MicroAI™ ESP32 APM SDK Version 1.0

Contents

Disclaimer 2

What is MicroAI™? 3

Why is MicroAI™ good for your project 3

Understanding MicroAI™ 4

Solution Types and Definitions 4

Requirements 5

Supported Devices 5

Tools & Versions 5

Getting Started 6

SDK Directory Structure 6

ESP-IDF Command Prompt 7

Project Configuration 8

APM (Using External Sensors) 8

Enabling Sensor inputs 8

Configuring your X-Code 9

Enabling MQTT 9

Configuring your Y-Code 10

Configuring task timing 11

SD Card setup 11

Attaching the ESP board 12

Programming the Flash device on the Target 13

Training the Model 14

Running the Model 15

Tuning the output data 15

Limitations and Dependencies 16

Reference Materials 16

Disclaimer

Your use of the SDK is at your sole risk. You will be solely responsible for any damage to your computer system or loss of data that results from the download or use of the SDK. To the maximum extent permitted by applicable law, in no event shall ONE Tech or its suppliers be liable for any special, incidental, indirect, or consequential damages whatsoever (including, without limitation damages for loss of business profits or revenue; business interruption or work stoppage; computer failure or malfunction; loss of business information, data or data use; loss of goodwill; or any other pecuniary loss) arising out of the use of or inability to use the SDK or the provision of or failure to provide support services, even if ONE Tech or its supplier has been advised of the possibility of such damages.

What is MicroAI™?

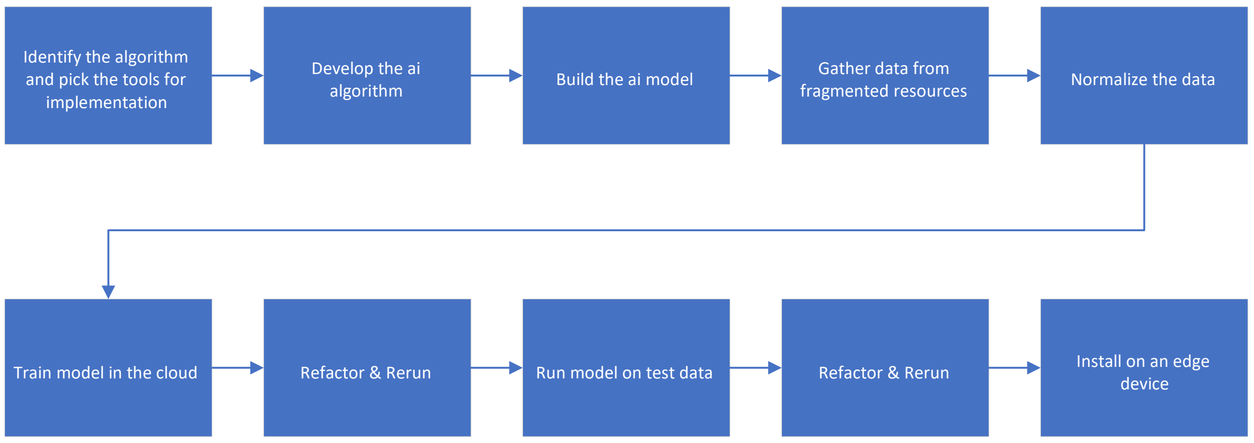
Simply put, MicroAI™ is an AI engine that can operate on low power edge and endpoint devices. It can learn the pattern of any and all time series data and can be used to detect anomalies or abnormalities, make one step ahead predictions/forecasts, and calculate the remaining life of entities (whether it is industrial machinery, small devices or the like).

Why is MicroAI™ good for your project

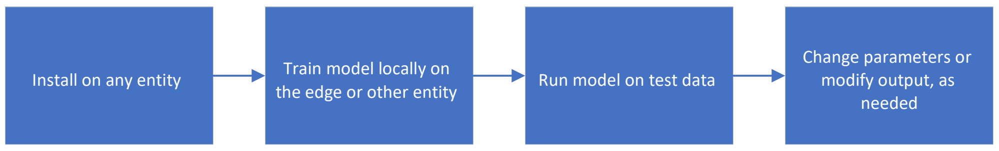
This MicroAI™ SDK enables your ESP32-based device or other entity with our proprietary AI (referred to in this document as 'MicroAI™') to help you build and gather feedback on your use cases, as well as the tutorials covered in this documentation. MicroAI™ can aid in the following tasks:

1. Data Collection
2. Data Organization
3. Model Building
4. Display, edit, and act upon the data gathered
5. Communicate with configured accessories and services to get them to perform actions, like turning on a light or receiving a notification on your phone.

Without MicroAI™ the process for building and integrating your own AI would work as follows:



Using MicroAI™ will save you development and cut down time to production significantly:


Install on any entity
Train model locally on the edge or other entity
Run model on test data
Change parameters or modify output, as needed

Understanding MicroAI™

The MicroAI™ SDK is made up of several key parts. The two most notable are the X-code and Y-code. However, all the parts can be broken down as follows:

Data Channels: As an MCU, the ESP32 typically utilizes sensor data for the data channels, but any time series data received by the controller could be assigned to a data channel.

Data Acquisition(X-code): X-Code is better known as 'input'. One of the foundational pieces of MicroAI™ is that it is data source agnostic, meaning, that it can ingest data from a variety of sources. The MCU can be configured to input data from a wide variety of sensors, utilizing the many communication channels available (I2C, SPI, etc.) or simply read via analog input. Common sources include accelerometers, gyroscopes, temperature, and humidity. Following data acquisition, the sensor data may be filtered as needed before being assigned to an input.

MicroAI™: Is made up of 4 steps: 1. Define and assign the input data channels (X-code) 2. Training the model 3. Engineering the features 4. Hosting the AI model. However, to define MicroAI simply, it provides insights into device behavior by comparing live data to comprehensive machine learning models.

Output Layer: Once the inputs (X-code) have been configured for the necessary sensors and the AI model has been trained, the MicroAI™ will process data in the output layer. In this layer, MicroAI™ is customized to the user's needs.

Application Layer(Y-code): Y-Code is better known as 'output'. However, the application layer and the output layer are dependent on each other. The Application layer provides users flexibility of how they would like to deploy. Essentially, the application layer (Y-Code) can be created to run on top of the output layer for a customized experience. Examples include email alerts, light changes, and turning the network connection on/off.

Visualization: This is exactly what it sounds like this can be in the form of dashboards and other outputs. This is dependent upon the target application and availability of local displays but can include sending data to servers capable of providing dashboard viewing.

Solution Types and Definitions

There are two type of solutions available:

APM: APM stands for Asset Performance Management, but to put it simply, when we refer to APM what we are really saying is that we are attaching external sensors to our ESP32 device in order to monitor an activity or entity.

Security: Simply put, security is the use of internal monitoring to predict abnormalities in usage consistent with the likes of security breaches or vulnerabilities.

This SDK only references the APM solution. The Security solution is provided in a separate SDK.

Requirements

Supported Devices

MicroAI™ can be supported on any ESP32 based board. This SDK is tailored to the ESP-WROVER-KIT which provides an on-board Micro-SD card slot, LCD panel, and I/O expansion capabilities. https://www.gridconnect.com/products/esp-wrover-kit-for-esp32-chip-module

For APM, you will need some form of outside data from sensors. This SDK has been configured to interface with the following sensors:

- Grove - I2C High Accuracy Temperature Sensor - MCP9808
- Grove - 6-Axis Accelerometer & Gyroscope
- Grove - Sound Sensor Based on LM358 amplifier

These are just examples, but any sensor that connects via the supported communication interfaces will work. Note that the provided code that reads these sensors is disabled by default to allow the user to get their system operational even without the sensors.

Tools & Versions

There are several development environments available that support the ESP32 devices. The one discussed in this SDK is provided free of charge by the manufacturer of the ESP32.

The following link provides a complete guide to the installation:

(https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/index.html)

However, the following steps provide a quick approach to process to install on the Windows development environment. Install all required tools by downloading and running the ESP-IDF Tools Installer for Windows (setup version 2.3).

https://dl.espressif.com/dl/esp-idf-tools-setup-2.3.exe

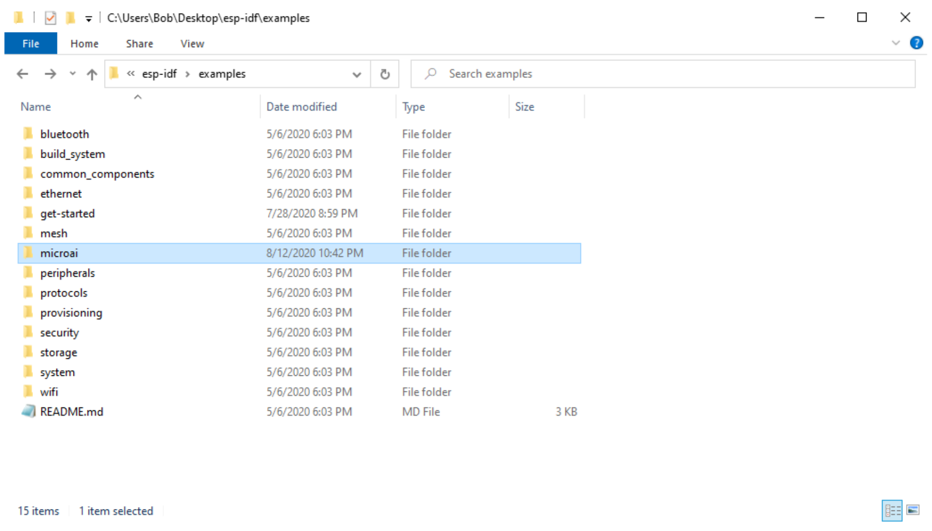
This setup program will prompt to download and install Python 3.7 and Git for Windows if they are not already installed. Additionally, it will prompt for the version of ESP-IDF. Please select "**4.0.1**". The scripts in this step install the compilation tools required by ESP-IDF in the user home directory (%USERPROFILE%\.espressif).

Following installation, numerous sample projects are available for testing features available on this development board. This provides a means to become familiar with the development environment. When familiar with this environment, proceed with the next section (Getting Started with the SDK).

Getting Started

Getting the SDK:

Once your ESP-IDF is installed and ready to begin the new project, go to the ONE Tech URL and download the SDK zip file. Then extract it into your choice of directory. This is commonly inserted into a new directory within the examples folder, as shown below:



C:\Users\Bob\Desktop\esp-idf\examples

« esp-idf › examples

Name	Date modified	Type	Size
bluetooth	5/6/2020 6:03 PM	File folder	
build_system	5/6/2020 6:03 PM	File folder	
common_components	5/6/2020 6:03 PM	File folder	
ethernet	5/6/2020 6:03 PM	File folder	
get-started	7/28/2020 8:59 PM	File folder	
mesh	5/6/2020 6:03 PM	File folder	
microai	8/12/2020 10:42 PM	File folder	
peripherals	5/6/2020 6:03 PM	File folder	
protocols	5/6/2020 6:03 PM	File folder	
provisioning	5/6/2020 6:03 PM	File folder	
security	5/6/2020 6:03 PM	File folder	
storage	5/6/2020 6:03 PM	File folder	
system	5/6/2020 6:03 PM	File folder	
wifi	5/6/2020 6:03 PM	File folder	
README.md	5/6/2020 6:03 PM	MD File	3 KB

15 items 1 item selected

SDK Directory Structure

Within the SDK directory shown above as "microai", you will find the following contents:

- **MicroAI Atom Evaluation License Agreement.pdf –** This will cover the legal license agreement you agree to by using MicroAI
- **MicroAI ESP32 APM SDK Version 1.0.pdf** (this file) **–** This is the manual you will use to guide you through the setup, installation, and tutorials of the SDK
- **README.md –** explains at a high level what is in the SDK package
- **CMakeLists.txt –** used at compilation time to identify required components
- **sdkconfig –** file generated by the IDF, this contains the configuration settings for the project. These setting are viewed and edited by running the "idf.py menuconfig" script.

Within "main"

- **esp32ai.c –** source code for initialization, main task, and X-code to populate the AI Engine data channels, and the first part of Y-code that determines data output.
- **esp32ai.h –** include file with defines used for conditional compilation
- **sensors.c –** source code for the specific sensors utilized in this SDK. This includes a separate task for reading analog to digital converter inputs on the MCU.

- **sensors.h –** include file
- **mqtt.c –** source code to support the MQTT communication. This is the output channel for the SDK and as such, is also referred to as the Y-Code. This includes a separate task to control the timing of messaging output.
- **mqtt.h –** include file
- **CMakeLists.txt –** used to specify the C source code modules to be compiled as well as reference to the location of the AI Engine library.
- **LSM6DS3.h –** include file to support the accel/gyro sensor board
- **Kconfig.projbuild –** text file used by the IDF to provide configuration options in addition to the standard options provided in the "sdkconfig" file.

Within "components/AiEngine"

- **AiEngine.h –** include file for integrating the required library
- **libAiEngine.a –** AI Engine library file
- **CMakeLists.txt –** used at compilation time to register the component library in this directory

Within "SDcard"

- **cfg –** sample configuration file to be loaded onto the micro SD card. This includes parameters used by various roles in the MicroAI™ system.
- **cfgExplain.txt –** a brief description of each parameter.
- **algPar –** sample configuration file to be loaded onto the micro SD card. This includes parameters for the individual input channels of the AI Engine.

ESP-IDF Command Prompt

The ESP-IDF tools installation added a desktop icon to begin a command window for ESP-IDF. Open that window now. This sets up the necessary parameters for running the tools. Upon entering this window, the prompt is typically at the base of the ESP-IDF tool set.

```
...\esp-idf>
```

Next, change the directory to the "microai" directory that was loaded from the SDK.

```
...\microai>
```

Project Configuration

When setting up any project in the ESP32 environment, the configuration script must be run before building the project. This script is run from within the project's root. In the case of this SDK, it is in the "microai" folder. To run the configuration, type the following:

```
...\microai>idf.py menuconfig
```

This opens the configuration editor. Most configuration parameters remain with their default settings. For this SDK, the following exceptions should be confirmed:

Example Configuration ---> I2C Master ---> (**19**) SCL GPIO Num
Example Configuration ---> I2C Master ---> (**18**) SDA GPIO Num
Example Configuration ---> I2C Master ---> (**1**) Port Number
Example Configuration ---> I2C Master ---> (**100000**) Master Frequency
Example Configuration ---> I2C Slave ---> (**26**) SCL GPIO Num
Example Configuration ---> I2C Slave ---> (**25**) SDA GPIO Num
Example Configuration ---> I2C Slave ---> (**0**) Port Number
Example Configuration ---> I2C Slave ---> (**0x28**) ESP Slave Address
Example Connection Configuration ---> Connect using (**Wi-Fi**)
Example Connection Configuration ---> (**"router SSID here"**) WiFi SSID
Example Connection Configuration ---> (**"router password here"**) WiFi Password

APM (Using External Sensors)

Enabling Sensor inputs

By default, the Sensor Input functionality is disabled. That merely allows compiling and running the code on the ESP32 board. To provide the value needed to test the MicroAI Engine, this functionality must be enabled.

To enable the Sensors inputs, uncomment the following lines in the "esp32ai.h" file:

```
        //#define ENABLE_SENSORS      // if enabled, we read sensors, else they are all zeros
```

Then continue with the following section to assign the sensor inputs to specific MicroAI Engine input channels. Until this functionality is enabled, the following section may be skipped (Configuring your X-Code).

Configuring your X-Code

(see pg. 4 for definition of APM)

When setting up an APM for the ESP32, there are 3 main steps.

1. Set up the sensor inputs by first defining the parameter channels to be obtained. List them in the following structure in "sensors.h". This indicates the current values available in the SDK.

```
typedef enum {
        E_ACCEL_X,
        E_ACCEL_Y,
        E_ACCEL_Z,
        E_GYRO_X,
        E_GYRO_Y,
        E_GYRO_Z,
        E_SOUND,
        E_TEMP,
        E_HUMID,
        E_ROTATE_Z,
        E_ROTATE_Z_DIFF,

        NOF_SENSOR_CHANNELS
} ENUM_SENSOR_CHANNELS;
```

   All of these channels will be read when the main task calls the read_sensor_data() function.

2. Modify the read_sensor_data() function to acquire each one of these channels. This is located in "sensors.c". This is where you add any necessary filtering for this data as well any other required Feature Engineering.

3. In the module "esp32ai.c", define the AI input channels in the following structure:

```
typedef enum {
        E_AI_CH_GYRO_NOW,
        E_AI_CH_GYRO,
        E_AI_CH_TEMPERATURE,
        E_AI_CH_HUMIDITY,
        E_AI_CH_SOUND,

        NOF_AI_CHANNELS
} ENUM_AI_CHANNELS_USED;
```

   Then assign the sensor channel data to the AI input channel in the main task. The sample in this SDK assigns the following 5 channels to the AI input channels:

```
signalSource[E_AI_CH_GYRO_NOW]       = sensor_array.data[E_ROTATE_Z];
signalSource[E_AI_CH_GYRO]           = sensor_array.data[E_ROTATE_Z_DIFF];
signalSource[E_AI_CH_TEMPERATURE]    = sensor_array.data[E_TEMP];
signalSource[E_AI_CH_HUMIDITY]       = sensor_array.data[E_HUMID];
signalSource[E_AI_CH_SOUND]          = sensor_array.data[E_SOUND];
```

   The resulting data channels are then fed into the AI Engine for processing.

Enabling MQTT

By default, the MQTT functionality is disabled, but you can still use the SDK to train and run your model. To use this module to provide output messaging to an MQTT server and enable this functionality, you need to make the following changes:

Un-comment these two lines from “esp32ai.h”

```
//#define MQTT_ENABLE

//#define MQTT_ALERT_ENABLE
```

Then continue with the following section to configure the MQTT settings.

Configuring your Y-Code

The SDK contains a module that provides output messaging to an MQTT server. This is where this APM contains the Y-Code.

Note: If MQTT is disabled (default setting in the SDK), this section may be disregarded. In this case, the output from the AI Engine will simply be viewed in the Terminal window.

This example utilizes FreeRTOS and its built in ability to provide multi-tasking. For the Y-Code and thus MQTT in this case, we have a task called “mqtt_task”. It is initiated from the startup code (in esp32ai.c) with the following FreeRTOS function:

```
xTaskCreate(mqtt_task, "MqttTask", 2048 * 2, (void *)2, 10, &mqtt_task_handle);
```

The code for this task is in the mqtt.c module and contains the following functions that require application-specific modification:

1. initMqtt() function – the initialization of this task’s parameters include the various MQTT channels being used. In this demo, we have 2 channels. One is used for sending regular updates and the other is for sending immediate alert messages when an error is detected. This allows for less frequent regular updates to save bandwidth while ensuring alert messages will be sent as soon as the error is detected.
   Each channel requires the following:
   - mqtt_cfg.host (server IP address)
   - mqtt_cfg.port (server port)
   - mqtt_cfg.client_id (client mac address, used by server to identify the incoming process)

   During this initialization, a connection is established with each channel.

2. mqtt_task() function – this function is initiated when the task is created. It runs continuously and pauses for the amount of time set by the configuration parameter 7mqttRateSecs. This provides the ability to change the regular messaging rate as fast as every second.
   Each loop through the task, the code assembles the messages as defined by the function: send_mqtt_msgs() and by the number of output channels defined in the “mqtt.h” file.

   Y-Code: These send message functions contain the Y-Code that allows the user to retrieve the latest data available from the MicroAI™ engine (in this case, by calling the updateDataBuffer function) and format the data to be transmitted (in this case by calling the send_mqtt_msg function).

The result is a published MQTT message for each defined output channel.

3. send_mqtt_alert_msg() function – called from the main task loop upon alert generation by the MicroAI™ engine. In this example, the schema for the alert is different from the regular published messages. Any additional Y-Code can be added here to send the necessary data.

   Note: All MQTT messages are being published with the use of the following command.

   ```
   esp_mqtt_client_publish(mqtt_alert_client, "<topic name inserted here>", charBuffer, 0,
   <Quality Of Service inserted here>, 0);
   ```

   Therefore, the user must insert the Topic name and QOS as required by their MQTT server.

Configuring task timing

This example utilizes FreeRTOS and its built in ability to provide multiple tasks. There are 3 tasks in this example and the timing between the 3 is important. Two of these are assigned in the module "esp32ai.h".

```
#define MAIN_TASK_LOOP_DELAY_MS          1000

#define SOUND_TASK_LOOP_DELAY_MS         500
```

These are delays in each of these 2 tasks, in milliseconds. This results in the sound task being executed 2 times per second and the main task running once per second.

The third task is assigned in the cfg configuration file. This is for the MQTT delay and is more commonly adjusted following training, thus the ability to change it in the SD card's configuration file. This parameter is shown in the next section.

SD Card setup

Prepare the MicroSD card used in this SDK by following these steps:

1. Format with the FAT/FAT16 file format. This results in a maximum size of 4 GB. This is required since almost all SD cards available are initially formatted with the FAT32 file format. There are many on-line guides with instructions on how to do this. One of them is the following: https://www.instructables.com/id/Format-USB-Flash-Drive-to-FATFAT16-not-FAT32/

2. After completing the format, copy the files from the SDK directory “SDcard” to the root directory of the SD card.
    a. cfg
    b. cfg-explain.txt
    c. algPar

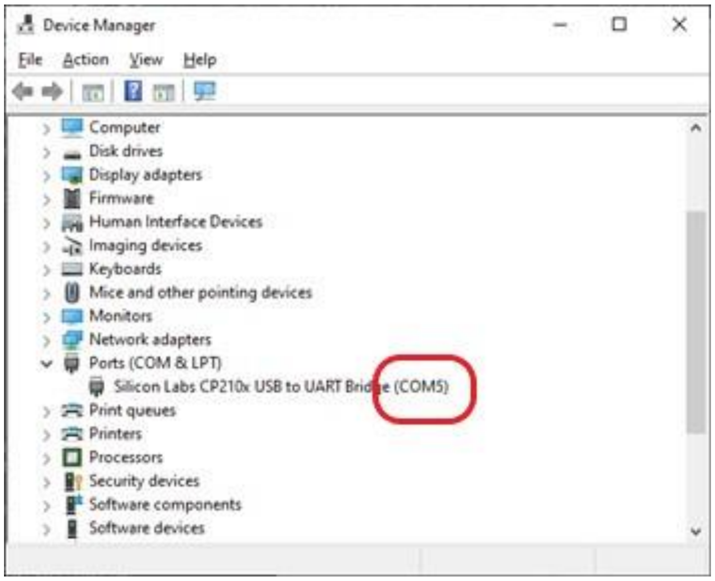
3. The file “cfg-explain.txt” contains an abbreviated description of each entry in the cfg file. These descriptions will be referenced later in this document. The other 2 files contain the configuration parameters used by the program and are editable with a simple text editor.
   Contents of “cfg-explain.txt”:
    - 0isModelBuild (set to a value of 1 the first time the system is run to build the model)
    - 1deleteoldmodel
    - 2meanCalGain
    - 3stdCalGain
    - 4initialPeriod
    - 5splitCounter
    - 6trainingSeconds
    - 7mqttRateSecs
    - 8smallThreshold

4. Following any changes to the configuration, eject the SD card from the PC and insert into the ESP32 board.

Attaching the ESP board

Connect the board using the USB cable to the development computer. The Windows computer should recognize the board and assign a COM port. The easiest way to determine which port has been assigned is to run the Device Manager program. Do this by typing “Device Manager” in the Windows start menu and run the program.

When the USB cable connection is established from the ESP32 board to the computer, there should be a listing under “Ports (COM & LPT)”, as shown in the example below:



This indicates the use of COM5. Every system is different, so make a note of this value since it will be used in the subsequent steps of this SDK. If the ESP32 board adds 2 com ports, use the 2nd one listed.

Programming the Flash device on the Target

After any coding changes are complete, the system is compiled with the following script:

```
…\microai>idf.py build
```

Upon successful completion of this step, the target system is updated using this command:

```
…\microai>idf.py -p COM5 flash
```

The COM port number used is system dependent and can be found by using a program such as Device Manager in Windows.

Immediately upon updating, the device is rebooted and execution begins. However, this com port is disconnected upon completion of the update. Therefore, to view any data sent to the com port, you must either run a terminal emulation program, or use the following script to place into monitor mode:

```
…\microai>idf.py -p COM5 monitor
```

This monitor mode can be halted by typing the key combination: Ctrl -]

Training the Model

Before the MicroAI™ engine can be run, it must first construct a training model. The more data points the training model has, the more accurate it should be. It should be noted that having more data points does not increase the size of the model but does increase the accuracy.

There are 4 configuration variables used in the process of constructing the model:

- 0isModelBuild – setting this to 1 causes the code to begin constructing the model upon powering up. This will be set to 0 for normal operation, following the construction.
- 1deleteoldmodel – set to 1, this deletes the old model files upon initiation of the model construction process. If set to 0, training data is added to the existing model files.
- 5splitCounter – this is the number of data points read into the AI Engine before storing another model entry.
- 6trainingSeconds – the number of seconds after power on that the model construction takes place. To train for 3 minutes, this value would then be 180.

To understand the affects of these numbers, consider the following example:

5splitCounter = 20

6trainingSeconds = 180

Data Point frequency = 1 per second (programmed into the X-code)

The total number of model entries would then be:

(180 secs * 1 data point per sec) / 20 data points per sample = 9 samples

When these parameters have been edited on the SD card, re-insert into the ESP32.

Power on and enter the following into the command line:

```
…\microai>idf.py -p COM5 monitor
```

This will reboot the system and monitor the output in the command window. As samples are recorded into the model, the following entry is displayed for each sample:

```
model rows ADDED
```

When the training process is complete, the following message is displayed:

```
*** TRAINING COMPLETE ***
```

At that point, power off the ESP32 board (move the power slide switch to the off position). Then remove the SD card and modify the cfg file entry for 0isModelBuild to a 0. This allows the system to run with the model that has been saved. When viewing the SD card files, you will now see the additional model files stored. They are named “modelX” and “modelY”.

Running the Model

With the cfg file set to run mode (0isModelBuild = 0), the system will begin running as soon as power is applied. No additional input is required.

If any results are programmed to be sent to the com port, it may be monitored in 2 ways:

1. Use the previously described method by entering the following Python command into the command window:

   ```
   …\microai>idf.py -p COM5 monitor
   ```

2. Run another terminal emulation program, such as PuTTy or TeraTerm, to observe the com port data and log it if desired.

Note that each time a connection is established to the com port, the ESP32 reboots its program.

Tuning the output data

As discussed earlier, the MicroAI Engine has a specific number of input channels as defined in the section regarding configuring the X-Code. These same channels provide data for output. In our example, we have 5 channels.

Be sure that the other configuration file "algPar" contains the same number of entries, with one line for each channel. In our sample file, we have the following entries:

```
3,5
3,5
3,5
3,5
3,5
```

Each value can be set to a range of 1 to 10. The first value is used for establishing the width of acceptance between high and low boundaries. Increasing this value widens this range and reduces the sensitivity.

The second value is used to set the sensitivity for the reported alarm condition. In a similar manner, raising this number reduces the sensitivity.

These values are available to tune the alert performance of the system channel by channel.

Limitations and Dependencies

- This is an evaluation version only and should not be expected to support a full production environment.
- Future versions of packages and devices may not be supported. Additional documentation and doc updates will be given to accommodate updates.

Reference Materials

- For questions or to get help please post your questions here: http://developers.ONE Tech.ai/community/