelConfigs[0],&radioConfigChangedHandler);RAIL_ConfigChannels(gRailHandle2, channelConfigs[1],&radioConfigChangedHandler);//
Configures the most useful callbacks plus catch a few errors.RAIL_ConfigEvents(gRailHandle1,
                RAIL_EVENTS_ALL,
                RAIL_EVENT_TX_PACKET_SENT
                | RAIL_EVENT_TX_ABORTED
                | RAIL_EVENT_TX_UNDERFLOW
                | RAIL_EVENT_SCHEDULER_STATUS
                | RAIL_EVENT_RX_PACKET_RECEIVED
                | RAIL_EVENT_RX_FRAME_ERROR // invalid CRC| RAIL_EVENT_RX_ADDRESS_FILTERED);RAIL_ConfigEvents(gRailHandle2,
                RAIL_EVENTS_ALL,
                RAIL_EVENT_TX_PACKET_SENT
                | RAIL_EVENT_TX_ABORTED
                | RAIL_EVENT_TX_UNDERFLOW
                | RAIL_EVENT_SCHEDULER_STATUS
                | RAIL_EVENT_RX_PACKET_RECEIVED
                | RAIL_EVENT_RX_FRAME_ERROR // invalid CRC| RAIL_EVENT_RX_ADDRESS_FILTERED);}

int main(void){
  RAIL_SchedulerInfo_t schedulerInfo;
  uint32_t sendTime;// Initializes the radio and creates the two RAIL handles.radioInitialize();// Only set priority because transactionTime is
meaningless for infinite// operations and slipTime has a reasonable default for relative operations.
  schedulerInfo =(RAIL_SchedulerInfo_t){.priority =200};RAIL_StartRx(gRailHandle1,0,&schedulerInfo);// Starts the first send 2 seconds after getting
set up.
  sendTime =RAIL_GetTime()+2000000;// Issues a transmit periodically on the second RAIL handle.while(true){if(packetSendComplete){
     RAIL_Status_t res;
     RAIL_ScheduleTxConfig_t scheduledTxConfig ={.when = sendTime,.mode = RAIL_TIME_ABSOLUTE };
     uint8_t packetData[]={0,1,2,3,4,5,6,7,7,8,10};// This assumes the Tx time is around 10 ms but should be tweaked based on// the specific PHY
configuration.
     schedulerInfo =(RAIL_SchedulerInfo_t){.priority =100,.slipTime =50000,.transactionTime =10000};// Loads the transmit buffer with something to
send.RAIL_WriteTxFifo(gRailHandle2, packetData,sizeof(packetData),true);// Transmits this packet at the specified time or up to 50 ms late.
     res =RAIL_StartScheduledTx(gRailHandle2,0,
                        RAIL_TX_OPTIONS_DEFAULT,&scheduledTxConfig,&schedulerInfo);if(res == RAIL_STATUS_NO_ERROR){
       packetSendComplete =false;
       sendTime +=2000000;// Add 2 seconds to the previous time.}else{// In the current configuration this should never
happen.assert(false);}}}return0;}
```

Understanding the Protocol Switch Time

Current EFR32 chips that run dynamic multiprotocol code only have a single radio in their hardware. As a result, when DMP switches protocols, it must fully reconfigure the radio accordingly. The time this reconfiguration takes, as well as the time it takes the scheduler to decide which radio task to run, and timings passed into RAIL via RAIL_SetStateTiming (idleToRx, idleToTx), make up the Protocol Switch Time. This is essentially the amount of time in advance the radio must be made aware of new tasks in order to execute them on time. If this time is not respected at higher layers, users may get a RAIL_SchedulerStatus_t that indicates a failed scheduled event. By default, RAIL uses TRANSITION_TIME_US as the time for all protocol switches. The following sections explain how to characterize this time for a specific app and reconfigure it.

Measuring the Protocol Switch Time

For users using only Silicon Labs provided standards-based stacks, the provided TRANSITION_TIME_US will be accurate. However, users implementing proprietary stacks or complex callbacks around scheduler events (RAIL_EVENT_CONFIG_UNSCHEDULED, RAIL_EVENT_CONFIG_SCHEDULED, and RAIL_EVENT_SCHEDULER_STATUS) may need to recharacterize this time if they find that the current switch time is causing scheduling failures in their application. Users can empirically determine this time for their system with the following algorithm: On the EFR, set a small (e.g. 0) transition time using RAIL_SetTransitionTime (this API must be called before RAIL_Init is called on any protocols). On protocol 1, enter infinite RX with RAIL_StartRx. On protocol 2, request an absolutely scheduled TX for some time in future (e.g. `RAIL_GetTime() + 1000000`). For small transition time values, the application should get RAIL_EVENT_SCHEDULER_STATUS on protocol 2, with a scheduler status of RAIL_SCHEDULER_STATUS_SCHEDULED_TX_FAIL. Repeat this test (resetting the device each time between runs to ensure RAIL_SetTransitionTime) is called before RAIL_Init) with increasing transition times. Eventually, for some large enough value, the application should begin getting RAIL_EVENT_TX_PACKET_SENT events, indicating that the supplied transition

time was sufficient, and thus the radio was able to transition in time to complete the scheduled transmit at the correct time. Ideally this test should be run in both directions (i.e. repeat the test, swapping protocols 1 and 2).

Sample code for the above algorithm is provided below:

```
#include <stdint.h>
#include <stdbool.h>
#include "rail.h"
#include "rail_config.h" // Generated by radio calculator

static RAIL_Handle_t gRailHandle1 = NULL;
static RAIL_Handle_t gRailHandle2 = NULL;

// Boolean to track when a packet has been sent.
static bool packetSendComplete = true;

// Creates transmit FIFOs for each PHY.
#define TX_FIFO_SIZE 128
static uint8_t txFifo1[TX_FIFO_SIZE];
static uint8_t txFifo2[TX_FIFO_SIZE];

uint32_t transitionTime = 0;

void success(void) {
  // Indicate test success to user
}

void fail(void) {
  // Indicate test failure to user
}

static void radioConfigChangedHandler(RAIL_Handle_t railHandle,
                                      const RAIL_ChannelConfigEntry_t *entry)
{
  bool isSubgig = (entry->baseFrequency < 1000000000UL);

  // ... handle radio configuration change, e.g., select the desired PA possibly
  // using isSubgig to handle multiple configurations.
  RAIL_ConfigTxPower(railHandle, &railTxPowerConfig);

  // Reapply the Tx power after changing the PA above.
  RAIL_SetTxPowerDbm(railHandle, txPower);
}

static void radioEventHandler(RAIL_Handle_t railHandle,
                              RAIL_Events_t events)
{
  // Note that two different callbacks above could be used,
  // but this example only uses one which is split based on the handle.
  if (railHandle == gRailHandle1) {
    // We don't care about events on protocol 1 for this test
    return;
  } else if(railHandle == gRailHandle2) {
    // Handle any packet completion event (success or failure) and set us up
    // to send another packet.
    if (events & RAIL_EVENT_TX_PACKET_SENT) {
      success();
    }
    if ((events & RAIL_EVENT_SCHEDULER_STATUS)
        && (RAIL_GetSchedulerStatus(railHandle) & RAIL_SCHEDULER_STATUS_SCHEDULED_TX_FAIL)) {
      fail();
    }
  }
}

// Initializes the two PHY configurations for the radio.
void radioInitialize(void)
{
  // transitionTime should be swept via testing instrumentation between
  // runs
  RAIL_SetTransitionTime(transitionTime)
```

```
// Creates each RAIL handle with their own configuration structures.
  RAIL_Config_t railCfg1 ={.eventsCallback =&radioEventHandler,};
  RAIL_Config_t railCfg2 ={.eventsCallback =&radioEventHandler,};
  gRailHandle1 =RAIL_Init(&railCfg1, NULL);
  gRailHandle2 =RAIL_Init(&railCfg2, NULL);// Sets up transmit FIFOs.RAIL_SetTxFifo(gRailHandle1, txFifo1,0,
TX_FIFO_SIZE);RAIL_SetTxFifo(gRailHandle2, txFifo2,0, TX_FIFO_SIZE);// Configures radio according to the generated radio
settings.RAIL_ConfigChannels(gRailHandle1, channelConfigs[0],&radioConfigChangedHandler);RAIL_ConfigChannels(gRailHandle2,
channelConfigs[1],&radioConfigChangedHandler);// Only need a few events on the second handle for this testRAIL_ConfigEvents(gRailHandle2,
                RAIL_EVENTS_ALL,
                RAIL_EVENT_TX_PACKET_SENT
                | RAIL_EVENT_SCHEDULER_STATUS);}

int main(void){
  RAIL_SchedulerInfo_t schedulerInfo;
  uint32_t sendTime;// Initializes the radio and creates the two RAIL handles.radioInitialize();// Get into receive on handle 1
  schedulerInfo =(RAIL_SchedulerInfo_t){.priority =200,.transactionTime =1000,.slipTime =0};RAIL_StartRx(gRailHandle1,0,&schedulerInfo);// Starts
the first send 1 second after getting set up.
  sendTime =RAIL_GetTime()+1000000;// Request a TX for some time in the future
  RAIL_ScheduleTxConfig_t scheduledTxConfig ={.when = sendTime,.mode = RAIL_TIME_ABSOLUTE };

  schedulerInfo.priority =100;
  res =RAIL_StartScheduledTx(gRailHandle2,0,
                        RAIL_TX_OPTIONS_DEFAULT,&scheduledTxConfig,&schedulerInfo);return0;}
```

Minimizing the Protocol Switch Time

Currently, the best way to minimize the switch time is by proper design of RAIL_Config_t::eventsCallback. Specifically, the three scheduler events RAIL_EVENT_CONFIG_UNSCHEDULED, RAIL_EVENT_CONFIG_SCHEDULED, and RAIL_EVENT_SCHEDULER_STATUS, will be passed to the callback independently of all other events. Additionally, they are only raised during protocol switch process, and so their handling is part of the critical path for the protocol switch. Ideally, if they are not needed, RAIL_ConfigEvents should disable these events. Otherwise they should be handled first, after which the event handler should return immediately.

Additionally, short transition times should be specified in RAIL_SetStateTiming. Note that 0 for these values can be unreliable in a DMP context, as that tells the radio to go "as fast as possible" as opposed to a reliable, known value.

Release Notes

Release Notes

RAIL Library 2.16.0 GA

Dec 13, 2023

Release Highlights

- Added support for new Fast channel switching PHYs on the EFR32xG24.
- Added support for IEE802154 2.4 GHz coherent PHYs on the EFR32xG28.
- Added RAILTEST support for the MGM240x modules.
- Added Channel sounding support on the EFR32xG24.
- Added collision detection support on the EFR32xG25 to detect and switch to a new stronger packet that is detected while in receive.
- To support chip-agnostic library builds that use RAIL, formerly chip-specific RAIL data structures, defines, and APIs are now, for the most part, exposed to all chips in rail_types.h and rail.h.
- Bugfixes and minor improvements.

Release Details

- API Changes
- RAIL Changelist
- RAILtest Changelist
- Deprecated List

Modules

RAIL Changelist

RAILtest Changelist

API Changes

RAIL Changelist

RAIL Changelist

2.16.0

- **663884:** Added better information for Front End Module (FEM) Utility component and its usage.
- **689770:** Added support for a new assert, which will be thrown if the loaded PHY is not supported by the software defined modem on the EFR32xG25.
- **748817:** Added new RAIL_GetAutoAckFifo() API and allow NULL for RAIL_WriteAutoAckFifo() or RAIL_IEEE802154_WriteEnhAck() ackData parameter, which gives applications direct access to the AutoAck FIFO to construct Ack packets in pieces.
- **751902:** Updated the default PTI rate to 3200000 bps on the EFR32XG25.
- **858169:** Updated documentation on the usage of RAIL_EVENT_RX_TIMING_DETECT, RAIL_EVENT_RX_TIMING_LOST, RAIL_EVENT_RX_PREAMBLE_DETECT and RAIL_EVENT_RX_PREAMBLE_LOST events and the possible issues that may be encountered when using these with certain demodulators on EFR32xG2x chips.
- **1079816:** Fixed a race condition on the EFR32xG22 and later during RX channel hopping or duty-cycling where frame detection occurring close to when a hop should happen could leave the radio stuck in reception but not receiving anything, with the only remedy being to idle the radio.
- **1088439:** Fixed an issue which would cause the incorrect antenna to be reported for a received packet on the EFR32xG25 when using OFDM and antenna diversity.
- **1104367:** Added support for antenna selection through the applicable RAIL_RxOptions_t and RAIL_TxOptions_t values when using OFDM on the EFR32xG25.
- **1112324:** Added a new Software Modem (SFM) sequencer image selection Utility component to allow selection of modulations supported by EFR32xG25 software modem (SFM). These changes can save considerable flash space by reducing the set of modulations to just those that are needed.
- **1117349:** No longer include rail_chip_specific.h and rail_features.h (anything dependent on em_device.h) when SLI_LIBRARY_BUILD is defined. This will allow the user to build their radio code in a way that depends on RAIL generically but is not chip-specific. When doing this the code cannot depend on things that are inherently chip-specific and are still in those files like RAIL_RF_PATHS, RAIL_NUM_PA, or any of the compile-time RAIL_SUPPORTS_xxx defines. Code will need to call appropriate runtime APIs or split itself between the generic and chip specific portions and build them separately.
- **1117349:** To better support chip-agnostic builds RAIL_TxPowerMode_t is now a superset representing all possible PAs across all chips. Any code depending on the number of or consecutive ordering of chip-supported PAs will likely need to be updated.
- **1117349:** To better support chip-agnostic builds RAIL_CalValues_t and subordinate RAIL_IrCalValues_t have grown to encompass the superset of fields needed on all chips, affecting all chips except EFR32xG25.
- **1117349:** To better support chip-agnostic builds RAIL_TransitionTime_t and therefore RAIL_StateTiming_t have grown to a superset type, affecting EFR32xG1.
- **1117349:** To better support chip-agnostic builds RAIL_FIFO_ALIGNMENT is now universally 32-bit, affecting EFR32xG1x and EFR32xG21, but is still only enforced on chips actually requiring that alignment.
- **1129262:** Added support for Sidewalk PHYs on the EFR32xG23 and EFR32xG28 chips.
- **1129280:** Added an assert RAIL_ASSERT_INVALID_XTAL_FREQUENCY on the EFR32xG1x and EFR32xG2x chips, to fire when a radio config that is loaded is incompatible with a device due to the defined crystal frequency of the config not matching with the device's crystal frequency.
- **1131398:** Added the RAIL_TX_REPEAT_OPTION_START_TO_START option to measure the delay between repeated transmits from the start of TX to start of TX instead of the default from end of TX to start of TX.
- **1153679:** Fixed an issue in Coexistence Utility on the EFR32xG24 where a GRANT signal pulse less than 100us might result in the radio not being properly placed in hold off after GRANT is deasserted.
- **1156980:** Fixed an issue with channel hopping on the EFR32xG22 and later where use of RAIL_RX_CHANNEL_HOPPING_OPTION_RSSI_THRESHOLD can prevent the timed RX channel hopping modes including RX duty-cycling from timing out properly.
- **1157301:** Added support for GCC 12.2.1 and IAR 9.40.1 compilers.

- **1159773:** Added support for Fast Channel Switching PHYs on the EFR32xG24.
- **1161043:** Added support for RAIL_IEEE802154_SUPPORTS_G_MODESWITCH on the EFR32xG28.
- **1163285:** Added support for the IEE802154 2.4 GHz coherent PHYs via RAIL_SUPPORTS_IEEE802154_BAND_2P4 on the EFR32xG28.
- **1171683:** Added a new RAIL_RxOptions_t option to enable collision detection on the EFR32xG25. Once enabled, when a collision with a strong enough packet is detected, the demod will stop the current packet decoding and try to detect the preamble of the incoming packet.
- **1174185:** Updated the documentation of RAIL_RxChannelHoppingMode_t::RAIL_RX_CHANNEL_HOPPING_MODE_MANUAL and its current limitation of not issuing RAIL_EVENT_RX_CHANNEL_HOPPING_COMPLETE.
- **1175032:** Updated the documentation mentioning that while configuring DPLL on all Series 2 chips, Phase Lock mode should typically be used instead of Frequency Lock mode.
- **1175684:** Fixed an issue with the RAIL_IDLE form of RAIL_Idle() and RAIL_STOP_MODE_PENDING form RAIL_StopTx() during the LBT/CSMA phase of a transmit which could previously hang. Pending LBT/CSMA and scheduled transmits are now stopped or idled with RAIL_EVENT_TX_BLOCKED triggered, except for idle mode RAIL_IDLE_FORCE_SHUTDOWN_CLEAR_FLAGS.
- **1183040:** Updated all PHYs built into RAIL on Series 2 platforms using the latest radio calculator to reduce observed high reference spurs.
- **1184982:** Fixed an issue with RAIL_StartAverageRssi() that could cause execution to hang in interrupt context just before raising RAIL_EVENT_RSSI_AVERAGE_DONE. This was primarily an issue on the EFR32xG21 platform. Note that it is possible, though unlikely, that RAIL_GetAverageRssi() might still return RAIL_RSSI_INVALID after the average RSSI period has finished.
- **1184982:** Fixed an issue with RAIL_StartAverageRssi() that caused it to mistakenly idle the radio in the newly-activated protocol if a dynamic protocol switch occurred during the averaging operation of the suspended protocol.
- **1187168:** Updated the default power curves for the 10dBm High Power PA on EFR32xG24.
- **1188083:** Fixed an issue where RAIL_Idle() could hang waiting for the radio to idle when using IEEE 802.15.4 fast RX channel switching.
- **1190187:** Fixed an issue where idling the radio before a Scheduled Receive window ends could cause a subsequent packet that should be silently filtered and rolled back to instead be received with RAIL_RX_PACKET_READY_CRC_ERROR.
- **1196683:** Added support of 802.15.4 IMM-ACK with OFDM and OQPSK modulations (FCS 4 bytes only) on the EFR32xG25.
- **1201506:** Fixed an issue in multiprotocol applications where the wrong sync word would be used if two protocols used the same radio configuration and only one of those protocols set a custom sync word with RAIL_ConfigSyncWords() API.
- **1211224:** Updated the RAIL Library component to include the Built-in PHYs for alternate HFXO frequencies component automatically to better support different HFXO frequencies on supported parts.

2.15.2

- **634068:** Reduced the delay between TX completion and the start of PA ramp down on the EFR32xG24.
- **663884:** Added better information for Front End Module (FEM) Utility component and its usage.
- **689770:** Added support for a new assert, which will be thrown if the loaded PHY is not supported by the software defined modem on EFR32xG25.
- **1131398:** Added the RAIL_TX_REPEAT_OPTION_START_TO_START option to measure the delay between repeated transmits from the start of TX to start of TX instead of the default from end of TX to start of TX.
- **1191666:** Fixed an issue where high reference spurs may appear during radio communication on EFR32xG22 and newer platforms.

2.15.1

- **707731:** Fixed an issue when using RAIL_BLE_SetNextTxRepeat() that would cause an incorrect Protocol Config ID to appear in the packet trace for repeated transmits.
- **748817:** Added new RAIL_GetAutoAckFifo() API and allow NULL for RAIL_WriteAutoAckFifo() or RAIL_IEEE802154_WriteEnhAck() ackData parameter, which gives applications direct access to the AutoAck FIFO to construct Ack packets in pieces.
- **1079816:** Fixed a race condition on EFR32xG22 and later during RX channel hopping or duty-cycling where frame detection occurring close to when a hop should happen could leave the radio stuck in reception but not receiving anything, with the only remedy being to idle the radio.
- **1088439:** Fixed an issue which would cause the incorrect antenna to be reported for a received packet on the EFR32xG25 when using OFDM and antenna diversity.
- **1097499:** Updated the RAIL_PacketTimeStamp_t::packetDurationUs field on the EFR32xG25 to be populated for non-OFDM packets.

- **1104367:** Added support for antenna selection through the applicable RAIL_RxOptions_t and RAIL_TxOptions_t values when using OFDM on the EFR32xG25.
- **1112324:** Added a new Software Modem (SFM) sequencer image selection Utility component to allow selection of modulations supported by EFR32XG25 software modem (SFM). These changes can save considerable flash space by reducing the set of modulations to just those that are needed.
- **1129262:** Added support for Sidewalk PHYs for EFR32xG23 and EFR32xG28 chips.
- **1141539:** Fixed an issue to improve CCA behavior for regional certification test suites for EFR32xG25 chips.
- **1153679:** Fixed an issue in Coexistence Utility where a GRANT signal pulse less than 100us might result in the radio not being properly placed in hold off after GRANT is deasserted.
- **1153921:** Added a new component to switch between Coexistence, Antenna Diversity and FEM Utilities for EFR32xG21 and EFR32xG24 chips supporting 15.4 Fast Channel Switching feature.
- **1156980:** Fixed an issue with channel hopping on EFR32xG22 and later where use of RAIL_RX_CHANNEL_HOPPING_OPTION_RSSI_THRESHOLD can prevent the timed RX channel hopping modes including RX duty-cycling from timing out properly.
- **1167235:** Fixed an issue with RAIL_SupportsFastRx2Rx() where it used to return an incorrect value on supported platforms.

2.15.0

- **757123:** Added support for PHY-specific RSSI offsets on the EFR32xG27 and EFR32xG28 platforms.
- **774988:** Added support for a new RX_DIRECT_SYNCHRONOUS_MODE_DATA RAIL RX data source to capture direct mode data in sync with the configured bit rate for the PHY. This requires a PHY that supports this mode of capture from the Radio Calculator and is only currently supported on the EFR32xG23.
- **824355:** Fixed an issue in IEEE802.15.4 MAC address filtering when receiving small OFDM packets.
- **832743:** Clarified use of RAIL_SetNextTxRepeat() must be prior to initiating a transmit operation via API call and fixed an issue where it did not properly return an error when called while a transmit operation was in progress.
- **1032820:** Updated Packet Trace on the EFR32xG25 and EFR32xG28 when using the WiSUN protocol to have a more informative PHY identifier and to support the whole channel number range.
- **1055824:** Fixed an issue with low-side synth injection (negative IF) on proprietary 2.4GHz PHYs when using EFR32xG22 and newer chips. This fix requires regenerating the PHY with the latest version of the Radio Configurator to work.
- **1058480:** Fixed an RX FIFO corruption on EFR32xG25 that occurred when receiving/sending certain OFDM packets using FIFO mode.
- **1060146:** Fixed the RAIL_PA_CURVES_2P4_LP power curves on the EFR32xG24 to better match characterization data.
- **1061665:** Added new RAIL_EnableCacheSynthCal function to enable the radio sequencer to cache calibration values instead of recalculating them on every RX and TX event. This allows you to lower the minimum transition time for most RAIL_StateTiming_t transitions in typical cases.
- **1068856:** Corrected the sign of the frequency error reported by RAIL_GetRxFreqOffset() when using OFDM on the EFR32xG25 to match how this was handled for other modulations (e.g., Freq_error=current_freq- expected_freq).
- **1077623:** Added new RAIL_ZWAVE_OPTION_PROMISCUOUS_BEAM_MODE to trigger RAIL_EVENT_ZWAVE_BEAM on all beam frames.
- **1077623:** Added RAIL_ZWAVE_GetBeamHomeIdHash() to retrieve the beam frame's HomeIdHash when handling that event and made sure that the HomeIdHash byte is now present on PTI for Z-Wave beam frames even when NodeId does not match.
- **1077623:** Fixed an issue on EFR32ZG23 where multiple beam frames were lumped together on PTI as one large beam chain.
- **1081396:** Adjusted channel power restrictions for the 802.15.4 PHYs on new xGM210 modules.
- **1082274:** Fixed an issue on the EFR32xG22, EFR32xG23, EFR32xG24, and EFR32xG25 chips that could cause the chip to lock up if the application attempts to re-enter EM2 within ~10 us after wake-up and hits a <0.5 us timing window. If hit, this lockup requires a power-on reset to restore normal operation to the chip.
- **1083615:** Fixed an issue for certain ramp time and power level combinations on the EFR32xG21 where the PA ramp would stop one power level short of the desired output level.
- **1086162:** Added a new RAIL_RX_OPTION_FAST_RX2RX which will force the radio sequencer to immediately transition to RXSEARCH to get ready to receive the next packet while still processing the previous one. This will minimize the RX to RX state transition time. This is only supported on chips that have RAIL_SUPPORTS_FAST_RX2RX set to true.
- **1090336:** Fixed an issue in the Protocol Utility component where BLE would be required to select a Zigbee PHY.
- **1090512:** Fixed an issue in the Power Amplifier (PA) Utility component where certain functions would attempt to use the RAIL_TX_POWER_MODE_2P4GIG_HIGHEST macro even though they didn't support it. This would result in undefined behavior previously, but will now correctly error.

- **1090728:** Fixed a possible RAIL_ASSERT_FAILED_UNEXPECTED_STATE_RX_FIFO issue on EFR32xG12 with RAIL_IEEE802154_G_OPTION_GB868 enabled for a FEC-capable PHY which can happen when aborting a packet at frame detection, for instance by idling the radio.
- **1092769:** Fixed an issue when using Dynamic Multiprotocol and BLE Coded PHYs where a transmit could underflow depending on what protocol was active when the PHY and syncword were loaded.
- **1096663:** Fixed a compilation error in Coexistence Utility when the Coexistence WiFi TX GPIO is enabled.
- **1096665:** Fixed a compilation issue in Coexistence Utility when the SL_RAIL_UTIL_COEX_WIFI_TX_PORT is defined.
- **1097152:** Added RAIL_PacketTimeStamp_t::packetDurationUs field which is currently set only on EFR32xG25 for received OFDM packets.
- **1097696:** Added RAIL support for the MGM240L lighting modules.
- **1099794:** Added separate curves when the 20 dBm PA is used at 3.3V and 1.8V for EFR32xG24.
- **1099829:** Added the new RAIL_WMBUS_Config API to allow configuring WMBUS and simultaneous M2O RX of T and C mode packets.
- **1103587:** Added a new API RAIL_SetTxFifoAlt() which provides a new start offset parameter to specify where the data begins in the TX FIFO.
- **1103966:** Fixed an unexpected Rx packet abort on the EFR32xG25 when using the Wi-SUN OFDM option4 MCS0 PHY.
- **1104033:** Fixed an issue in the RAIL_ZWAVE_ReceiveBeam function so that it idles the radio regardless of whether a beam is detected on the EFR32ZG23.
- **1104441:** Fixed an issue with the Coexistence Utility counters for Zigbee that could prevent them from ticking as expected depending on how things are linked.
- **1105134:** Fixed an issue when switching between certain PHYs that could cause the first received packet to be reported as RAIL_RX_PACKET_READY_CRC_ERROR instead of RAIL_RX_PACKET_READY_SUCCESS. This issue could potentially impact EFR32xG22 and newer chips.
- **1105529:** Fixed an issue on EFR32xG22 and later platforms when using a FrameType decoding PHY where a bad frame type packet was mis-reported as RAIL_RX_PACKET_ABORT_ABORTED instead of the proper RAIL_RX_PACKET_ABORT_FORMAT.
- **1107288:** Added support for RAIL_IEEE802154_SupportsRxChannelSwitching on the EFR32xG21. This is also still supported at an alpha quality level on the EFR32xG24.
- **1109574:** Fixed an issue on EFR32xG22 and newer chips where a radio sequencer assert could cause the application to hang in an ISR rather than report the assert via RAILCb_AssertFailed().
- **1117127:** Added RAIL_IEEE802154_SetRxToEnhAckTx() to allow IEEE 802.15.4 stacks to specify a different rxToTx state transition turnaround time for Enhanced ACKs, which generally need more time to construct and secure. Immediate ACKs will continue to use the rxToTx time specified in RAIL_IEEE802154_Config_t::timings.
- **1117595:** Added Tx packet duration information for EFR32xG22 and newer chips.
- **1118063:** Fixed issue with recent RAIL_ZWAVE_OPTION_PROMISCUOUS_BEAM_MODE on EFR32xG13 and EFR32xG14 where the NodeId of the promiscuous beam was not properly recorded for RAIL_ZWAVE_GetBeamNodeId(), causing it to report 0xFF.
- **1121630:** Increased EFR32ZG13 and EFR32ZG14 Z-Wave long-range beam detect time to improve FLiRS performance.
- **1126343:** Fixed an issue on EFR32xG24 when using the IEEE 802.15.4 PHY where the radio could become stuck when doing an LBT transmit if a frame is received during the CCA check window.
- **1134223:** Fixed an issue when using Coexistence Utility where the request line is left asserted after TX is aborted following a sync detect.
- **1135418:** Fixed incorrect RAIL_RxPacketInfo_t::filterMask on received 802.15.4 Beacon frames, which now reflects which PanId and address the Beacon's Source PanId and Source Address match, if any. Note that RAIL generally accepts all Beacons so the filterMask can be 0x00.
- **1138522:** Fixed an issue on the EFR32xG25 for SUN FSK PHYs where receiving a packet after calling RAIL_IEEE802154_Init() but before configuring RAIL_IEEE802154_ConfigGOptions could break reception.
- **1140569:** Fixed a rare timing issue on EFR32xG24 where an ACK timeout might cause the next packet to be received as RAIL_RX_PACKET_READY_CRC_ERROR instead of RAIL_RX_PACKET_READY_SUCCESS.
- **1150779:** Fixed the 15.4 channel configurations on the MGM240PA32 and MGM240PB32 modules to use the correct frequency for channel 26.

2.14.3

- **1058480:** Fixed an RX FIFO corruption on EFR32xG25 that occurred when receiving/sending certain OFDM packets using FIFO mode.
- **1109993:** Fix an issue in the Coexistence Utility component so that it simultaneously asserts request and priority if request and priority share the same GPIO port and polarity.

- **1118063:** Fixed issue with recent RAIL_ZWAVE_OPTION_PROMISCUOUS_BEAM_MODE on EFR32xG13 and xG14 where the NodeId of the promiscous beam was not properly recorded for RAIL_ZWAVE_GetBeamNodeId(), causing it to report 0xFF.
- **1126343:** Fixed an issue on EFR32xG24 when using the IEEE 802.15.4 PHY where the radio could become stuck when doing an LBT transmit if a frame is received during the CCA check window.

2.14.2

- **747041:** Fixed an issue on the EFR32xG23 and EFR32xG25 that could cause certain radio actions to delay for extended periods of time when the main core enters EM2 while the radio is still running.
- **1077623:** Added new RAIL_ZWAVE_OPTION_PROMISCUOUS_BEAM_MODE to trigger RAIL_EVENT_ZWAVE_BEAM on all beam frames.
- **1077623:** Added RAIL_ZWAVE_GetBeamHomeIdHash() to retrieve the beam frame's HomeIdHash when handling that event and made sure that the HomeIdHash byte is now present on PTI for Z-Wave beam frames even when NodeId does not match.
- **1077623:** Fixed an issue on EFR32ZG23 where multiple beam frames were lumped together on PTI as one large beam chain.
- **1090512:** Fixed an issue in the Power Amplifier (PA) Utility component where certain functions would attempt to use the RAIL_TX_POWER_MODE_2P4GIG_HIGHEST macro even though they didn't support it. This would result in undefined behavior previously, but will now correctly error.
- **1090728:** Fixed a possible RAIL_ASSERT_FAILED_UNEXPECTED_STATE_RX_FIFO issue on EFR32xG12 with RAIL_IEEE802154_G_OPTION_GB868 enabled for a FEC- capable PHY which can happen when aborting a packet at frame detection, for instance by idling the radio.
- **1092769:** Fixed an issue when using Dynamic Multiprotocol and BLE Coded PHYs where a transmit could underflow depending on what protocol was active when the PHY and syncword were loaded.
- **1097152:** Added RAIL_PacketTimeStamp_t::packetDurationUs field which is currently set only on EFR32xG25 for received OFDM packets.
- **1103966:** Fixed an unexpected Rx packet abort on the EFR32xG25 when using the Wi-SUN OFDM option4 MCS0 PHY.
- **1105134:** Fixed an issue when switching certain PHYs that could cause the first received packet to be reported as RAIL_RX_PACKET_READY_CRC_ERROR instead of RAIL_RX_PACKET_READY_SUCCESS. This issue could potentially impact EFR32xG22 and newer chips.
- **1109574:** Fixed an issue on EFR32xG22 and newer chips where a radio sequencer assert could cause the application to hang in an ISR rather than report the assert via RAILCb_AssertFailed().

2.14.1

- **1068856:** Corrected the sign of the frequency error reported by RAIL_GetRxFreqOffset() when using OFDM on the EFR32xG25 to match how this was handled for other modulations (e.g. Freq_error=current_freq- expected_freq).
- **1074384:** The RAIL_SetTune() and RAIL_GetTune() functions now use the CMU_HFXOCTuneSet() and CMU_HFXOCTuneGet() functions respectively on EFR32xG2x and newer devices.
- **1077611:** Fixed an issue on the EFR32xG25 that would cause a 40us porch before an OFDM TX.
- **1082274:** Fixed an issue on the EFR32xG22, EFR32xG23, EFR32xG24, and EFR32xG25 chips that could cause the chip to lock up if the application attempts to re-enter EM2 within ~10 us after wake-up and hits a <0.5 us timing window. If hit, this lockup requires a power on reset to restore normal operation to the chip.

2.14.0

- **670716:** The RAIL_ConfigRfSenseSelectiveOokWakeupPhy() will now return an error when run on the EFR32xG21 platform because this device cannot support the wakeup PHY.
- **751910:** Added HFXO temperature compensation in RAIL on platforms that support RAIL_SUPPORTS_HFXO_COMPENSATION. This feature can be configured with the new RAIL_ConfigHFXOCompensation() API. The user will also need to be sure to handle the new RAIL_EVENT_THERMISTOR_DONE event to trigger a call to RAIL_CalibrateHFXO to perform the compensation.
- **765112:** Added options in Protocol Utility to control whether Z-Wave, 802.15.4 2.4 GHz and Sub-GHz, and BLE are enabled so that the user can save space in their application by disabling unused protocols.
- **843708:** Moved function declarations from rail_features.h to rail.h to avoid a convoluted include dependency order.
- **844325:** Fixed RAIL_SetTxFifo() to properly return 0 (error) rather than 4096 for an undersized FIFO.
- **845608:** Fixed an issue with the RAIL_ConfigSyncWords API when using certain underlying demodulator hardware on EFR32xG2x parts.
- **846240:** Added a new API RAIL_ZWAVE_PerformIrcal to help perform an IR calibration across all the different PHYs used by a Z-Wave device.
- **851150:** Fixed an issue on EFR32xG2 series devices where the radio would trigger RAIL_ASSERT_SEQUENCER_FAULT when PTI is used and GPIO configuration is locked. GPIO configuration can only be locked when PTI is disabled. See

RAIL_EnablePti() for further information.
- **853682:** Added 40MHz crystal support on EFR32xG24 devices to the Built-in PHYs for alternate HFXO frequencies component.
- **855048:** Updated the pa_customer_curve_fits.py helper script to accept floating point value for the maximum power argument, similar to the increment argument.
- **857210:** Added support in Coexistence Utility for configuring priority options when directional priority is enabled but no static priority GPIO is defined.
- **857267:** Fixed an issue when using the Coexistence Utility component with TX abort, the signal identifier feature and DMP.
- **858567:** Added support for IEEE 802.15.4 fast RX channel switching with the new RAIL_IEEE802154_ConfigRxChannelSwitching API on supported platforms (RAIL_IEEE802154_SupportsRxChannelSwitching). This feature allows us to simultaneously detect packets on any two 2.4GHz 802.15.4 channels with a slight reduction in overall sensitivity of the PHY.
- **988492:** Added support for the BGM220SC23HNA2 module.
- **1015152:** Fixed an issue on EFR32xG2x devices where RAIL_EVENT_RX_FIFO_ALMOST_FULL or RAIL_EVENT_TX_FIFO_ALMOST_EMPTY could trigger improperly when the event is enabled or the FIFO is reset.
- **1017609:** Fixed an issue where PTI appended information could be corrupted when RAIL_RX_OPTION_TRACK_ABORTED_FRAMES is in effect when RAIL_IDLE_FORCE_SHUTDOWN or RAIL_IDLE_FORCE_SHUTDOWN_CLEAR_FLAGS is used. Also clarified that RAIL_RX_OPTION_TRACK_ABORTED_FRAMES is not useful with coded PHYs.
- **1019590:** Fixed an issue when using the Coexistence Utility component with BLE where the sl_bt_system_get_counters() function would always return 0 for GRANT denied counts.
- **1019794:** Eliminated compiler warning in Initialization Utility component when few of its features are enabled.
- **1022182:** Added a new Thermal Protection feature, on platforms that support RAIL_SUPPORTS_THERMAL_PROTECTION, to track temperature and prevent transmits when the chip is too hot.
- **1022692:** Added new table based OFDM and FSK PAs for EFR32xG25 based devices. The output power of these can be modified through a new customer provided look-up table. Ask support or look for an updated app note on how to configure the values in this table for your board.
- **1023016:** Fixed an issue on EFR32xG22 and newer chips where waits in between radio activity would consume slightly more power than necessary after the first 13ms. This was especially noticeable when using RAIL_ConfigRxDutyCycle with large off time values.
- **1024294:** Removed support for the deprecated BGX220P22HNA and BGX220S22HNA modules.
- **1024457:** Added support for the MGM240SA22VNA, BGM240SA22VNA, and BGM241SD22VNA modules and updated the configurations for the BGM240SB22VNA, MGM240SB22VNA, and the MGM240SD22VNA.
- **1029740:** Fixed issue where RAIL_GetRssi()/RAIL_GetRssiAlt() could return a "stale" RSSI value (the value was from previous RX state instead of the current one) if called quickly upon entering receive.
- **1040814:** Added support to the Coexistence Utility component for configuring the coexistence request priority on sync detect when using BLE.
- **1041734:** Broke up some EFR32xG12 802.15.4 dynamic FEC code to save code size for Zigbee and BLE which never need this functionality.
- **1056207:** Fixed an issue with IQ sampling when using the Angle of Arrival/Departure (AoX) Utility component with only 0 or 1 antennas selected.
- **1057830:** Remove Coexistence Utility dependency from the RAIL Utility, Coulomb Counter component.
- **1062712:** Fixed an issue where the Coexistence Utility component wouldn't always update request states correctly which could lead to missed events triggered by new requests.
- **1062940:** Prevent the Coexistence Utility component from aborting BLE transmits when SL_RAIL_UTIL_COEX_BLE_TX_ABORT is disabled.
- **1063152:** Fixed an issue where radio reception would not be fully cleaned up when a receive error occurs with receive state transitions set to idle on error but transmit on success, a configuration mostly associated with BLE. On the EFR32xG24 this could cause a SYNTH calibration to not be properly restored and eventually cause the radio to stop working.
- **1063502:** The RAIL_PrepareChannel() function has been made dynamic multi-protocol safe and will no longer return an error if called when your protocol is inactive.

2.13.3

- **1041997:** Fixed the librail_config libraries for the following xGM240 modules: BGM240PA22VNA, BGM240PA32VNA, BGM240PA32VNN, BGM240PB22VNA, BGM240PB32VNA, BGM240PB32VNN, MGM240PA22VNA, MGM240PA32VNA, MGM240PA32VNN. Without this update these modules will assert when trying to load the supported BLE and IEEE 802.15.4 PHYs.

2.13.2

- **813171:** Added early support for IEEE802.15.4G dynamic forward error correction PHYs on the EFR32xG12 platform. Using this at this time will require help from support.
- **844377:** Fixed a BLE 2Mbps AoX issue on EFR32xG24 when using a 38.4MHz crystal.
- **1026914:** Improved PA configurations for the xGM240 modules based on additional test data.
- **1029710:** Fixed an issue with RAIL's PA auto mode that would cause us to select an unsupported RAIL_TxPowerMode_t on chip OPNs that are missing the higher power PAs.

2.13.1

- **819544:** Improved reception on EFR32xG23 and EFR32xG24 when using a PHY with fast detect enable (preamble sense mode).
- **843708:** Moved function declarations from rail_features.h to rail.h to avoid a convoluted include dependency order.
- **844325:** Fixed RAIL_SetTxFifo() to properly return 0 (error) rather than 4096 for an undersized FIFO.
- **844936:** Fixed an issue where using RAIL_SetNextTxRepeat() could cause a brownout reset on EFR32xG23.
- **853714:** Fixed an issue with xGM240 modules causing them to assert during initialization.
- **857210:** Added support in Coexistence Utility for configuring priority options when directional priority is enabled but no static priority GPIO is defined.
- **988518:** Fixed an issue where the radio sequencer would leave portions of the chip enabled after AoX CTE packet reception, preventing the device from going into EM2 sleep and potentially causing missed packet receive events.

2.13.0

- **376658:** Fixed an issue with the BLE coded PHY on EFR32xG21 where a packet received with a corrupt coding indicator would result in an invalid start of packet timestamp.
- **645362:** The Packet Trace Interface (PTI) Utility component will now validate that the correct set of pins are in use for the desired PTI mode.
- **653200:** RAIL will now error if attempting to start a CSMA or LBT transmit while a scheduled RX is still in progress or vice versa.
- **714958:** The RAIL channel of a received packet is now available in the packet's RAIL_RxPacketDetails_t::channel field. This can be of value when scanning or hopping across multiple channels while letting packets accumulate in the receive FIFO for later processing.
- **739371:** Added the RAIL_ConfigPaAutoEntry API to allow for easier configuration of PA auto mode operation in RAIL.
- **746775:** Added PA curves for BGM240P and MGM240P modules.
- **748680:** Added the RAIL_SetRssiDetectThreshold API to allow the user to detect when the RSSI is at or above a configurable threshold. Once configured the RAIL_EVENT_DETECT_RSSI_THRESHOLD event can be used to detect when this happens.
- **758341:** Restricted the SL_RAIL_UTIL_PA_RAMP_TIME_US to 10us on some EFR32 modules to match the certification conditions.
- **772769:** Fixed an issue when running IR Calibration on the EFR32xG23 using RAIL_CalibrateIrAlt where we could compute an invalid IRCAL value for certain PHYs and chips.
- **777427:** Fixed support for using the signal identifier CCA modes simultaneously with a user-enabled signal identifier trigger event.
- **812815:** Added support for the MGM240L022RNF module.
- **813381:** Added support for the FGM230SA27HGN and FGM230SBHGN modules.
- **819644:** Fixed an issue with frame-type decoding PHYs running at more than 500kbps on EFR32xG22 and later.
- **825083:** Fixed an issue on EFR32xG23 and EFR32xG24 where PTI could merge multiple receive packets into the same transaction when interrupt latency is significant.
- **828716:** Added the RAIL_GetChannelAlt API. This function returns the channel the radio is currently using. If using DMP and run on the inactive protocol it returns the channel that will be used when next switching to that protocol. When using channel hopping, mode switch, and other features that change channels dynamically this may be different than what is returned by RAIL_GetChannel as this function will track what channel the radio is actually on at that moment and not what it started on.
- **829499:** Fixed an issue where RAIL_GetRadioStateDetail would not report the correct state information when frame detection was disabled or during an LBT operation.
- **830214:** Ensure that the RAIL_RadioConfigChangedCallback_t is called for all RAIL handles in a dynamic multi-protocol application where multiple handles use the same underlying PHY configuration.

- **835299:** Fixed an issue with dynamic handling of whitening and FCS in FSK when only RAIL_IEEE802154_E_OPTION_GB868 was enabled.
- **844600:** Fixed issue of not being able to receive packets during a RAIL_ScheduleRx configured with a zero relative start time when Power Manager sleep is enabled and configured with an EM2 or lower energy requirement.

2.12.2

- **759793:** Fixed an issue with BLE long-range reception on EFR32xG21 that corrupted packet data and tripped RAIL_ASSERT_FAILED_UNEXPECTED_STATE_RX_FIFO.
- **764346:** BLE and 802.15.4 built-in PHYs are now exposed as public symbols in header files. Do not modify these without explicit instructions by Silicon Labs.
- **764347:** Added a new Built-in PHYs for alternate HFXO frequencies component to enable the built-in phys to operate with either 38.4 MHz or 39 MHz crystals on EFR32xG24 devices.
- **774883:** Updated power curves for ZGM230SA27HGN, ZGM230SA27HNN, ZGM230SB27HGN modules to provide more accurate output powers at the lower and higher end of the dBm range.
- **777290:** The PA auto mode configuration is fixed to use both HP and LP PA modes on the 10dBm EFR32xG24 chips.
- **777427:** Fixed support for using the signal identifier CCA modes simultaneously with a user enabled signal identifier trigger event.
- **812938:** Fixed the RAIL_RX_CHANNEL_HOPPING_MODE_MULTI_SENSE mode on EFR32xG22 and EFR32xG23 to properly stay active after detecting Timing and Preamble when using the standard demodulator.
- **813182:** Previously selecting an invalid CCA mode using RAIL_IEEE802154_ConfigCcaMode would fail silently and continue to use the RSSI based CCA mode. Now RAIL_IEEE802154_ConfigCcaMode will return an error if an invalid CCA mode is selected.

2.12.1

- **758341:** Restricted the SL_RAIL_UTIL_PA_RAMP_TIME_US to 10us on some EFR32 modules to match the certification conditions.
- **764234:** Changed Quuppa channel 2 frequency from 2480 MHz to 2403 MHz.
- **765843:** Updated the Z-Wave PHYs for the EFR32xG23 to prevent a sensitivity degradation on the R2 (9.6 kbps) PHY.
- **773178:** Fixed compiler warning in Callbacks Utility component when app_assert component ignores asserts.

2.12.0

- **646980:** An attempt to use an unsupported built-in radio channel configuration, e.g. on a module that does not support that protocol or configuration, will now trip RAIL_ASSERT_FAILED_INVALID_CHANNEL_CONFIG rather than returning success and ignoring the configuration.
- **671651:** Fixed timing problems with certain State Transitions or RX Channel Hopping delay values on the EFR32xG22 and newer parts.
- **682739:** Fixed an issue with the BLE coded PHY's modulation index on the EFR32xG21 parts that could cause deviation measurements to fail.
- **688211:** Added IEEE802.15.4 Coexistence and FEM PHYs to EFR32xG12 and EFR32xG13 based modules.
- **688211:** Updated IEEE802.15.4 FEM PHYs on EFR32xG12 and EFR32xG13 based modules for improved performance.
- **697705:** Added support for the ZGM230SA27HGN, ZGM230SA27HNN, and ZGM230SB27HGN modules.
- **708206:** Added RAIL_GetTxPacketsRemaining() API for use when handling one of the RAIL_EVENTS_TX_COMPLETION to get a sense of how many transmits remain in a RAIL_SetNextTxRepeat() sequence.
- **712012:** Updated all header files to have extern "C" when being built with C++ for compatibility.
- **714271:** Fixed an issue where RAIL_IEEE802154_Config2p4GHzRadio*() and RAIL_IEEE802154_ConfigGB*Radio() functions were improperly clearing or setting certain RAIL_IEEE802154_EOptions_t. Also documented that these functions still implicitly clear or set certain RAIL_IEEE802154_GOptions_t suitable for that configuration.
- **714709:** Made the RAIL_EnableRxDutyCycle() API safe to call in a multiprotocol application.
- **715123:** Added PA curves for HP, MP, LP and LLP modes on all EFR32xG23 radio boards.
- **716369:** Fixed an issue where incorrect radio transition times were being applied at higher temperatures when using the high power PA on EFR32xG22 parts.
- **718290:** Changed the LQI measurement to use chip errors on IEEE802.15.4 PHYs for EFR32xG24 parts.
- **719194:** Added RAIL_PA_BAND_COUNT to count RAIL_PaBand_t.
- **720948:** Added a new RAIL_RxDataSource_t to capture direct mode data on supported devices.
- **721163:** To save both flash and RAM, moved information formerly contained in RAIL_Config_t::protocol, RAIL_Config_t::scheduler, and RAIL_Config_t::buffer internal to RAIL and sized appropriately for single vs. multiprotocol. RAIL multiprotocol now provides two internal state buffers for two protocols by default. An application that needs more must now

call RAIL_AddStateBuffer3() or RAIL_AddStateBuffer4() to add a 3rd and 4th buffer, respectively. Otherwise RAIL_Init() will fail when trying to initialize a 3rd or 4th protocol.

- **723098:** Fixed RAIL_SetFixedLength(handle, RAIL_SETFIXEDLENGTH_INVALID) to restore dynamic frame length operation if the current PHY was originally configured for that.
- **723558:** Added support for BGM240PA22VNA, BGM240PA32VNA, BGM240PA32VNN, BGM240PB22VNA, BGM240PB32VNA, BGM240PB32VNN, MGM240PA22VNA, MGM240PA32VNA, MGM240PA32VNN, MGM240PB22VNA, MGM240PB32VNA and MGM240PB32VNN modules.
- **723605:**Coexistence Utility GPIO interrupt numbers are now chosen at runtime to avoid conflicts.
- **726491:** A new RAIL API RAIL_GetSchedulerStatusAlt will now return more descriptive radio scheduler events as well as the RAIL_Status_t of the RAIL API invoked by the radio scheduler. As a part of the new API, new RAIL_SchedulerStatus_t events have been added while retaining the previous ones for backwards compatibility. Note that the underlying values of the existing RAIL_SchedulerStatus_t events may have changed.
- **727721:** Updated the RAIL_IEEE802154_Config2p4GHzRadio*Fem PHYs on the EFR32xG12 and EFR32xG13 devices to improve performance.
- **727728:** Added a new RAIL_IEEE802154_Config2p4GHzRadioCustom1 API to configure an alternate IEEE 802.15.4 PHY with slightly different performance characteristics for the EFR32xG12 and EFR32xG13 parts. Use this API if instructed by Silicon Labs for your use case.
- **738931:** Fixed an issue with the BLE Coded PHYs on the EFR32xG22 device that could cause some packets to be improperly sent and not trigger a RAIL_EVENT_TX_PACKET_SENT event.
- **739594:** Fixed an issue with the RX_IQDATA_FILTLSBRAIL_RxDataSource_t on EFR32xG23 parts where the data did not properly saturate and was instead just the lower 16 bits of IQ sample data.
- **744323:** Fixed an issue when using BLE AoX where non-AoX packets were transmitted on an undefined antenna. They will now always use the first antenna in the configured RAIL_BLE_AoxConfig_t::antArrayAddr pattern.
- **745528:** Fixed some incorrect RAIL_RxPacketInfo_t::filterMask values for 802.15.4 ACKs when promiscuous, or when the PanId coordinator received a packet with only source PanId and no destination address.
- **753655:** The example CSV files referred to in AN1127: Power Amplifier Power Conversion Functions in RAIL 2.x are updated with realistic values.
- **753860:** Fixed an issue when running IR Calibration on the EFR32xG23 (RAIL_CalibrateIrAlt) where we could compute a completely invalid IRCAL value for certain PHYs and chips.
- **754219:** Increased maximum BLE Coexistence request window setting, SL_RAIL_UTIL_COEX_REQ_WINDOW, in Coexistence Utility from 255 to 5000.

2.11.3

- **737292:** Fixed an issue on the EFR32xG21 that was causing some BLE coded PHY packets to be sent with a corrupted sync word and payload breaking the connection.
- **738158:** Fixed the RX_DIRECT_MODE_DATARAIL_RxDataSource_t data source on the EFR32xG23.
- **741120:** Fixed an issue where the IQ data captured during BLE's CTE could be all zeros on the 2Mbps PHY.
- **743258:** Fixed an issue that could cause the antenna switching pattern in BLE AoX applications to sometimes repeat a previous sequence instead of following the expected antenna sequence.

2.11.2

- **274536:** Added a new RAIL_ConfigDirectMode() API for configuring direct mode settings on all chips. This can set whether the data stream is synchronous or asynchronous and which GPIOs are used for the feature.
- **671651:** Fixed timing problems with certain State Transitions or RX Channel Hopping delay values on the EFR32xG22 and newer parts.
- **714271:** Fixed an issue where RAIL_IEEE802154_Config2p4GHzRadio*() and RAIL_IEEE802154_ConfigGB*Radio() functions were improperly clearing or setting certain RAIL_IEEE802154_EOptions_t. Also documented that these functions still implicitly clear or set certain RAIL_IEEE802154_GOptions_t suitable for that configuration.
- **715123:** Added PA curves for EFR32XG23 for HP, MP, LP and LLP modes for both 14dBm and 20dBm variants.
- **716369:** Fixed an issue where incorrect radio transition times were being applied at higher temperatures when using the high power PA on EFR32xG22 parts.
- **720948:** Added a new RAIL_RxDataSource_t to capture direct mode data on supported devices.
- **735862:** Now when using a RAIL_RxDataSource_t other than RX_PACKET_DATA the receiver will be disabled any time a RAIL_EVENT_RX_FIFO_OVERFLOW occurs and need to be manually restarted. If the RAIL_EVENT_RX_FIFO_OVERFLOW event is not enabled then the receiver will continue to run and lose some amount of data until the buffer is processed which matches the old behavior.

2.11.1

- **646980:** An attempt to use an unsupported built-in radio channel configuration, e.g. on a module that does not support that protocol or configuration, will now trip RAIL_ASSERT_FAILED_INVALID_CHANNEL_CONFIG rather than returning success and ignoring the configuration.
- **675252:** Fixed an antenna diversity regression where a transmitted auto- ACK would incorrectly go out the currently configured TX antenna rather than the antenna used to receive the packet being acknowledged.
- **676896:** Fixed an issue in OOK PHYs where dynamic adjustments made to receive a packet with a high RSSI could persist after the packet and decrease the ability to receive packets with a lower RSSI.
- **696198:** Fixed an issue on PHYs that do not support dual sync words. If RAIL_RX_OPTION_ENABLE_DUALSYNC is called when one of these PHYs is loaded the RAIL_ConfigRxOptions function will now return RAIL_STATUS_INVALID_PARAMETER.
- **697097:** Fixed a rare situation where a premature RAIL_EVENT_TX_BLOCKED event might occur when auto-ACK is enabled and a scheduled transmit using RAIL_SCHEDULED_TX_DURING_RX_ABORT_TX is pending when an erroneous packet is received.
- **700439:** Fixed an issue where configuring a Selective RF Sense Wakeup packet by calling RAIL_ConfigRfSenseSelectiveOokWakeupPhy followed by RAIL_SetRfSenseSelectiveOokWakeupPayload would put RAIL in fixed length mode and leave it there even after a new PHY was loaded. We will now revert the fixed length settings to their default when loading a new PHY after the Selective RF Sense Wakeup PHY was loaded so that the incoming PHY's settings are used.
- **710273:** Fixed an issue using RAIL_TX_REPEAT_OPTION_HOP with RAIL_SetNextTxRepeat() or RAIL_BLE_SetNextTxRepeat() when they return an error found in the channel-hopping configuration, yet would still attempt to repeat the next transmit anyway.
- **710983:** Added the new RAIL_STREAM_CARRIER_WAVE_PHASENOISERAIL_StreamMode_t for phase noise measurement.
- **719194:** Added RAIL_PA_BAND_COUNT to count RAIL_PaBand_t.

2.11.0

- **369712:** Added RAIL_StartScheduledCcaCsmaTx and RAIL_StartScheduledCcaLbtTx APIs to allow applications to schedule a CSMA/LBT transmit.
- **456701:** Fixed an issue on EFR32xG1x parts where calling RAIL_Init() with the MSC->CTRL.CLKDISFAULTEN bit set would cause a bus fault.
- **524885:** Added support for a new RAIL_EVENT_ZWAVE_LR_ACK_REQUEST_COMMAND, triggered on the reception of a Z-Wave Long range packet with acknowledgement request bit set, following which the application must call RAIL_ZWAVE_SetLrAckData to populate the fields of the Z-Wave Long range acknowledgement packet.
- **626961:** Add support for the RAIL_TX_POWER_MODE_*_HIGHEST options in the RAIL_GetTxPowerCurve() function.
- **654600:** On EFR32XG21, a watchdog has been added to terminate an RSSI averaging operation in case the RAIL_EVENT_RSSI_AVERAGE_DONE event does not occur.
- **655541:** Fixed an issue on EFR32XG22 and later where packet filtering might be incomplete in FEC-enabled radio configurations causing good packets to be improperly dropped. Note that if packet filtering fails close to the end of an otherwise successfully received packet, the packet may be flagged RAIL_RX_PACKET_READY_CRC_ERROR rather than RAIL_RX_PACKET_ABORT_FILTERED. This issue is still possible on earlier chips due to hardware filtering restrictions.
- **660021:** Changed when RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND event is issued to better facilitate support for 802.15.4E-2012 Enhanced ACKing and reduce time spent in the event handler. The event is now issued **after** receiving the Auxiliary Security Header (if present) in the MAC Header of the incoming frame, and for MAC Command frames, after receiving the MAC Command byte (which may be encrypted). This change is **not** backwards compatible with prior releases for Enhanced ACK requesting frames, but is for Immediate ACK requesting frames. Use of RAIL_IEEE802154_EnableEarlyFramePending() is no longer required to support Enhanced ACKing; the notion of early frame pending notification has also shifted to after the Auxiliary Security Header for Enhanced ACK requesting frames.
- **661651:** Changed RAIL Timer Synchronization over sleep on the EFR32xG21 to use the RTCC instead of the PRORTC to reduce current consumption in EM2.
- **665705:** Fixed an issue where a transmit with RAIL_TX_OPTION_SYNC_WORD_ID 1 to use SYNC2 would improperly indicate SYNC1 was used in the packet trace appended information.
- **666275:** Fixed potential delays when using RAIL's channel hopping or duty cycling features in EM1P mode on the EFR32xG22 and newer parts.
- **667103:** Fixed RAIL_ReadRxFifo() to behave as documented when passed NULL dataPtr: the data is thrown away rather than copied out.
- **671817:** Fixed an issue when switching between certain radio configurations (e.g. Z-Wave) where use of RAIL_TX_OPTION_REMOVE_CRC can become permanently stuck.

- **673333:** Fixed an issue with RAIL_TX_OPTION_WAIT_FOR_ACK transmits where an RX overflow during the ACK wait period would silently abort the ACK timer resulting in no RAIL_EVENT_RX_ACK_TIMEOUT being generated.
- **681133:** Added RAIL_TX_OPTION_RESEND to allow re-transmitting the packet most recently loaded into the Transmit FIFO. This can be used in combination with RAIL_SetNextTxRepeat() on supported platforms to repeatedly transmit the same packet.
- **681135:** Added new RAIL_SetNextTxRepeat() API and RAIL_TxRepeatConfig_t to allow configuration of repeated transmits triggered by an initial transmit.
- **681136:** Added ability to configure TxToTx state transition time via RAIL_StateTiming_t::txToTx and RAIL_SetStateTiming(). This time is generally used between an autoACK transmit and a user transmit that was pending. It's also used during repeated transmits by default, but can be overridden by the RAIL_TxRepeatConfig_t::delayOrHop or RAIL_BLE_TxRepeatConfig_t::delayOrHop configuration.
- **681141:** Added ability to hop channels during repeated transmits in RAIL_TxRepeatConfig_t using RAIL_TX_REPEAT_OPTION_HOP.
- **681143:** Added BLE-specific hooks for repeated transmits with channel hopping in RAIL_BLE_SetNextTxRepeat() and RAIL_BLE_TxRepeatConfig_t.
- **682032:** Fix an issue where setting RAIL_SCHEDULED_TX_DURING_RX_ABORT_TX for a scheduled transmit could cause subsequent non-scheduled transmits to be blocked.
- **695484:** The posting of RAIL_EVENT_TX_ABORTED now occurs before the PA ramps down, at the same time other transmit completion events get posted.
- **696665:** Fixed an issue in RAIL_ConvertRawToDbm for PAs which use piecewise-linear line segment fit, where the minimum raw power level was incorrectly compared to the minimum deci-dBm value.
- **697097:** Fixed a rare situation where a premature RAIL_EVENT_TX_BLOCKED event might occur when auto-ACK is enabled and a scheduled transmit using RAIL_SCHEDULED_TX_DURING_RX_ABORT_TX is pending when an erroneous packet is received.
- **699890:** Fixed missing C++ compatibility in the Initialization Utility component's generated header file.
- **701604:** Fixed an issue where using RX Channel Hopping with channels in different frequency bands would cause RAIL to assert.
- **703788:** Fixed an issue on EFR32xG2x devices where RAIL would not allow you to initialize the radio with voltage scaling enabled even though this is supported on these devices.
- **705595:** Fixed an issue where custom PA curves provided through the PA Module were not respected when building Silicon Labs Zigbee applications.
- **705802:** Allow RAIL_ZWAVE_ReceiveBeam to be run on US LR regions instead of always returning an error.
- **706542:** The Callbacks Utility component can now be configured to not provide the RAILCb_AssertFailed() function for situations where the application wants to provide its own implementation.
- **708511:** Fixed possible RAIL_ASSERT_FAILED_UNEXPECTED_STATE_TX_FIFO when RAIL_SetTxFifo() is called after the transmit FIFO had been filled enough to cause its write offset to wrap.

2.10.2

- **652028:** If the selected RAIL_TxPower_t is not supported by the chip, then by default RAIL_TX_POWER_MODE_2P4_HIGHEST is chosen as the power mode.
- **666275:** Fixed potential delays when using RAIL's channel hopping or duty cycling features in EM1P mode on the EFR32xG22 and newer parts.
- **666917:** Previously runtime configuring the Zigbee coexistence PHY select timeout to 255 ms would only guarantee 255 ms on the coexistence optimized PHY. Now the coexistence optimized PHY will be selected indefinitely after runtime configuring the coexistence PHY select timeout to 255 ms.
- **667555:** Fixed an issue with the Front End Module (FEM) Utility component where it would generate an error if the Rx pin was not required and turned off by the user.
- **669697:** Updated the configuration libraries on BGM210PB22JIA and BGM210PB32JIA modules to fix issues using these parts in the field.
- **671817:** Fixed an issue when switching between certain radio configurations (e.g. Z-Wave) where use of RAIL_TX_OPTION_REMOVE_CRC can become permanently stuck.
- **672904:** The minimum and maximum power levels for BGM220P and BGM220SC modules are updated to be 0 and 15 respectively.
- **672909:** Fixed an issue with PA auto mode where it might try to use a RAIL_TxPowerMode_t that is not valid for the current part.
- **673246:** A multiprotocol scheduled receive with a relative start time and an absolute end time now respects that end time regardless of when the receive actually starts. Before, the end time was made relative and pushed out based on the actual

start time to keep the receive window the same width in duration. This is a backwards-incompatible change, but should align better with what users expected when specifying an absolute end time.
- **684407:** Fixed an issue where calling RAIL_ConfigAntenna() on EFR32xG22 and newer devices could fault if the GPIO block clock was not explicitly enabled beforehand.
- **687455:** Fixed the conversion of power level to dBm to fetch correct values for BGM220P and BGM220S modules when using the RAIL_TX_POWER_MODE_2P4GIG_HP power mode and power levels above the maximum.

2.10.1

- **635037:** Added support on the EFR32xG22 parts for a new BLE PHY that can receive 1Mbps and LR Coded packets simultaneously. See RAIL_BLE_ConfigPhySimulscan().
- **652969:** Restored automatic IR calibration on EFR32xG22 at RAIL_Init() time.
- **653955:** Fixed an issue when using the Coexistence Utility component where PWM was enabled if SL_RAIL_UTIL_COEX_PWM_DEFAULT_ENABLED was set, even if SL_RAIL_UTIL_COEX_PWM_REQ_ENABLED was disabled. Now SL_RAIL_UTIL_COEX_PWM_DEFAULT_ENABLED is ignored when SL_RAIL_UTIL_COEX_PWM_REQ_ENABLED is disabled.
- **654726:** Fixed an issue where antenna diversity was not being enabled on EFR32xG1x devices when configured through the Antenna Diversity Utility component.
- **656175:** Fixed an issue where any RAIL_ChannelConfigEntry_t in a RAIL_ChannelConfig_t with a maximum power less than 0 dBm would be stuck at the maximum power no matter what power was requested.
- **663815:** Fixed an issue when using the Antenna Diversity Utility component where moving from TX antenna mode HAL_ANTENNA_MODE_ENABLE2 to HAL_ANTENNA_MODE_DIVERSITY would cause the antenna selection to get stuck on antenna 2.
- **665161:** Fixed an issue on EFR32xG22 and later PTI where the network analyzer could misrepresent the RSSI of incoming packets.

2.10.0

- **244222:** Added support for reporting more detailed transmit errors on the Packet Trace Interface (PTI).
- **318768:** Added new API RAIL_GetRadioStateDetail that provides more detailed radio state information than RAIL_GetRadioState.
- **362133:** The default RSSI offset on the EFR32xG1, EFR32xG12, EFR32xG13, EFR32xG14, and EFR32xG21 chips does not compensate for a known internal hardware offset. This offset is chip specific and can be found using the new Received Signal Strength Indicator (RSSI) Utility component which will load the correct value for your chip by default when the RSSI Offset component or Hardware Configurator peripheral is enabled. Since the hardware and antenna design can also impact this offset it is recommended that you measure this value for your particular hardware for the best accuracy. This correction is not enabled by default on the chips listed above to prevent changing radio behavior significantly without the user opting into this change. For the EFR32xG22 and future chips the hardware offset is measured and included by default. Please be aware that if any prior CCA token modifications have been made to modify the threshold for CSMA/LBT transmits, adding an RSSI Offset value could reduce or prevent TX.
- **446308:** Updated RAIL_ZWAVE_ReceiveBeam() to automatically idle the radio when RAIL_ZWAVE_ReceiveBeam() finishes even when no beam is detected.
- **451099:** Added the ability to use the Initialization Utility component multiple times when creating a multiprotocol application.
- **471715:** Fixed an issue when using RAIL_ConfigAntenna() on the EFR32xG22 with an RF path other than 0 since these parts do not have multiple RF paths.
- **478948:** Added RAIL_RxPacketInfo_t::filterMask field of type RAIL_AddrFilterMask_t which is a bitmask representing which address filter(s) the packet has passed.
- **493557:** Added the ability for RAIL_GetRssi() to wait for a valid RSSI in radio states that are transitioning into RX. Additionally, a maximum wait timeout for a valid RSSI can be configured using the new API RAIL_GetRssiAlt().
- **494571:** Added a new RAIL_ZWAVE_ConfigRxChannelHopping() API to configure Z-Wave Rx channel hopping using the recommended hopping parameters.
- **494571:** Added a new RAIL_ZWAVE_GetRegion() API to determine the currently selected Z-Wave region.
- **495497:** Changed RAIL_PacketTimeStamp_t::totalPacketBytes from uint32_t to uint16_t to reduce RAM usage.
- **497196:** The Initialization Utility component now defaults most options to a disabled state, instead of enabled. Now you have to opt-in, instead of opt- out, of RAIL init functionality.
- **501510:** Added a new RAIL_SupportsTxPowerModeAlt() API to get the minimum and maximum power levels for a specific power mode if the power mode is supported by the chip.
- **519195:** The EFR32xG21 will now use RTCC channel 0, as opposed to the PRORTC, to perform sleep timer synchronization. This will help lower the EM2 current consumption for this chip.

- **519288:** The Power Amplifier (PA) Utility component now enables PA calibration by default to ensure that PA power remains consistent chip-to-chip.
- **520521:** Added a new API RAIL_SetAddressFilterAddressMask() that allows for setting a bit mask pattern for packet data in the address filters.
- **520937:** Add new RAIL_EVENT_RF_SENSED as an alternative to the current RAIL_StartRfSense() callback parameter.
- **520943:** Added a new API RAIL_ConfigSleepAlt() to allow configuring the PRS channel, RTCC channel, and whether sleep is enabled in one call.
- **524046:** Added support for MGM210PB22JIA, MGM210PB32JIA, BGM210PB22JIA and BGM210PA32JIA modules.
- **580312:** Created a new Protocol Utility component for setting up RAIL to use one of the standards based PHYs by default.
- **630457:** On custom boards, the Packet Trace Interface (PTI) Utility component no longer reserves pins for use without being configured.
- **632723:** The EFR32xG22 will limit going to EM1P sleep mode when an 80MHz HRFRCO PLL system clock is selected. Going to EM1P sleep is not supported when using the DPLL on this hardware as it can cause clock drift which would impact radio timing and tuning.
- **635237:** Added an event RAIL_EVENT_PA_PROTECTION to indicate the power protection circuit has kicked in.
- **638067:** Fixed a DMP issue that poached transmit power when switching between protocols using the same channel configuration and channel.
- **638239:** Created a Callbacks Utility component for application- level callbacks.
- **639833:** Fixed a potential radio hang on a corrupted BLE packet when doing BLE AoX.
- **642893:** Reduced RAIL library flash data alignment needs on the EFR32xG22.
- **645641:** Fixed an EFR32xG22 issue where a state transition to receive after a transmit from EM2 sleep would drop packets.
- **650072:** In multiprotocol RAIL, when the supplied handle is not the active handle, RAIL_GetRadioState now returns RAIL_RF_STATE_RX rather than RAIL_RF_STATE_IDLE if a background receive is currently scheduled.

2.9.2

- **501674:** Reduced the time from last modulated bit on air to ramp down on EFR32xG1x devices.
- **623922:** Fix to set the right power on EFR32xG21 when the PA uses VDD around 1.8V.

2.9.1

- **493409:** Fixed an issue where output power might not be set correctly when changing frequency bands within one channel configuration.
- **497061:** Corrected an issue on EFR32xG1x chips where specifying a large ramp time could erroneously add extra delay to a transmit.
- **519195:** EFR32xG21 will now use RTCC channel 0, as opposed to the PRORTC, to perform sleep timer synchronization. This will help lower the EM2 current consumption for this chip.
- **524046:** Added support for MGM210PB22JIA, MGM210PB32JIA, BGM210PB22JIA and BGM210PA32JIA modules.

2.9.0

- **204557:** Added support for setting the default value of the FramePending bit in outgoing IEEE 802.15.4 data poll ACKs to true. This means that the user would then be responsible for clearing this bit in the frame pending callback instead of having it default to cleared and having to set it. See the RAIL_IEEE802154_Config_t::defaultFramePendingInOutgoingAcks field for configuring this feature.
- **298829:** Added a new API RAILCb_ConfigSleepTimerSync() to allow for configuration of the PRS and RTCC channels used for timer sync operations.
- **300348:** Added support for a new RAIL_EVENT_SCHEDULED_RX_STARTED and RAIL_EVENT_SCHEDULED_TX_STARTED, triggered when a scheduled receive or transmit begins. These are the same value because a scheduled receive and transmit cannot occur at the same time. Note: This new event shifted the bit positions of some events in RAIL_Events_t.
- **381485:** Provided new RAIL_RX_CHANNEL_HOPPING_MODE_MULTI_SENSE along with RAIL_RxChannelHoppingConfigMultiMode_t to configure its parameters. This mode can be configured to tolerate brief loss of timing and/or preamble making it less susceptible to hopping than the single-sense modes. It can also be used with RX duty cycling.
- **402203:** Provided new RAIL_RX_CHANNEL_HOPPING_OPTION_RSSI_THRESHOLD to augment each of the RAIL_RxChannelHoppingMode_t modes with one RSSI Threshold check on entering receive for the channel. If the RSSI is below that specified by RAIL_RxChannelHoppingConfigEntry_t::rssiThresholdDbm then hop (or if below RAIL_RxDutyCycleConfig_t::rssiThresholdDbm, suspend RX). RAIL_RxDutyCycleConfig_t has been augmented not only with this new field, but also now includes RAIL_RxDutyCycleConfig_t::options field. For those options to be recognized by

RAIL_ConfigRxDutyCycle() (due to backwards compatibility) RAIL_RxDutyCycleConfig_t::mode must be one of the new WITH_OPTIONS modes, e.g. RAIL_RX_CHANNEL_HOPPING_MODE_TIMEOUT_WITH_OPTIONS.

- **411017:** Relax constraints in RAIL to allow calling RAIL_SetRxTransitions, RAIL_SetTxTransitions, RAIL_ScheduleRx, and all of the RAIL_BLE_ConfigPhy functions before the radio is completely IDLE.
- **414398:** Added support for the Silicon Labs Power Manager component. When enabled via RAIL_InitPowerManager() RAIL will communicate directly with the power manager to configure sleep modes. See the Sleep section for more documentation.
- **429513:** Added support for RAIL's FrameType based length on the EFR32xG22.
- **442248:** Updated the pa_customer_curve_fits.py helper script to work with Python 3 as well as Python 2.
- **442978:** Changed pa_customer_curve_fits.py to take maxpower as a parameter to generate better curves. When maxpower and increment are different than the defaults they will now be included in the output curve. The current power curve limits can be read at runtime from the new RAIL_GetTxPowerCurveLimits API.
- **450750:** The RAIL_GetRadioEntropy() API will now ensure a valid radio configuration has been loaded using RAIL_ConfigChannels() since it can cause problems if the radio is used before this.
- **454096:** Updated the function RAIL_GetRxPacketDetailsAlt to return the time position of the received packet timestamp corresponding to the default location in the packet.
- **454096:** Added a new function, RAIL_GetTxPacketDetailsAlt2, which allows a RAIL_TxPacketDetails_t structure to be passed as an argument.
- **454096:** Added a new function, RAIL_GetTxTimePreambleStartAlt, which allows a RAIL_TxPacketDetails_t structure to be passed as an argument.
- **454096:** Added a new function, RAIL_GetTxTimeSyncWordEndAlt, which allows a RAIL_TxPacketDetails_t structure to be passed as an argument.
- **454096:** Added a new function, RAIL_GetTxTimeFrameEndAlt, which allows a RAIL_TxPacketDetails_t structure to be passed as an argument.
- **456338:** Fixed an issue with RAIL state transitions where an internal timer wrapping could cause incorrect transition times. This error would previously affect at most one packet every 15 minutes.
- **459581:** Fixed an issue with the EFR32xG21 medium power PA that could cause the output power to be too low for certain power level and ramp time combinations.
- **464534:** Fixed issue where RAIL_StartAverageRssi() ran twice as long as it should have.
- **464734:** Regenerated the power curves for the EFR32xG22 to allow access to the maximum power level available on the chip.
- **464735:** Closed tiny timing window on EFR32xG13 that might corrupt PTI appended info when idling the radio.
- **465096:** Fixed an issue where RAIL_Idle() was not properly terminating an ongoing RAIL_StartAverageRssi() process.
- **465220:** Bluetooth Angle of Arrival and Angle of Departure support has been removed from the EFR32xG13 chip. For future usage of this feature please look to the EFR32xG22.
- **466012:** Fixed an issue where the CRC could be disabled indefinitely on transmit when switching configs in a multiphy setup.
- **474678:** Fixed an issue with duty cycle receive and channel hopping on the EFR32xG1x parts where some parts of the radio would be left on even with long delay parameters causing extra current to be used. This allows for a noticeable improvement in power consumption when using the RAIL_ConfigRxDutyCycle() API with delays of more than a hundred microseconds.
- **475184:** Fixed an issue on the EFR32xG22 where the receiver was not automatically re-calibrated if the temperature changed significantly while sitting in receive. This could cause the radio to go off channel for significant temperature changes resulting in receive problems.
- **477621:** Improved frequency accuracy on EFR32xG1x devices when the radio configuration has an entry with a large number of channels. Previously, small errors in the channel spacing calculation could exist. If they did, then when computing the channel frequency this error would be multiplied by the channel number minus the start channel for that entry causing some drift for higher order channels. This was not much an issue in most cases, but for certain PHY and crystal combinations it could be worse.
- **477833:** Some radio configurations on the EFR32XG22 are not usable with RAIL address filtering and RAIL 802.15.4 filtering. Add an assert to catch those cases.
- **479539:** Fixed a bug where the RAIL_ConfigTxPower() would override the PA capacitor tuning values for transmit and receive without caching them. In a multiprotocol scenario this could cause us to apply incorrect PA capacitor tuning values set prior to a call to RAIL_ConfigTxPower. Note that you must still call RAIL_SetPaCTune after any making any changes to the power configuration via RAIL_ConfigTxPower.
- **479665:** Fixed an issue where RAIL_SetRxFifo() would not reject a buffer smaller than 64 bytes and would mistakenly think it is very big. In addition, RAIL_ASSERT_FAILED_RX_FIFO_BYTES can no longer occur with a 64 byte buffer when doing IR calibration.
- **482007:** Fixed a bug in multiprotocol RAIL where running an IR calibration during a protocol switch would fail. The calibration function will now return RAIL_STATUS_INVALID_STATE if called in such a scenario.

- **483688:** Fixed an issue where the power mode selected when using RAIL_TX_POWER_MODE_2P4GIG_HIGHEST on supported chips was not saved in a multiprotocol context and could cause problems.
- **484374:** Fixed regression from 2.8.1 on EFR32xG21 where Bluetooth LE did not include packet sync word on PTI.
- **488752:** Fixed an issue with MGM1x and BGM1x modules that caused PHY changes to be slower than when using stand alone parts. This could cause problems in certain BLE or DMP use cases where reconfiguration time is critical.
- **489214:** Fixed an issue where calling RAIL_IEEE802154_CalibrateIr2p4Ghz, RAIL_IEEE802154_CalibrateIrSubGhz, or RAIL_BLE_CalibrateIr with a NULL imageRejection parameter would result in a crash.

2.8.10

- **456701:** Fixed an issue on EFR32xG1x parts where calling RAIL_Init() with the MSC->CTRL.CLKDISFAULTEN bit set would cause a bus fault.
- **702377:** The GB868 PHYs have been modified to improve performance for the EFR32xG1, EFR32xG12, EFR32xG13 and EFR32xG14.

2.8.9

- **645641:** Fixed an issue on EFR32xG22 where a state transition to receive after a transmit from EM2 sleep would drop packets.
- **661651:** Changed RAIL Timer Synchronization over sleep on the EFR32xG21 to use the RTCC instead of the PRORTC to reduce current consumption in EM2.
- **671168:** Fixed an issue on EFR32xG22 parts where current consumption would be higher than expected after radio activity.

2.8.8

- **639833:** Fixed a potential radio hang on a corrupted BLE packet when doing BLE AoX.

2.8.7

- **499216:** Fixed an issue with the EFR32xG21 medium power PA that could cause the output power to be too low for certain power level and ramp time combinations.
- **499366:** Added support for new BGM220 modules.

2.8.6

- **477833:** Some radio configurations on the EFR32XG22 are not usable with RAIL address filtering and RAIL 802.15.4 filtering. Add an assert to catch those cases.
- **484374:** Fixed regression from 2.8.1 on EFR32xG21 where BLE did not include packet sync word on PTI.

2.8.4

- **344182:** Added new RAIL_BLE_ConfigAoxAntenna API to configure which GPIO pins are used for BLE AoX.
- **411017:** Relax constraints in RAIL to allow calling RAIL_SetRxTransitions, RAIL_SetTxTransitions, RAIL_ScheduleRx, and all of the RAIL_BLE_ConfigPhy before the radio is completely IDLE
- **465096:** Fixed an issue where RAIL_Idle() was not properly terminating an ongoing RAIL_StartAverageRssi() process.
- **467589:** Updated default dynamic multiprotocol (DMP) transition timings to make them work with Zigbee and BLE DMP applications work. The previously suggested workaround of adding 30 us to the default transition time using RAIL_SetTransitionTime() is no longer required.
- **471373:** Fixed an issue on the EFR32xG22 where loading IEEE 802.15.4 and BLE PHYs without a reset would cause an assert with error code RAIL_ASSERT_CACHE_CONFIG_FAILED.
- **471955:** Fixed an issue with BGM220 modules that caused an assert, RAIL_ASSERT_INVALID_MODULE_ACTION, when using them in previous releases.

2.8.3

- **387749:** Revamped Features and rail_features.h, providing runtime RAIL_SupportsSomeFeature() APIs for each of the features as some features may be restricted to certain chips within a family. Also added more consistent RAIL_SUPPORTS_ compile-time synonyms for the features while retaining the existing RAIL_FEAT_ defines for backwards compatibility. These defines can now be used in C code and not just preprocessor #if statements.
- **442248:** Updated the pa_customer_curve_fits.py helper script to work with Python 3 as well as Python 2.
- **464735:** Closed tiny timing window on EFR32xG13 that might corrupt PTI appended info when idling the radio.

- **464774:** * Calling RAIL_ConfigSleep() with RAIL_SLEEP_CONFIG_TIMERSYNC_ENABLED on chips that use the PRORTC for synchronization (EFR32xG13 and newer) will now only configure the choose the LF clock source if the PRORTC IRQ is disabled. This allows for other code to safely configure the PRORTC like the Silicon Labs generic sleep timer.
- **466396:** Added support for new BGM220 modules.
- **469015:** Fixed an issue on the EFR32xG21 that could cause the RAIL_GetRadioEntropy() function to return the same first 4 bytes when called with the radio off after a reset.

2.8.2

- **456338:** Fixed an issue with RAIL state transitions where an internal timer wrapping could cause incorrect transition times. This error would previously affect a maximum of one packet every 15 minutes.
- **460062:** Fixed a RAIL_ScheduleRx() issue where RAIL_EVENT_RX_SCHEDULED_RX_END might not be posted when the Rx RAIL_StateTransitions_t::error transition is to RAIL_RF_STATE_IDLE and the Rx window ended during receipt of an erroneous packet.

2.8.1

- **251720:** Added the new RAIL_STREAM_10_STREAMRAIL_StreamMode_t to allow you to send a 1010 stream for debugging.
- **316106:** Added a new function, RAIL_StartTxStreamAlt, which allows the specific antenna to be specified for a stream transmit.
- **383197:** Added new RAIL_RX_PACKET_HANDLE_OLDEST_COMPLETE packet handle to allow the user to get a reference to the oldest unreleased complete packet.
- **435689:** Added a new External Thermistor interface to RAIL. This allows access the user to connect and read the impedance of an external thermistor on supported chips.
- **444205:** Fixed a transmit-from-idle issue with RAIL_StartCcaCsmaTx() or RAIL_StartCcaLbtTx() which would always fail when the RAIL_StateTiming_t::idleToRx is configured below the minimum the radio is capable of achieving (typically 65-100 microseconds depending on platform).
- **450750:** The RAIL_GetRadioEntropy() API will now ensure a valid radio configuration has been loaded using RAIL_ConfigChannels() since it can cause problems if the radio is used before this.
- **452628:** Fixed an issue where idling the radio from an Rx antenna diversity mode would consume extra power.
- **452690:** Fixed an issue where Rx antenna diversity could be left enabled after switching to a radio configuration that lacks diversity support.
- **455495:** Added RAIL_IEEE802154_ConvertRssiToEd() and RAIL_IEEE802154_ConvertRssiToLqi() to assist Zigbee 802.15.4 certification testing.
- **456225:** Changed the value of RAIL_FREQUENCY_OFFSET_INVALID from -1 to -32768 since -1 is a reasonable frequency offset to pass to RAIL_SetFreqOffset(). Also added convenience definitions RAIL_FREQUENCY_OFFSET_MIN and RAIL_FREQUENCY_OFFSET_MAX to specify the valid range of offset values the radio supports.

2.8.0

- **197573:** Suppressed extraneous RAIL_EVENT_TX_START_CCA events that might occur during long CCA durations. Now only one such event should occur per CCA try.
- **295265:** Added two new APIs, RAIL_GetSyncWords and RAIL_ConfigSyncWords(), to allow getting and setting the sync word configuration of your PHY at runtime.
- **301472:** On EFR32xG12 thru EFR32xG14, added support for 802.15.4G-2012 SUN PHY dynamic frame payload whitening on reception and transmit based on the PHY header's Data Whitening flag setting. This feature is automatically enabled when RAIL_IEEE802154_ConfigGOptions()' RAIL_IEEE802154_G_OPTION_GB868 is enabled, and assumes the radio configuration specifies the appropriate whitening algorithm and settings.
- **301488:** On EFR32xG12 thru EFR32xG14, added support for 802.15.4G-2012 SUN PHY dynamic frame payload 2/4-byte FCS (CRC) on reception and transmit based on the PHY header's FCS Type flag setting. This feature is automatically enabled when RAIL_IEEE802154_ConfigGOptions()' RAIL_IEEE802154_G_OPTION_GB868 is enabled. The radio configuration's (single) CRC algorithm settings are ignored, overridden by RAIL.
- **301501:** On EFR32xG12 thru EFR32xG14, 802.15.4 AutoACK behavior has also been updated so transmitted immediate ACKs reflect the Whitening and 2/4-byte FCS of the received frame being acknowledged.
- **369722:** Added RAIL_TX_OPTION_CCA_ONLY to just perform CCA (CSMA/LBT), stopping short of automatically transmitting when the channel is clear.
- **369727:** Added support for a new RAIL_EVENT_TX_STARTED, triggered when the first preamble bit is about to go on-air. Also included the ability to retrieve the equivalent RAIL_PACKET_TIME_AT_PREAMBLE_START timestamp of that event from the

event's handler via RAIL_GetTxTimePreambleStart(). Note: This new event shifted the bit positions of some events in RAIL_Events_t.

- **369727:** Also changed RAIL_GetRxTimePreambleStartAlt(), RAIL_GetRxTimeSyncWordEndAlt(), and RAIL_GetRxTimeFrameEndAlt() to properly update its pPacketDetails' RAIL_PacketTimeStamp_t::timePosition to reflect the adjusted RAIL_PacketTimeStamp_t::packetTime rather than leaving it as RAIL_PACKET_TIME_DEFAULT.
- **384878:** Clarified documentation of the RAIL_EVENT_RX_ACK_TIMEOUT event and RAIL_AutoAckConfig_t::ackTimeout period which extends only to packet sync word detection of an expected ACK, not packet completion of that ACK.
- **397072:** Added an API, RAIL_StopInfinitePreambleTx, that can stop an infinite preamble on PHYs configured to use infinite preambles.
- **402991:** Added additional information to the packet trace stream for the Z-Wave protocol to indicate what region is currently active to help with decoding.
- **406080:** Added support for RFSENSE Selective (OOK) mode for supported chips, which currently includes only EFR32xG22 devices. Please refer to RAIL internal chip specific documentation for more details.
- **406871:** Documented RAIL's internal 16-packet metadata FIFO which exists on EFR32 platforms supplementing the receive FIFO of packet data. Refer to Data Management and EFR32 for details. Included is support for a new RAIL_EVENT_RX_FIFO_FULL, triggered with any packet completion event in which the receive FIFO or packet metadata FIFO are full. This tells the application it must free up the oldest packets/data ASAP to reduce the chance of RAIL_EVENT_RX_FIFO_OVERFLOW (however, overflow may already have occurred). Note: This new event shifted the bit positions of some events in RAIL_Events_t.
- **411498:**RAIL_StartAverageRssi() now returns RAIL_STATUS_INVALID_STATE if called when the radio is not idle, enforcing its documented behavior.
- **414114:** Added a new PA mode which will attempt to automatically choose the PA which consumes the least amount of current to reliably produce the requested output power. See RAIL_EnablePaAutoMode() for details.
- **417340:** Fixed an issue where RAIL_RxPacketDetails_t::isAck would incorrectly be set true for non-ACK or unexpected ACK packets received successfully (e.g. when RAIL_IEEE802154_ACCEPT_ACK_FRAMES is enabled) or aborted while waiting for the expected ACK. Note that when RAIL_RX_OPTION_IGNORE_CRC_ERRORS is in effect, an expected ACK includes one that fails CRC, and will have isAck set true.
- **418493:** RAIL_ConfigRadio will now return RAIL_STATUS_INVALID_STATE if called from the inactive config in dynamic multiprotocol instead of returning success but not applying the change.
- **427934:** Fixed a race condition that could cause a device to not re- enable frame detection after an Rx overflow event if the overflow was processed and cleared very quickly.
- **430026:** Enforced and clarified that RAIL_Init() should not be called more than once per protocol.
- **430081:** Fixed an issue where the first Clear Channel Assessment (CCA) of a CSMA/LBT transmit from radio idle state would consistently fail when the RAIL_CsmaConfig_t::ccaBackoff or RAIL_LbtConfig_t::lbtBackoff time is smaller than the RAIL_StateTiming_t::idleToRx time.
- **436163:** Fixed a post-receive transition timing issue for received packets that were on the air longer than 32 milliseconds. AutoACK turnaround timing should now behave properly at low data rates.
- **437054:** Fixed an issue with the pa_customer_curve_fits.py that caused values below -12 to not be considered when computing the fit. Re-generated default, Silicon Labs-provided curves to consume this fix, resulting in minor changes to the lowest-power segment in curve-fit based PA's. If using a custom power curve created using the documentation in AN1127 customers should re-run the script on the already collected output data to get slightly more accurate curves.
- **441635:** Return the correct RAIL_TxPowerMode_t value of RAIL_TX_POWER_MODE_NONE from RAIL_GetTxPowerConfig if called before RAIL_ConfigTxPower.
- **446289:** Fixed RAIL_IDLE_ABORT to idle the radio sooner when in RAIL_RF_STATE_RX, especially now that RAIL_RxChannelHoppingConfigEntry_t::delay can extend the time in that state.
- **447578:** Fixed an issue where setting a transmit power over the maximum allowed for a given channel would result in no change in the output power instead of using the maximum allowed value.
- **450187:** Fixed an issue where calling RAIL_Idle() with RAIL_IDLE_FORCE_SHUTDOWN while in receive with channel hopping enabled could corrupt some internal channel hopping state and trigger a bus fault or other radio problems.

2.7.4

- **436215:** Fixed an issue on the EFR32xg13 parts that caused the 2Mbps BLE IFS time to be incorrect when receiving very short packets.
- **443144:** Fixed the module specific config libraries for the xGM210xA modules: BGM210LA22JIF, BGM210LA22JNF, BGM210PA22JIA, BGM210PA22JNA, BGM210PA32JIA, BGM210PA32JNA, MGM210LA22JIF, MGM210LA22JNF, MGM210PA22JIA, MGM210PA22JNA, MGM210PA32JIA, MGM210PA32JNA. Without this change these modules would assert with RAIL_ASSERT_INVALID_MODULE_ACTION.

2.7.3

- **425464:** Improved interoperability with many common devices for the 1Mbps Viterbi BLE PHY on the EFR32xG13 chips. Note that this does revert a previous change to improve interoperability from changelist_2_6_4 that attempted to improve interoperability with a few specific devices. For best interoperability we recommend using the RAIL_BLE_ConfigPhy1Mbps() standard PHY.
- **431229:** Resolved an issue on EFR32xG1x devices which could cause the transmit power to be incorrect when using the low power 2.4GHz PA (RAIL_TX_POWER_MODE_2P4_LP).

2.7.2

- **336739:** Added new RAIL_EVENT_RX_SCHEDULED_RX_MISSED and RAIL_EVENT_TX_SCHEDULED_TX_MISSED events to inform the user when these are missed after waking from RAIL_Sleep(). This can happen if the specified wakeup time is not long enough.
- **421410:** Fixed an issue introduced in 2.7.1 with TX PTI on EFR32XG13 including an extra byte of appended information, confusing the network analyzer.

2.7.1

- **337468:** Fixed an issue where calling RAIL_Sleep() if there were no RAIL events pended would not stop and synchronize the clock source as requested but would return success indicating it had. The clock source will now be properly stopped and synchronized even if there are no events pending and it will be the user's responsibility to wake up on time.
- **337468:** Fixed an issue where calling RAIL_Wake() without a successful call to RAIL_Sleep() first could cause the clock source to drift even when using RAIL_SLEEP_CONFIG_TIMERSYNC_ENABLED.
- **389462:** Fixed an issue with RAIL_Calibrate() in multiprotocol, which would return RAIL_STATUS_INVALID_STATE if it is called with an inactive railHandle. Now, RAIL_Calibrate() will make the given railHandle active, if not already, and perform calibration.
- **389462:** Fixed an issue with RAIL_Calibrate() where after completing calibration the radio would no longer be able to receive any packets, sometimes, until it was reset.
- **396786:** Added support for the following EFR32xG21-based modules: BGM210L022JIF1, BGM210L022JIF2, BGM210L022JNF1, BGM210L022JNF2, BGM210LA22JIF1, BGM210LA22JIF2, BGM210LA22JNF1, BGM210LA22JNF2, BGM210P022JIA1, BGM210P022JIA2, BGM210P022JNA2, BGM210P032JIA1, BGM210P032JIA2, BGM210P032JNA2, BGM210PA22JIA1, BGM210PA22JIA2, BGM210PA22JNA2, BGM210PA32JIA1, BGM210PA32JIA2, BGM210PA32JNA2, MGM210L022JIF1, MGM210L022JIF2, MGM210L022JNF1, MGM210L022JNF2, MGM210LA22JIF1, MGM210LA22JIF2, MGM210LA22JNF1, MGM210LA22JNF2, MGM210P022JIA1, MGM210P022JIA2, MGM210P022JNA2 MGM210P032JIA1, MGM210P032JIA2, MGM210P032JNA2 MGM210PA22JIA1, MGM210PA22JIA2, MGM210PA22JNA2 MGM210PA32JIA1, MGM210PA32JIA2, MGM210PA32JNA2
- **401826:** Fixed several issues and race conditions with RAIL_ScheduleRx() window end handling and event reporting to conform to its intended design:
  - When the RX window ends with no RAIL_EVENTS_RX_COMPLETION imminent, RAIL_EVENT_RX_SCHEDULED_RX_END is posted.
  - When the RX window is implicitly ended by one of the RAIL_EVENTS_RX_COMPLETION, e.g. because it results in an RX transition to Idle or because RAIL_ScheduleRxConfig_t::rxTransitionEndSchedule is non-zero, the event(s) posted depend on that setting:
    - When zero, both events are posted simultaneously.
    - When non-zero, only the appropriate RAIL_EVENTS_RX_COMPLETION is posted, unless that event is not enabled, in which case RAIL_EVENT_RX_SCHEDULED_RX_END is substituted instead.
  - When the RX window ends while receiving a packet, it is deferred to the expected RX completion event (which includes aborting that packet when RAIL_ScheduleRxConfig_t::hardWindowEnd is non-zero). Event(s) reported at that time are the same as in the previous case.
- **406276:** Restored ability for RAIL_StartCcaCsmaTx() and RAIL_StartCcaLbtTx() to perform an immediate transmit when their respective RAIL_CsmaConfig_t::csmaTries or RAIL_LbtConfig_t::lbtTries is 0. This functionality was improperly removed in 2.6.0.
- **407047:** Fixed an EFR32xG21 issue where CSMA/LBT CCA durations were significantly shorter than specified.
- **408096:** Fixed the 802.15.4 ACK turnaround time on the EFR32xG21 platform. Due to a calculation error this was actually 18 us too short which could cause interoperability problems.
- **414257:** Added RAIL_TX_POWER_LEVEL_MAX which can be used to set the max PA power level across PA's. RAIL_TX_POWER_LEVEL_INVALID was added in 2.7.0 as the value 255. Some customers were using 255 to set max power across PA's with RAIL_SetTxPower, which previously worked, but will now return an error.

2.7.0

- **379150:** Added support to RAIL_Sleep() for the PLFRCO on EFR32xG13 Rev D parts.
- **357922:** Changed the LQI metric for the 2.4GHz IEEE802.15.4 PHY configurations to be scaled from 0 - 255 and to include more data to make it more stable. This can impact existing applications that are using the LQI values returned in prior RAIL versions.
- **378035:** Use of the unsafe enum GPIO_Port_TypeDef within RAIL aggregate types RAIL_PtiConfig_t and RAIL_AntennaConfig_t has been replaced by safe uint8_t.
- **376229:** Fixed an issue with Rx antenna diversity operation that prevented CCA from working, causing CSMA failures.
- **315849:** Reduced switch time overhead for dynamic multiprotocol applications. The new switch times as well as information about them are documented at Understanding the Protocol Switch Time.
- **370805:** Fixed an issue with the EFR32xG21 reporting a phantom packet on PTI after reset.
- **377026:** Improved documentation of RAIL_RxPacketStatus_t values and their corresponding RAIL_Events_t events.
- **400303:** Corrected an issue where RAIL_EVENT_TX_CHANNEL_BUSY due to RAIL_CsmaConfig_t::csmaTimeout or RAIL_LbtConfig_t::lbtTimeout would prevent further transmits.
- **400303:** Corrected an issue where an invalid RAIL_CsmaConfig_t::ccaDuration or RAIL_LbtConfig_t::lbtDuration too large for the radio configuration to handle would not fail the respective transmit; this now returns RAIL_STATUS_INVALID_PARAMETER.
- **400303:** Clarified that RAIL_EVENT_TX_BLOCKED and RAIL_EVENT_TX_CHANNEL_BUSY leave the TX FIFO intact without consuming any of its packet data.
- **392350:** Corrected an issue where the radio might be left in receive after a RAIL_EVENT_TX_BLOCKED or RAIL_EVENT_TX_CHANNEL_BUSY when the transmit RAIL_StateTransitions_t::error is RAIL_RF_STATE_IDLE.
- **251786:** Added new 802.15.4 RAIL APIs RAIL_IEEE802154_EnableEarlyFramePending() and RAIL_IEEE802154_EnableDataFramePending() to support Thread Basil-Hayden Enhanced Frame Pending feature.
- **369736:** Added new 802.15.4 RAIL APIs RAIL_IEEE802154_ConfigGOptions() and RAIL_IEEE802154_ConfigEOptions() for configuring certain 802.15.4E-2012 and G-2012 features needed by GB868. Also, RAIL may now suppress the RAIL_EVENT_IEEE802154_DATA_REQUEST_COMMAND when latency has delayed the handling of that event too late to influence the outgoing ACK for it.
- **385960:** Added information to packet trace for every protocol switch in dynamic multiprotocol telling the user what protocol RAIL has changed to as well the radio event that triggered this switch. This information is visible in Network Analyzer for better debugging of DMP applications.
- **389439:** The EFR32xG13 AoX RAIL library is no longer present as that functionality has been merged into the standard RAIL library for EFR32xG13.
- **360371:** Fixed an issue where calling RAIL_GetTxPowerDbm prior to calling RAIL_SetTxPower would return -500 i.e. -50dBm. As a part of the fix, we now return an invalid dBm value, RAIL_TX_POWER_MIN, if RAIL_SetTxPower was not called before calling RAIL_GetTxPowerDbm or if RAIL_SetTxPower it returns an error status.
- **382575:** Added a new API RAIL_UseDma(), which can be used to enhance RAIL startup speed if called before RAIL_Init().
- **376711:** Added support for Z-Wave node ID based packet filtering via the RAIL_ZWAVE_OPTION_NODE_ID_FILTERING option.
- **386163:** Added support for the radio sending an ACK packet automatically when in Z-Wave mode as long as node ID filtering and Auto-ACK features are enabled and a packet requesting an ACK is sent to the device.

2.6.5

- Corrected the frequencies used by Z-Wave in the China, Malaysia, and India regions to match the specification. China was using the EU frequencies, Malaysia was using 868.10 MHz, and India was not applying the required 20 kHz offset to the R1 channel.

2.6.4

- Fix an issue on the EFR32xG21 where the medium power and low power PAs might not complete ramping and end up transmitting at a lower power than desired.
- Improved interoperability of the BLE 1Mbps PHY on EFR32xG13 devices.
- Improved performance of OOK PHYs.
- Updated packet trace format to include extra information for BLE packets.

2.6.3

- Fixed an issue introduced in 2.6.2 where issuing a RAIL_Idle() with mode RAIL_IDLE_FORCE_SHUTDOWN_CLEAR_FLAGS could sometimes cause the receive FIFO to be corrupted and trigger a RAIL assert with code (RAIL_ASSERT_FAILED_ILLEGAL_RXLEN_ENTRY_STATUS).
- Various bug fixes and improvements for the EFR32xG21 parts.

2.6.2

- Updated the PA curves on the EFR32xG21 chips to slightly improve transmit power accuracy.
- Resolved an issue where RAIL_GetChannelHoppingRssi would return an RSSI value even if receive was disabled.
- Resolved an issue where RAIL_GetChannelHoppingRssi would not update the RSSI value while receiving packets.
- Resolved an issue in multiprotocol where an absolute event with a non zero slip time would always work even if scheduled in the past.

2.6.1

- Resolved an issue on the EFR32xG21 platform that could cause the chip to hang when coming out of a reset if RAIL_Idle() was called at the wrong time.
- Added RAIL_GetChannelHoppingRssi API to allow the user to read the RSSI from each channel when channel hopping is enabled.
- Fixed an issue where CRC-failed packets received with RAIL_RX_OPTION_IGNORE_CRC_ERRORS would be reported with RAIL_RX_PACKET_READY_SUCCESS instead of RAIL_RX_PACKET_READY_CRC_ERROR when the RAIL_EVENT_RX_FRAME_ERROR event isn't enabled.
- Fixed an issue on the EFR32xG21 platform that caused the chip to incorrectly compute the PA ramp time at high power levels. If this happened, the ramp time would be higher than expected and cause large delays in the RAIL_StateTiming_t::rxToTx time which is used for sending ACK packets.
- Fixed an issue on the EFR32xG21 platform that could cause erroneous Rx overflow events and failed receives when using the 125Kbps BLE coded PHY.

2.6.0

- Fixed an issue causing the receive antenna to always be reported as antenna 0 when using receive antenna diversity.
- Added alternative APIs for calculating receive timestamps, which give correct results for Long-Range BLE: RAIL_GetRxTimeFrameEndAlt, RAIL_GetRxTimePreambleStartAlt, RAIL_GetRxTimeSyncWordEndAlt.
- Fixed an issue where receive events could corrupt the transmit packet length in frame type based length mode if they came in between the start transmit call and the actual transmit.
- Fixed an Auto-ACK problem where ACKs were being sent for CRC error frames when RAIL_RX_OPTION_IGNORE_CRC_ERRORS is in effect.
- Fixed a math error that could cause the RAIL_CsmaConfig_t::ccaDuration to be shorter than requested. This would happen whenever the (ccaDuration * rx_baudrate) exceeded the size of a 32 bit integer.
- Fixed an issue in the multiprotocol RAIL library where the RAIL_GetTxPower() function could return the wrong output power if called with any RAIL handle other than the active one. We will now return the actual power that would be set if that protocol is active.
- Fixed the PA conversion code so that it better handles error conditions and will not trigger a fault when given NULL parameters.
- Added new API to retrieve the channel index of the most recent Beam packet accepted: RAIL_ZWAVE_GetBeamChannelIndex.
- Added a new RAIL_DelayUs() API for short blocking delays using the RAIL time base.
- Fixed a problem where state timings would not properly be updated when switching if both protocols used the same channel configuration structure.
- Fixed an issue on the EFR32xG14 series that prevented Rx antenna diversity from functioning properly. This was introduced in the RAIL 2.5.0 release.
- Restored pre-2.5.0 behavior with RAIL_SetRxFifoThreshold() to disable threshold events from occurring when passed an rxThreshold of the FIFO's size or larger. A 2.5.0 change allowed such rxThreshold settings to cause the event to fire when the buffer became full, though it's been mis-documented as one byte away from becoming full for quite a while – something that was never actually implemented and is now removed from the documentation.
- Fixed RAIL_StartCcaCsmaTx() and RAIL_StartCcaLbtTx() to return RAIL_STATUS_INVALID_PARAMETER when passed invalid configuration parameters in their respective RAIL_CsmaConfig_t or RAIL_LbtConfig_t, whose documentation has been updated accordingly. In earlier releases the requested operation would proceed with an invalid or incomplete hardware configuration, resulting in undefined behavior.

2.5.1

- Worked around a BLE longrange problem where Rx abort could cause deafness.

2.5.0

- Added a channel hopping feature with RAIL_ConfigRxChannelHopping and RAIL_EnableRxChannelHopping. This feature allows users to specify a list of channels to scan over during receive. This feature allows for much faster scanning with less delay (about 100us) than simply repeatedly calling RAIL_StartRx() with different channels.
- Included two default transmit power curves in the library: RAIL_TxPowerCurvesDcdc and RAIL_TxPowerCurvesVbat. When the pa-conversions plugin is used, one of these can be passed to the plugin, based on the way the power amplifier is powered.
- Added the ability to specify a Link Quality Indicator (LQI) conversion callback with RAIL_ConvertLqi such that the hardware's LQI value can be changed before application consumption of that value.
- Fixed an issue where RAIL_StopTxStream() was taking an extended amount of time to complete.
- Certain received packet details in RAIL_RxPacketDetails_t, which were formerly available only for successfully received packets, are now also available for unsuccessfully received packets.
- RAIL_DataMethod_t and related documentation have been updated to reflect that many FIFO_MODE concepts can now be used in PACKET_MODE, and the distinction between these modes has become nearly invisible for transmit. Also eliminated the previous restriction that both receive and transmit RAIL_DataMethod_t must match in the RAIL_DataConfig_t passed to RAIL_ConfigData().
- RAIL_ConfigData() no longer resets the receive FIFO when switching to RAIL_DataMethod_t::FIFO_MODE.
- Added a new RAIL_TxOptions_t option RAIL_TX_OPTION_CCA_PEAK_RSSI to allow the user to use the peak RSSI instead of the average RSSI over the CCA period during CSMA and LBT transmits. This is only supported on EFR32xG12 and newer silicon.
- Added a new RAIL_TxOptions_t option RAIL_TX_OPTION_ALT_PREAMBLE_LEN to allow the user to transmit with a runtime-specified preamble length. This alternate preamble length is set with RAIL_SetTxAltPreambleLength.
- Added Z-Wave PHY configurations and functionality to the RAIL library.
- Fixed an issue that could cause register corruption in very rare conditions due to a bus access race condition. The specific registers impacted were limited to those related to the radio and the corruption was most likely when running the coherent IEEE802.15.4 PHY on the EFR32xG12 and EFR32xG13 platforms. On other platforms and PHY configurations the issue is possible though very unlikely and never observed (id: 317234).
- Resolved a multiprotocol scheduler issue that could cause some interrupted events to not be reported to the caller. Since these events never run there are no other related events either so state machines relying on RAIL events could get stuck in invalid states (id: 341124).
- Enhanced RAIL_SetRxFifoThreshold() and RAIL_SetTxFifoThreshold() to now support a threshold value of RAIL_FIFO_THRESHOLD_DISABLED to explicitly disable a threshold previously established.
- Fixed a documentation issue where a variety of RAIL APIs were incorrectly shown as "Referenced by RAIL_CopyRxPacket()".

2.4.1

- Fixed an issue with RAIL_Sleep() that could cause the wake event to be set very far in the future. This would prevent waking at the proper time and cause problems for the application.
- Fixed an issue in the multiprotocol library where the txToRxSearch time set by RAIL_SetStateTiming() would be lost after a protocol switch if there were no timings stored in the channel configuration.
- Fixed an issue preventing radio coexistence transmit abort counter from incrementing.
- Fixed radio coexistence HAL_CONFIG support for shared priority and shared request.

2.4.0

- Added ability to set custom receive FIFO sizes via the RAIL_SetRxFifo() function. The new RAILCb_SetupRxFifo() callback may be used to override the default FIFO initialization to save RAM.
- Added alternate functions to get packet details, in a way that helps lower code size: RAIL_GetRxPacketDetailsAlt() and RAIL_GetTxPacketDetailsAlt(). These come with functions to adjust the returned time stamp to different places in the packet: RAIL_GetRxTimePreambleStart(), RAIL_GetRxTimeSyncWordEnd(), RAIL_GetRxTimeFrameEnd(), RAIL_GetTxTimePreambleStart(), RAIL_GetTxTimeSyncWordEnd(), and RAIL_GetTxTimeFrameEnd().
- Broke up the IR calibration code to save code size for Zigbee and BLE where the results are stored on chip.

- Added a new RAIL_SetRssiOffset() command to offset all RSSIs used by RAIL to match any hardware offsets on a given board.
- Added a new RAIL_StopTx() function to stop only pending or current transmits without impacting receive.
- For multiprotocol, added the ability to update a task's priority after it started with RAIL_SetTaskPriority().
- Added the ability to pre-apply a radio configuration to hardware before a TX or RX operation begins via RAIL_PrepareChannel().
- Added the ability to query RAIL for the channel configuration currently configured for use via RAIL_GetChannel().
- Fixed an issue where relative tasks with a non-zero slip time could start a few hundred microseconds earlier than they were supposed to.
- Fixed an issue that could cause antenna diversity to not function correctly.
- Fixed an issue where RAIL_Idle() with RAIL_IDLE_FORCE_SHUTDOWN could cause an erroneous RAIL_EVENT_TX_ABORTED event to fire.

2.3.1

- Fixed one case where calling RAIL_Idle() with the mode set to RAIL_IDLE_FORCE_SHUTDOWN could cause us to output partial packet trace appended information.
- Deferred multiprotocol event failures until just before the event must start when the expected runtime of a task causes it to not fit. These were previously reported immediately even though future tasks could change.

2.3.0

- Added a RAIL_CopyRxPacket() helper for use after RAIL_GetRxPacketInfo().
- Added a fast, but inconsistent, transition times mode. This can be activated by setting state timings to 0.
- Resolved a potential receive FIFO corruption that could occur when calling RAIL_HoldRxPacket() or RAIL_PeekRxPacket().
- A multi-timer API was added that allows for multiple software timers all driven off the one hardware timer. This must be explicitly enabled and will use more code than the standard APIs, but can be done through RAIL_ConfigMultiTimer(). Once in use, all timer APIs must use the multi timer versions to prevent conflicts with the hardware timer.
- Added diagnostic functions RAIL_Verify() and RAIL_ConfigVerification() to provide a means of verifying internal radio memory contents.
- Added alternative calibration APIs to RAIL_Calibrate, for potential code size savings: RAIL_CalibrateTemp, RAIL_CalibrateIr, and RAIL_ApplyIrCalibration.
- Fixed an issue where requesting a small fixed CSMA backoff could result in a very large value from being selected instead.
- Made PHY configuration improvements for BLE and Zigbee on the efr32xg14 platform.

2.2.0

- In FIFO mode on the EFR32 the FIFO could only be read after 8 bytes were received. This restriction has now been lifted though the user must be careful and understand that in certain cases they could be reading appended information and not packet bytes.
- On the EFR32XG14 some BLE packets showed up with invalid timestamps. These are now filtered and reported as aborted to the user so that the stack does not time sync to an invalid timestamp and lose the connection.
- Fixed an issue where state transition times would not be recomputed when changing the PA ramp time on the fly.
- Fixed a regression in 2.1.1 that caused Frame Type based length configurations to lock up the receiver. There was an issue where bits in the byte that were not supposed to be a part of the frame type were being included some times and could cause problems in the receive processing logic.
- Prevented RAIL_StartRx() from resetting the receive FIFO. This was a legacy feature that was not documented and, though it generally worked, would cause problems to users processing deferred receive events or users that called RAIL_StartRx() in an interrupt disabled context. If this behavior was required it can be replicated by idling the radio and then calling RAIL_ResetFifo() with rxFifo set to true before any calls to RAIL_StartRx().
- Fixed an issue in the RAIL_IEEE802154_SetAddresses() function that caused it to never set any long address except for the first one and to set that one to the value given in the final long address entry.
- Updated the documentation for the IEEE 802.15.4 and BLE protocol functions to show properly usage with the new RAIL 2.0 API.
- Dynamic Multiprotocol fixes:
  - Added support for each protocol having auto ACK enabled by allocating space for independent ACK buffers.

- Ensured that the sync word is properly output in BLE Long Range no matter when the packet data is loaded into the transmit FIFO.
- Prevented the sync word in packet trace feature from being turned on in Zigbee when it's enabled by BLE.

2.1.2

- Removed unimplemented functions RAIL_GetActiveChannelConfig() and RAIL_GetActiveChannelConfigEntry().

2.1.1

- RAIL_PauseTxAutoAck() was internally pausing the Rx AutoAck logic. This has now been corrected.
- EFR32XG12 and EFR32XG13 devices were detecting a high number of false 15.4 sync words. Even though these packets were eventually filtered they would show up on PTI. In 2.1.1 there should be significantly fewer of these packets as some of this was caused by a bug in a configuration algorithm.
- Channel power limits for BLE PHYs were not always being respected. This could have caused us to use a power which might result in certification issues.
- The RAIL_StateBuffer_t type was incorrectly sized resulting in a waste of RAM for RAIL applications. This has now been trimmed down to the actual size required.

2.1.0

- Packet receive APIs were totally revamped to allow faster and out-of-order deferred processing of received packets and their data outside of interrupt/callback context. Gone are RAIL_GetRxPacketHandle(), RAIL_ReadRxFifoAppendedInfo(), and the memory management RAILCb_ callbacks, replaced with RAIL_GetRxPacketInfo() that provides pointers directly to the received data held in RAIL's internal receive FIFO buffer allowing for zero-copy operation. See other RAIL 2.x porting documentation and new routines RAIL_GetRxPacketInfo(), RAIL_GetRxPacketDetails(), RAIL_HoldRxPacket(), and RAIL_ReleaseRxPacket(), along with updated RAIL_PeekRxPacket() and RAIL_ReadRxFifo() behavior.
- On the transmit side, zero-copy is also now supported by allowing applications to specify their own buffers for the transmit FIFO, eliminating RAIL's former internal fixed-size 512-byte one. See new RAIL_SetTxFifo().
- The former RAIL_LoadTxData() has been merged into an updated RAIL_WriteTxFifo() API.
- The RAIL_Event_t type was renamed RAIL_Events_t now that multiple events can be presented to the application simultaneously in RAIL's generic events callback, rather than relying on the internal arbitrary order that RAIL imposed previously, which may not match the application's or protocol's expectations.
- Event/callback configuration is now managed entirely by RAIL_ConfigEvents(), eliminating the old RAIL_ConfigTx() and RAIL_SetCallbackConfig() APIs.
- Channel numbers were increased in range from 8 bits to 16 bits, and channel configurations made much more flexible allowing for multi-band radio configurations and an easy way to specify TX power limits on certain channels that otherwise share a common configuration. See new APIs RAIL_GetActiveChannelConfig() and RAIL_GetActiveChannelConfigEntry() along with changes to RAIL_ConfigChannels().
- Power levels were given a formal RAIL_TxPowerLevel_t type, and RAIL_SetTxPower() now returns a RAIL_Status_t not the power level actually achieved – must now use RAIL_GetTxPower() to get that.
- PA power curves and the conversion between RAIL power level and dBm have been fully abstracted to live outside of RAIL, allowing customers complete flexibility to characterize their designs and PA selection. See new APIs RAIL_ConfigTxPower(), RAIL_ConvertDbmToRaw(), RAIL_ConvertRawToDbm(), RAIL_GetTxPowerConfig(), RAIL_GetTxPowerDbm(), RAIL_SetTxPowerDbm(), and RAIL_EnablePaCal().
- Protocols using Frame Types to determine packet length are better integrated into the Studio radio configurator and channel configuration. Gone are APIs like RAIL_ConfigFrameTypeLength() and RAIL_EnableAddressFilterByFrameType() whose functionality is now subsumed within channel configurations.
- APIs to support RAIL timebase synchronization across sleep/wake were added: RAIL_ConfigSleep(), RAIL_Sleep(), and RAIL_Wake().
- State transitions for success and error are now encapsulated in a new RAIL_StateTransitions_t structure. This affected the RAIL_SetRxTransitions() and RAIL_SetTxTransitions() routines.
- The former RAIL_EVENT_TX_BUFFER_UNDERFLOW has been broken out into non-ACK (RAIL_EVENT_TX_UNDERFLOW) and ACK (RAIL_EVENT_TXACK_UNDERFLOW) flavors like the other transmit completion events.
- Diagnostic toning APIs were merged into the streaming APIs as the RAIL_StreamMode_t::RAIL_STREAM_CARRIER_WAVE.
- RAIL_ConfigCal() was revamped to use RAIL_CalMask_t.
- RAIL_Calibrate()'s bool calSave argument was removed, being redundant with a NULL vs. non-NULL calValues argument.

- Support for Packet Traffic Arbitration (PTA) or Coexistence was added in the form of an API to prevent transmits from going on the air: RAIL_EnableTxHoldOff(), along with RAIL_IsTxHoldOffEnabled() to check the current state.
- The former RAIL_AutoAckDisable() was merged into RAIL_ConfigAutoAck() by means of its RAIL_AutoAckConfig_t::enable parameter field.
- The former RAIL_EnableRxFifoThreshold() was replaced by using RAIL_ConfigEvents() to enable/disable the RAIL_EVENT_RX_FIFO_ALMOST_FULL event.
- The former blocking RAIL_PollAverageRssi() was eliminated; use the non-blocking RAIL_StartAverageRssi() instead.
- Several APIs were renamed for consistency and updated accordingly:
  - former RAIL_GetTxPacketInfo() is now RAIL_GetTxPacketDetails();
  - former RAIL_UseTxBufferForAutoAck() is now RAIL_UseTxFifoForAutoAck();
  - former RAIL_LoadAutoAckBuffer() is now RAIL_WriteAutoAckFifo().
- More API functions were updated to take a RAIL_Handle_t argument: RAIL_SetFreqOffset(), RAIL_GetRxFreqOffset(), and the RAIL_IEEE802154_ protocol APIs.
- RAIL_IEEE802154_GetAddress() was updated to return RAIL_Status_t rather than RAIL_IEEE802154_Address_t which is now passed as an output argument.
- RAIL_EnableDirectMode() and RAIL_SetTune() now return RAIL_Status_t rather than void.
- A multiprotocol scheduler API was added to get its status: RAIL_GetSchedulerStatus().
- The RAILCb_AssertFailed() callback returns with a RAIL_Handle_t parameter that might be NULL since assert conditions might occur during initialization prior to handle assignment or during multiprotocol switching.
- Many APIs with pointer arguments were clarified to use const pointers where the API doesn't modify the data pointed to.

2.0.0

- All API names were updated to follow a strictly VerbNoun convention.
- RAIL_SetAbortScheduledTxDuringRx() is no longer a separate API, rather it was added to RAIL_ScheduleTxConfig_t. Set the desired TX during RX behavior when that struct is passed to RAIL_StartScheduledTx().
- Add support for building single-prototocol, multi-protocol, and core RAIL libraries.
- Changed RAIL_RadioState_t to a bitmask. Supported RAIL_RadioState_t bits are: RAIL_RF_STATE_ACTIVE, RAIL_RF_STATE_RX, and RAIL_RF_STATE_TX; combinations of these bits yield the additional states RAIL_RF_STATE_INACTIVE, RAIL_RF_STATE_IDLE, RAIL_RF_STATE_RX_ACTIVE, and RAIL_RF_STATE_TX_ACTIVE.
- Added RAIL_RX_OPTION_TRACK_ABORTED_FRAMES to configure seeing full packets via Packet Trace, even after they have been aborted.
- Added RAIL 15.4 PTI Rx error code UNDESIRED_ACK to indicate an ACK filtered because it did not have the desired sequence number matching that in the recently-transmitted ACK-requesting frame.
- Updated return type from uint8_t to RAIL_Status_t for many functions.
- Removed deprecated bit error-rate APIs RAIL_BerConfigSet(), RAIL_BerRxStart(), RAIL_BerRxStop(), RAIL_BerStatusGet(), along with deprecated types RAIL_BerConfig_t and RAIL_BerStatus_t.
- TX related events are now broken out to specify whether the event is relevant to an ACK packet or user-initiated TX. Specifically those are RAIL_EVENT_TX_ABORTED and RAIL_EVENT_TXACK_ABORTED, RAIL_EVENT_TX_BLOCKED and RAIL_EVENT_TXACK_BLOCKED, and RAIL_EVENT_TX_PACKET_SENT and RAIL_EVENT_TXACK_PACKET_SENT. One of each type (i.e. one ACK, one non-ACK) can occur before being handled.
- RAIL_StartTx() no longer takes a pre-TX operation function pointer to do scheduled or CCA transmits. Instead, there are now separate API's for each type of transmit: RAIL_StartTx(), RAIL_StartScheduledTx(), RAIL_StartCcaLbtTx(), RAIL_StartCcaCsmaTx().
- RAIL_TxOptions_t has been converted to a 32-bit bitmask, so that RX and TX options are both represented as bitmasks. A single TX option mask is passed into each TX call.
- RAIL_ConfigEvents() now also takes two bitmasks, one to indicate which events should be affected, and the second indicating which of those events should be enabled.
- Similarly, RAIL_ConfigRxOptions() takes two bitmasks, one to indicate which options to update, and the second indicating the desired value of those options, which affect all future receives.
- Options for removing RX appended info and ignoring CRC are now controlled exclusively by RAIL_ConfigRxOptions().

1.6.1

- Added new BLE radio configurations for EFR32xG13 parts to fix a transmit overshoot issue on the coded PHY configurations for BLE Long Range.
- Fixed DCDC voltage droop during RAIL_RfInit()
- Added an API (RAIL_EnablePaCal()) that enables loading of chip to chip PA calibration data.

1.6.0

- Fixed an issue where RAIL_RfStateGet() would mistakenly return RAIL_RF_STATE_IDLE instead of RAIL_RF_STATE_TX when a Transmit operation's LBT is active.
- Added an API (RAIL_SetAbortScheduledTxDuringRx) that sets the behavior when a scheduled TX fires during an RX or ACK. If the API is passed false, TX's will be delayed until the RX (or ACK to the RX) is complete. That behavior is the default and what always happened previously. Passing the function true however will abort scheduled TX's that fire during the RX (or ACK to the RX) and fire the RAILCb_TxRadioStatus callback.
- Added new RAILCb_AssertFailed callback to give the flexible handling of asserts within RAIL. This includes defines of assert codes and error reasons in rail_assert_error_codes.h. The default implementation hangs in an infinite loop.
- Added RAIL_ENUM define to override enum size ambuiguity in the ARM EABI by making them actually a uint8_t. For documentation purposes they are still shown as enums.
- Added new TX options: removeCrc, to override whether CRC is sent; and syncWordId, to choose which SYNC word to transmit when multiple SYNC words are configured.
- Added new RX option: RAIL_RX_OPTION_ENABLE_DUALSYNC, which allows multiple SYNC words to be searched for on PHYs that support it.
- Added RAIL_PeekRxPacket to allow reading part of a packet before it has been fully received.
- Changed BER Test functionality in RAILTest such that if a RX overflow is detected (due to the incoming datarate being too high), the test now aborts and displays an appropriate status message.
- Fixed an issue where the code could hang in a RX overflow state when receiving incoming data streams.
- RAIL_TxToneStart() and RAIL_TxStreamStart() now idle the radio before transmitting.
- Fixed an issue where calling RAIL_TimerSet() while in the middle of a stream or tone transmission could lock up the radio.
- Added new frequency offset related functions, RAIL_GetRxFreqOffset and RAIL_SetFreqOffset, and the associated RAIL_FrequencyOffset_t type definition.

1.5.2

- Fixed an issue in 1.5.1 that would cause scheduled receive, scheduled transmit, and RAIL timer operations to fail on EFR32xG12 with a relative delay of 0.
- Fixed an issue in 1.5.1 that would cause scheduled receive and scheduled transmit to fail if called with an absolute time from [0, rxWarmTime) or [0, txWarmTime) respectively.
- Fixed an issue where a failed scheduled transmit operation would prevent any transmit from succeeding until RAIL_RfIdle() or RAIL_RfIdleExt() were called.
- Fixed an issue in 1.5.1 where calling idle while looking for an ACK could cause the radio to get stuck in the receive state.

1.5.1

- Added official support for the EFR32xG12 parts through a new librail_efr32xg12.a version of the library.
- Added RAIL_SetRxOptions API for configuration of receive options, such as RAIL_RX_OPTION_STORE_CRC.
- Increase transmit and receive fifo sizes to 512 bytes for all EFR32xG1 parts. This impacts both packet and fifo modes.
- Added RAIL_RxDataSource_t::RX_DEMOD_DATA in order to receive demodulated data via FIFO method.
- Shortened the duration that the transmit buffer lock in RAIL_TxDataLoad() is held, as it was overprotective. RAIL will still report an error if the application attempts to manipulate the transmit buffer while actively transmitting data.
- Improved state transition timings for enhanced precision and accuracy. To improve precision we have switched to a more stable receive complete event since our previous trigger could jitter for certain radio configurations. To improve accuracy we measured and removed PHY specific overhead for BLE applications. Note that this measurement was BLE specific and does **not** completely correct accuracy on any arbitrary PHY configuration. For custom PHY configurations you will still have to manually calibrate out any offsets by measurement and any previous measurements are likely different now.
- Moved BER test implementation out of the RAIL library and into RAILTest. Deprecated the existing BER test functions in the RAIL API. With BER test code now existing in RAILTest, you must use version 2.2.1 or later of the radio configurator.
- Disabled receive buffer overflow and transmit buffer underflow as BUFC_IRQHandler() events. These events now only occur in FRC_IRQHandler().

- Sped up RAIL transmit and receive buffer read/write operations.
- Fixed a race condition that could cause the receive packet callback to be delayed until the next receive packet event.

1.5.0

- Added other methods of Data Management. The application can configure RAIL to provide/retrieve data via FIFO method.
- The new FIFO method of data management brings with it two new callbacks: RAILCb_TxFifoAlmostEmpty(), and RAILCb_RxFifoAlmostFull(). Note that you must implement these new callbacks or stub them out if you do not intend to use them.

1.4.2

- Fixed a bug in how we handled IEEE 802.15.4 data request packets. Specifically, we were not issuing the RAILCb_IEEE802154_DataRequestCommand() callback for data request packets that were sent with MAC security enabled.
- Fixed a bug where calling RAIL_TxDataLoad() at an inopportune time could cause that and all subsequent such calls to fail.

1.4.1

- Implemented LQI measurement. The field in RAIL_AppendedInfo_t is now populated on every packet receive.
- Allow transmit while receive is active, if the channel is the same. After transmitting, the radio will follow the configured state transitions as before.
- Allow shorter ccaBackoff times if receive is active during the CSMA or LBT process.
- Allow calling RAIL_RxConfig while receiving.
- Added new receive event: RAIL_RX_CONFIG_PACKET_ABORTED
- Added new RAIL_TxConfig API, along with more transmit events: RAIL_TX_CONFIG_CHANNEL_CLEAR, RAIL_TX_CONFIG_CCA_RETRY, RAIL_TX_CONFIG_START_CCA
- Fixed a filtering bug in the IEEE 802.15.4 code. We were incorrectly filtering packets with a 0xFFFF source PANID while the coordinator which caused problems in the joining process. The new logic will only use the source PANID for filtering if we are the coordinator, there is no destination address, and the frame type is MAC command or Data.
- Migrated to Gecko SDK 5.0, including the new em_core APIs for interrupt enabling and disabling. This adds required dependencies on the CORE_EnterCritical() and CORE_ExitCritical() functions.
- Added new RSSI functions for hardware and software averaging: RAIL_StartAverageRSSI(), RAIL_GetAverageRSSI(), RAIL_AverageRSSIReady(), RAIL_PollAverageRSSI().
- Added new RSSI callback: RAILCb_RssiAverageDone().
- Added the ability to switch between multiple BLE phys, including 2Mbps phys, on chips that support the new phys.

1.4.0

- Added support for the EFR32MG12P part family with larger RAM and Flash size variants along with some new chip features.
- Renamed anything that was just efr32 in the previous releases to efr32xg1 to differentiate it from the new efr32xg12 family of parts.
- Add support for the new BLE 2MBit PHY to the BLE acceleration and configuration APIs on EFR32xG12 chips.
- Better optimize IR Calibration run times for all EFR32xG1 parts. You must use version 0.69 or later of the radio configurator.

1.3.2

- Added RAIL_SetPtiProtocol() and RAIL_PtiProtocol_t. This function allows the application to configure RAIL's packet trace interface to output protocol specific data.
- Fix BER functions to operate at 1Mbps data rates.
- RAIL_RfIdle() does not clear the transmit buffer anymore. RAIL_TxDataLoad() resets the transmit buffer with every packet set. This fixes the case where RAIL_RfIdle() is called after RAIL_TxDataLoad().
- Fixed bug when using fixed packet lengths configured via the Radio Configurator. RAIL_TxStart() will not overwrite the configured fixed length to the amount of data loaded by RAIL_TxDataLoad().
- Fixed bug in RADIO_PA_Init() where ramp times greater than 11.2 ms would be truncated to a very short ramp.

1.3.1

- Added IEEE 802.15.4 hardware acceleration. The new APIs can be found in protocol/ieee802154/rail_ieee802154.h. Documentation for these APIs can be found in IEEE 802.15.4. This includes a new callback: RAILCb_IEEE802154_DataRequestCommand().
- Added Auto Ack hardware acceleration. Documentation for these APIs can be found in Auto-ACK.
- Added RAIL_TxStartWithOptions(). This new API takes a pointer to RAIL_TxOptions_t that contains options to modify the upcoming transmit.
- Increased minimum state transition time to 100 us.
- Added RAIL_STATUS_INVALID_CALL
- Added RAIL_AppendedInfo_t::isAck

1.3.0

- Added BLE hardware acceleration. The new APIs can be found in protocol/ble/rail_ble.h. Documentation for these APIs can be found in BLE.
- Added RAIL_ScheduleRx() and RAIL_ScheduleRxConfig_t. This allows the application to schedule a RX window using the RAIL timebase.
- Added RAIL_TIME_DISABLED enum to RAIL_TimeMode_t.
- Added an extended idle function which gives you more control of how the radio is idled. See RAIL_RfIdleExt() and RAIL_RfIdleMode_t for more details.
- Added RAIL_RX_CONFIG_TIMEOUT and RAIL_RX_CONFIG_SCHEDULED_RX_END
- Added RAIL_TX_CONFIG_TX_ABORTED and RAIL_TX_CONFIG_TX_BLOCKED
- Added an extended radio status callback that supports up to 32 different status reasons (RAILCb_RxRadioStatusExt()). You may implement this instead of RAILCb_RxRadioStatus() or continue using the old version if you do not need access to more than the first 8 status values.

1.2.7

- Added FEM control signals
- Added RAIL_SetPaCTune in order to tune the PA capacitor value for TX and RX.
- Fix typo in RAIL_StreamMode_t : PSUEDO_RANDOM_STREAM to PSEUDO_RANDOM_STREAM

1.2.6

- Added ability to reset demod.

1.2.5

- Fixed register settings for BER testing.
- Fixed packet buffer for contents larger than 170 bytes.

1.2.4

- Presented worst case RfSense wake period

1.2.3

- Optimized address filtering code for operation at higher data rates.
- Clarified data rate limitations of address filtering.

1.2.2

- Added RAIL_SetTime() in order to allow the user to change the RAIL timebase with microsecond granularity.
- Removed spurious calls to RAIL_RxRadioStatus() with the RAIL_RX_CONFIG_BUFFER_OVERFLOW argument, which were happening when receive was aborted due to events such as address filtering failures.
- Fixed incorrect PA output power calculations.

1.2.1

- Allow RAIL_RxStart() and RAIL_TxStart() to not error if the radio is heading to idle after a call to RAIL_RfIdle(). Application code does not have to wait until the radio is completely idle before calling a receive or transmit operation as long as the radio is heading to idle.

1.2.0

- Fixed IR calibration such that calibration values are more accurate for radio configurations with datarates below 10Kbps.
- Fixed IR calibration such that the default calibration value is applied when a better calibration value is unable to be calculated.
- Added RAIL_CalInit() for calibration initialization.
- Added BER test API and structures for diagnostic use - RAIL_BerConfigSet(), RAIL_BerRxStart(), RAIL_BerRxStop(), RAIL_BerStatusGet(), RAIL_BerConfig_t, RAIL_BerStatus_t.
- Enabled DC Calibration during initialization.

1.1.1

- Fixed CCA timing to prevent invalid CCA failures when starting out of IDLE state.
- Fixed RX state transitions to TX when ignoring CRC errors. Will properly take the success route when a CRC error occurs while ignoring CRC errors.
- Fixed TX error state transition after CCA failure.

1.1.0

- RAIL_RX_CONFIG_INVALID_CRC changed to RAIL_RX_CONFIG_FRAME_ERROR
  - This now accurately represents the callback flag.
  - A frame error is either an invalid crc, a frame coding error, or a variable length error. On detection of a frame error, the radio will proceed to the defined error state.
- RAIL_SetRxTransitions() and RAIL_SetTxTransitions() APIs added
  - Add ability to configure radio to automatically to transition to a given radio state on completion of a receive or transmit event.
- RAIL_SetStateTiming() API added
  - State transition times can be configured. The given timestamps will not perfectly match on-air times, due to the RX and TX chain delays, which are dependent on the radio configuration.
- RAIL_AddressFilterByFrameType() added
  - If address filtering and frame type length decoding are both enabled, this gives the ability to enable or disable address filtering per frame type

1.0.0

- Return quarter dBm resolution from RAIL_RxGetRSSI(). This changes the return value to an int16_t from an int8_t.
- Added calibration APIs (Calibration)
  - The RAIL library will notify the application via callback, RAILCb_CalNeeded(), whenever it detects a calibration is needed.
  - An API is provided allowing the application to choose when to calibrate the radio, to pass in a known calibration value, or force a calibration to occur.
- RAIL_TxPowerSet() returns the power level that can be set which might not match the requested power level.

Beta 1 - November 16, 2015

- Exposed EFR32 RF Energy Sensing capability via a RAIL API.
  - This functionality allows the radio to be woken from sleep by a large amount of energy over the air.
- Added new RAIL Timer APIs (System Timing)
  - This provides access to a hardware timer running at the RAIL timebase. You can use this to implement simple protocol timing that may be needed by your low level radio code.
  - This change also adds a new RAILCb_TimerExpired() that must be implemented by your application.
- Added APIs to access the bit rate and symbol rate for the current PHY (RAIL_BitRateGet(), RAIL_SymbolRateGet()).
- Cleaned up RAIL type names so that they all begin with the RAIL_ prefix. This will require updates to existing application code.
- Initial integration with Simplicity Studio's application builder tools

Alpha 2 - September 14, 2015

- Filtering of receive packets based on defined address fields and compare values. See Address Filtering for more details.
- Added different Pre-Transmit options for the Tx function.
  - Scheduling of packet transmission at an absolute time or after a defined delay.

    - Packet transmission conditional on Clear Channel Assessment (CSMA, LBT).
- The ability to read RSSI manually with RAIL_RxGetRSSI().
- Support for building RAIL applications using both GCC and IAR toolchains.

Alpha 1

- Initial release with support for transmit receive and basic radio configuration.

RAILtest Changelist

RAILtest Changelist

RAIL 2.16.x:

- **1175616:** Added support for RAILTEST on the MGM240x modules.
- **1185980:** Added support for channel masks during Wi-SUN mode switch in RAILTEST application on the EFR32xG25.

RAIL 2.15.x:

- **1103500:** When printRxErrors is in effect, erroneous packets now show RSSI and any erroneous payload received when Rx FIFO mode is used.
- **1105873:** RAILtest now displays board name (if known) on bootup.

RAIL 2.13.x:

- **825100:** Added a clearer error message when setRssiOffset is used and the radio is not idle.
- **828171:** Added support for all types of line endings to the sweepTxPower command to prevent situations where advancing to the next power level could trigger twice in a row.
- **845292:** Added resetCause (in hex) to railtest's bootup display.

RAIL 2.12.x:

- **721006:** Any pin on ports A, B, C, or D can now be selected to use as a debug signal in railtest using the setDebugSignal command.

RAIL 2.11.x:

- **679427:** Altered RAILtest to allow for more dependencies to be removed - antenna diversity, RF path, CLI, CLI RAM storage, CLI delay.
- **689827:** Fixed a hanging issue caused by Rx channel hopping being enabled without a successful Rx channel hopping configuration. This was previously a recommended practice but is now required.
- **700831:** RAILtest sleep command again allows distinguishing energy modes 4h and 4s.
- **712345:** Added additional bootup messages to RAILtest showing radio, system, and PTI configuration information.
- **737886:** Fixed the help text for the setTimings command which had the order swapped for the rxToTx and txToRx time parameters.

RAIL 2.10.x:

- **605067:** The SL_RAIL_TEST_PER warning has been removed from RAILtest when a PER pin is not configured for use.

RAIL 2.9.x:

- **381485:** Added `configChannelHoppingMulti` command for pre-configuring channel hopping multi-sense mode parameters prior to issuing `configRxChannelHopping` command. Also augmented `configRxDutyCycle` command to allow parameterization of this mode.
- **402203:** Augmented `configChannelHoppingOptions` command to allow specifying RSSI threshold when enabling that option. Augmented `configRxDutyCycle` command to allow specifying options in general including the RSSI threshold.
- **427841:** Updated the Command Line Interface (CLI) help display format.
- **488425:** CLI commands are now displayed back to the user using the same capitalization as they are entered (e.g. entering command `configTXoptions` will display `configTXoptions` and not `configTxOptions`).
- **454890:** Restructured RAILtest to use a common main.c template with a software component-based architecture to select software features.
- **487098:** Disabled graphics, buttons, LEDs and PER test GPIO pin by default.
- **484333:** Standardized spelling of `RAILtest` for consistency.

- Updated bootup message example: `{{(reset)}{App:RAILtest}{Built:Jul 19 2019 13:36:24}}`
- **633846:** Altered CLI input of command `setDebugSignal`.
- **630457:** Fixed an issue where the RAIL PTI component would automatically reserve hardware pins when no pins were specified for use.

RAIL 2.8.x:

- **344361:** Added new command `getCteSampleRate` to retrieve the actual ADC sample rate used to capture the CTE.
- **369727:** Added new commands `setTxTimePos` and `setRxTimePos` to designate the time position of displayed packet timestamps among the RAIL_PacketTimePosition_t choices. In txEnd messages, now also display the lastTxStart time, and fixed a bug with the lastTxStatus display.
- **406293:** Fixed an issue where a memory pointer could be discarded from the event queue on queue overflow and never released. This can occur if the memory buffer pool is greater than the event queue size.
- **415017:** Updated the RAILTest bootup message to incorporate the `reset` command, the application name and the date and time of the RAILTest build.
  - Updated bootup message example: `{{(reset)}{App:RAIL Test App}{Built:Jul 19 2019 13:36:24}}`
- **406080:** Added new commands `configRfSenseWakeupPhy` and `setRfSenseTxPayload` to support waking up chips that support RFSENSE Selective(OOK) mode. Updated `sleep` and `rfsense` commands to allow setting the syncword length and syncword for EFR32XG22 chips which support RFSENSE Selective(OOK) mode. The `tx` command will always overwrite the TX FIFO with a default payload, hence `setRfSenseTxPayload` must be used with `fifoModeTestOptions` set to use `txFifoManual` to prevent the FIFO being overwritten.
- **460516:** Fixed minor railtest problem where `setConfigIndex` might not restart receiver listening like `setChannel` would if the receiver was on when the command was issued.

RAIL 2.7.x:

- **368510:** Fixed an issue if receive was entered because of a transmit state transition where we would not change the channel when calling setChannel and would ignore calls to `rx 1` to enter normal receive mode.
- **394431:** Fixed an issue where the configured antenna diversity settings in HAL config would not be applied to the radio at startup.

RAIL 2.6.x:

- Fixed `setMultiTimer` which was ignoring its mode argument; `abs` mode now works.
- The commands `setRxOptions` and `configTxOptions` can now be called without any parameters to print the current state of these options.
- RAILTest now uses its config-changed callback for both external and internal radio configs to ensure that the appropriate PA level is always selected.
- Added 802.15.4 `setDataReqLatency` command for testing purposes.
- Fixed some asynchronous event print outs related to Z-Wave beam frame reception since they could corrupt the output.
- Added `getChannelHoppingRssi` command to read the RSSI when in channel hopping mode on each available channel.

RAIL 2.5.x:

- Added the following multitimer commands.
  - `enableMultiTimer`, `setMultiTimer`, `multiTimerCancel`, `printMultiTimerStats`
- Added `printRxErrors` command to enable/disable printing of unsuccessfully received packets.
- Added the ability to save command scripts to flash and run them from flash. Renamed command `startScript` to `runScript`. Added command `clearScript`.

RAIL 2.4.x:

- Added the RAM Modem Reconfiguration commands. These commands allow the dynamic configuration of the modem via a set of studio scripted commands.

RAIL 2.0.x:

- `setChannel` no longer requires the radio be idle.

RAIL 1.6.x:

- simple `tx` command transmits with defaults (i.e. don't wait for ack, send CRC, use sync word 0)
- configTxOptions now controls 3 options
  - BIT0: Wait for ack
  - BIT1: Send CRC
  - BIT2: Sync Word ID
- setRxOptions now controls 3 options
  - BIT0: Store CRC
  - BIT1: Ignore CRC errors
  - BIT2: Enable dual sync

RAIL 1.5.x:

- Added the following commands:
  - `rxConfig`, `setRxOptions`, `setFixedLength`, `fifoStatus`, `dataConfig`, `setTxFifoThreshold`, `setRxFifoThreshold`, `iqCapture`

RAIL 1.4.1:

- Added the following commands:
  - `setPtiProtcol`, `sweepPower`, `startAvgRssi`, `getAvgRssi`, `setBle1Mbps`, `setBle2Mbps`

RAIL 1.3.1:

- Added Bluetooth Low Energy commands
  - `setBleMode`, `getBleMode`, `setBleChannelParams`, `setBleAdvertising`
- Fixed help text for `setPanId802154`, `setshortAddr802154`, `setLongAddr802154`
- Changed parameters for `acceptFrames` separating the bitmask into individual elements

RAIL 1.3.0:

- Added Auto Acknowledgement commands
  - `autoAckConfig`, `autoAckDisable`, `setAckPayload`, `setAckLength`, `printAckPacket`, `getAutoAck`, `autoAckPause`, `setTxAckOptions`
- Added IEEE 802.15.4 commands
  - `enable802154`, `config2p4GHz802154`, `acceptFrames`, `setPromiscuousMode`, `setPanCoordinator`, `setPanId802154`, `setShortAddr802154`, `setLongAddr802154`
- Added scheduled RX command `rxAt`
- Added `transmitWithOptions` and `configTxOptions` commands
- Added `setDebugSignal` command.

RAIL 1.2.1:

- Added `setPaCtune` which can configure custom PA capacitor tuning values for TX and RX
- Fixed RSSI slope requiring a 4dB power change for 1dB change in the received packet print

RAIL 1.2.0:

- Added bit error rate test commands - `setBerConfig`, `berRx`, and `berStatus`.
- Added packet error rate test commands - `perRx`, `perStatus`.
- Added the ability to set the RAIL timebase in microseconds - `setTime`.
- RxOverflow counter now only counts number of overflows, instead of mistakenly counting some non-overflow events.
- Removed the RfUs output parameter from the 'sleep' command due to it being incorrect.

RAIL 1.1 - May 15, 2016:

- Automatic state transitions and timings added via `setRxTransitions`, `setTxTransitions`, and `setTimings`.
- Ability to selectively enable the address filter on certain frame types added through `addressFilterByFrame` command.
- Independent status booleans were removed in favor of a single `AppMode` status variable, which keeps track of the state of RAILtest.
- More validation is done to ensure that certain RAIL commands are only run in the correct AppMode and radio state. The radio must be IDLE for `setChannel`, `freqOverride`, `setPower`, and `setCtune` to succeed.
- `setTestMode` was removed in favor of individual commands for `setTxUnderflow`, `setRxOverflow`, and `setCalibrations`
- `toggleTone` became `setTxTone`, and `toggleStream` became `setTxStream`.

- Removed command `getTicks`, since the RAIL_GetTime() function made the timer used unnecessary.
- Removed command `txTimeReport`, since it was exposing implemenation-specific behavior encapsulated by RAIL.

Beta 1 - November 16, 2015:

- New library features for RF Energy Sensing exposed via new `rfSense` and `sleep` commands.
- New RAIL timer functionality exposed via the `setTimer`, `timerCancel`, and `printTimerStats` commands.
- New `printDataRates` command to get information about the current PHY's data rates.
- Documentation clean up.

Alpha 2 - September 14, 2015:

- Commands for filtering of receive packets based on defined address fields and compare values.
- Scheduling of packet transmission at an absolute time or after a defined delay.
- Packet transmission conditional on Clear Channel Assessment (CSMA, LBT).
- The ability to read RSSI manually.
- Capability to switch between multiple PHY configurations at runtime.
- Revised help text (more uniform parameter descriptions).

API Changes

API Changes

RAIL Library 2.16.0

Added

New API Name	Return Value	Comment
RAILCb_IEEE802154_IsModeSwitchNewChannelValid(uint32_t currentBaseFreq, uint8_t newPhyModeId, const RAIL_ChannelConfigEntry_t *configEntryNewPhyModeId, uint16_t *pChannel)	RAIL_Status_t	A new API to check if the mode switch channel is valid
RAIL_SupportsCollisionDetection(RAIL_Handle_t railHandle)	bool	A new API to indicate if the chip supports Collision detection feature

RAIL Library 2.15.2

Added

New API Name	Return Value	Comment
RAIL_SupportsTxRepeatStartToStart(RAIL_Handle_t railHandle)	bool	See RAIL Changelist 1131398

RAIL Library 2.15.1

Added

New API Name	Return Value	Comment
RAIL_GetAutoAckFifo(RAIL_Handle_t railHandle, uint8_t **ackBuffer, uint16_t *ackBufferBytes)	RAIL_Status_t	See RAIL Changelist 748817
RAIL_Sidewalk_ConfigPhy2GFSK50kbps(RAIL_Handle_t railHandle)	RAIL_Status_t	Function to load the Sidewalk 2GFSK 50kbps PHY
RAIL_SupportsProtocolSidewalk(RAIL_Handle_t railHandle)	bool	API to determine at runtime when Sidewalk PHYs are supported

Modified

Old API Name	Return Value	New API Name	Return Value	Comment
RAIL_ConfigVerification(RAIL_Handle_t railHandle, RAIL_VerifyConfig_t *configVerify, const uint32_t *radioConfig, RAIL_VerifyCallbackPtr_t cb)	RAIL_Status_t	RAIL_ConfigVerification(RAIL_Handle_t railHandle, RAIL_VerifyConfig_t *configVerify, RAIL_RadioConfig_t radioConfig, RAIL_VerifyCallbackPtr_t cb)	RAIL_Status_t	

RAIL Library 2.15.0

Added

New API Name	Return Value	Comment
RAIL_EnableCacheSynthCal(RAIL_Handle_t railHandle, bool enable)	RAIL_Status_t	See RAIL Changelist item 1061665.
RAIL_GetSetEffClpcFemdata(RAIL_Handle_t railHandle, uint8_t *newMode, bool changeMode)	RAIL_Status_t	New alpha API.
RAIL_IEEE802154_SetRxToEnhAckTx(RAIL_Handle_t railHandle, RAIL_TransitionTime_t *pRxToEnhAckTx)	RAIL_Status_t	See RAIL Changelist item 1117127.
RAIL_SetTxFifoAlt(RAIL_Handle_t railHandle, uint8_t *addr, uint16_t startOffset, uint16_t initLength, uint16_t size)	uint16_t	See RAIL Changelist item 1103587.
RAIL_SupportsFastRx2Rx(RAIL_Handle_t railHandle)	bool	Added to help determine Fast RX2RX support at runtime.
RAIL_SupportsProtocolWiSUN(RAIL_Handle_t railHandle)	bool	Added to help determine WiSUN support at runtime.
RAIL_WMBUS_Config(RAIL_Handle_t railHandle, bool enableSimultaneousTCRx)	RAIL_Status_t	See RAIL Changelist item 1099829.
RAIL_WMBUS_SupportsSimultaneousTCRx(RAIL_Handle_t railHandle)	bool	See RAIL Changelist item 1099829.

Removed

Old API Name	Return Value	Comment
RAIL_GetSetEffMode(RAIL_Handle_t railHandle, uint8_t *newMode, bool changeMode)	RAIL_Status_t	Removed alpha API.

Modified

Old API Name	Return Value	New API Name	Return Va
RAIL_GetSetClpcEnable(RAIL_Handle_t railHandle, uint8_t *newClpcEnable, bool changeClpcEnable)	RAIL_Status_t	RAIL_GetSetEffClpcEnable(RAIL_Handle_t railHandle, uint8_t *newClpcEnable, bool changeClpcEnable)	RAIL_Stat
RAIL_GetSetClpcFastLoop(RAIL_Handle_t railHandle, RAIL_EffModeSensor_t modeSensorIndex, uint16_t *newTargetMv, uint16_t *newSlopeMvPerPaLevel, bool changeValues)	RAIL_Status_t	RAIL_GetSetEffClpcFastLoop(RAIL_Handle_t railHandle, RAIL_EffModeSensor_t modeSensorIndex, uint16_t *newTargetMv, uint16_t *newSlopeMvPerPaLevel, bool changeValues)	RAIL_Stat
RAIL_GetSetClpcFastLoopCal(RAIL_Handle_t railHandle, RAIL_EffModeSensor_t modeSensorIndex, RAIL_EffCalConfig_t *calibrationEntry, bool changeValues)	RAIL_Status_t	RAIL_GetSetEffClpcFastLoopCal(RAIL_Handle_t railHandle, RAIL_EffModeSensor_t modeSensorIndex, RAIL_EffCalConfig_t *calibrationEntry, bool changeValues)	RAIL_Stat
RAIL_GetSetClpcFastLoopCalSlp(RAIL_Handle_t railHandle, RAIL_EffModeSensor_t modeSensorIndex, int16_t *newSlope1e1MvPerDdbm, int16_t *newoffset290Ddbm, bool changeValues)	RAIL_Status_t	RAIL_GetSetEffClpcFastLoopCalSlp(RAIL_Handle_t railHandle, RAIL_EffModeSensor_t modeSensorIndex, int16_t *newSlope1e1MvPerDdbm, int16_t *newoffset290Ddbm, bool changeValues)	RAIL_Stat
RAIL_GetSetEffBypassDwellTimeMs(RAIL_Handle_t railHandle, uint32_t *newDwellTime, bool changeDwellTime)	RAIL_Status_t	RAIL_GetSetEffLnaBypassDwellTimeMs(RAIL_Handle_t railHandle, uint32_t *newDwellTime, bool changeDwellTime)	RAIL_Stat

Old API Name	Return Value	New API Name	Return Value
RAIL_GetSetEffControl(RAIL_Handle_t railHandle, uint16_t tempBuffer[(52U)/sizeof(uint16_t)], bool reset)	RAIL_Status_t	RAIL_GetSetEffClpcControl(RAIL_Handle_t railHandle, uint16_t tempBuffer[(52U)/sizeof(uint16_t)], bool reset)	RAIL_Status
RAIL_GetSetEffRuralUrbanMv(RAIL_Handle_t railHandle, uint16_t *newTrip, bool changeTrip)	RAIL_Status_t	RAIL_GetSetEffLnaRuralUrbanMv(RAIL_Handle_t railHandle, uint16_t *newTrip, bool changeTrip)	RAIL_Status
RAIL_GetSetEffUrbanBypassMv(RAIL_Handle_t railHandle, uint16_t *newTrip, bool changeTrip)	RAIL_Status_t	RAIL_GetSetEffLnaUrbanBypassMv(RAIL_Handle_t railHandle, uint16_t *newTrip, bool changeTrip)	RAIL_Status
RAIL_GetSetEffUrbanDwellTimeMs(RAIL_Handle_t railHandle, uint32_t *newDwellTime, bool changeDwellTime)	RAIL_Status_t	RAIL_GetSetEffLnaUrbanDwellTimeMs(RAIL_Handle_t railHandle, uint32_t *newDwellTime, bool changeDwellTime)	RAIL_Status

RAIL Library 2.14.3

No changes

RAIL Library 2.14.2

Added

New API Name	Return Value	Comment
RAIL_GetPowerSettingTable(RAIL_Handle_t railHandle, RAIL_TxPowerMode_t mode, RAIL_TxPower_t *minPower, RAIL_TxPower_t *maxPower, RAIL_TxPowerLevel_t *step)	const RAIL_PaPowerSetting_t	
RAIL_ZWAVE_GetBeamHomeIdHash(RAIL_Handle_t railHandle, RAIL_ZWAVE_HomeIdHash_t *pBeamHomeIdHash)	RAIL_Status_t	

Removed

Old API Name	Return Value	Comment
RAIL_GetFemProtectionConfig(RAIL_Handle_t railHandle, RAIL_FemProtectionConfig_t *femConfig)	RAIL_Status_t	Removed alpha API for EFR32xG25+EFF01 support
RAIL_SetFemProtectionConfig(RAIL_Handle_t railHandle, const RAIL_FemProtectionConfig_t *newFemConfig)	RAIL_Status_t	Removed alpha API for EFR32xG25+EFF01 support

RAIL Library 2.14.1

No changes

RAIL Library 2.14.0

Added

New API Name	Return Value	Comment
RAIL_CalibrateHFXO(RAIL_Handle_t railHandle, int8_t *crystalPPMError)	RAIL_Status_t	See RAIL Changelist item 751910

New API Name	Return Value	Comment
RAIL_CompensateHFXO(RAIL_Handle_t railHandle, int8_t crystalPPMError)	RAIL_Status_t	See RAIL Changelist item 751910
RAIL_ConfigHFXOCompensation(RAIL_Handle_t railHandle, const RAIL_HFXOCompensationConfig_t *pHfxoCompensationConfig)	RAIL_Status_t	See RAIL Changelist item 751910
RAIL_ConfigThermalProtection(RAIL_Handle_t genericRailHandle, const RAIL_ChipTempConfig_t *chipTempConfig)	RAIL_Status_t	See RAIL Changelist item 1022182
RAIL_GetHFXOCompensationConfig(RAIL_Handle_t railHandle, RAIL_HFXOCompensationConfig_t *pHfxoCompensationConfig)	RAIL_Status_t	See RAIL Changelist item 751910
RAIL_GetPaPowerSetting(RAIL_Handle_t railHandle)	RAIL_PaPowerSetting_t	
RAIL_GetSetClpcFastLoopCal(RAIL_Handle_t railHandle, RAIL_EffCalConfigEnum_t calibrationIndex, RAIL_EffCalConfig_t *calibrationEntry, bool changeValues)	RAIL_Status_t	
RAIL_GetSetClpcFastLoopCalSlp(RAIL_Handle_t railHandle, RAIL_EffCalConfigEnum_t calibrationIndex, int16_t *newSlope1e1, int16_t *newoffset290Ddbm, bool changeValues)	RAIL_Status_t	
RAIL_GetThermalProtection(RAIL_Handle_t genericRailHandle, RAIL_ChipTempConfig_t *chipTempConfig)	RAIL_Status_t	See RAIL Changelist item 1022182
RAIL_IEEE802154_ConfigRxChannelSwitching(RAIL_Handle_t railHandle, const RAIL_IEEE802154_RxChannelSwitchingCfg_t *pConfig)	RAIL_Status_t	See RAIL Changelist item 858567
RAIL_IEEE802154_SupportsRxChannelSwitching(RAIL_Handle_t railHandle)	bool	See RAIL Changelist item 858567
RAIL_SetPaPowerSetting(RAIL_Handle_t railHandle, RAIL_PaPowerSetting_t paPowerSetting, RAIL_TxPower_t minPowerDbm, RAIL_TxPower_t maxPowerDbm, RAIL_TxPower_t currentPowerDbm)	RAIL_Status_t	
RAIL_SupportsHFXOCompensation(RAIL_Handle_t railHandle)	bool	See RAIL Changelist item 751910
RAIL_SupportsIEEE802154Band2P4(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsThermalProtection(RAIL_Handle_t railHandle)	bool	See RAIL Changelist item 1022182
RAIL_ZWAVE_PerformIrcal(RAIL_Handle_t railHandle, RAIL_ZWAVE_IrcalVal_t *pIrCalVals, bool forceIrcal)	RAIL_Status_t	See RAIL Changelist item 846240

Removed

Old API Name	Return Value	Comment
Old API Name	Return Value	Comment
RAIL_GetSetClpcSlowLoop(RAIL_Handle_t railHandle, uint16_t *newTarget, uint16_t *newSlope, bool changeValues)	RAIL_Status_t	Removed alpha API for EFR32xG25+EFF01 support
RAIL_GetSetInternalTempThreshold(RAIL_Handle_t railHandle, uint16_t *newThreshold, bool changeThreshold)	RAIL_Status_t	Removed alpha API for EFR32xG25 support

Modified

Old API Name	Return Value	New API Name	Return Value	Comment
RAIL_GetSetClpcFastLoop(RAIL_Handle_t railHandle, uint16_t *newTarget, uint16_t *newSlope, bool changeValues)	RAIL_Status_t	RAIL_GetSetClpcFastLoop(RAIL_Handle_t railHandle, uint16_t *newTargetFsk, uint16_t *newSlopeFsk, uint16_t *newTargetOfdm, uint16_t *newSlopeOfdm, bool changeValues)	RAIL_Status_t	Modified alpha API

RAIL Library 2.13.3

No changes

RAIL Library 2.13.2

No changes

RAIL Library 2.13.1

No changes

RAIL Library 2.13.0

Added

New API Name	Return Value	Comment
RAIL_ChangedDcdc(void)	RAIL_Status_t	
RAIL_ComputeHFXOPPMError(RAIL_Handle_t railHandle, int16_t crystalTemperatureC, int8_t *crystalPPMError)	RAIL_Status_t	
RAIL_ConfigEff(RAIL_Handle_t railHandle, const RAIL_EffConfig_t *config)	RAIL_Status_t	
RAIL_ConfigHFXOThermistor(RAIL_Handle_t railHandle, const RAIL_HFXOThermistorConfig_t *hfxoThermistorConfig)	RAIL_Status_t	
RAIL_ConfigPaAutoEntry(RAIL_Handle_t railHandle, const RAIL_PaAutoModeConfigEntry_t *paAutoModeEntry)	RAIL_Status_t	
RAIL_ConvertThermistorImpedance(RAIL_Handle_t railHandle, uint32_t thermistorImpedance, int16_t *thermistorTemperatureC)	RAIL_Status_t	
RAIL_GetChannelAlt(RAIL_Handle_t railHandle, uint16_t *channel)	RAIL_Status_t	
RAIL_GetFemProtectionConfig(RAIL_Handle_t railHandle, RAIL_FemProtectionConfig_t *femConfig)	RAIL_Status_t	
RAIL_GetRssiDetectThreshold(RAIL_Handle_t railHandle)	int8_t	

New API Name	Return Value	Comment
RAIL_GetSetClpcEnable(RAIL_Handle_t railHandle, uint8_t *newClpcEnable, bool changeClpcEnable)	RAIL_Status_t	
RAIL_GetSetClpcFastLoop(RAIL_Handle_t railHandle, uint16_t *newTarget, uint16_t *newSlope, bool changeValues)	RAIL_Status_t	
RAIL_GetSetClpcSlowLoop(RAIL_Handle_t railHandle, uint16_t *newTarget, uint16_t *newSlope, bool changeValues)	RAIL_Status_t	
RAIL_GetSetEffBypassDwellTimeMs(RAIL_Handle_t railHandle, uint32_t *newDwellTime, bool changeDwellTime)	RAIL_Status_t	
RAIL_GetSetEffControl(RAIL_Handle_t railHandle, uint16_t tempBuffer[(52U)/sizeof(uint16_t)], bool reset)	RAIL_Status_t	
RAIL_GetSetEffMode(RAIL_Handle_t railHandle, uint8_t *newMode, bool changeMode)	RAIL_Status_t	
RAIL_GetSetEffRuralUrbanMv(RAIL_Handle_t railHandle, uint16_t *newTrip, bool changeTrip)	RAIL_Status_t	
RAIL_GetSetEffTempThreshold(RAIL_Handle_t railHandle, uint16_t *newThreshold, bool changeThreshold)	RAIL_Status_t	
RAIL_GetSetEffUrbanBypassMv(RAIL_Handle_t railHandle, uint16_t *newTrip, bool changeTrip)	RAIL_Status_t	
RAIL_GetSetEffUrbanDwellTimeMs(RAIL_Handle_t railHandle, uint32_t *newDwellTime, bool changeDwellTime)	RAIL_Status_t	
RAIL_GetSetInternalTempThreshold(RAIL_Handle_t railHandle, uint16_t *newThreshold, bool changeThreshold)	RAIL_Status_t	
RAIL_GetTemperature(RAIL_Handle_t railHandle, int16_t tempBuffer[(13U)], bool reset)	RAIL_Status_t	
RAIL_SetFemProtectionConfig(RAIL_Handle_t railHandle, const RAIL_FemProtectionConfig_t *newFemConfig)	RAIL_Status_t	
RAIL_SetRssiDetectThreshold(RAIL_Handle_t railHandle, int8_t rssiThresholdDbm)	RAIL_Status_t	
RAIL_SupportsAuxAdc(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsEff(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsRssiDetectThreshold(RAIL_Handle_t railHandle)	bool	

Removed

Old API Name	Return Value	Comment
RAIL_IEEE802154_Config863MHzSUNOFDMOptRadio(RAIL_Handle_t railHandle, RAIL_IEEE802154_OFDMOption_t ofdmOption)	RAIL_Status_t	Removed unnecessary alpha API.
RAIL_IEEE802154_Config902MHzSUNOFDMOptRadio(RAIL_Handle_t railHandle, RAIL_IEEE802154_OFDMOption_t ofdmOption)	RAIL_Status_t	Removed unnecessary alpha API.

Modified

Old API Name	Return Value	New API Name	Return Value	Comm
RAIL_BLE_ConfigSignalIdentifier(RAIL_Handle_t railHandle, const RAIL_BLE_SignalIdentifierMode_t signalIdentifierMode)	RAIL_Status_t	RAIL_BLE_ConfigSignalIdentifier(RAIL_Handle_t railHandle, RAIL_BLE_SignalIdentifierMode_t signalIdentifierMode)	RAIL_Status_t	Remov unnece const

Old API Name	Return Value	New API Name	Retu
RAIL_BLE_EnableSignalIdentifier(RAIL_Handle_t railHandle, bool enable)	RAIL_Status_t	RAIL_BLE_EnableSignalDetection(RAIL_Handle_t railHandle, bool enable)	RAIL
RAIL_IEEE802154_ConfigCcaMode(RAIL_Handle_t railHandle, const RAIL_IEEE802154_CcaMode_t ccaMode)	RAIL_Status_t	RAIL_IEEE802154_ConfigCcaMode(RAIL_Handle_t railHandle, RAIL_IEEE802154_CcaMode_t ccaMode)	RAIL
RAIL_IEEE802154_ConfigSignalIdentifier(RAIL_Handle_t railHandle)	RAIL_Status_t	RAIL_IEEE802154_ConfigSignalIdentifier(RAIL_Handle_t railHandle, RAIL_IEEE802154_SignalIdentifierMode_t signalIdentifierMode)	RAIL
RAIL_IEEE802154_EnableSignalIdentifier(RAIL_Handle_t railHandle, bool enable)	RAIL_Status_t	RAIL_IEEE802154_EnableSignalDetection(RAIL_Handle_t railHandle, bool enable)	RAIL

RAIL Library 2.12.2

No changes

RAIL Library 2.12.1

No changes

RAIL Library 2.12.0

Added

New API Name	Return Value	Comment
RAIL_AddStateBuffer3(RAIL_Handle_t genericRailHandle)	RAIL_Status_t	See RAIL Changelist item 721163
RAIL_AddStateBuffer4(RAIL_Handle_t genericRailHandle)	RAIL_Status_t	See RAIL Changelist item 721163
RAIL_BLE_ConfigSignalIdentifier(RAIL_Handle_t railHandle, const RAIL_BLE_SignalIdentifierMode_t signalIdentifierMode)	RAIL_Status_t	
RAIL_BLE_EnableSignalIdentifier(RAIL_Handle_t railHandle, bool enable)	RAIL_Status_t	
RAIL_BLE_GetCteSampleOffset(RAIL_Handle_t railHandle)	uint8_t	

New API Name	Return Value	Comment
RAIL_BLE_SupportsSignalIdentifier(RAIL_Handle_t railHandle)	bool	
RAIL_GetSchedulerStatusAlt(RAIL_Handle_t railHandle, RAIL_SchedulerStatus_t *pSchedulerStatus, RAIL_Status_t *pRailStatus)	RAIL_Status_t	See RAIL Changelist item 726491
RAIL_GetTxPacketsRemaining(RAIL_Handle_t railHandle)	uint16_t	See RAIL Changelist item 708206
RAIL_IEEE802154_ComputeChannelFromPhyModeId(RAIL_Handle_t railHandle, uint8_t newPhyModeId, uint16_t *pChannel)	RAIL_Status_t	
RAIL_IEEE802154_Config2p4GHzRadioCustom1(RAIL_Handle_t railHandle)	RAIL_Status_t	See RAIL Changelist item 727728
RAIL_IEEE802154_Config863MHzSUNOFDMOptRadio(RAIL_Handle_t railHandle, RAIL_IEEE802154_OFDMOption_t ofdmOption)	RAIL_Status_t	
RAIL_IEEE802154_Config902MHzSUNOFDMOptRadio(RAIL_Handle_t railHandle, RAIL_IEEE802154_OFDMOption_t ofdmOption)	RAIL_Status_t	
RAIL_IEEE802154_ConfigCcaMode(RAIL_Handle_t railHandle, const RAIL_IEEE802154_CcaMode_t ccaMode)	RAIL_Status_t	
RAIL_IEEE802154_ConfigSignalIdentifier(RAIL_Handle_t railHandle)	RAIL_Status_t	
RAIL_IEEE802154_EnableSignalIdentifier(RAIL_Handle_t railHandle, bool enable)	RAIL_Status_t	
RAIL_IEEE802154_SupportsCustom1Phy(RAIL_Handle_t railHandle)	bool	
RAIL_IEEE802154_SupportsDualPaConfig(RAIL_Handle_t railHandle)	bool	
RAIL_IEEE802154_SupportsGDynFec(RAIL_Handle_t railHandle)	bool	
RAIL_IEEE802154_SupportsGModeSwitch(RAIL_Handle_t railHandle)	bool	
RAIL_IEEE802154_SupportsSignalIdentifier(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsOFDMPA(RAIL_Handle_t railHandle)	bool	

RAIL Library 2.11.3

No changes

RAIL Library 2.11.2

Added

New API Name	Return Value	Comment
RAIL_ConfigDirectMode(RAIL_Handle_t railHandle, const RAIL_DirectModeConfig_t *directModeConfig)	RAIL_Status_t	
RAIL_EnableDirectModeAlt(RAIL_Handle_t railHandle, bool enableDirectTx, bool enableDirectRx)	RAIL_Status_t	
RAIL_GetDefaultRxDutyCycleConfig(RAIL_Handle_t railHandle, RAIL_RxDutyCycleConfig_t *config)	RAIL_Status_t	
RAIL_SetMfmPingPongFifo(RAIL_Handle_t railHandle, const RAIL_MFM_PingPongBufferConfig_t *config)	RAIL_Status_t	
RAIL_SupportsMfm(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsRxDirectModeDataToFifo(RAIL_Handle_t railHandle)	bool	

RAIL Library 2.11.1

Added

New API Name	Return Value	Comment
RAIL_GetPtiProtocol(RAIL_Handle_t railHandle)	RAIL_PtiProtocol_t	
RAIL_GetRadioClockFreqHz(RAIL_Handle_t railHandle)	uint32_t	

RAIL Library 2.11.0

Added

New API Name	Return Value	Comment
RAIL_BLE_SetNextTxRepeat(RAIL_Handle_t railHandle, const RAIL_BLE_TxRepeatConfig_t *repeatConfig)	RAIL_Status_t	
RAIL_SetNextTxRepeat(RAIL_Handle_t railHandle, const RAIL_TxRepeatConfig_t *repeatConfig)	RAIL_Status_t	
RAIL_StartScheduledCcaCsmaTx(RAIL_Handle_t railHandle, uint16_t channel, RAIL_TxOptions_t options, const RAIL_ScheduleTxConfig_t *scheduleTxConfig, const RAIL_CsmaConfig_t *csmaConfig, const RAIL_SchedulerInfo_t *schedulerInfo)	RAIL_Status_t	
RAIL_StartScheduledCcaLbtTx(RAIL_Handle_t railHandle, uint16_t channel, RAIL_TxOptions_t options, const RAIL_ScheduleTxConfig_t *scheduleTxConfig, const RAIL_LbtConfig_t *lbtConfig, const RAIL_SchedulerInfo_t *schedulerInfo)	RAIL_Status_t	
RAIL_SupportsTxToTx(RAIL_Handle_t railHandle)	bool	
RAIL_ZWAVE_GetBeamRssi(RAIL_Handle_t railHandle, int8_t *pBeamRssi)	RAIL_Status_t	
RAIL_ZWAVE_GetRxBeamConfig(RAIL_ZWAVE_BeamRxConfig_t *pConfig)	RAIL_Status_t	
RAIL_ZWAVE_SetDefaultRxBeamConfig(RAIL_Handle_t railHandle)	RAIL_Status_t	
RAIL_ZWAVE_SetLrAckData(RAIL_Handle_t railHandle, const RAIL_ZWAVE_LrAckData_t *pLrAckData)	RAIL_Status_t	

RAIL Library 2.10.2

Added

New API Name	Return Value	Comment
RAIL_IEEE802154_GetPtiRadioConfig(RAIL_Handle_t railHandle)	RAIL_IEEE802154_PtiRadioConfig_t	

RAIL Library 2.10.1

No changes

RAIL Library 2.10.0

Added

New API Name	Return Value	Comment
RAIL_ApplyIrCalibrationAlt(RAIL_Handle_t railHandle, RAIL_IrCalValues_t *imageRejection, RAIL_AntennaSel_t rfPath)	RAIL_Status_t	
RAIL_BLE_ConfigPhySimulscan(RAIL_Handle_t railHandle)	RAIL_Status_t	

New API Name	Return Value	Comment
RAIL_BLE_SupportsSimulscanPhy(RAIL_Handle_t railHandle)	bool	
RAIL_CalibrateIrAlt(RAIL_Handle_t railHandle, RAIL_IrCalValues_t *imageRejection, RAIL_AntennaSel_t rfPath)	RAIL_Status_t	
RAIL_ConfigRetimeOptions(RAIL_Handle_t railHandle, RAIL_RetimeOptions_t mask, RAIL_RetimeOptions_t options)	RAIL_Status_t	
RAIL_ConfigSleepAlt(RAIL_Handle_t railHandle, RAIL_TimerSyncConfig_t *syncConfig)	RAIL_Status_t	
RAIL_GetRadioStateDetail(RAIL_Handle_t railHandle)	RAIL_RadioStateDetail_t	
RAIL_GetRetimeOptions(RAIL_Handle_t railHandle, RAIL_RetimeOptions_t *pOptions)	RAIL_Status_t	
RAIL_GetRfPath(RAIL_Handle_t railHandle, RAIL_AntennaSel_t *rfPath)	RAIL_Status_t	
RAIL_GetRssiAlt(RAIL_Handle_t railHandle, RAIL_Time_t waitTimeout)	int16_t	
RAIL_GetTuneDelta(RAIL_Handle_t railHandle)	int32_t	
RAIL_IEEE802154_Config2p4GHzRadioAntDivCoexFem(RAIL_Handle_t railHandle)	RAIL_Status_t	
RAIL_IEEE802154_Config2p4GHzRadioAntDivFem(RAIL_Handle_t railHandle)	RAIL_Status_t	
RAIL_IEEE802154_Config2p4GHzRadioCoexFem(RAIL_Handle_t railHandle)	RAIL_Status_t	
RAIL_IEEE802154_Config2p4GHzRadioFem(RAIL_Handle_t railHandle)	RAIL_Status_t	
RAIL_IEEE802154_SupportsFemPhy(RAIL_Handle_t railHandle)	bool	
RAIL_SetAddressFilterAddressMask(RAIL_Handle_t railHandle, uint8_t field, const uint8_t *bitMask)	RAIL_Status_t	
RAIL_SetTuneDelta(RAIL_Handle_t railHandle, int32_t delta)	RAIL_Status_t	
RAIL_SupportsAddrFilterAddressBitMask(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsAddrFilterMask(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsDirectMode(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsRfSenseEnergyDetection(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsSQPhy(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsTxPowerModeAlt(RAIL_Handle_t railHandle, RAIL_TxPowerMode_t *powerMode, RAIL_TxPowerLevel_t *maxPowerLevel, RAIL_TxPowerLevel_t *minPowerLevel)	bool	
RAIL_ZWAVE_ConfigRxChannelHopping(RAIL_Handle_t railHandle, RAIL_RxChannelHoppingConfig_t *config)	RAIL_Status_t	
RAIL_ZWAVE_GetLrBeamTxPower(RAIL_Handle_t railHandle, uint8_t *pLrBeamTxPower)	RAIL_Status_t	
RAIL_ZWAVE_GetRegion(RAIL_Handle_t railHandle)	RAIL_ZWAVE_RegionId_t	
RAIL_ZWAVE_SupportsConcPhy(RAIL_Handle_t railHandle)	bool	
RAIL_ZWAVE_SupportsEnergyDetectPhy(RAIL_Handle_t railHandle)	bool	

RAIL Library 2.9.2

No changes

RAIL Library 2.9.1

No changes

RAIL Library 2.9.0

Added

New API Name	Return Value	Comment
RAILCb_ConfigSleepTimerSync(RAIL_TimerSyncConfig_t *timerSyncConfig)	void	
RAIL_BLE_ConfigPhyQuuppa(RAIL_Handle_t railHandle)	RAIL_Status_t	
RAIL_BLE_SupportsQuuppa(RAIL_Handle_t railHandle)	bool	
RAIL_DeinitPowerManager(void)	RAIL_Status_t	
RAIL_GetTxPacketDetailsAlt2(RAIL_Handle_t railHandle, RAIL_TxPacketDetails_t *pPacketDetails)	RAIL_Status_t	
RAIL_GetTxTimeFrameEndAlt(RAIL_Handle_t railHandle, RAIL_TxPacketDetails_t *pPacketDetails)	RAIL_Status_t	
RAIL_GetTxTimePreambleStartAlt(RAIL_Handle_t railHandle, RAIL_TxPacketDetails_t *pPacketDetails)	RAIL_Status_t	
RAIL_GetTxTimeSyncWordEndAlt(RAIL_Handle_t railHandle, RAIL_TxPacketDetails_t *pPacketDetails)	RAIL_Status_t	
RAIL_InitPowerManager(void)	RAIL_Status_t	

RAIL Library 2.8.10

No changes

RAIL Library 2.8.9

No changes

RAIL Library 2.8.8

No changes

RAIL Library 2.8.7

No changes

RAIL Library 2.8.6

No changes

RAIL Library 2.8.4

Added

New API Name	Return Value	Comment
RAIL_BLE_ConfigAoxAntenna(RAIL_Handle_t railHandle, RAIL_BLE_AoxAntennaConfig_t *antennaConfig)	RAIL_Status_t	
RAIL_BLE_InitCte(RAIL_Handle_t railHandle)	RAIL_Status_t	

RAIL Library 2.8.3

Added

New API Name	Return Value	Comment
RAIL_BLE_Supports1Mbps(RAIL_Handle_t railHandle)	bool	
RAIL_BLE_Supports1MbpsNonViterbi(RAIL_Handle_t railHandle)	bool	
RAIL_BLE_Supports1MbpsViterbi(RAIL_Handle_t railHandle)	bool	
RAIL_BLE_Supports2Mbps(RAIL_Handle_t railHandle)	bool	
RAIL_BLE_Supports2MbpsNonViterbi(RAIL_Handle_t railHandle)	bool	
RAIL_BLE_Supports2MbpsViterbi(RAIL_Handle_t railHandle)	bool	
RAIL_BLE_SupportsAntennaSwitching(RAIL_Handle_t railHandle)	bool	
RAIL_BLE_SupportsCodedPhy(RAIL_Handle_t railHandle)	bool	
RAIL_BLE_SupportsCte(RAIL_Handle_t railHandle)	bool	
RAIL_BLE_SupportsIQSampling(RAIL_Handle_t railHandle)	bool	
RAIL_BLE_SupportsPhySwitchToRx(RAIL_Handle_t railHandle)	bool	
RAIL_IEEE802154_SupportsCancelFramePendingLookup(RAIL_Handle_t railHandle)	bool	
RAIL_IEEE802154_SupportsCoexPhy(RAIL_Handle_t railHandle)	bool	
RAIL_IEEE802154_SupportsEEnhancedAck(RAIL_Handle_t railHandle)	bool	
RAIL_IEEE802154_SupportsEMultipurposeFrames(RAIL_Handle_t railHandle)	bool	
RAIL_IEEE802154_SupportsESubsetGB868(RAIL_Handle_t railHandle)	bool	
RAIL_IEEE802154_SupportsEarlyFramePendingLookup(RAIL_Handle_t railHandle)	bool	
RAIL_IEEE802154_SupportsG4ByteCrc(RAIL_Handle_t railHandle)	bool	
RAIL_IEEE802154_SupportsGSubsetGB868(RAIL_Handle_t railHandle)	bool	
RAIL_IEEE802154_SupportsGUnwhitenedRx(RAIL_Handle_t railHandle)	bool	
RAIL_IEEE802154_SupportsGUnwhitenedTx(RAIL_Handle_t railHandle)	bool	
RAIL_Supports2p4GHzBand(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsAlternateTxPower(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsAntennaDiversity(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsChannelHopping(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsDualBand(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsDualSyncWords(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsExternalThermistor(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsPrecisionLFRCO(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsProtocolBLE(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsProtocolIEEE802154(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsProtocolZWave(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsRadioEntropy(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsRfSenseSelectiveOok(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsSubGHzBand(RAIL_Handle_t railHandle)	bool	
RAIL_SupportsTxPowerMode(RAIL_Handle_t railHandle, RAIL_TxPowerMode_t powerMode, RAIL_TxPowerLevel_t *pMaxPowerLevel)	bool	
RAIL_ZWAVE_SupportsRegionPti(RAIL_Handle_t railHandle)	bool	

RAIL Library 2.8.2

No changes

RAIL Library 2.8.1

Added

New API Name	Return Value	Comment
RAILCb_PaAutoModeDecision(RAIL_Handle_t railHandle, RAIL_TxPower_t *power, RAIL_TxPowerMode_t *mode, const RAIL_ChannelConfigEntry_t *chCfgEntry)	RAIL_Status_t	
RAIL_GetThermistorImpedance(RAIL_Handle_t railHandle, uint32_t *thermistorImpedance)	RAIL_Status_t	
RAIL_IEEE802154_ConvertRssiToEd(int8_t rssiDbm)	uint8_t	
RAIL_IEEE802154_ConvertRssiToLqi(uint8_t origLqi, int8_t rssiDbm)	uint8_t	
RAIL_StartThermistorMeasurement(RAIL_Handle_t railHandle)	RAIL_Status_t	
RAIL_StartTxStreamAlt(RAIL_Handle_t railHandle, uint16_t channel, RAIL_StreamMode_t mode, RAIL_TxOptions_t options)	RAIL_Status_t	

RAIL Library 2.8.0

Added

New API Name	Return Value	Comment
RAIL_BLE_GetCteSampleRate(RAIL_Handle_t railHandle)	uint32_t	
RAIL_ConfigRfSenseSelectiveOokWakeupPhy(RAIL_Handle_t railHandle)	RAIL_Status_t	
RAIL_ConfigSyncWords(RAIL_Handle_t railHandle, const RAIL_SyncWordConfig_t *syncWordConfig)	RAIL_Status_t	
RAIL_EnablePaAutoMode(RAIL_Handle_t railHandle, bool enable)	RAIL_Status_t	
RAIL_GetChannelMetadata(RAIL_Handle_t railHandle, RAIL_ChannelMetadata_t *channelMetadata, uint16_t *length, uint16_t minChannel, uint16_t maxChannel)	RAIL_Status_t	
RAIL_GetRxTransitions(RAIL_Handle_t railHandle, RAIL_StateTransitions_t *transitions)	RAIL_Status_t	
RAIL_GetSyncWords(RAIL_Handle_t railHandle, RAIL_SyncWordConfig_t *syncWordConfig)	RAIL_Status_t	
RAIL_GetTxTransitions(RAIL_Handle_t railHandle, RAIL_StateTransitions_t *transitions)	RAIL_Status_t	
RAIL_IsPaAutoModeEnabled(RAIL_Handle_t railHandle)	bool	
RAIL_SetRfSenseSelectiveOokWakeupPayload(RAIL_Handle_t railHandle, uint8_t preamble, uint8_t numSyncwordBytes, uint32_t syncword)	RAIL_Status_t	
RAIL_StartSelectiveOokRfSense(RAIL_Handle_t railHandle, RAIL_RfSenseSelectiveOokConfig_t *config)	RAIL_Status_t	
RAIL_StopInfinitePreambleTx(RAIL_Handle_t railHandle)	RAIL_Status_t	
RAIL_ZWAVE_ConfigBeamRx(RAIL_Handle_t railHandle, RAIL_ZWAVE_BeamRxConfig_t *config)	RAIL_Status_t	
RAIL_ZWAVE_ReceiveBeam(RAIL_Handle_t railHandle, uint8_t *beamDetectIndex, const RAIL_SchedulerInfo_t *schedulerInfo)	RAIL_Status_t	

Modified

Old API Name	Return Value	New API Name	Return Value	Comment
RAIL_ZWAVE_ConfigRegion(RAIL_Handle_t railHandle, const RAIL_ZWAVE_RegionConfig_t regionCfg)	RAIL_Status_t	RAIL_ZWAVE_ConfigRegion(RAIL_Handle_t railHandle, const RAIL_ZWAVE_RegionConfig_t *regionCfg)	RAIL_Status_t	

RAIL Library 2.7.4

No changes

RAIL Library 2.7.3

No changes

RAIL Library 2.7.2

No changes

RAIL Library 2.7.1

No changes

RAIL Library 2.7.0

Added

New API Name	Return Value	Comment
RAIL_GetRxIncomingPacketInfo(RAIL_Handle_t railHandle, RAIL_RxPacketInfo_t *pPacketInfo)	void	
RAIL_IEEE802154_ConfigEOptions(RAIL_Handle_t railHandle, RAIL_IEEE802154_EOptions_t mask, RAIL_IEEE802154_EOptions_t options)	RAIL_Status_t	
RAIL_IEEE802154_ConfigGOptions(RAIL_Handle_t railHandle, RAIL_IEEE802154_GOptions_t mask, RAIL_IEEE802154_GOptions_t options)	RAIL_Status_t	
RAIL_IEEE802154_EnableDataFramePending(RAIL_Handle_t railHandle, bool enable)	RAIL_Status_t	
RAIL_IEEE802154_EnableEarlyFramePending(RAIL_Handle_t railHandle, bool enable)	RAIL_Status_t	
RAIL_IEEE802154_WriteEnhAck(RAIL_Handle_t railHandle, const uint8_t *ackData, uint8_t ackDataLen)	RAIL_Status_t	
RAIL_UseDma(uint8_t channel)	RAIL_Status_t	
RAIL_ZWAVE_GetTxLowPower(RAIL_Handle_t railHandle)	RAIL_TxPowerLevel_t	
RAIL_ZWAVE_GetTxLowPowerDbm(RAIL_Handle_t railHandle)	RAIL_TxPower_t	
RAIL_ZWAVE_SetTxLowPower(RAIL_Handle_t railHandle, uint8_t powerLevel)	RAIL_Status_t	
RAIL_ZWAVE_SetTxLowPowerDbm(RAIL_Handle_t railHandle, RAIL_TxPower_t powerLevel)	RAIL_Status_t	

RAIL Library 2.6.5

No changes

RAIL Library 2.6.4

No changes

RAIL Library 2.6.3

No changes

RAIL Library 2.6.2

No changes

RAIL Library 2.6.1

Added

New API Name	Return Value	Comment
New API Name	Return Value	Comment
RAIL_GetChannelHoppingRssi(RAIL_Handle_t railHandle, uint8_t channelIndex)	int16_t	

RAIL Library 2.6.0

Added

New API Name	Return Value	Comment
RAILCb_ConfigFrameTypeLength(RAIL_Handle_t railHandle, const RAIL_FrameType_t *frameType)	void	
RAIL_BLE_ConfigAox(RAIL_Handle_t railHandle, const RAIL_BLE_AoxConfig_t *aoxConfig)	RAIL_Status_t	
RAIL_BLE_CteBufferIsLocked(RAIL_Handle_t railHandle)	bool	
RAIL_BLE_LockCteBuffer(RAIL_Handle_t railHandle, bool lock)	bool	
RAIL_BLE_PhySwitchToRx(RAIL_Handle_t railHandle, RAIL_BLE_Phy_t phy, uint16_t railChannel, uint32_t startRxTime, uint32_t crcInit, uint32_t accessAddress, uint16_t logicalChannel, bool disableWhitening)	RAIL_Status_t	
RAIL_ConfigRxDutyCycle(RAIL_Handle_t railHandle, const RAIL_RxDutyCycleConfig_t *config)	RAIL_Status_t	
RAIL_DelayUs(RAIL_Time_t microseconds)	RAIL_Status_t	
RAIL_EnableRxDutyCycle(RAIL_Handle_t railHandle, bool enable)	RAIL_Status_t	
RAIL_GetRxTimeFrameEndAlt(RAIL_Handle_t railHandle, RAIL_RxPacketDetails_t *pPacketDetails)	RAIL_Status_t	
RAIL_GetRxTimePreambleStartAlt(RAIL_Handle_t railHandle, RAIL_RxPacketDetails_t *pPacketDetails)	RAIL_Status_t	
RAIL_GetRxTimeSyncWordEndAlt(RAIL_Handle_t railHandle, RAIL_RxPacketDetails_t *pPacketDetails)	RAIL_Status_t	
RAIL_GetTransitionTime(void)	RAIL_Time_t	
RAIL_IEEE802154_Config2p4GHzRadioAntDivCoex(RAIL_Handle_t railHandle)	RAIL_Status_t	
RAIL_IEEE802154_Config2p4GHzRadioCoex(RAIL_Handle_t railHandle)	RAIL_Status_t	
RAIL_IncludeFrameTypeLength(RAIL_Handle_t railHandle)	void	
RAIL_IsInitialized(void)	bool	
RAIL_SetTransitionTime(RAIL_Time_t transitionTime)	void	
RAIL_ZWAVE_GetBeamChannelIndex(RAIL_Handle_t railHandle, uint8_t *pChannelIndex)	RAIL_Status_t	

RAIL Library 2.5.1

No changes

RAIL Library 2.5.0

Added

New API Name	Return Value	Comment
RAIL_ConfigRxChannelHopping(RAIL_Handle_t railHandle, RAIL_RxChannelHoppingConfig_t *config)	RAIL_Status_t	
RAIL_ConvertLqi(RAIL_Handle_t railHandle, RAIL_ConvertLqiCallback_t cb)	RAIL_Status_t	
RAIL_EnableRxChannelHopping(RAIL_Handle_t railHandle, bool enable, bool reset)	RAIL_Status_t	

New API Name	Return Value	Comment
RAIL_IEEE802154_Config2p4GHzRadioAntDiv(RAIL_Handle_t railHandle)	RAIL_Status_t	
RAIL_SetTxAltPreambleLength(RAIL_Handle_t railHandle, uint16_t length)	RAIL_Status_t	
RAIL_VerifyTxPowerCurves(const struct RAIL_TxPowerCurvesConfigAlt *config)	void	
RAIL_ZWAVE_ConfigOptions(RAIL_Handle_t railHandle, RAIL_ZWAVE_Options_t mask, RAIL_ZWAVE_Options_t options)	RAIL_Status_t	
RAIL_ZWAVE_ConfigRegion(RAIL_Handle_t railHandle, const RAIL_ZWAVE_RegionConfig_t regionCfg)	RAIL_Status_t	
RAIL_ZWAVE_Deinit(RAIL_Handle_t railHandle)	RAIL_Status_t	
RAIL_ZWAVE_GetBeamNodeId(RAIL_Handle_t railHandle, RAIL_ZWAVE_NodeId_t *pNodeId)	RAIL_Status_t	
RAIL_ZWAVE_Init(RAIL_Handle_t railHandle, const RAIL_ZWAVE_Config_t *config)	RAIL_Status_t	
RAIL_ZWAVE_IsEnabled(RAIL_Handle_t railHandle)	bool	
RAIL_ZWAVE_SetHomeId(RAIL_Handle_t railHandle, RAIL_ZWAVE_HomeId_t homeId, RAIL_ZWAVE_HomeIdHash_t homeIdHash)	RAIL_Status_t	
RAIL_ZWAVE_SetNodeId(RAIL_Handle_t railHandle, RAIL_ZWAVE_NodeId_t nodeId)	RAIL_Status_t	

RAIL Library 2.4.1

No changes

RAIL Library 2.4.0

Added

New API Name	Return Value	Comment
PRORTC_IRQHandler(void)	void	
RAILCb_SetupRxFifo(RAIL_Handle_t railHandle)	RAIL_Status_t	
RAIL_BLE_CalibrateIr(RAIL_Handle_t railHandle, uint32_t *imageRejection)	RAIL_Status_t	
RAIL_GetChannel(RAIL_Handle_t railHandle, uint16_t *channel)	RAIL_Status_t	
RAIL_GetRssiOffset(RAIL_Handle_t railHandle)	int8_t	
RAIL_GetRxPacketDetailsAlt(RAIL_Handle_t railHandle, RAIL_RxPacketHandle_t packetHandle, RAIL_RxPacketDetails_t *pPacketDetails)	RAIL_Status_t	
RAIL_GetRxTimeFrameEnd(RAIL_Handle_t railHandle, uint16_t totalPacketBytes, RAIL_Time_t *pPacketTime)	RAIL_Status_t	
RAIL_GetRxTimePreambleStart(RAIL_Handle_t railHandle, uint16_t totalPacketBytes, RAIL_Time_t *pPacketTime)	RAIL_Status_t	
RAIL_GetRxTimeSyncWordEnd(RAIL_Handle_t railHandle, uint16_t totalPacketBytes, RAIL_Time_t *pPacketTime)	RAIL_Status_t	
RAIL_GetTxPacketDetailsAlt(RAIL_Handle_t railHandle, bool isAck, RAIL_Time_t *pPacketTime)	RAIL_Status_t	
RAIL_GetTxTimeFrameEnd(RAIL_Handle_t railHandle, uint16_t totalPacketBytes, RAIL_Time_t *pPacketTime)	RAIL_Status_t	
RAIL_GetTxTimePreambleStart(RAIL_Handle_t railHandle, uint16_t totalPacketBytes, RAIL_Time_t *pPacketTime)	RAIL_Status_t	
RAIL_GetTxTimeSyncWordEnd(RAIL_Handle_t railHandle, uint16_t totalPacketBytes, RAIL_Time_t *pPacketTime)	RAIL_Status_t	
RAIL_IEEE802154_CalibrateIr2p4Ghz(RAIL_Handle_t railHandle, uint32_t *imageRejection)	RAIL_Status_t	
RAIL_IEEE802154_CalibrateIrSubGhz(RAIL_Handle_t railHandle, uint32_t *imageRejection)	RAIL_Status_t	

New API Name	Return Value	Comment
RAIL_IEEE802154_ConfigGB863MHzRadio(RAIL_Handle_t railHandle)	RAIL_Status_t	
RAIL_IEEE802154_ConfigGB915MHzRadio(RAIL_Handle_t railHandle)	RAIL_Status_t	
RAIL_PrepareChannel(RAIL_Handle_t railHandle, uint16_t channel)	RAIL_Status_t	
RAIL_SetRssiOffset(RAIL_Handle_t railHandle, int8_t rssiOffset)	RAIL_Status_t	
RAIL_SetRxFifo(RAIL_Handle_t railHandle, uint8_t *addr, uint16_t *size)	RAIL_Status_t	
RAIL_SetTaskPriority(RAIL_Handle_t railHandle, uint8_t priority, RAIL_TaskType_t taskType)	RAIL_Status_t	
RAIL_StopTx(RAIL_Handle_t railHandle, RAIL_StopMode_t mode)	RAIL_Status_t	

Deprecated List

Deprecated List

- Global RAIL_ApplyIrCalibration (RAIL_Handle_t railHandle, uint32_t imageRejection)
  Please use RAIL_ApplyIrCalibrationAlt instead.
- Global RAIL_BLE_AOX_OPTIONS_DO_SWITCH
  Obsolete AOX option
- Global RAIL_BLE_AOX_OPTIONS_LOCK_CTE_BUFFER_SHIFT
  Please use RAIL_BLE_AOX_OPTIONS_DISABLE_BUFFER_LOCK_SHIFT instead.
- Global RAIL_BLE_AOX_OPTIONS_RX_ENABLED
  Obsolete AOX option
- Global RAIL_BLE_AOX_OPTIONS_TX_ENABLED
  Obsolete AOX option
- Global RAIL_BLE_Coding_125kbps_DSA
  Will be removed in a future version of RAIL
- Global RAIL_BLE_Coding_500kbps_DSA
  Will be removed in a future version of RAIL
- Global RAIL_CalibrateIr (RAIL_Handle_t railHandle, uint32_t *imageRejection)
  Please use RAIL_CalibrateIrAlt instead.
- Global RAIL_IEEE802154_CalibrateIrSubGhz (RAIL_Handle_t railHandle, uint32_t *imageRejection)
  Please use RAIL_CalibrateIrAlt instead.
- Global RAIL_RX_CHANNEL_HOPPING_OPTION_DEFAULT
  Please use RAIL_RX_CHANNEL_HOPPING_OPTIONS_DEFAULT instead.
- Global RAIL_RxChannelHoppingConfigEntry_t::delayMode
  Set delayMode to RAIL_RX_CHANNEL_HOPPING_DELAY_MODE_STATIC.
- Global RAIL_RxChannelHoppingDelayMode_t
  Set only to RAIL_RX_CHANNEL_DELAY_MODE_STATIC
- Global RAIL_TX_POWER_MODE_2P4_HIGHEST
  Please use RAIL_TX_POWER_MODE_2P4GIG_HIGHEST instead.
- Global RAIL_TX_POWER_MODE_2P4_HP
  Please use RAIL_TX_POWER_MODE_2P4GIG_HP instead.
- Global RAIL_TX_POWER_MODE_2P4_LP
  Please use RAIL_TX_POWER_MODE_2P4GIG_LP instead.
- Global RAIL_TX_POWER_MODE_2P4_MP
  Please use RAIL_TX_POWER_MODE_2P4GIG_MP instead.
- Global RAIL_TX_POWER_MODE_OFDM_PA
  Please use RAIL_TX_POWER_MODE_OFDM_PA_POWERSETTING_TABLE instead.
- Global RAIL_TX_POWER_MODE_SUBGIG
  Please use RAIL_TX_POWER_MODE_SUBGIG_HP instead.

About RAIL Examples

RAIL Examples

A number of examples are provided with Simplicity Studio and the Proprietary Flex SDK. Each example has an associated README that explains the purpose of the example and how to use it. Some of the examples have more extensive documentation, and that is included in this section. Because new examples are always being added, the GSDK may contain other examples not documented here.

Many of these examples are intended only to test specific aspects of the radio. They are not intended as a starting point for your own application, although they may contain code for some features you want to use. **RAIL - SoC Empty** is a recommended starting point.

RAIL - SoC Empty: The Empty Example application initializes RAIL to support a Single PHY radio configuration and defines an empty radio event handler. These are the necessary parts for every RAIL-based application.

RAIL - SoC Burst Duty Cycle: The Burst Duty Cycle example is a complete duty cycle solution implemented on top of RAIL. It implements both the Master (sender) and the Slave (listener) nodes. In receive mode the device spends most of the time in sleep state (EM1 or EM2) and wakes up periodically for short periods. In transmit mode the device transmits multiple packets in quick succession. This application can be used to optimize radio configuration and designing LDC (Low Duty-cycle) periods on the receiver in order to save as much energy as possible without PER (Packet Error Rate) reduction.

RAIL - SoC Long Preamble Duty Cycle: The Long Preamble Duty Cycle example is a complete duty cycle solution implemented on top of RAIL. This example supports both receive and transmit modes. In receive mode the device spends most of the time in a sleep state (EM1 or EM2) and wakes up periodically for a short periods. In transmit mode the device transmits one packet using long preamble. The low duty cycle mode of the EFR32 radios is a simple and effective way to reduce the current consumption of the system by autonomously waking up the receiver to look for transmitted packets, while the MCU remains in its lowest power mode.

RAIL - SoC Energy Mode: The Energy Mode example application serves as a complete, ready-to-use program with which you can accelerate the evaluation of the of energy consumption on your custom radio design by iterating over the available energy modes or using TX/RX radio operations.

RAIL - SoC Simple TRX: The Simple TRX example demonstrates the simplest exchange of transmit and receive operations between two nodes implemented in RAIL. It can be used to set up a simple link test, where the nodes are listening except for the short periods where packet transmission takes place.

RAIL - SoC Simple TRX with Auto-ACK: The Simple TRX with Auto-ACK example demonstrates how RAIL's Auto-ACK feature works. It can be used to set up a simple link test that requires Auto-ACK functionality.

RAIL - SoC Simple TRX Standards: The Simple TRX Standards example demonstrates the simplest packet exchange of transmit and receive operations between two nodes implemented in RAIL compatible with Bluetooth LE, IEEE 802.15.4 (2.4 GHz only) and IEEE 802.15.4g (subGHz only) standards. It can be used to set up a simple link test, where the nodes are listening except for the short periods where packet transmission takes place.

RAIL - SoC RAILtest: The RAILtest application (RAILtest) is a simple tool for testing the radio and the functionality of the RAIL library. It is fully documented in the RAILtest User's Guide (UG409).

RAIL - SoC Range Test: Range Test is a standalone test application that creates a radio link between devices and sends a predefined number of packets from the transmitter side to the receiver. The Range Test demo implements packet error rate (PER) measurement. PER is a commonly-used technique for measuring the quality of RF links in wireless systems under particular conditions. Range Test applications are documented in Flex SDK v3.x Range Test Demo User's Guide (UG471).

RAIL - SoC Wireless M-bus Meter and Collector: Wireless M-bus applications are documented in Using RAIL for Wireless M-Bus Applications with EFR32 (AN1119).