Machine Learning

- Overview
- Getting Started Guides
  - Getting Started With Machine Learning
  - Add Machine Learning to a New or Existing Project
  - Update or Replace a .tflite File
  - Tensorflow Lite Micro from Scratch
  - Developing a Model
- Tensorflow Lite for Microcontrollers
  - Tensorflow Lite for Microcontrollers
  - Flatbuffer Converter Tool
  - MVP Accelerator
  - Sample Applications
- API Documentation
  - Audio Feature Generator
    - sl_ml_audio_feature_generation_init
    - sl_ml_audio_feature_generation_frontend_init
    - sl_ml_audio_feature_generation_update_features
    - sl_ml_audio_feature_generation_get_features_raw
    - sl_ml_audio_feature_generation_fill_tensor
    - sl_ml_audio_feature_generation_get_new_feature_slice_count
    - sl_ml_audio_feature_generation_get_feature_buffer_size
    - sl_ml_audio_feature_generation_reset
  - TensorFlow Lite Micro Debug
    - sl_tflite_micro_enable_debug_log
    - sl_tflite_micro_is_debug_log_enabled
  - TensorFlow Lite Micro Init
    - sl_tflite_micro_estimate_arena_size
    - sl_tflite_micro_allocate_tensor_arena
    - sl_tflite_micro_get_error_reporter
    - sl_tflite_micro_get_interpreter
    - sl_tflite_micro_get_input_tensor
    - sl_tflite_micro_get_output_tensor
    - sl_tflite_micro_opcode_resolver
    - sl_tflite_micro_init
  - API Documentation

Overview

Machine Learning

Silicon Labs natively supports TensorFlow Lite for Microcontrollers (TFLM) in the GSDK. TensorFlow is one of the most widely used neural network development platforms. Silicon Labs also supports other Machine Learning (ML) methods through the use of third-party software tools and solutions.

This page will help you decide:

- What software and tools are most applicable to you and your application.
- What examples, demos, and/or tutorials are available to help you get started.

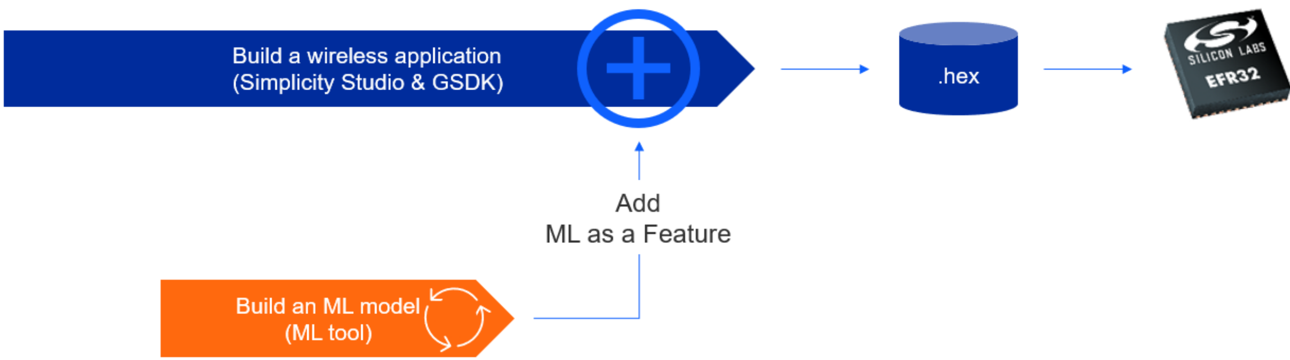
You may also want to visit:

- silabs.com/ai-ml for a general overview of Silicon Labs' Machine Learning approach and product offerings.
- docs.silabs.com/TFLM overview for more information on the native GSDK support of Tensorflow Lite for Microcontrollers.
- siliconlabs.github.io/mltk for the Silicon Labs Machine Learning Toolkit, one of several tool options mentioned below, an open-source, self-serve, community supported, python reference package for Tensorflow developers.
- UG103.19-Machine Learning Fundamentals for a more in-depth explanation of Machine Learning including discussion about models, deploying models, and some key challenges.

Machine Learning and the Development Workflow

Developing an application that incorporates machine learning as a feature requires two distinct workflows:

- The embedded application development workflow used to create a wireless application (with Simplicity Studio or your favorite IDE).
- The machine learning workflow used to create the machine learning feature that will be added to the embedded application.



Familiarity with the standard embedded application development workflow is assumed. The tools presented on this page cover the various options for developing the machine learning feature.

Know Your Machine Learning Developer Skills

We have defined three segments of developer skills expected when using an ML toolchain:

- **ML Expert** : The developer is experienced with TensorFlow and Python. The Expert tools require that the developer is very proficient with these skills, or is willing to learn.
- **ML Explorer** : The developer knows the basic machine learning concepts and workflow (data capture, training, testing & converting), but wants a curated approach that steps the developer through all the steps needed to implement machine learning as a feature in their product.
- **ML Solutions** : The developer wants to use machine learning technology but does not have any knowledge about machine learning nor is willing to learn. Instead, they want a black box solution they add as a value-added feature to their product.

Native TFLM Support for Experts

For the **ML Expert**, Silicon Labs provides native TensorFlow Lite for Microcontrollers (TFLM) support. There are two implementation aspects to consider when using these tools:

1. **Training a model**. For training there are two options to consider: use TensorFlow directly, or use the Silicon Labs Machine Learning Toolkit MLTK.
2. **Integrating the model**. Integrating a `.tflite` model file into the embedded application and running inference. The getting started sections below refer to examples that can help the developer with integration.

Training a Model with TensorFlow

The GSDK supports TensorFlow Lite for Microcontrollers (TFLM) natively. The developer can create a quantized `.tflite` model file using the TensorFlow environment directly and integrate it into the GDSK.

Note:

- Use the MLTK Profiler Utility to determine the RAM and Flash requirements of your `.tflite` model file to confirm the fit within your embedded application.

Getting Started

- Developing a Model Using TensorFlow and Keras (docs.silabs.com)

Training a Model with the Silicon Labs MLTK

The Machine Learning Toolkit (MLTK) was created for these reasons to help the Tensorflow developer. It is a set of python scripts designed to follow the typical machine learning workflow for Silicon Labs embedded devices.

Note:

- This package is made available as an open source, self-serve, community supported, reference package with a comprehensive set of on-line documentation. There are no Silicon Labs support services for this software at this time.
- These scripts are a reference implementations for the use case covered by the documented tutorials. Support for other use cases is the responsibility of the Expert developer.

Getting Started

- Create a Keyword Spotting Model for a GSDK example. (GitHub.io) - MLTK Tutorial

Integrating the Model

The trained model, whether using TensorFlow directly or the MLTK, is represented in a `.tflite` file. To add the `.tflite` file to your embedded application see developing an inference application.

There are several GSDK examples that include machine learning and show how to run inference using pre-trained models. The pre-trained models can be replaced with a model trained from one of the methods above.

Getting Started

- Start with any one of these Sample Applications that are provided with the GSDK.

Third Party Partner Toolchains for Explorers

For the **ML Explorer**, these tools curate the end-to-end workflow for creating a machine learning application specific model and accompanying embedded software to include in your application. They offer:

- An end-to-end coverage of the machine learning workflow
- An easy-to-follow developer experience
- Extra value added features like AutoML, or object detection

Requires:

- Understanding of machine learning concepts (date collection, training, verification, inference, confusion matrix, etc.).
- Ability to collect data that represents the real world scenarios for your product deployment.

The following tools fit well with developing ML as a feature for a Silicon Labs-based embedded application.

Edge Impulse

Edge Impulse is ushering in the future of embedded machine learning by empowering developers to create and optimize solutions with real-world data. They are making the process of building, deploying, and scaling embedded ML applications easier and faster than ever, unlocking massive value across every industry, with millions of developers making billions of devices smarter.

Getting Started

Using Edge Impulse Studio with the xG24 Dev Kit

SensiML

SensiML offers cutting edge AutoML embedded code generation software for implementing AI at the IoT edge. With SensiML Analytics Toolkit, developers have an AI development tool which supports rapid data collection, labeling, feature extraction, ML classification and optimized firmware code generation. Automation built into the tool drastically reduces development time and cost, allowing projects ranging from single users to large teams to generate optimized edge AI sensor algorithms in a fraction of the time that would have otherwise been required with hand coding.

Getting Started

Using SensiML Analytics Toolkit with the xG24 Dev Kit

Third Party Partner Solutions

These tools do not require any knowledge of machine learning to take full advantage of their features. They are designed for specific use cases.

Anomaly Detection

MicroAI's AtomML is an Edge-native, self-correcting, semi-supervised learning engine that aggregates data from internal device sensors, to tune itself to create a behavioral profile of the asset, which then detects and acts upon abnormal behavior. AtomML brings big infrastructure intelligence into a single piece of equipment or device. Unlike traditional AI-driven asset management solutions that rely on the cloud, AtomML is deployed directly to smart devices and sensors. AtomML operates within the small environment of the device itself, providing a more efficient method for asset analytics and generating real-time alerts. AtomML brings an intimate, local approach to asset management for producing a host of operational efficiencies.

Getting Started

Using MicroAI AtomML with the Thunderboard Sense 2

Wake-word/Voice Commands

Sensory creates a safer and superior UX through vision and voice technologies. Sensory technologies are widely deployed in consumer electronics applications including mobile phones, automotive, wearables, toys, IoT, PCs, medical products, and various home electronics. Sensory's product line includes TrulyHandsfree voice control, TrulySecure biometric authentication, and Trul-yNatural large vocabulary natural language embedded speech recognition. Sensory's technologies have shipped in over three bil-lion units of leading consumer products.

Getting Started

Using Sensory Truly Hands Free with the xG24 Dev Kit

Choosing a Machine Learning Tool Based on Use Case

This section provides getting started links for tools available or suggested, based on use case. The links provided may be any one of and an example, demo, or tutorial.

Sensor Signal Processing

Sensor signal processing is the use of low data rate sensors, like accelerometers, gyroscopes, air quality sensors, temperature sensors, pressure sensors, and so on. These use cases can be found in many types of markets, such as preventive maintenance, medical devices or smoke detectors.

- TensorFlow Magic Wand Example - GSDK Platform Example
- Predictive Maintenance Demo - SensiML Demo (video 5 min)
- Build the Predictive Maintenance Demo - SensiML Tutorial (video 80 min) with procedure.pdf
- Air Quality Anomaly Detection Demo - Micro.ai Demo (video 1.5 min)

Audio Pattern Matching

Audio pattern matching uses a microphone to detect different sounds. The types of sounds detected can be a wide range of non-speech related sounds, such as squeaky bearing, jingling keys, breaking glass, water running, animal sounds, and so on.

- Creating an Acoustic Smart Home Door Lock Demo - SensiML Tutorial (5 part blog)
- Introduction to Building an Audio Classifier Demo - Edge Impulse Tutorial Introduction (video 10 min)
- Building and Audio Classifier Demo - Edge Impulse Tutorial (video 70 min), with pre-work instructions, start on page 12
- Audio classifier Example - MLTK Example

Voice Command

Voice commands is a specific sub-set of audio patterns that are the recognition of a small set of spoken words. This is also referred to as keyword spotting. It can be used in a variety of use cases, such as waking up a voice service (i.e., Alexa), or using voice activation for smart home devices or appliances.

- Zigbee Voice Controlled Light Switch Example - GSDK Zigbee Example
- Voice Controlled LED Example - GSDK Platform Example
- Create a Voice Command Model for a GSDK Example - MLTK Tutorial
  - This tutorial can be used for either example above to create a model (i.e., `.tflite` file).
- Building a Voice Recognition Demo - Edge Impulse Tutorial (video 90 min)
- Voice Command/Wake-word Example - Sensory Truly Hands Free Example

Low-Resolution Vision

Low resolution vision uses a camera with resolution around 100x100 to detect objects for presence detection, people counting, video wake-up or more.

- People Flow and Counting Example - GSDK Platform Example
- People Counting Demo - Edge Impulse Example
- Build a Rock, Paper, Scissors Detector Demo MLTK Tutorial
- Build a Fingerprint Authenticator Demo MLTK Tutorial

Summary of Machine Learning Tools

Below is a table that summarizes our different tool options organized by developer skills and target use cases

	ML EXPERT	ML EXPLORER	ML SOLUTIONS
LOW-RATE SENSORS	TensorFlow	EDGE IMPULSE SensiML	Micro.ai Anomaly Detection
AUDIO PATTERN MATCHING	TensorFlow Machine Learning Toolkit	EDGE IMPULSE SensiML	
VOICE COMMANDS	TensorFlow Machine Learning Toolkit	EDGE IMPULSE SensiML	sensory Voice Command
LOW-RESOLUTION VISION	TensorFlow Machine Learning Toolkit	EDGE IMPULSE	

Getting Started With Machine Learning

Getting Started with Machine Learning

Introduction

Silicon Labs integrates TensorFlow Lite for Microcontrollers as a component within the Gecko SDK and Project Configurator for EFx32 series microcontrollers, making it simple to add machine learning capability to any application. This guide covers how TensorFlow Lite for Microcontrollers is integrated with the Gecko SDK for use Silicon Labs' EFx32 devices.

TensorFlow Lite for Microcontrollers

TensorFlow is a widely used deep learning framework, with capability for developing and executing neural networks across a variety of platforms. TensorFlow Lite provides an optimized set of tools specifically catered towards machine learning for mobile and embedded devices.

TensorFlow Lite for Microcontrollers (TFLM) specifically provides a C++ library for running machine learning models in embedded environments with tight memory constraints. Silicon Labs provides tools and support for loading and running pre-trained models that are compatible with this library.

Gecko SDK TensorFlow Integration

The Gecko SDK includes TensorFlow Lite for Microcontrollers (TFLM) as a third-party package, allowing for easy integration and testing with Silicon Labs' projects. Note that the included TFLM version may differ from the latest content upstream, as TFLM practices continuous delivery, and integration into the Gecko SDK only happens biannually.

Additionally, TensorFlow Software Components in the Project Configurator simplify the process of including the necessary dependencies to use TFLM in a project.

Training and Quantizing a Model

To perform neural network inference on an EFx32 device, one first needs a trained model in the TFLite Flatbuffer format. There are two approaches to consider for developers experienced with TensorFlow:

- Using the Silicon Labs Machine Learning Toolkit, a Python reference package that combines and simplifies all the necessary TensorFlow training steps.
- Following published tutorials for training neural networks using TensorFlow, as outlined in the section Developing a Model.

Developing an Inference Application Using Simplicity Studio and the Gecko SDK

After you have a trained and quantized TFLite model, the next step is to set up the TFLM libraries to run inference on the EFx32 device.

Project Configurator Setup

The Project Configurator includes TFLM libraries as software components. These software components may be added to any existing project. They are described in the SDK Component Overview. The core components needed for any machine learning project are as follows:

1. TensorFlow Lite Micro. This is the core software component that pulls in all the TFLM dependencies.
2. A supported TFLM kernel implementation. A kernel is a specific hardware/platform implementation of a low level operation used by TensorFlow. Kernel selection can drastically change the performance and computation time of a neural network. By default, the best kernel implementation for the given device is selected automatically.

3. A supported TFLM debug logger. The Project Configurator defaults to using the I/O Stream implementation of the logger. To disable logging entirely, add the "Debug Logging Disabled" component.

In addition to required TFLM components, software components for obtaining and pre-processing sensor data can be added to the project. As an example for audio applications, Silicon Labs provides an audio feature generator component that includes powerful DSP features to filter and extract features from raw audio data, to be used as a frontend for microphone-based applications. Silicon Labs developed drivers for microphones, accelerometers, and other sensors provide a simple interface for obtaining sensor data to feed to a network.

Add Machine Learning to a New or Existing Project

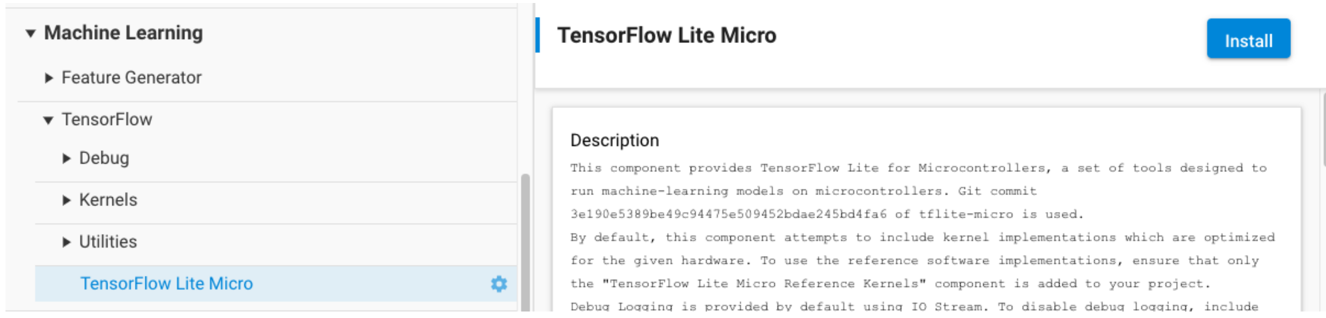
Add Machine Learning to a New or Existing Project

This guide provides details of adding Machine Learning to a new or existing project, making use of the wrapper APIs for TensorFlow Lite for Microcontrollers provided by Silicon Labs for automatic initialization of the TFLM framework.

The guide assumes that a project already exists in the Simplicity Studio workspace. If you're starting from scratch, you may start with any sample application or the Empty C++ application. TFLM has a C++ API, so the application code interfacing with it will also need to be written in C++. If you're starting with an application that is predominantly C code, see the section on interfacing with C code for tips on how to structure your project by adding a separate C++ file for the TFLM interface.

Install the TensorFlow Lite Micro Component

Browse the Software Components library in the project to find the TensorFlow Lite Micro component in `Machine Learning > TensorFlow`.



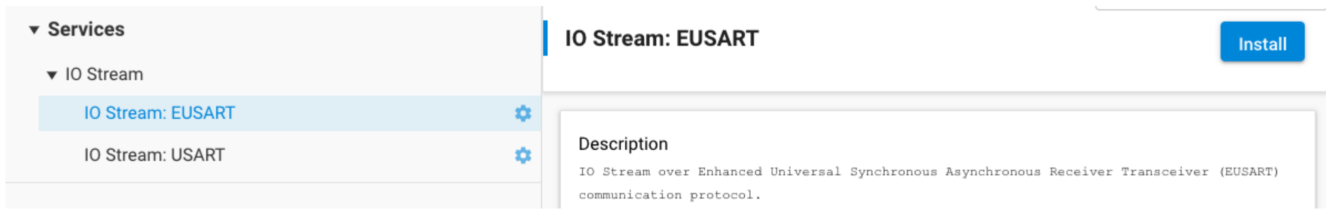
If your project didn't already contain an I/O Stream implementation, you may get a dependency validation warning. This is not a problem, but simply means that a choice of I/O Stream backend needs to be made. The USART or EUSART backends are the most common, as these can communicate with a connected PC through the development kit virtual COM port (VCOM).

Close

Validation Errors

Validation Errors found when adding TensorFlow Lite Micro. Due to these errors, automatic project generation did not occur.

iostream needs iostream_transport_core(may be satisfied by adding one of the following components: IO Stream: USART Core, IO Stream: SWO, IO Stream debug, IO Stream: DUMMY, IO Stream: RTT, IO Stream: EUSART Core, IO Stream VUART, IO Stream: STDIO, IO Stream: CPC)



Accept the default suggestion of "vcom" as the instance name, which will automatically configure the pinout to connect to the development board's VCOM lines. If you're using your own hardware, you can set any instance name and configure the pinout manually.

Model Inclusion

With the TensorFlow Lite Micro component added in the Project Configurator, the next step is to load the model file into the project. To do this, create a `tflite` directory inside the `config` directory of the project, and copy the `.tflite` model file into it. The project configurator provides a tool that will automatically convert `.tflite` files into `sl_tflite_micro_model` source and header files. The full documentation for this tool is available at Flatbuffer Converter Tool.

Automatic Initialization

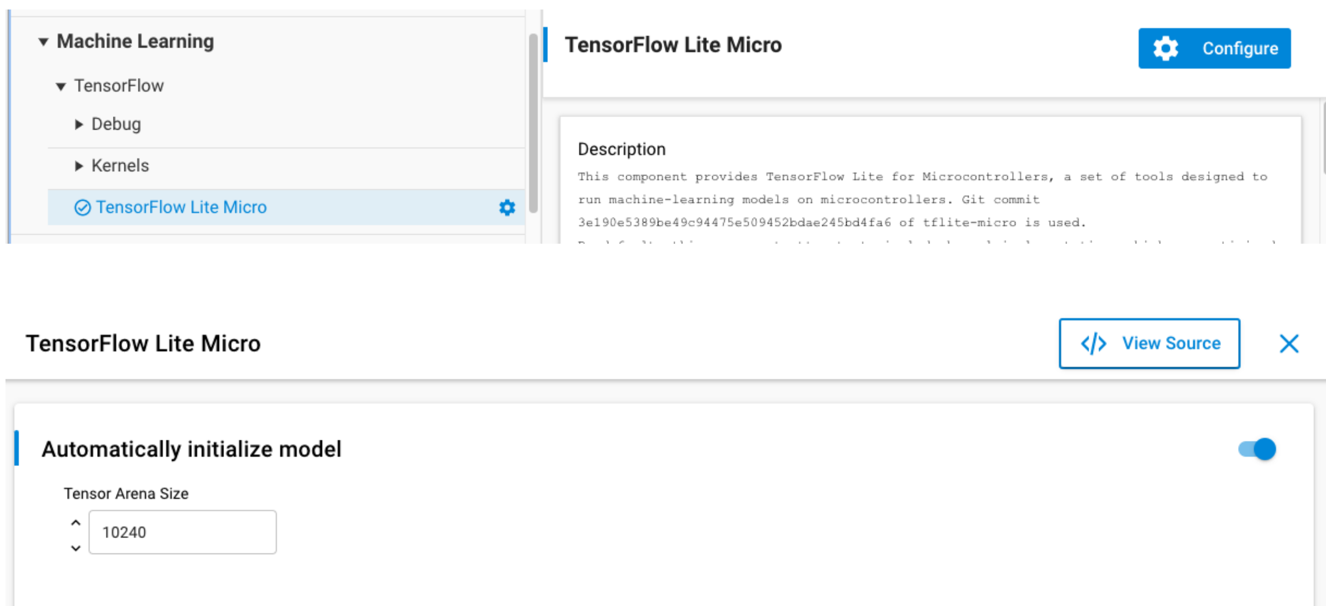
The TensorFlow framework is automatically initialized using the system initialization framework described in SDK Programming Model. This includes allocating a tensor arena, instantiating an interpreter and loading the model.

Configuration

If the model was produced using the Silicon Labs Machine Learning Toolkit (MLTK), it already contains metadata indicating the required size of the Tensor Arena, the memory area used by TensorFlow for runtime storage of input, output, and intermediate arrays. The required size of the arena depends on the model used.

If not using the MLTK, the arena size needs to be configured. This can be done in two ways:

- [Automatic] Set the arena size to -1, and it will attempt to automatically infer the size upon initialization.
- [Manual] Start with a large number during development, and reduce the allocation until initialization fails as part of size optimization.



Run the Model

Include the Silicon Labs TensorFlow Init API

```
#include "sl_tflite_micro_init.h"
```

For default behavior in bare metal application, it is recommended to run the model during `app_process_action()` in `app.cpp` to ensure that periodic inferences occur during the standard event loop. Running the model involves three stages:

Provide Input to the Interpreter

Sensor data is pre-processed (if necessary) and then is provided as input to the interpreter.

```
TfLiteTensor* input = sl_tflite_micro_get_input_tensor();
// stores 0.0 to the input tensor of the model
input->data.f[0] = 0.;
```

Run Inference

The interpreter is then invoked to run all layers of the model.

```
TfLiteStatus invoke_status = sl_tflite_micro_get_interpreter()->Invoke();
if (invoke_status != kTfLiteOk) {
TF_LITE_REPORT_ERROR(sl_tflite_micro_get_error_reporter(),
                    "Bad input tensor parameters in model");
return;
}
```

Read Output

The output prediction is read from the interpreter.

```
TfLiteTensor* output = sl_tflite_micro_get_output_tensor();
// Obtain the output value from the tensor
float value = output->data.f[0];
```

At this point, application-dependent behavior based on the output prediction should be performed. The application will run inference on each iteration of `app_process_action()` .

Full Code Snippet

After following the steps above, the resulting `app.cpp` now appears as follows:

```
#include "sl_tflite_micro_init.h"

/***************************************************************************//**
 * Initialize application.
 ******************************************************************************/
void app_init(void)
{
  // Init happens automatically
}

/***************************************************************************//**
 * App ticking function.
 ******************************************************************************/
void app_process_action(void)
{
  TfLiteTensor* input = sl_tflite_micro_get_input_tensor();
  // stores 0.0 to the input tensor of the model
  input->data.f[0] = 0.;

  TfLiteStatus invoke_status = sl_tflite_micro_get_interpreter()->Invoke();
  if (invoke_status != kTfLiteOk) {
  TF_LITE_REPORT_ERROR(sl_tflite_micro_get_error_reporter(),
                       "Bad input tensor parameters in model");
  return;
  }

  TfLiteTensor* output = sl_tflite_micro_get_output_tensor();
  // Obtain the output value from the tensor
  float value = output->data.f[0];
}
```

Addendum: Interfacing with C code

If your project is written in C rather than C++, place the code interfacing with TFLM into a separate file that exports a C API through an interface header. For this example, a filename `app_ml.cpp` is assumed that implements the function `ml_process_action()` with the same content as in the example above.

app_ml.h

```
#ifdef __cplusplus
extern "C" {
#endif

void ml_process_action(void);

#ifdef __cplusplus
}
#endif
```

app_ml.cpp

```
#include "app_ml.h"
#include "sl_tflite_micro_init.h"

extern "C" void ml_process_action(void)
{
  // ...
}
```

app.c

```
#include "app_ml.h
// ...
void app_process_action(void)
{
  ml_process_action();
}
```

Update or Replace a .tflite File

Updating or Replacing the .tflite File in a Project

This guide describes how to swap out the model in an existing project. It assumes that the project uses the Flatbuffer Converter Tool.

Replace the Model

To replace the model in an existing project, drag-and-drop the new model file into the `config/tflite/` directory of the project. If the new model has the same file name as the previous model, accept the prompt to overwrite the file. If the new model has a different name, delete or rename the old `.tflite` file such that it no longer has the `.tflite` extension. The Flatbuffer Converter Tool will automatically execute when the directory watcher notices that a different `.tflite` file is present.

Note: If you have multiple `.tflite` files in the `config/tflite/` directory, the converter tool will pick the first file in alphabetical order. Hence the recommendation to rename or delete any old models to ensure that the tool uses the correct file.

Inspect the Model

You can take steps to ensure that the automatic regeneration of the C arrays and headers from the `.tflite` file executed as expected. Any of the below steps can be taken if you're ever unsure of what model is part of your application binary.

Check the Model Size

The generated file `autogen/sl_tflite_micro_model.c` defines a size variable `sl_tflite_model_len`. This number can be compared to the file size of the `.tflite` file in bytes, e.g., by right-clicking the `.tflite` file and looking at the value of Properties > Resource > Size.

Check the Operators Used by the Model

The generated file `autogen/sl_tflite_micro_opcode_resolver.h` contains multiple calls to `tflite::MicroMutableOpResolver::AddXXX`, where `XXX` is the name of operators used by the model. This can be compared to the operators you know the model *should* use.

Check the Model Parameters

If the `.tflite` file was generated using the Silicon Labs Machine Learning Toolkit, it contains metadata that is generated into `autogen/sl_tflite_micro_model_parameters.h`. These parameters can be compared to the expected parameters.

Force Generation of C files

If anything happens that makes the generated `.c` and `.h` get out of sync with the `.tflite` file, the project can be forcefully regenerated by pressing the "Force Generation" button on the Project Details pane of the Project Configurator.

Tensorflow Lite Micro from Scratch

Machine Learning on Silicon Labs Devices from Scratch

This guide assumes familiarity with the content of the Getting Started Guide. It provides details of working with the TensorFlow API, as an alternative to the automatic initialization provided by Silicon Labs as described in the guide on Adding Machine Learning to a New Project.

Model Inclusion

With the TensorFlow Lite Micro component added in the Project Configurator, the next step is to load the model file into the project. To do this, copy the `.tflite` model file into the `config/tflite` directory of the project. The project configurator provides a tool that will automatically convert `.tflite` files into a `sl_tflite_micro_model` source and header files. The full documentation for this tool is available at Flatbuffer Converter Tool.

To do this step manually, a C array can be created from the `.tflite` using a tool such as `xxd` .

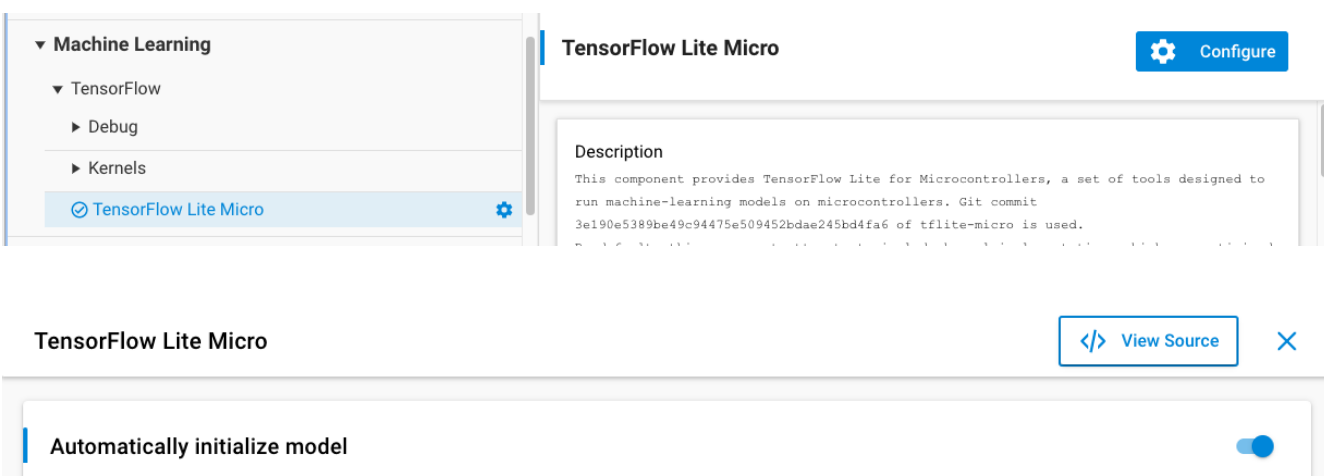
TFLM Initialization and Inference

To instantiate and use the TensorFlow APIs, follow the steps below to add TFLM functionality to the application layer of a project. This guide closely follows the TFLM Getting Started Guide, and is adapted for use with Silicon Labs' projects.

A special note needs to be taken regarding the operations used by TensorFlow. Operations are specific types of computations executed by a layer in the neural network. All operations may be included in a project at once, but doing this may increase the binary size dramatically (>100kB). The more efficient option is to only include the operations necessary to run a specific model. Both options are described in the steps below.

0. Disable the automatic initialization provided by the TensorFlow Lite Micro component

To manually set up TensorFlow, first disable automatic initialization of the model by disabling "Automatically initialize model" in the configuration header for the TensorFlow Lite Micro component (`sl_tflite_micro_config.h`).



1. Include the library headers

If using a custom, limited set of operations (recommended):

```
#include "tensorflow/lite/micro/micro_mutable_op_resolver.h"
#include "tensorflow/lite/micro/tflite_bridge/micro_error_reporter.h"
#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/schema/schema_generated.h"
#include "tensorflow/lite/version.h"
```

If using all operations:

```
#include "tensorflow/lite/micro/all_ops_resolver.h"
#include "tensorflow/lite/micro/micro_error_reporter.h"
#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/schema/schema_generated.h"
#include "tensorflow/lite/version.h"
```

2. Include the model header

Because the `autogen/` folder is always included in the project include paths, an imported model may be generically included in any project source file with:

```
#include "sl_tflite_micro_model.h"
```

If not using the Flatbuffer Converter Tool, include the file containing your model definition instead.

3. Define a Memory Arena

TensorFlow requires a memory arena for runtime storage of input, output, and intermediate arrays. This arena should be statically allocated, and the size of the arena depends on the model used. It is recommended to start with a large arena size during prototyping.

```
constexpr int tensor_arena_size = 10 * 1024;
uint8_t tensor_arena[tensor_arena_size];
```

Note: After prototyping, it is recommended to manually tune the memory arena size to the model used. After the model is finalized, start with a large arena size and incrementally decrease it until interpreter allocation (described below) fails.

4. Set up Logging

This should be performed even if the `Debug Logging Disabled` component is used. It is recommended to instantiate this statically and call the logger init functions during the `app_init()` sequence in `app.cpp`

```
static tflite::MicroErrorReporter micro_error_reporter;
tflite::ErrorReporter* error_reporter = &amp;micro_error_reporter;
```

5. Load the Model

Continuing during the `app_init()` sequence, the next step is to load the model into `tflite`:

```
const tflite::Model* model = ::tflite::GetModel(sl_tflite_model_array);
if (model->version() != TFLITE_SCHEMA_VERSION) {
  TF_LITE_REPORT_ERROR(error_reporter,
     "Model provided is schema version %d not equal "
     "to supported version %d.\n",
     model->version(), TFLITE_SCHEMA_VERSION);
}
```

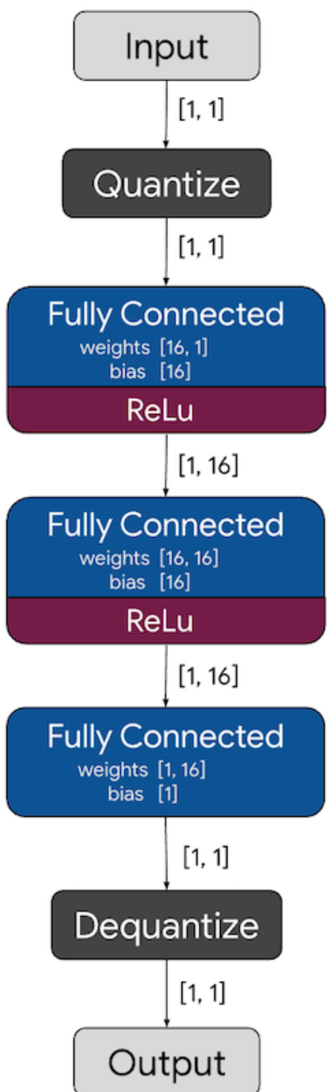
6. Instantiate the Operations Resolver

If using all operations, this is very straightforward. During `app_init()`, statically instantiate the resolver via:

```
static tflite::AllOpsResolver resolver;
```

Note: loading all operations will result in large increases to the binary size. It is recommended to use a custom set of operations.

If using a custom set of operations, a mutable ops resolver must be configured and initialized. This will vary based on the model and application. To determine the operations utilized in a given `.tflite` file, third party tools such as netron may be used to visualize the network and inspect which operations are in use.



Netron visualization from the TensorFlow Lite Micro hello world example

The example below loads the minimal operators required for the TensorFlow hello_world example model. As shown in the Netron visualization, this only requires fully connected layers:

```
#define NUM_OPS 1

static tflite::MicroMutableOpResolver<NUM_OPS> micro_op_resolver;
if (micro_op_resolver.AddFullyConnected() != kTfLiteOk) {
  return;
}
```

If using the Flatbuffer Converter Tool, it generates a C preprocessor macro that automatically sets up the optimal `tflite::MicroMutableOpResolver` for the a flatbuffer:

```
#include "sl_tflite_micro_opcode_resolver.h"

SL_TFLITE_MICRO_OPCODE_RESOLVER(micro_op_resolver, error_reporter);
```

7. Initialize the Interpreter

The final step during `app_init()` is to instantiate an interpreter and allocate buffers within the memory arena for the interpreter to use:

```
// static declaration
tflite::MicroInterpreter* interpreter = nullptr;

 // initialization in app_init
 tflite::MicroInterpreter interpreter(model, micro_op_resolver, tensor_arena,
                                      tensor_arena_size, error_reporter);
 interpreter = &interpreter_struct;
 TfLiteStatus allocate_status = interpreter.AllocateTensors();
 if (allocate_status != kTfLiteOk) {
   TF_LITE_REPORT_ERROR(error_reporter, "AllocateTensors() failed");
   return;
 }
```

The allocation will fail if the arena is too small to fit all the operations and buffers required by the model. Adjust the `tensor_arena_size` accordingly to resolve the issue.

8. Run the Model

For default behavior in bare metal application, it is recommended to run the model during `app_process_action()` in `app.cpp` in order for periodic inferences to occur during the standard event loop. Running the model involves three stages:

1. Sensor data is pre-processed (if necessary) and then is provided as input to the interpreter.

```
TfLiteTensor* input = interpreter.input(0);
// stores 0.0 to the input tensor of the model
input->data.f[0] = 0.;
```

It is important to match the shape of the incoming sensor data to the shape expected by the model. This can optionally be queried by checking properties defined in the `input` struct. An example of this for the hello_world example is shown below:

```
TfLiteTensor* input = interpreter->input(0);
if ((input->dims->size != 1) || (input->type != kTfLiteFloat32)) {
  TF_LITE_REPORT_ERROR(error_reporter,
                       "Bad input tensor parameters in model");
  return;
}
```

1. The interpreter is then invoked to run all layers of the model.

```
TfLiteStatus invoke_status = interpreter->Invoke();
if (invoke_status != kTfLiteOk) {
  TF_LITE_REPORT_ERROR(error_reporter, "Invoke failed on x_val: %f\n",
                       static_cast<double>(x_val));
  return;
}
```

1. The output prediction is read from the interpreter.

```
TfLiteTensor* output = interpreter->output(0);
// Obtain the output value from the tensor
float value = output->data.f[0];
```

At this point, application-dependent behavior based on the output prediction should be performed. The application will run inference on each iteration of `app_process_action()`.

Full Code Snippet

After following the steps above and choosing to use the mutable ops resolver, the resulting `app.cpp` now appears as follows:

```
#include "tensorflow/lite/micro/micro_mutable_op_resolver.h"
#include "tensorflow/lite/micro/micro_error_reporter.h"
#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/schema/schema_generated.h"
#include "tensorflow/lite/version.h"

#include "sl_tflite_micro_model.h"

#define NUM_OPS 1

constexpr int tensor_arena_size = 10 * 1024;
uint8_t tensor_arena[tensor_arena_size];

tflite::MicroInterpreter* interpreter = nullptr;

/***************************************************************************//**
 * Initialize application.
 ******************************************************************************/
void app_init(void)
{
  static tflite::MicroErrorReporter micro_error_reporter;
  tflite::ErrorReporter* error_reporter = &micro_error_reporter;

  const tflite::Model* model = ::tflite::GetModel(g_model);
  if (model->version() != TFLITE_SCHEMA_VERSION) {
   TF_LITE_REPORT_ERROR(error_reporter,
       "Model provided is schema version %d not equal "
       "to supported version %d.\n",
       model->version(), TFLITE_SCHEMA_VERSION);
  }



  static tflite::MicroMutableOpResolver<NUM_OPS> micro_op_resolver;
  if (micro_op_resolver.AddFullyConnected() != kTfLiteOk) {
     return;
  }

  static tflite::MicroInterpreter interpreter_struct(model, micro_op_resolver, tensor_arena,
                                   tensor_arena_size, error_reporter);
  interpreter = &interpreter_struct;
  TfLiteStatus allocate_status = interpreter.AllocateTensors();
  if (allocate_status != kTfLiteOk) {
   TF_LITE_REPORT_ERROR(error_reporter, "AllocateTensors() failed");
   return;
  }
}

/***************************************************************************//**
 * App ticking function.
 ******************************************************************************/
void app_process_action(void)
{
  // stores 0.0 to the input tensor of the model
  TfLiteTensor* input = interpreter->input(0);
  input->data.f[0] = 0.;

  TfLiteStatus invoke_status = interpreter->Invoke();
  if (invoke_status != kTfLiteOk) {
   TF_LITE_REPORT_ERROR(error_reporter, "Invoke failed on x_val: %f\n",
                   static_cast<double>(x_val));
   return;
  }

  TfLiteTensor* output = interpreter->output(0);
  float value = output->data.f[0];
}
```

Examples

As described in the Sample Application Overview, examples developed by the TensorFlow team demonstrating the `hello_world` example described in this guide, as well as a simple speech recognition example `micro_speech`, are included in the Gecko SDK.

Note that the `micro_speech` example demonstrates use of the `MicroMutableOpResolver` to only load required operations.

Developing a Model

Developing a Machine Learning Model

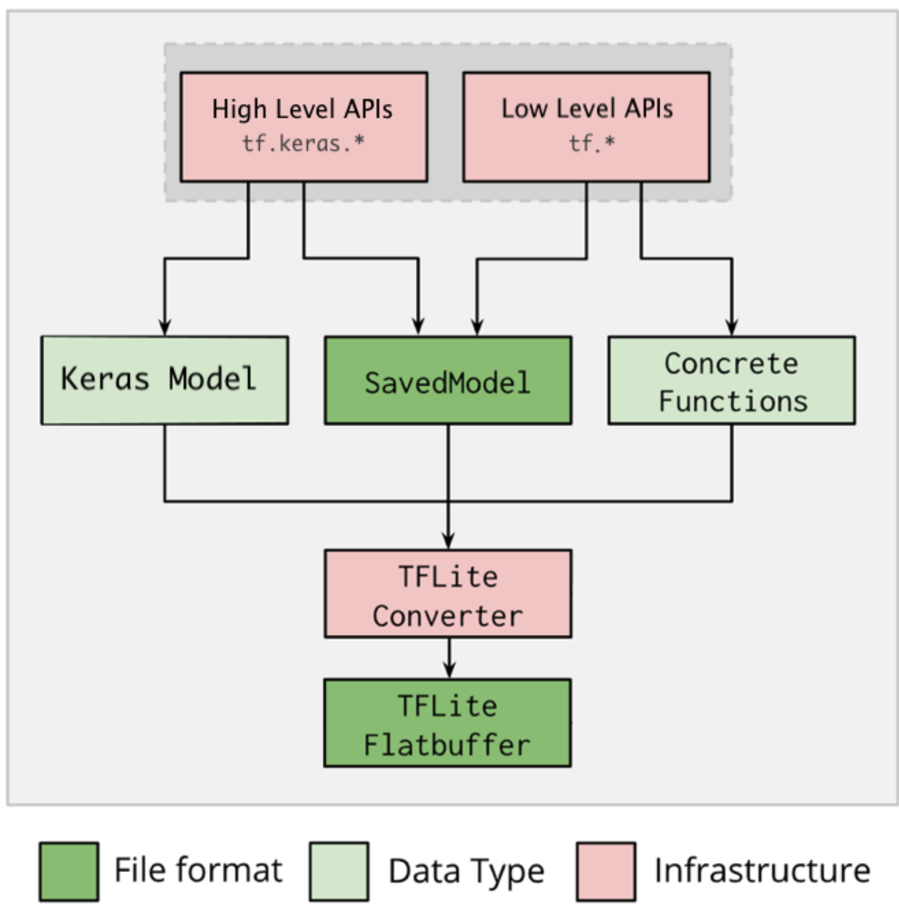
Developing a Model using the MLTK

The Silicon Labs Machine Learning Toolkit (MLTK) is a Python package that implements a layer above TensorFlow to help the TensorFlow developer build models that can be successfully deployed on Silicon Labs chips. These scripts are a reference implementation for the audio use case, which includes the use of the Audio Feature Generator on both the training and inference side. This is modified version of the TensorFlow "microfrontend" audio front end. It is expected that the MLTK is used by an ML Expert with deep knowledge of TensorFlow and Python, or by a developer willing to learn.

The MLTK is offered as a self-serve, self-support, fully documented, Python reference package published through GitHub. We are delivering this as an Experimental package – which means it's available "as-is", un-tested, and without support.

See the MLTK documentation for more information.

Developing a Model Manually using TensorFlow and Keras



When developing and training neural networks for use in embedded systems, it is important to note the limitations on TFLM that apply to model architecture and training. Embedded platforms also have significant performance constraints that must be considered when designing and evaluating a model. The embedded TLFM documentation links describe these limitations and considerations in detail.

Additionally, the TensorFlow Software Components in Studio require a quantized `*.tflite` representation of the trained model. As a result, TensorFlow and Keras are the recommended platforms for model development and training because both platforms are supported by the TensorFlow Lite Converter that generates `.tflite` model representations.

Both TensorFlow and Keras provide guides on model development and training:

- TensorFlow Basic Training Loops
- Keras Training and Evaluation

After a model has been created and trained in TensorFlow or Keras, it needs to be converted and serialized into a `*.tflite` file. During model conversion, it is important to optimize the memory usage of the model by quantizing it. It is highly recommended to use integer quantization on Silicon Labs devices.

- TensorFlow Lite Converter
- Quantization Overview

A complete example demonstrating the training, conversion, and quantization of a simple TFLM compatible neural network is available from TensorFlow:

- TensorFlow Hello World Training Example
- Trained Hello World Model

Tensorflow Lite for Microcontrollers

TensorFlow Lite for Microcontrollers

TensorFlow Lite for Microcontrollers is a framework that provides a set of tools for running neural network inference on microcontrollers. It contains a wide selection of kernel operators with good support for 8-bit integer quantized networks. The framework is limited to model inference and does not support training. For information about how to train a neural network, see the Silicon Labs Machine Learning Toolkit (MLTK).

Silicon Labs provides an integration of TensorFlow Lite for Microcontrollers with the Gecko SDK. This overview gives an overview of the integration. See the Getting Started Guides for step-by-step instructions on how to make use of Machine Learning in your project.

SDK Component Overview

The software components required to use TensorFlow Lite for Microcontrollers can be found under `Machine Learning > TensorFlow` in the software component browser UI in the Simplicity Studio project configurator.

TensorFlow Lite Micro

This component contains the full TensorFlow Lite for Microcontrollers framework, and automatically pulls in the most optimal implementation of kernels for the device selected for the project by default. To use TensorFlow Lite Micro, this component is the only one that needs to be explicitly installed. It is however possible to manually install different kernel implementations if so desired, for instance to compare inference performance or code size, and to manually install a different debug logging implementation.

By default, the TensorFlow Lite Micro component makes use of the Flatbuffer Converter Tool to convert a `.tflite` file into a C array and to initialize this neural network model automatically. See the section on automatic initialization for more details.

Kernel Implementations

Reference Kernels

This component provides unoptimized software implementations of all kernels. This is a default implementation that is designed to be easy to read and can run on any platform. As a result, these kernels may run more slowly than an optimal implementation.

CMSIS-NN Optimized Kernels

Some kernels have implementations that have been optimized for certain CPU architectures using the CMSIS-NN library. Using these kernels when available can improve inference performance significantly. By enabling this component, the available optimized kernel implementations are added to the project, replacing the corresponding reference kernel implementations. The remaining kernels fall back to using the reference implementations by depending on the reference kernel component.

MVP Accelerated Kernels

Some kernels have implementations optimized for the MVP accelerator available on select Silicon Labs parts. Using these kernels will improve inference performance. By enabling this component, the available accelerated kernel implementations are added to the project, replacing the corresponding optimized or reference kernel implementations. The remaining kernels fall back to use the optimized or reference implementations by depending on the corresponding components. See more details about the accelerator to learn what kernels are supported, and what constraints apply.

Debug Logging using I/O Stream / Disabled

Debug logging is used in TensorFlow to display debug and error information. Additionally, it can be used to display inference results. Debug logging is enabled by default, with an implementation that uses I/O Stream to print over UART to the virtual COM port on development kits (VCOM). Logging can be disabled by ensuring that the component "Debug Logging Disabled" is included in the project.

TensorFlow Third Party Dependencies

A specific version of the CMSIS-NN library is used as with TensorFlow Lite for Microcontrollers to optimize certain kernels. This library is included in the project together with TensorFlow Lite for Microcontrollers. TensorFlow depends on a bleeding-edge version of CMSIS-NN, while the rest of the Gecko SDK uses a stable CMSIS release. It is strongly recommended to avoid using functions from the Gecko SDK version of CMSIS-DSP and CMSIS-NN elsewhere in the project and instead use the version bundled with TensorFlow Lite for Microcontrollers to avoid versioning conflicts between the two.

Audio Feature Generator

The audio feature generator can be used to extract time-frequency features from an audio signal for use with machine learning (ML) audio classification applications. The generated feature array is a mel-scaled spectrogram, representing the frequency information of the signal of a given sample length of audio.

When used together with the Flatbuffer Converter Tool, the audio feature generator by default consumes its configuration settings from the model parameters of the `.tflite` flatbuffer. Such metadata can be added to the flatbuffer by using the Silicon Labs Machine Learning Toolkit. This ensures that the settings used during inference on the embedded device match the settings used during training. If models without such metadata are used, the configuration option "Enable Manual Frontend Configurations" can be enabled, and configuration values set in the configuration header `sl_ml_audio_feature_generation_config.h`.

Automatic Initialization of Default Model

When the TensorFlow Lite Micro component is added to the project, it will by default attempt to automatically initialize a default model using the TFLite Micro Init API. It performs initialization of TensorFlow Lite Micro by creating an opcode resolver and interpreter for the given flatbuffer. In addition, it creates the tensor arena buffer.

The model used by the automatic initialization code comes from the Flatbuffer Converter Tool. If the flatbuffer was produced using the MLTK, it may contain metadata about the necessary tensor arena size. If such information is present, it will be automatically initialized to the correct size. If a non-MLTK flatbuffer is used, the tensor arena size must be configured manually using the configuration file for the TensorFlow Lite Micro component.

If automatic initialization at startup is not desired, this can be turned off using the **Automatically initialize model** (`SL_TFLITE_MICRO_INTERPRETER_INIT_ENABLE`) configuration option.

Version

The Gecko SDK incorporates TensorFlow Lite for Microcontrollers version #3e7d66cc86c12a96edad58954ed35944b9bc4dc3 in `util/third_party/tflite-micro/`. The core TensorFlow Lite for Microcontrollers offering is unpatched, all additional content for Silicon Labs devices is delivered in the `util/third_party/tensorflow_extra/` directory.

Third-party Tools and Partners

Tools

- Netron is a visualization tool for neural networks, compatible with `.tflite` model files. This is useful for viewing the operations used in a model, the sizes of tensors and kernels, etc.

AI/ML Partners

Silicon Labs AI/ML partners provide expertise and platforms for data collection, model development, and training. See the technology partner pages to learn more.

Flatbuffer Converter Tool

Flatbuffer Converter Tool

The Flatbuffer Converter Tool helps to convert a `.tflite` file into a C array that can be compiled into a binary for an embedded system. This array can be used with the TensorFlow Lite for Microcontrollers API, which takes a void pointer to a buffer containing the model as an argument to its `tflite::GetModel()` init function.

In addition to converting the flatbuffer into a C array, the tool supports emitting model parameters embedded as metadata in the `.tflite` file as C preprocessor macros.

Input

The tool takes a directory containing one or more `.tflite` files as input. If the directory consists of multiple files, only the first file in alphabetical order is converted.

Tip: If you have multiple files in the directory, but the one you want to convert isn't the first file in alphabetical order, you can rename the other files to add a `.bak` extension or rename the target file accordingly.

Output

The tool writes its output into multiple files in a single output directory.

Model Array

The tool always emits a pair of files `sl_tflite_micro_model.c` / `.h`, which declares the variables as follows

- `const uint8_t sl_tflite_model_array[]` containing the full contents of the `.tflite` file
- `const uint32_t sl_tflite_model_len` containing the length of the model array

Opcode Resolver

A header file `sl_tflite_micro_opcode_resolver.h` is also emitted. This file declares a C preprocessor macro `SL_TFLITE_MICRO_OPCODE_RESOLVER(opcode_resolver, error_reporter)` that instantiates a `tflite::MicroMutableOpResolver` object and automatically registers the set of operators required to parse the model.

This macro can be used as part of an initialization sequence to automatically initializing the optimal opcode resolver.

Model Parameters

If the `.tflite` file contains model parameters in its metadata section, a third header file `sl_tflite_micro_model_parameters.h` is emitted.

For every model parameter key-value pair, a C preprocessor macro `SL_TFLITE_MODEL_<key>` is created with the relevant value.

See the MLTK documentation to learn more about embedding model parameters in the `.tflite` file.

SLC Project Configuration Integration

The Flatbuffer Converter Tool integrates with the SLC project configuration tools in Simplicty Studio and on the command line using SLC-CLI. When using these tools, the Flatbuffer Converter is automatically run with the `config/tflite/` directory of the project as input, and the `autogen/` directory as output.

In other words, any file with the `.tflite` extension in the `config/tflite/` directory in the project will be automatically converted when the project is generated. If using Simplicity Studio, a directory watcher ensures that the project is automatically

generated if a `.tflite` file is added or removed. This means that it is sufficient to drag-and-drop a `.tflite` file into the project, and it is automatically converted into C code.

Manual Usage

If conversion is desired outside of a full SLC project generation cycle, the flatbuffer converter can be invoked manually. This is generally not necessary, but is an option for advanced users. This can only be done using the SLC-CLI, users of Simplicity Studio are recommended to regenerate the full project.

As SLC Project Generator

SLC-CLI supports running a single project generation tool by passing the `-tools` to the `slc generate` command.

The Flatbuffer Converter step of the project generation process can be run standalone by running the following command:

```
slc generate -p <my_project.slcp> -tools tflite
```

As Standalone SLC-CLI Command

The flatbuffer converter is also available as a standalone command in SLC-CLI for use outside of the context of a project. Execute the following command using absolute paths for both input and output directories.

```
slc tflite generate -contentFolder [/path/to/config/tflite] -generationOutput [/path/to/autogen]
```

MVP Accelerator

MVP Accelerator

The MVP accelerator is a co-processor designed to perform matrix and vector operations. Using hardware accelerated kernel implementations will reduce neural network inference time, as well as off-load the main processor to allow it to perform other tasks or go to sleep.

Silicon Labs has implemented common neural network operators as programs to be executed on the MVP and integrated these with TensorFlow Lite for Microcontrollers. The MVP has 5 array controllers, each of which can support iterating in 3 independent dimensions. Each dimension is limited to 1024 elements, with a stride between each element of 2047. The limiting factor for most neural network operations is therefore the product of the width and depth dimensions, since this becomes the stride in the height dimension.

All MVP-accelerated operations take signed 8-bit integers as input and output. If the inner dimension of the tensor has even size, each element can contain two int8 values, interpreted as a single complex int8 value by the accelerator. The accelerator can then effectively support 2048 int8 values. If the inner dimension is odd, the accelerator must perform one computation at a time, which reduces performance and limits the dimension size to 1024 int8 values.

The operators listed below will be accelerated using the MVP if tensor sizes allow. If a specific tensor cannot be accelerated, the implementation will automatically fall back to using optimized (CMSIS-NN) or reference kernel implementations at runtime. To maximize the likelihood that an operator is supported by the accelerator, use even-valued numbers of channels when designing the model.

Internally, the MVP accelerator uses 16-bit floating point math, even when taking 8-bit integers as input. This means that there is a slight reduction in accuracy of computations, which may be especially noticeable when performing operations that accumulate many elements.

For more information about the MVP hardware accelerator, see the reference manual for EFR32xG24.

Accelerated TensorFlow operators

Add

TensorFlow operator name: `ADD`

Any tensor size is supported.

FullyConnected (Dense)

TensorFlow operator name: `FULLY_CONNECTED` , `FULLY_CONNECTED_INT8`

Supports tensors where all dimensions are within the 1024 element limit. Also supports larger tensors where the size of the last dimension is decomposable into two factors that are both within the 1024 element limit.

AveragePool2D

TensorFlow operator name: `AVERAGE_POOL_2D`

Supports tensors where `width*channels` is within the 2047 element stride limit and all dimensions are within the 1024 element limit.

MaxPool2D

TensorFlow operator name: `MAX_POOL_2D`

Supports tensors where `width*channels` is within the 2047 element stride limit and all dimensions are within the 1024 element limit.

Conv2D

TensorFlow operator name: `CONV_2D`

Supports tensors where `width*channels` is within the 2047 element stride limit and all dimensions are within the 1024 element limit.

DepthwiseConv2D

TensorFlow operator name: `DEPTHWISE_CONV_2D`

Supports tensors where `width*channels` is within the 2047 element stride limit and all dimensions are within the 1024 element limit. Dilation is not supported.

TransposeConv2D

TensorFlow operator name: `TRANSPOSE_CONV_2D`

Supports tensors where `width*channels` is within the 2047 element stride limit and all dimensions are within the 1024 element limit. Dilation is not supported.

Suspending Execution While Waiting for Accelerator

The software API for the MVP accelerator is blocking, meaning that any call to the MVP driver will wait for completion before returning from the function call. To save energy, the driver can optionally suspend execution of the main processor while waiting for the accelerator to complete an operation. By default, the main processor busy-waits for the accelerator.

No sleep (0)

When the "No sleep" option is used, the MCU core will busy-wait for the MVP to finish. This is the option which provides the fastest MVP execution time. The "No sleep" option can be used in a bare metal application or an application using a real-time operating system (RTOS).

Enter EM1 (1)

When the "Enter EM1" option is used, the MCU will be put into Energy Mode 1 whenever the driver waits for an MVP program to complete. The "Enter EM1" option is not safe to use in an application using RTOS because it will prevent proper RTOS scheduling.

Yield RTOS thread (2)

When the "Yield RTOS thread" option is used, the task waiting for the MVP program to complete will yield, allowing other tasks in the system to run or potentially let the scheduler put the system into a sleep mode. The "Yield RTOS thread" requires that the application is using an RTOS.

The power mode of the MVP driver can be configured by setting the `SL_MVP_POWER_MODE` configuration option in the `sl_mvp_config.h` configuration header.

Sample Applications

Sample Applications

The following applications demonstrate the use of the TensorFlow Lite for Microcontrollers framework with the Gecko SDK.

Voice Control Light

This application demonstrates a neural network with TensorFlow Lite for Microcontrollers to detect the spoken words "on" and "off" from audio data recorded on the microphone in a Micrium OS kernel task.

The detected keywords are used to control an LED on the board. The audio data is sampled continuously and preprocessed using the Audio Feature Generator component. Inference is run every 200 ms on the past ~1 s of audio data.

This sample application uses the Flatbuffer Converter Tool to add the .tflite file to the application binary.

Z3SwitchWithVoice

This application combines voice detection with Zigbee 3.0 to create a voice-controlled switch node that can be used to toggle a light node. The application uses the same model as Voice Control Light to detect the spoken keywords "on" and "off". Upon detection, the switch node sends On/Off commands over the Zigbee network.

This sample application uses the Flatbuffer Converter Tool to add the .tflite file to the application binary.

TensorFlow Lite Micro - Hello World

This application demonstrates a model trained to replicate a sine function and use the inference results to fade an LED. The application is originally written by TensorFlow, but has been ported to the Gecko SDK.

The model is approximately 2.5 KB. The entire application takes around 157 KB flash and 15 KB RAM. This application uses large amounts of flash memory because it does not manually specify which operations are used in the model and, as a result, compiles all kernel implementations.

The application illustrates a minimal inference application and serves as a good starting point for understanding the TensorFlow Lite for Microcontrollers model interpretation flow.

This sample application uses a fixed model contained in `hello_world_model_data.cc`.

TensorFlow Lite Micro - Micro Speech

This application demonstrates a 20 KB model trained to detect simple words from speech data recorded from a microphone. The application is originally written by TensorFlow, but has been ported to the Gecko SDK.

This application uses around 100 KB flash and 37 KB of RAM. Around 10 KB of the RAM usage is related to FFT frontend and to store audio data. With a clock speed of 38.4 MHz and using the optimized kernel implementations, the inference time on ~1 s of audio data is approximately 111 ms.

This application illustrates the process of generating features from audio data and doing detections in real time. It also demonstrates how to manually specify which operations are used in the network, which saves a significant amount of flash.

This sample application uses a fixed model contained in `micro_speech_model_data.cc`.

TensorFlow Lite Micro - Magic Wand

This application demonstrates a 10 KB model trained to recognize various hand gestures using an accelerometer to detect the motion. The detected gestures are printed to the serial port. The application is originally written by TensorFlow, but has

been ported to the Gecko SDK.

This application uses around 104 KB flash and 25 KB of RAM. This application demonstrates how to use accelerometer data as inference input and also shows how to manually specify which operations are used in the network, which saves a significant amount of flash.

This sample application uses the Flatbuffer Converter Tool to add the .tflite file to the application binary.

TensorFlow Model Profiler

This application is designed to profile a TensorFlow Lite Micro model on Silicon Labs hardware. The model used by the application is provided by a TensorFlow Lite flatbuffer file called model.tflite in the config/tflite subdirectory. The profiler will measure the number of CPU clock cycles and elapsed time in each layer of the model when performing an inference. It will also produce a summary when inference is done. The input layer of the model is filled with all zeroes before performing a single inference. Profiling results are transmitted over VCOM.

To run the application with a different .tflite model, you can replace the file called model.tflite with a new TensorFlow Lite Micro flatbuffer file. This new file must also be called "model.tflite" and be placed inside the config/tflite subdirectory to be picked up by the sample application. After the model has been replaced, regenerate the project.

To load and perform inference on a TensorFlow Lite Micro model, allocate a number of bytes to a "tensor arena" to hold state needed by the TensorFlow Lite Micro. The size of this tensor arena depends on the size of the model and the number of operators. The TensorFlow Model Profiler application can be used to measure the amount of RAM needed by the tensor arena to load the specific TensorFlow Lite Micro model. This is measured by dynamically allocating RAM for the tensor arena and reporting the number of bytes needed on VCOM. The number of bytes needed for the tensor arena can later be used to statically allocate memory when the model is used in a different application.

This sample application uses the Flatbuffer Converter Tool to add the .tflite file to the application binary.

API Documentation

API Documentation

List of modules	Description
TensorFlow Lite Micro Init	Initialize the Tensorflow Lite Micro Runtime.
TensorFlow Lite Micro Debug	Additional SL utilities for logging in TensorFlow Lite Micro.
Audio Feature Generator	Extracts mel-filterbank features from an audio signal to use with machine learning audio classification applications.

Audio Feature Generator

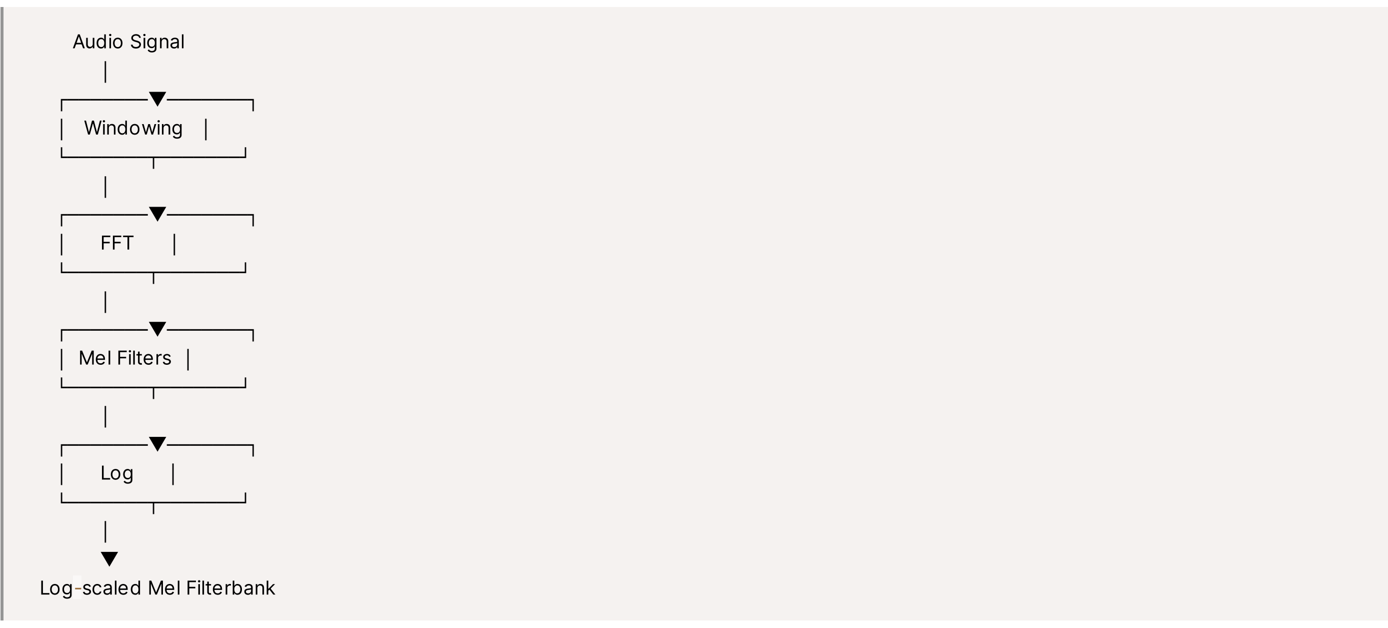
Audio Feature Generator

The audio feature generator extracts mel-filterbank features from an audio signal to use with machine learning audio classification applications using a microphone as an audio source.

Feature Generation

The Mel scale replicates the behavior of the human ear, which has a higher resolution for lower frequencies and is less discriminative of the higher frequencies. To create a mel filterbank, a number of filters are applied to the signal, where the pass-band of the lower channel filters is narrow and increases towards higher frequencies.

The audio signal is split into short overlapping segments using a window function (Hamming). The Fast Fourier Transform (FFT) is applied to each segment to retrieve the frequency spectrum and then the power spectrum of the segment. The filterbank is created by applying a series of mel-scaled filters to the output. Finally, the log is applied to the output to increase the sensitivity between the lower channels.



The feature array is generated by stacking filterbanks of sequential segments together to form a spectrogram. The array is sorted such that the first element is the first channel of the oldest filterbank.

Usage

sl_ml_audio_feature_generation_init() initializes the frontend for feature generation based on the configuration in sl_ml_audio_feature_generation_config.h. It also initializes and starts the microphone in streaming mode, which places the audio samples into a ring-buffer.

If used together with the Flatbuffer Converter Tool and a compatible TensorFlow Lite model, the configuration is pulled from the TensorFlow Lite model by default. Set the configuration option `SL_ML_AUDIO_FEATURE_GENERATION_MANUAL_CONFIG_ENABLE` to override this behavior and use manually-configured options from the configuration header.

The features are generated when sl_ml_audio_feature_generation_update_features() is called. The feature generator then updates the features for as many new segments of audio as possible, starting from the last time the function was called up

until the current time. The new features are appended to the feature buffer, replacing the oldest features such that the feature array always contains the most up to date features.

Note that if the audio buffer is not large enough to hold all audio samples required to generate features between calls to sl_ml_audio_feature_generation_update_features(), audio data will simply be overwritten. The generator will not return an error. The audio buffer must therefore be configured to be large enough to store all new sampled data between updating features.

To retrieve the generated features, either sl_ml_audio_feature_generation_get_features_raw(), sl_ml_audio_feature_generation_get_features_quantized(), or sl_ml_audio_feature_generation_fill_tensor() must be called.

Example

When used with TensorFlow Lite Micro, the audio feature generator can be used to fill a tensor directly by using sl_ml_audio_feature_generation_fill_tensor(). However, the model has to be trained using the same feature generator configurations as used for inference, configured in sl_ml_audio_feature_generation_config.h.

```
#include "sl_tflite_micro_init.h"
#include "sl_ml_audio_feature_generation.h"

void main(void)
{
 sl_ml_audio_feature_generation_init();

 while(1){
  sl_ml_audio_feature_generation_update_features();

  if(do_inference){
   sl_ml_audio_feature_generation_fill_tensor(sl_tflite_micro_get_input_tensor());
   sl_tflite_micro_get_interpreter()->Invoke();
  }

  ...

 }
}
```

Note that updating features and retrieving them can be performed independently. Updating features should be done often enough to avoid overwriting the audio buffer while retrieving them only needs to be done prior to inference.

Functions

sl_status_t	sl_ml_audio_feature_generation_init() Set up the microphone as an audio source for feature generation and initialize the frontend for feature generation.
sl_status_t	sl_ml_audio_feature_generation_frontend_init() Initialize microfrontend according to the configuration in sl_ml_audio_feature_generation_config.h.
sl_status_t	sl_ml_audio_feature_generation_update_features() Update the feature buffer with the missing feature slices since the last call to this function.
sl_status_t	sl_ml_audio_feature_generation_get_features_raw(uint16_t *buffer, size_t num_elements) Retrieve the features as type uint16 and copy them to the provided buffer.
sl_status_t	sl_ml_audio_feature_generation_fill_tensor(TfLiteTensor *input_tensor) Fill a TensorFlow tensor with feature data of type int8.
int	sl_ml_audio_feature_generation_get_new_feature_slice_count() Return the number of new or unfetched feature slices that have been updated since the last call to sl_ml_audio_feature_generation_get_features_raw or sl_ml_audio_feature_generation_fill_tensor.

int sl_ml_audio_feature_generation_get_feature_buffer_size()
Return the feature buffer size.

void sl_ml_audio_feature_generation_reset()
Reset the state of the audio feature generator.

Function Documentation

sl_ml_audio_feature_generation_init

```
sl_status_t sl_ml_audio_feature_generation_init ()
```

Set up the microphone as an audio source for feature generation and initialize the frontend for feature generation.

Returns

- SL_STATUS_OK for success SL_STATUS_FAIL

Definition at line `171` of file `util/third_party/tensorflow_extra/inc/sl_ml_audio_feature_generation.h`

sl_ml_audio_feature_generation_frontend_init

```
sl_status_t sl_ml_audio_feature_generation_frontend_init ()
```

Initialize microfrontend according to the configuration in sl_ml_audio_feature_generation_config.h.

Returns

- SL_STATUS_OK for success SL_STATUS_FAIL

Definition at line `182` of file `util/third_party/tensorflow_extra/inc/sl_ml_audio_feature_generation.h`

sl_ml_audio_feature_generation_update_features

```
sl_status_t sl_ml_audio_feature_generation_update_features ()
```

Update the feature buffer with the missing feature slices since the last call to this function.

To retrieve the features, call sl_ml_audio_feature_generation_get_features_raw or sl_ml_audio_feature_generation_fill_tensor.

Note

- This function needs to be called often enough to ensure that the audio buffer isn't overwritten.

Returns

- SL_STATUS_OK for success SL_STATUS_EMPTY No new slices were calculated

Definition at line `219` of file `util/third_party/tensorflow_extra/inc/sl_ml_audio_feature_generation.h`

sl_ml_audio_feature_generation_get_features_raw

```
sl_status_t sl_ml_audio_feature_generation_get_features_raw (uint16_t *buffer, size_t num_elements)
```

Retrieve the features as type uint16 and copy them to the provided buffer.

Parameters

[out]	buffer	Pointer to the buffer to store the feature data
[in]	num_elements	The number of elements corresponding to the size of the buffer; If this is not large enough to store the entire feature buffer the function will return with an error.

Note

- This function overwrites the entire buffer.

Returns

- SL_STATUS_OK for success SL_STATUS_INVALID_PARAMETER num_elements too small

Definition at line `240` of file `util/third_party/tensorflow_extra/inc/sl_ml_audio_feature_generation.h`

sl_ml_audio_feature_generation_fill_tensor

```
sl_status_t sl_ml_audio_feature_generation_fill_tensor (TfLiteTensor *input_tensor)
```

Fill a TensorFlow tensor with feature data of type int8.

Parameters

[in]	input_tensor	The input tensor to fill with features.

The int8 values are derived by quantizing the microfrontend output, expected to be in the range 0 to 670, to signed integer numbers in -128 to 127 range.

Note

- This function overwrites the entire input tensor.
- Supports tensors of type kTfLiteInt8.

Returns

- SL_STATUS_OK for success SL_STATUS_INVALID_PARAMETER Tensor type or size does not correspond with configuration

Definition at line `344` of file `util/third_party/tensorflow_extra/inc/sl_ml_audio_feature_generation.h`

sl_ml_audio_feature_generation_get_new_feature_slice_count

```
int sl_ml_audio_feature_generation_get_new_feature_slice_count ()
```

Return the number of new or unfetched feature slices that have been updated since the last call to sl_ml_audio_feature_generation_get_features_raw or sl_ml_audio_feature_generation_fill_tensor.

Returns

- The number of unfetched feature slices

Definition at line `356` of file `util/third_party/tensorflow_extra/inc/sl_ml_audio_feature_generation.h`

sl_ml_audio_feature_generation_get_feature_buffer_size

```
int sl_ml_audio_feature_generation_get_feature_buffer_size ()
```

Return the feature buffer size.

Returns

- Size of the feature buffer

Definition at line `365` of file `util/third_party/tensorflow_extra/inc/sl_ml_audio_feature_generation.h`

sl_ml_audio_feature_generation_reset

```
void sl_ml_audio_feature_generation_reset ()
```

Reset the state of the audio feature generator.

Definition at line `371` of file `util/third_party/tensorflow_extra/inc/sl_ml_audio_feature_generation.h`

TensorFlow Lite Micro Debug

TensorFlow Lite Micro Debug

Additional SL utilities for logging in TensorFlow Lite Micro.

Functions

void	sl_tflite_micro_enable_debug_log(bool enable) Enable or disable debug logging.
bool	sl_tflite_micro_is_debug_log_enabled(void) Check if debug logging is enabled.

Function Documentation

sl_tflite_micro_enable_debug_log

```
void sl_tflite_micro_enable_debug_log (bool enable)
```

Enable or disable debug logging.

Parameters

[in]	enable	Whether the debug logging should be enabled or disabled.

Definition at line `47` of file `util/third_party/tensorflow_extra/inc/sl_tflite_micro_debug_log.h`

sl_tflite_micro_is_debug_log_enabled

```
bool sl_tflite_micro_is_debug_log_enabled (void)
```

Check if debug logging is enabled.

Parameters

N/A		

Returns

- Whether debug logging is enabled.

Definition at line `54` of file `util/third_party/tensorflow_extra/inc/sl_tflite_micro_debug_log.h`

TensorFlow Lite Micro Init

TensorFlow Lite Micro Init

The TensorFlow Lite Micro Init functions are autogenerated from the flatbuffer file provided in the configuration, which includes an opsResolver generated according to the included operators in the flatbuffer.

Helper functions to access the input and output tensors are also provided.

Functions

bool	sl_tflite_micro_estimate_arena_size(const tflite::Model *model, const tflite::MicroOpResolver &opcode_resolver, size_t *estimated_size) Estimate the arena size for a given model.
uint8_t *	sl_tflite_micro_allocate_tensor_arena(size_t arena_size, uint8_t **tensor_arena) Dynamically allocate a buffer that can be used for the tensor arena.
tflite::ErrorReporter *	sl_tflite_micro_get_error_reporter() Get a pointer to the TensorFlow Lite Micro error reporter created by the init function.
tflite::MicroInterpreter *	sl_tflite_micro_get_interpreter() Get a pointer to the TensorFlow Lite Micro interpreter created by the init function.
TfLiteTensor *	sl_tflite_micro_get_input_tensor() Get a pointer to the input tensor, set by the init function.
TfLiteTensor *	sl_tflite_micro_get_output_tensor() Get a pointer to the output tensor, set by the init function.
tflite::MicroOpResolver &	sl_tflite_micro_opcode_resolver() Get a pointer to the opcode resolver for the flatbuffer given by the configuration.
void	sl_tflite_micro_init(void) Create the error reporter and opcode resolver and initialize variables for the flatbuffer given by the configuration.

Function Documentation

sl_tflite_micro_estimate_arena_size

```
bool sl_tflite_micro_estimate_arena_size (const tflite::Model *model, const tflite::MicroOpResolver &opcode_resolver, size_t *estimated_size)
```

Estimate the arena size for a given model.

Parameters

[in]	model	Pointer to the model to estimate the arena size for.
[in]	opcode_resolver	The opcode resolver to use for the model.
[out]	estimated_size	The estimated size of the arena, as output.

Returns

- True if the estimation was successful, false otherwise.

Definition at line 57 of file util/third_party/tensorflow_extra/inc/sl_tflite_micro_init.h

sl_tflite_micro_allocate_tensor_arena

```
uint8_t * sl_tflite_micro_allocate_tensor_arena (size_t arena_size, uint8_t **tensor_arena)
```

Dynamically allocate a buffer that can be used for the tensor arena.

Parameters

[in]	arena_size	The size of the arena to allocate.
[out]	tensor_arena	Pointer to the allocated tensor arena buffer.

Returns

- A pointer to the allocated base buffer. Returns nullptr if the allocation failed. The base buffer is used for freeing the allocated memory.

Definition at line 66 of file util/third_party/tensorflow_extra/inc/sl_tflite_micro_init.h

sl_tflite_micro_get_error_reporter

```
tflite::ErrorReporter * sl_tflite_micro_get_error_reporter ()
```

Get a pointer to the TensorFlow Lite Micro error reporter created by the init function.

Returns

- A pointer to the error reporter.

Definition at line 75 of file util/third_party/tensorflow_extra/inc/sl_tflite_micro_init.h

sl_tflite_micro_get_interpreter

```
tflite::MicroInterpreter * sl_tflite_micro_get_interpreter ()
```

Get a pointer to the TensorFlow Lite Micro interpreter created by the init function.

Returns

- A pointer to the interpreter.

Definition at line 85 of file util/third_party/tensorflow_extra/inc/sl_tflite_micro_init.h

sl_tflite_micro_get_input_tensor

```
TfLiteTensor * sl_tflite_micro_get_input_tensor ()
```

Get a pointer to the input tensor, set by the init function.

Returns

- A pointer to the input tensor.

Definition at line 94 of file util/third_party/tensorflow_extra/inc/sl_tflite_micro_init.h

sl_tflite_micro_get_output_tensor

```
TfLiteTensor * sl_tflite_micro_get_output_tensor ()
```

Get a pointer to the output tensor, set by the init function.

Returns

- A pointer to the output tensor.

Definition at line `103` of file `util/third_party/tensorflow_extra/inc/sl_tflite_micro_init.h`

sl_tflite_micro_opcode_resolver

```
tflite::MicroOpResolver & sl_tflite_micro_opcode_resolver ()
```

Get a pointer to the opcode resolver for the flatbuffer given by the configuration.

Returns

- The address to the opcode resolver.

Definition at line `112` of file `util/third_party/tensorflow_extra/inc/sl_tflite_micro_init.h`

sl_tflite_micro_init

```
void sl_tflite_micro_init (void)
```

Create the error reporter and opcode resolver and initialize variables for the flatbuffer given by the configuration.

Parameters

N/A		

Definition at line `124` of file `util/third_party/tensorflow_extra/inc/sl_tflite_micro_init.h`

Copyright © 2023 Silicon Laboratories. All rights reserved.