Future Skills Centre | Centre des Compétences futures

# MACIOT Trainer Manual


**Revision: 1.0**

**Josh Dorland & Dr. Ishwar Singh**

W Booth School of Engineering Practice and Technology | McMaster University

# Acknowledgements

**We would like to thank Reiner Schmidt for his hardwork and dedictation in creating the McMaster Internet of Things board and opening the door for future students to learn and develop industry leading skills in internet of things.**

**We would also like to thank our partnership with the Future Skills Development program for helping to fund the McMaster Internet of Things board and corresponding manual and helping prepare student for the jobs of the future.**

**Finally we would like to thank Axibo for the amazing partnership and allowing future students to use Axibos family of products for learning and skills development.**

**I would like to thank Dr.Singh for giving me the amazing opportunity to work along side him in creating this lab manual as well as the opportunity to work with Axibo.**

# TABLE OF CONTENTS

# INTRODUCTION

## Background

Internet of Things (IoT) is the process of connecting anything that can be turned on/off to the internet and to other connected devices. Its applications are everywhere from smart sensing if someone is in a room, to measuring temperature and humidity inside a greenhouse. The MacIoT board aims to allow the user to learn step by step about IoT as well as communication protocols such as CAN which is used inside modern day vehicles and I2C which can be used to communicate with various sensors. The MacIoT board brings an onboard 9-axis accelerometer, gyroscope, and magnetometer, temperature and humidity sensor, light sensor as well as LoRa, I2C, and CAN.

## Schematic

The schematic section is meant to give a rundown of the MacIoT schematic created by Reiner Schmidt. It will cover the various circuitry to help improve the users overall understanding of the board. The full schematic and pinout diagrams can be found in Appendix A.

### ESP32 Controller Module

The ESP32-WROOM-32D is a low-cost, low-power system on a chip micrcontroller. The microcontroller has onboard WiFi and Bluetooth. It houses UART, SPI, I2C, and more module interfaces. The hardware has an operating voltage range of 3.0V $\sim$ 3.6V with an operating current of 80mA. Some important pins to remember on the board include:

- IO22 (Pin 36) $\rightarrow$ I2C_SCL
- IO21 (Pin 33) $\rightarrow$ I2C_SDA
- IO5 (Pin 29) $\rightarrow$ CAN_TXD
- IO4 (Pin 26) $\rightarrow$ CAN_RXD

### Arduino SHIELD Connector

The Arduino Shield Connector allows the MacIoT to extend its functionality and connect to another or multiple arduinos at once. Having this means that the MacIoT can stack CAN shields, LoRa shields, and more.

### LCD Connector

Allows the MacIoT to connect to an external LCD monitor.

### External Sensors

The external sensors which include the 9-axis IMU, temperature & humidity, and light sensor are connected and use I2C_SCL and I2C_SDA lines to transmit data. The clock for the SCL or serial clock line is provided by the ESP32 microcontroller.

### IO Expander

The IO expander is a very helpful addition to the MacIoT board as it allows for more input and ouput ports to be used. This means that more sensors and devices can be connected than previously allowed from just the ESP32. This is done through I2C communication from the ESP32 to the IO expander. As seen this IO expander connects 4 switches, 4 push buttons, and 7 LEDs.

## Required Software

Only a few pieces of software is required to get started using the MacIoT board. This software is optional and different IDEs can be used but it is highly recommended to use the same software for limited issues while completing the modules. The software is:

1. Visual Studio Code → `https://code.visualstudio.com/`
2. PlatformIO → `https://platformio.org/`

## Required Hardware

Lister below is all the relevant hardware that is required in order to complete all modules in the manual. This includes:

- MacIoT Board
- CAN Transeiver Board
- LoRa Transeiver Board

## Datasheet

The links for the datasheets for the ESP32, I2C IO expander, humidity and temperature sensors, 9-axis IMU, and light sensor are included below. Although all the information will be provided to you to complete the modules it is extremely importatnt to become well versed in effective manual reading as this will not only help in this manual but throughout your projects and career.

| Name | Part Number | Link |
|---|---|---|
| ESP32-WROOM-32D | ESP32-WROOM-32D | https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32d_esp32-wroom-32u_datasheet_en.pdf |
| 16-Bit I/O Expander with Serial Interface | MCP23017 | https://ww1.microchip.com/downloads/aemDocuments/documents/OTH/ProductDocuments/DataSheets/20001952C.pdf |
| I2C HUMIDITY AND TEMPERATURE SENSOR | Si7020-A20 | https://www.micro-semiconductor.com/datasheet/b3-SI7020-A20-YM0R.pdf |
| Ambient Light Sensor | APDS-9306-065 | https://docs.broadcom.com/doc/AV02-4755EN |
| 3D accelerometer, 3D gyroscope, 3D magnetometer | LSM9DS1 | https://www.st.com/content/ccc/resource/technical/document/datasheet/1e/3f/2a/d6/25/eb/48/46/DM00103319.pdf/files/DM00103319.pdf/jcr:content/translations/en.DM00103319.pdf |

# MODULE 1: THE START

## General Overview

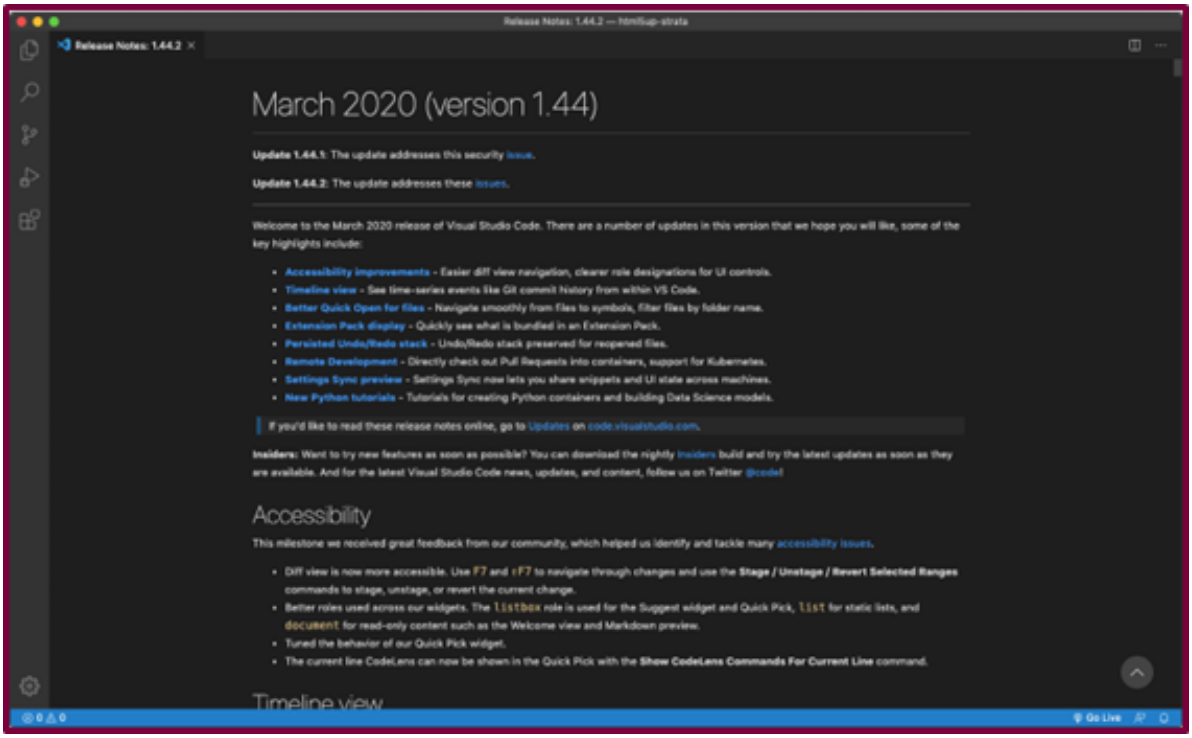
Module 1 covers how to get started with the MacIoT board. This will include setting up Visual Studio Code, PlatformIO, and creating a simple blinking LED project.

## Visual Studio Code

Before we get started, we must have the necessary software. We are going to download **Visual Studio Code** as the Integrated Development Environment (IDE) of our choice. Once we have that setup, we will install the **PlatformIO** extension within Visual Studio Code to help streamline our workflow for embedded development.

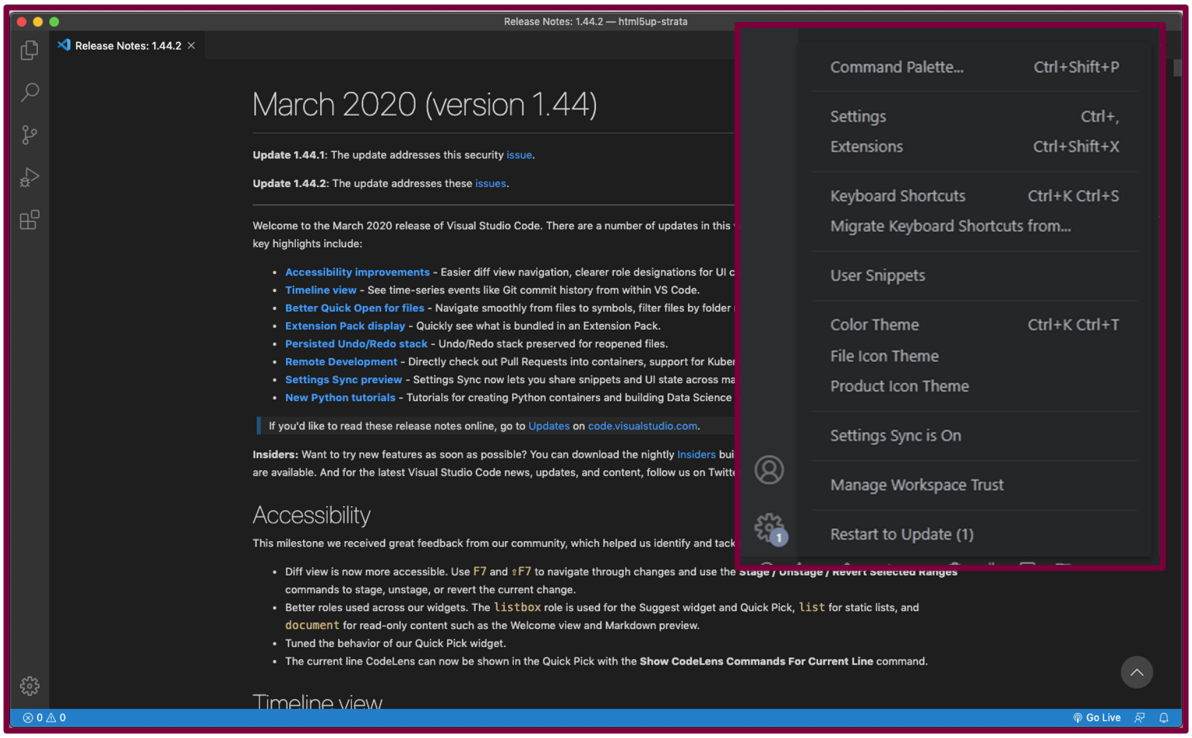
### Installation

Download Visual Studio Code here: `https://code.visualstudio.com/download` for whichever operating system you are currently working on. Once downloaded, go ahead, and install. For demonstration purposes the Windows 10 operating system was used.



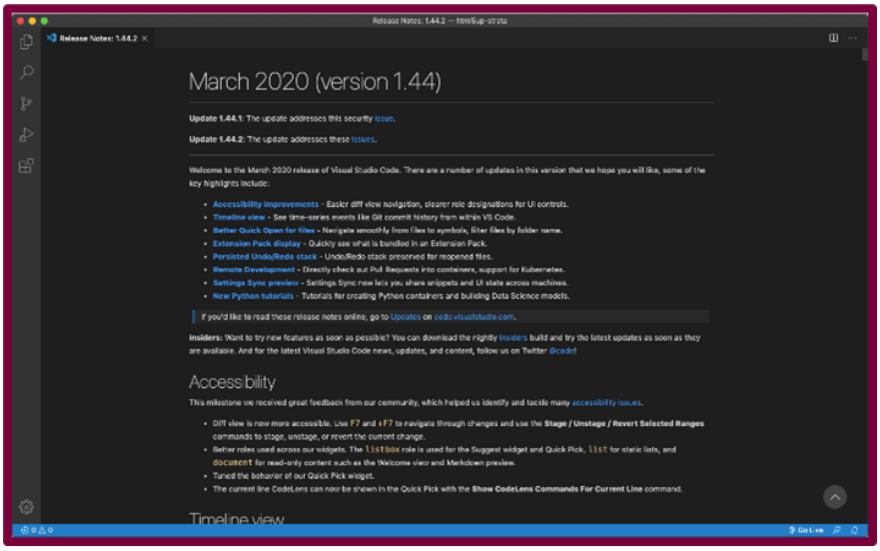
## Choosing a Theme

Now of course we all love to customize things to our own preferences. Let's change the theme! Click on the gear icon in the bottom left of the screen and you will find a Color Theme option. From there a menu appears and we can select the theme of our choice.




## Installing PlatformIO

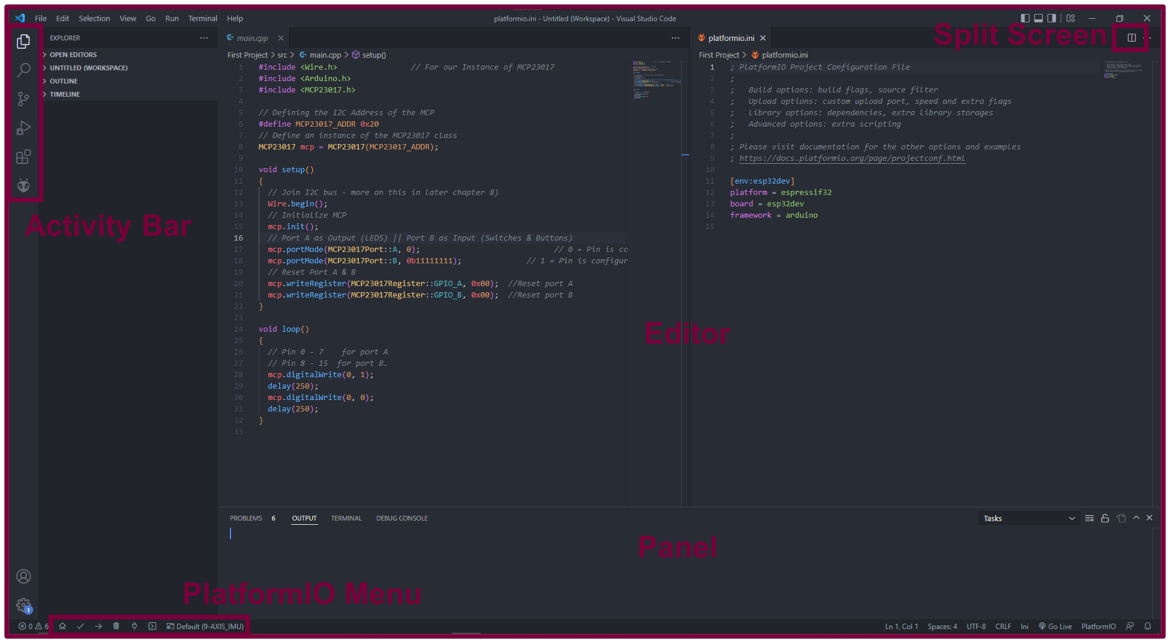
Go to the extensions tab and search for **PlatformIO IDE**. Go ahead and install the extension, and then restart Visual Studio Code.



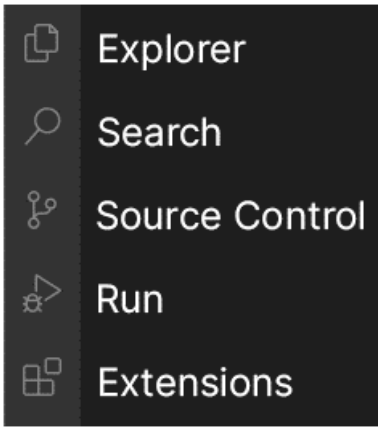

## Navigating the Workspace

Before we really dive deep, let's familiarize ourselves with our work environment. While Visual Studio Code may seem a bit daunting to some, it's actually not that bad! It's an extremely useful Integrated Development Environment (IDE), and all it takes is a little getting used to.

Highlighted below are some of the main features of the User Interface (UI) that we'll be familiarizing ourselves with. As mentioned in the earlier sections, there is the Activity Bar. At the bottom is the Status Bar, where the PlatformIO icons are located. We'll go over them soon. Then the environment can be broken into two main views, where you have the Editor and the Panels. The editor is where you edit your files and the panels will show various information such as output, debug, errors, etc. All very important. Finally, there is the Split Screen option for side-by-side development. For a more detailed overview: `https://code.visualstudio.com/docs/getstarted/userinterface`.



## Activity Bar




- **Explorer** - Shows the current file directories or document tree
- **Search** - Is self explanatory and allows you to find and replace text within files
- **Source Control** - Is for source control of git repositories.
- **Run** - Is for running debug files. We won't worry about this
- **Extensions** - Is for managing and installing extensions

## PlatformIO Menu

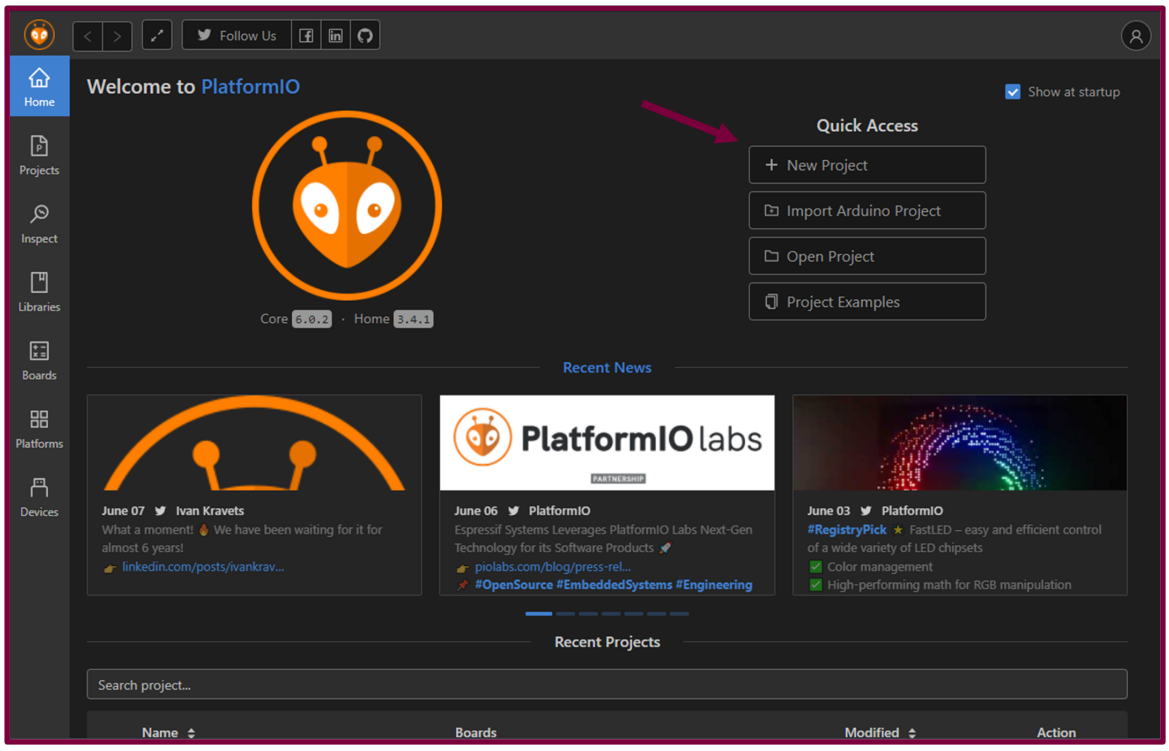

- **Home** - Brings up the PlatformIO home tab
- **Build** - Compiles the project and links any dependencies
- **Upload** - flashes the code onto the device defined on whichever serial **COM** (Windows) or **dev/tty/** (Mac) ports.
- **Upload to Remote Devices** - Uploads code onto remote devices. We won't be using this

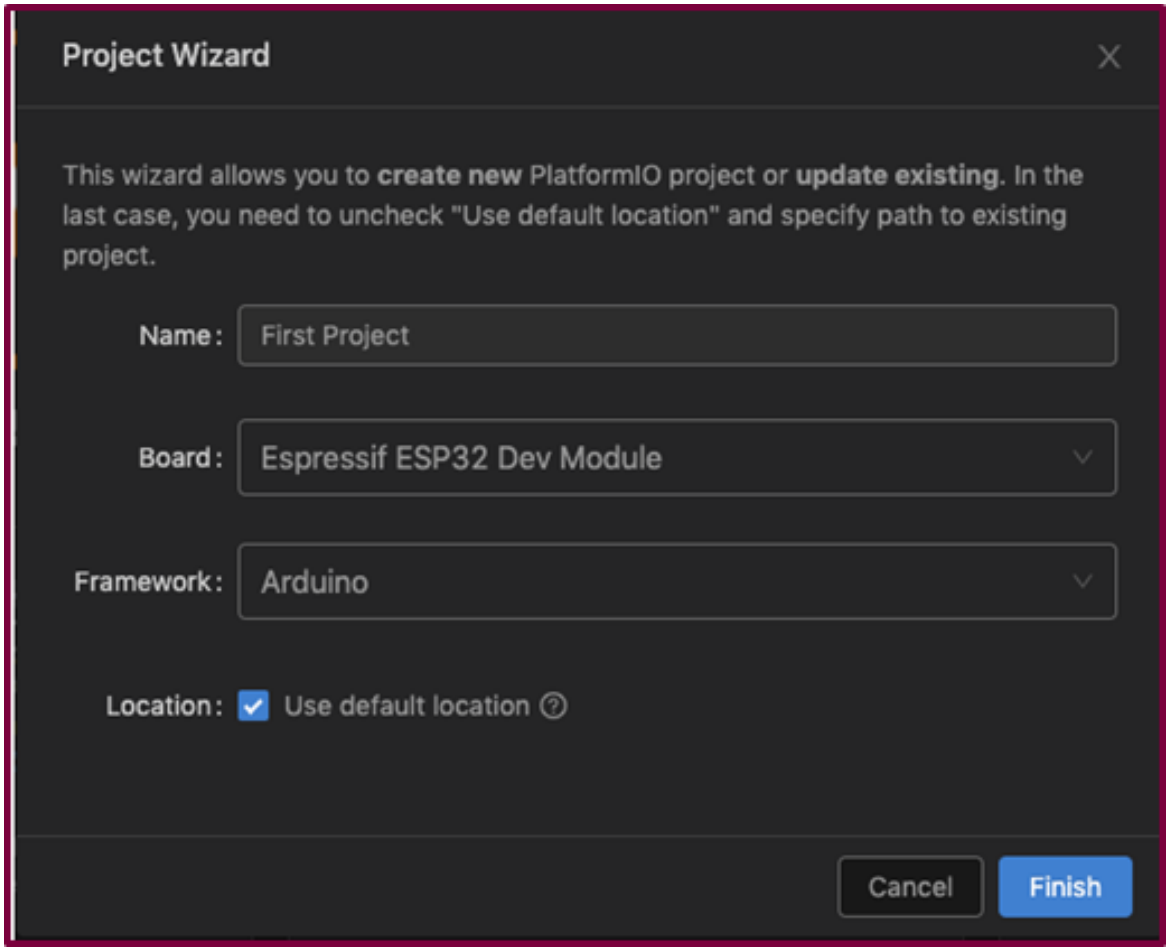
- **Clean** - Will delete compiled object files, libraries, and program binaries
- **Test** - Runs local tests from project
- **Run Task** - Runs specific tasks
- **Serial Monitor** - Opens the serial monitor on the defined baud rate
- **New Terminal** - Clears the terminal

## Creating a New Project

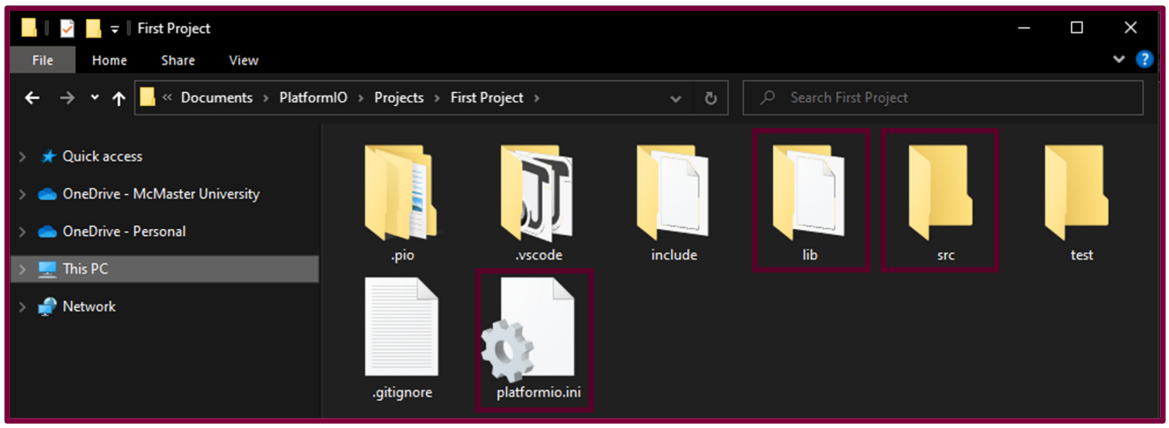
With Visual Studio Code open, navigate to the PlatformIO (PIO) home screen. If you don't see the home screen, click on the PIO Icon in the Activity Bar. Then, go ahead and click "Open". That should bring up the **PIO Home screen**.

Under "Quick Access" within PIO Home, let's go ahead and create a **New Project**. The Project Wizard will appear. Call your project whatever you want. I'll be calling mine "First Project". As you have noticed, PIO supports hundreds of boards. For our purposes, we'll select Espressif ESP32 Dev Module. This is because the MacIoT's system on a chip (SoC) microcontroller is the ESP32. Finally, the framework we'll be working with will be the Arduino framework.






You may choose anywhere as your file location. Hovering over the (?) will show the file path to the default location. Once you are finished creating your project, let's open it and inspect the contents.



Let's talk a bit about these files. A standard PIO project structure contains three main items:

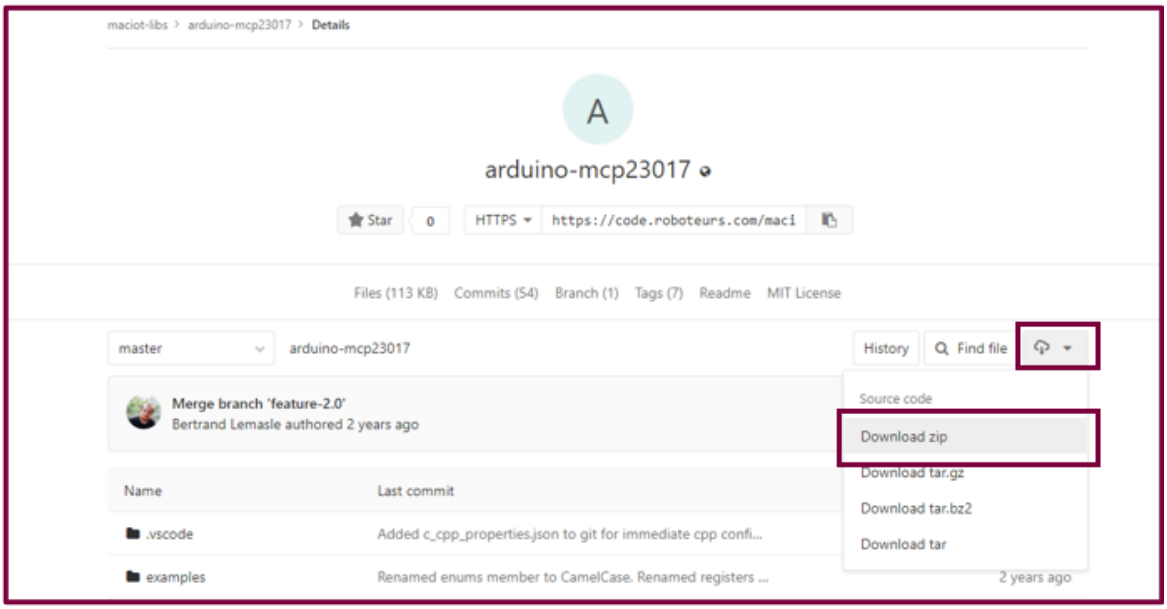
- **lib** - The directory which contains project specific (private) libraries
- **src** - The directory which contains your source code. Files such as (.c, .cpp, .ino, etc)
- **platform.ini** - Project configuration file. This is an "INI" file which defines things such as the platform (**espressif32**), board (**esp32dev**), framework (**Arduino**). When we were initially creating the project, we defined these parameters in the Project Wizard. This file will be modified as we define more things later on such as build options, library options, etc.

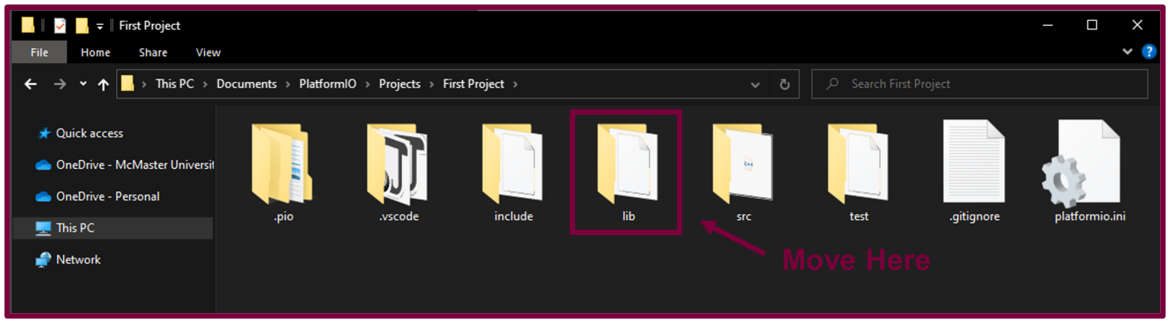

### Installing Libraries

Let's install our first library. But what exactly are libraries? Libraries provide a clean way of extending the functionalities of our code. We can have all sorts of libraries for different functions, and we just simply call the function in our main code. For the libraries that we'll use, they will have at least two files, **header ".h"** files and the **source code ".c/cpp"** files. Header files contain definitions of the functions defined in the source files. That's why in our main.cpp, we'll be including just the header file. We'll need to install a library that allows us to communicate with the IO expander that's on the I2C bus. The IO expander has all the buttons, switches, and LEDs. This library can be found here: `https://code.roboteurs.com/maciot-libs/arduino-mcp23017`.

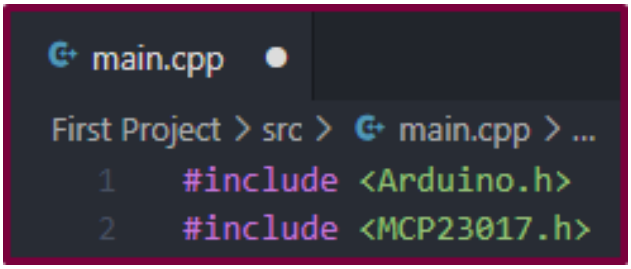
### Installing the MCP23017 Library

There are many ways to install a library. For now, we'll go with the simplest method which is just moving the library to our project's **"lib"** directory. Downloading the library as a **".zip"**, go ahead and unzip the file. Move the library folder to your project's **"lib"** folder.

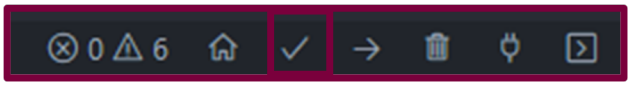



Now go back to **Visual Studio Code** and head to **main.cpp**. Add `#include <MCP23017.h>` to the top to include the library.




Now head to the **PlatformIO** Menu at the bottom and build the code.




When we build our project again, notice how the library is now being compiled. That's good! There should be no errors in compilation.

### Compiling the Project

Now that we have created our project and familiarized ourself with our work environment, let's build/compile the project. You can build it from the **Activity Bar** or from the **PlatformIO** build icon on the status bar at the bottom of the Visual Studio Code. The first time we build will take longer the successive builds.

Below shows the output from the first build. It looks long and complicated, but don't worry. We'll break it down by segments so you can gain a better understanding of what exactly happens when we build our project.

```
    > Executing task in folder Project: platformio run <

Processing esp32dev (platform: espressif32; board: esp32dev;
framework: arduino)
------------------------------------------------------------
Verbose mode can be enabled via `-v, --verbose` option
CONFIGURATION: https://docs.platformio.org/page/boards/espressif32
-/esp32dev.html
PLATFORM: Espressif 32 1.12.0 > Espressif ESP32 Dev Module
HARDWARE: ESP32 240MHz, 320KB RAM, 4MB Flash
DEBUG: Current (esp-prog) External (esp-prog, iot-bus-jtag, jlink,
minimodule, olimex-arm-usb-ocd, olimex-arm-usb-ocd-h, olimex-arm
-usb-tiny-h, olimex-jtag-tiny, tumpa)
PACKAGES:
 - framework-arduinoespressif32 3.10004.200129 (1.0.4)
 - tool-esptoolpy 1.20600.0 (2.6.0)
 - toolchain-xtensa32 2.50200.80 (5.2.0)
LDF: Library Dependency Finder -> http://bit.ly/configure-pio-ldf
LDF Modes: Finder ~ chain, Compatibility ~ soft
Found 26 compatible libraries
Scanning dependencies...
No dependencies
Building in release mode
Compiling .pio/build/esp32dev/src/main.cpp.o
Generating partitions .pio/build/esp32dev/partitions.bin
Archiving .pio/build/esp32dev/libFrameworkArduinoVariant.a
Indexing .pio/build/esp32dev/libFrameworkArduinoVariant.a
Compiling .pio/build/esp32dev/FrameworkArduino/Esp.cpp.o
Compiling .pio/build/esp32dev/FrameworkArduino/FunctionalInterrupt.cpp.o
Compiling .pio/build/esp32dev/FrameworkArduino/HardwareSerial.cpp.o
Compiling .pio/build/esp32dev/FrameworkArduino/IPAddress.cpp.o
Compiling .pio/build/esp32dev/FrameworkArduino/IPv6Address.cpp.o
Compiling .pio/build/esp32dev/FrameworkArduino/MD5Builder.cpp.o
Compiling .pio/build/esp32dev/FrameworkArduino/Print.cpp.o
Compiling .pio/build/esp32dev/FrameworkArduino/Stream.cpp.o
```

```
Compiling .pio/build/esp32dev/FrameworkArduino/StreamString.cpp.o
Compiling .pio/build/esp32dev/FrameworkArduino/WMath.cpp.o
Compiling .pio/build/esp32dev/FrameworkArduino/WString.cpp.o
Compiling .pio/build/esp32dev/FrameworkArduino/base64.cpp.o
Compiling .pio/build/esp32dev/FrameworkArduino/cbuf.cpp.o
Compiling .pio/build/esp32dev/FrameworkArduino/esp32-hal-adc.c.o
Compiling .pio/build/esp32dev/FrameworkArduino/esp32-hal-bt.c.o
Compiling .pio/build/esp32dev/FrameworkArduino/esp32-hal-cpu.c.o
Compiling .pio/build/esp32dev/FrameworkArduino/esp32-hal-dac.c.o
Compiling .pio/build/esp32dev/FrameworkArduino/esp32-hal-gpio.c.o
Compiling .pio/build/esp32dev/FrameworkArduino/esp32-hal-i2c.c.o
Compiling .pio/build/esp32dev/FrameworkArduino/esp32-hal-ledc.c.o
Compiling .pio/build/esp32dev/FrameworkArduino/esp32-hal-matrix.c.o
Compiling .pio/build/esp32dev/FrameworkArduino/esp32-hal-misc.c.o
Compiling .pio/build/esp32dev/FrameworkArduino/esp32-hal-psram.c.o
Compiling .pio/build/esp32dev/FrameworkArduino/esp32-hal-rmt.c.o
Compiling .pio/build/esp32dev/FrameworkArduino/esp32-hal-sigmadelta.c.o
Compiling .pio/build/esp32dev/FrameworkArduino/esp32-hal-spi.c.o
Compiling .pio/build/esp32dev/FrameworkArduino/esp32-hal-time.c.o
Compiling .pio/build/esp32dev/FrameworkArduino/esp32-hal-timer.c.o
Compiling .pio/build/esp32dev/FrameworkArduino/esp32-hal-touch.c.o
Compiling .pio/build/esp32dev/FrameworkArduino/esp32-hal-uart.c.o
Compiling .pio/build/esp32dev/FrameworkArduino/libb64/cdecode.c.o
Compiling .pio/build/esp32dev/FrameworkArduino/libb64/cencode.c.o
Compiling .pio/build/esp32dev/FrameworkArduino/main.cpp.o
Compiling .pio/build/esp32dev/FrameworkArduino/stdlib_noniso.c.o
Compiling .pio/build/esp32dev/FrameworkArduino/wiring_pulse.c.o
Compiling .pio/build/esp32dev/FrameworkArduino/wiring_shift.c.o
Archiving .pio/build/esp32dev/libFrameworkArduino.a
Indexing .pio/build/esp32dev/libFrameworkArduino.a
Linking .pio/build/esp32dev/firmware.elf
Building .pio/build/esp32dev/firmware.bin
Retrieving maximum program size .pio/build/esp32dev/firmware.elf
Checking size .pio/build/esp32dev/firmware.elf
Advanced Memory Usage is available via "PlatformIO Home > Project Inspect"
RAM:   [          ]   4.5% (used 14700 bytes from 327680 bytes)
Flash: [==        ]  16.0% (used 209795 bytes from 1310720 bytes)
esptool.py v2.6
============== [SUCCESS] Took 13.51 seconds ==============
```

### Breaking it Down

```
> Executing task in folder Project: platformio run <
Processing esp32dev (platform: espressif32; board: esp32dev; framework:
arduino)
```

The *platformio* run is the Command-Line Interface (CLI) command to build the project. So, when we build through VSC, we're running this command. In line three, *esp32dev* is the ID assigned to the board by PIO. The platform, board, and framework are defined in the **"platformio.ini"** file that gets created when we first initialized our project.

```
Verbose mode can be enabled via `-v, --verbose` optionCONFIGURATION:
https://docs.platformio.org/page/boards/espressif32/esp32dev.html-
PLATFORM: Espressif 32 1.12.0 > Espressif ESP32 Dev Module
HARDWARE: ESP32 240MHz, 320KB RAM, 4MB FlashDEBUG: Current (esp-prog)
External (esp-prog, iotbus-jtag, jlink, minimodule, olimex-arm-usb-ocd,
olimex-arm-usb-ocd-h, olimex-arm-usb-tiny-h, olimex-jtag-tiny, tumpa)
PACKAGES:
 - framework-arduinoespressif32 3.10004.200129 (1.0.4)
 - tool-esptoolpy 1.20600.0 (2.6.0) - toolchain-xtensa32 2.50200.80
 (5.2.0)
```

Verbose mode can be enabled with the *-v, –verbose* option/flag. So for example, when running the CLI command, you can say *platformio run -v* or *pio run -v*, which will show detailed information when it's processing the environments.

The next few lines contain information regarding the esp32dev chip we're working with. The *DEBUG* line shows various uploading and debugging options, *esp-prog* being the default. Our framework is *arduinoespressif32* which is the Arduino-ESP32 framework. The toolchain-xtensa32 package is simply just the compiler and linker used.

```
LDF: Library Dependency Finder -> http://bit.ly/configure-pio-ldfLDF
Modes: Finder ~ chain,
Compatibility ~ soft
Found 26 compatible libraries
Scanning dependencies...No dependencies
```

The Library Dependency Finder (LDF) looks for `#include < >` in C/C++ files to look for the header directories to include in compilation. The default *chain* mode will parse all C/C++ source files in the project.

```
Compiling .pio/build/esp32dev/src/main.cpp.oGenerating partitions
.pio/build/esp32dev/partitions.binArchiving .pio/build/esp32dev/lib-
FrameworkArduinoVariant.a Indexing
.pio/build/esp32dev/libFrameworkArduinoVariant.a
```

The compiler creates the *main.cpp.o* object file, and generates the *partitions.bin* binary file. The compiler creates an archive file called *libFrameworkArduinoVariant.a* and indexes it.

```
Compiling .pio/build/esp32dev/FrameworkArduino/Esp.cpp.oCompiling
.pio/build/esp32dev/FrameworkArduino/FunctionalInterrupt.cpp.o-
Compiling
.pio/build/esp32dev/FrameworkArduino/HardwareSerial.cpp.oCompiling
.pio/build/esp32dev/FrameworkArduino/IPAddress.cpp.oCompiling
.pio/build/esp32dev/FrameworkArduino/IPv6Address.cpp.oCompiling
.pio/build/esp32dev/FrameworkArduino/MD5Builder.cpp.oCompiling
.pio/build/esp32dev/FrameworkArduino/Print.cpp.oCompiling
.pio/build/esp32dev/FrameworkArduino/Stream.cpp.oCompiling
.pio/build/esp32dev/FrameworkArduino/StreamString.cpp.oCompiling
.pio/build/esp32dev/FrameworkArduino/WMath.cpp.oCompiling
.pio/build/esp32dev/FrameworkArduino/WString.cpp.oCompiling
.pio/build/esp32dev/FrameworkArduino/base64.cpp.oCompiling
.pio/build/esp32dev/FrameworkArduino/cbuf.cpp.oCompiling
.pio/build/esp32dev/FrameworkArduino/esp32-hal-adc.c.oCompiling
.pio/build/esp32dev/FrameworkArduino/esp32-hal-bt.c.oCompiling
.pio/build/esp32dev/FrameworkArduino/esp32-hal-cpu.c.oCompiling
.pio/build/esp32dev/FrameworkArduino/esp32-hal-dac.c.oCompiling
.pio/build/esp32dev/FrameworkArduino/esp32-hal-gpio.c.oCompiling
.pio/build/esp32dev/FrameworkArduino/esp32-hal-i2c.c.oCompiling
.pio/build/esp32dev/FrameworkArduino/esp32-hal-ledc.c.oCompiling
.pio/build/esp32dev/FrameworkArduino/esp32-hal-matrix.c.oCompiling
.pio/build/esp32dev/FrameworkArduino/esp32-hal-misc.c.oCompiling
.pio/build/esp32dev/FrameworkArduino/esp32-hal-psram.c.oCompiling
.pio/build/esp32dev/FrameworkArduino/esp32-hal-rmt.c.oCompiling
.pio/build/esp32dev/FrameworkArduino/esp32-hal-sigmadelta.c.o-
Compiling
.pio/build/esp32dev/FrameworkArduino/esp32-hal-spi.c.oCompiling
.pio/build/esp32dev/FrameworkArduino/esp32-hal-time.c.oCompiling
.pio/build/esp32dev/FrameworkArduino/esp32-hal-timer.c.oCompiling
.pio/build/esp32dev/FrameworkArduino/esp32-hal-touch.c.oCompiling
.pio/build/esp32dev/FrameworkArduino/esp32-hal-uart.c.oCompiling
.pio/build/esp32dev/FrameworkArduino/libb64/cdecode.c.oCompiling
.pio/build/esp32dev/FrameworkArduino/libb64/cencode.c.oCompiling
.pio/build/esp32dev/FrameworkArduino/main.cpp.oCompiling
.pio/build/esp32dev/FrameworkArduino/stdlib_noniso.c.oCompiling
.pio/build/esp32dev/FrameworkArduino/wiring_pulse.c.oCompiling
.pio/build/esp32dev/FrameworkArduino/wiring_shift.c.oArchiving
.pio/build/esp32dev/libFrameworkArduino.aIndexing
.pio/build/esp32dev/libFrameworkArduino.a
```

The *.o* ending on all the *FrameworkArduino* files indicate that these are object files, which are files you get post-compiling. Object files are mainly machine code, and a linker will take these object files together and create an executable file.

```
Linking .pio/build/esp32dev/firmware.elf Building
.pio/build/esp32dev/firmware.binRetrieving maximum program size
.pio/build/esp32dev/firmware.elfChecking size .pio/build/esp32dev/-
firmware.elf
```

The compiler links the *firmware.elf* which is an Executable and Linkable File (ELF), as well as builds the binary *firmware.bin* file.

```
Advanced Memory Usage is available via "PlatformIO Home > Project
Inspect"RAM: [ ] 4.5% (used 14700 bytes from 327680 bytes)Flash:
[== ] 16.0% (used 209795 bytes from 1310720 bytes)esptool.py
v2.6
```

Shows the memory usage.

## Blinking an LED

Now we're ready to do the microcontroller's equivalent of "Hello World!": blinking an LED. Because this is the first program, we'll provide a majority of the source code. However, in the future it'll be as self-directed as possible. So let's get started.

Currently in your *main.cpp* file, it should be the default Arduino starting template plus the `#include <MCP23017.h>` line we added during the **compiling** section. Before we begin coding, we should familiarize ourselves with the MCP23017 microchip, the 16-bit IO (Input / Output) expander. This is because the LEDs, switches, and buttons are all connected to the IO expander. And so if we want to turn on an LED, we would need to learn how to interface with the IO expander.

Using an I2C scanner or looking at the provided MacIoT pinout, we were able to determine the address of the IO expander, which is 0x20. Now that we have an address, we can communicate with the IO expander. Now since we're planning on using the library, we need to know what functions the library has. So, we need to take a look inside the *MCP23017.h* header file.

It looks like there's an MCP23017 class, so let's go ahead and instantiate it. It needs the I2C address of the IO expander, which we already know to be 0x20. Let's go ahead and define the instance of the class to be called *mcp*. Looking at the header, there are two ways to instantiate our class.

```
1  MCP23017 mcp = MCP23017(const MCP23017 &)
2  MCP23017 mcp = MCP23017(uint8_t address, TwoWire &bus = Wire)
```

The first is just using a constant MCP23017 address. That's the easiest and it'll be the one we go with. Notice how Wire is defaulted (TwoWire&bus = Wire), so we only need to just pass the address. Alternatively, you can specify the Wire instance you want for the second argument. For example *MCP23017(0x24, Wire1)*. This should also let us know that we need to use the Arduino

Wire library and add `#include <Wire.h>` in our main source file.

Let us use a `#define` directive to define our address. It essentially let's us define constant (non-changing) macros in our source code. So now, we should have something like this:

```
#include <Wire.h>                // For our Instance of MCP23017
#include <Arduino.h>
#include <MCP23017.h>
// Defining the I2C Address of the MCP
#define MCP23017_ADDR 0x20
// Define an instance of the MCP23017 class
MCP23017 mcp = MCP23017(MCP23017_ADDR);
```

Some of you might be wondering how we're able to pass our hexadecimal address 0x20 as an unsigned integer of 8 bits (or 1 byte), designated as `uint8_t`. If you convert 0x20 into a decimal, you will get 32, which falls within the range of `uint8_t`(0 - 255).

Now in our `void setup()`, we'll need to join the I2C, because currently we're not.
`Wire.begin(address)`initiates the Wire library and joins the I2C bus as the master if we do not include any address. We'll use `Wire.begin(Serial.begin(9600))`to begin serial communication at a baud rate (bits per second) of 9600.

Looking through our library once again, we realize we need to initialize the MCP chip. Therefore, we must run the function *mcp.init()* in our setup, as this should be done once. On the IO expander, there are Port A and Port B GPIOs (General Purpose Input Output) pins. The MacIoT board has it set up so that the LEDs are on Port A, and the switches and buttons are on Port B. With that in mind, we need to set Port A as output (because we are writing to the GPIOs to turn LEDs on or off) and Port B as input (because we need are reading the GPIOs to see if the mechanical elements are changed).

As well, we'll need to reset the Ports A and B. This is because we want to start with values we know, and it'll reset any previously written values on the board. We'll therefore use the following library functions:

```
void portMode(MCP23017Port port, uint8_t directions, uint8_t pullups = 0xFF,
uint8_t inverted = 0x00);
void writeRegister(MCP23017Register reg, uint8_t value);
```

Seeing this, we'll need a port of class *MCP23017Port* and a register of class *MCP23017Register*. Looking at the header once again, we will find the necessary objects in their respective classes. Using the scope resolution operator `::`, we can access members of those classes, as they are outside the current scope of our program.

```
void setup()
{
// Join I2C bus - more on this in later chapter 8)
Wire.begin();
// Initialize MCP
mcp.init();
// Port A as Output (LEDS) || Port B as Input (Switches & Buttons)
mcp.portMode(MCP23017Port::A, 0); // 0 = Pin is configured as an output.
mcp.portMode(MCP23017Port::B, 0b11111111); // 1 = Pin is configured as an input.
// Reset Port A & B
mcp.writeRegister(MCP23017Register::GPIO_A, 0x00);  //Reset port A
mcp.writeRegister(MCP23017Register::GPIO_B, 0x00);  //Reset port B
}
```

So we define the port direction of `MCP23017Port::A` as 0, which configures it as an output. The port direction of `MCP23017Port::B` is a binary 0b11111111 (integer value 255). This configures the port to be an input. Finally the GPIO pins for Port A and B are resetted.

Now all that's left is to blink an LED of our choice. We will use the MCP `digitalWrite` function to choose the GPIO pin of our choice on Port A (LEDs). I'll simply be using the first one.

```
void digitalWrite(uint8_t pin, uint8_t state);
```

The function requires a GPIO pin, as well as the state, where 1 = logic high and 0 = logic low. To access the GPIO pins on Port A, the pin values correspond from 0 to 7. For Port B, they go from 8 to 15.

```
void loop()
{
// Pin 0 - 7    for port A
// Pin 8 - 15  for port B.
mcp.digitalWrite(0, 1);
delay(250);
mcp.digitalWrite(0, 0);
delay(250);
}
```

Finally throw some delays in between and you got yourself a blinking LED! Not bad!

# MODULE 2: BITS & BYTES

## General Overview

Module 2 will cover bits and bytes. This will not be a comprehensive guide to bits and bytes but the information here will be useful in later modules. More resources can be found below if you are interested in diving deeper.

### The Basics

Bits are essentially how data is communicated and stored within memory registers. Bits are how I2C, CAN, UART, and other communication protocols work and transfer data as well as how you can change memory registers to set specific functions.

- Bits are represented as 0s and 1s. They can be put together to form values such as 0000, 0001, 0010, 0011 and so on (0, 1, 2, 3 respectively in decimal). This is what binary is. A sequence of 0s and 1s.
- 8 bits = 1 byte
- 4 bits = 1 nibble
- A word is ambiguous and can represent 16, 32, 64, or more bits but depends on the system architecture
- Leading 0s mean nothing (ex. 0010 = 10 = decimal 2)
- 0b represents binary values (ex. X = 0b1010, X holds the binary value 1010)
- 0x represents hex values (ex. X = 0x10, X holds the binary value 00010000)
- Each digit in hex represents 4 bits. For example, 0x42 $\rightarrow$ 0b01000010

| Decimal (base 10) | Binary (base 2) | Hexadecimal (base 16) |
|---|---|---|
| 0 | 0000 | 0 |
| 1 | 0001 | 1 |
| 2 | 0010 | 2 |
| 3 | 0011 | 3 |
| 4 | 0100 | 4 |
| 5 | 0101 | 5 |
| 6 | 0110 | 6 |
| 7 | 0111 | 7 |
| 8 | 1000 | 8 |

| Decimal (base 10) | Binary (base 2) | Hexadecimal (base 16) |
|---|---|---|
| 9 | 1001 | 9 |
| 10 | 1010 | A |
| 11 | 1011 | B |
| 12 | 1100 | C |
| 13 | 1101 | D |
| 14 | 1110 | E |
| 15 | 1111 | F |

This module was just a refresher on important concepts when working with bits and bytes especially for C programming. It is highly recommended that you read through these two excellent resources on binary and hexadecimal before continuing.

`https://learn.sparkfun.com/tutorials/binary#bits-nibbles-and-bytes`
`https://learn.sparkfun.com/tutorials/hexadecimal`

# MODULE 3: INPUT & OUTPUT

## General Overview

Module 3 will go over the MacloTs various input and ouput capabilities. This module will cover the serial monitor, buttons, switches, and LEDs.

## Serial Monitor

We'll be doing a quick dive into the Arduino Serial Monitor, a very useful tool in the Arduino framework for prototyping, developing, and debugging. As the name suggests, it uses **Serial** communication.

To communicate with the Serial Monitor, there is the Arduino function `Serial.begin(speed)` . where **speed** is the baud rate (bits per second). Thus, all `Serial.begin()` is doing is opening the serial port and setting the data communication rate to a speed. To print to the Serial monitor, we use functions `Serial.print()` or `Serial.println()` . The difference between the two is that `Serial.println()` prints with the `\n` new line ASCII as well as the carriage return `\r`.

The `Serial.print(value, format)` function prints the data as ASCII text. Numerical values are printed with ASCII digit characters. The second parameter is **format**, which specifies the base for integer values, and the number of decimal points for float values.

```
#include <Arduino.h>
int num = 10;
float f_num = 10.12345;
void setup(void) {
    Serial.begin(9600);
}
void loop(void) {
    Serial.println("Hello from the Serial Monitor!");
    Serial.println(num); // print as an ASCII-encoded decimal
    Serial.println(num, DEC); // print as an ASCII-encoded decimal
    Serial.println(num, HEX); // print as an ASCII-encoded hexadecimal
    Serial.println(num, OCT); // print as an ASCII-encoded octal
    Serial.println(num, BIN); // print as an ASCII-encoded binary
    Serial.println(f_num); // print as an ASCII-encoded float, two decimal points
    Serial.println(f_num, 0); // print as an ASCII-encoded float, no decimal points
    Serial.println(f_num, 2); // print as an ASCII-encoded float, two decimal points
    Serial.println(f_num, 4); // print as an ASCII-encoded float, four decimal points
    delay(1000);
```

```
19    }
```

Opening the PlatformIO Serial Monitor we can see the output.

```
Hello from the Serial Monitor!
10
10
A
12
1010
10.12
10
10.12
10.1235
```

## Format

But what about printing to the same line? As briefly mentioned earlier, we can use `Serial.print()` and can do something like this:

```
1    Serial.print("num: ");
2    Serial.print(num);
3    Serial.print(" and f_num: ");
4    Serial.print(f_num);
5    Serial.print(" \n"); // Prints the new line ASCII character.
```

Which will get you **num: 10** and **f_num: 10.12** but isn't that a little tiresome? You can format it in different ways as you like so that it doesn't take as many lines of code:

```
1    Serial.print("num: "); Serial.print(num); Serial.print(" and f_num: ");
2    Serial.print(f_num); Serial.print(" \n");
```

After all, when the compiler sees the semicolon terminator **;** it'll recognize a termination. But this isn't a very clean way of formatting our code either. There's gotta be something better. And there is! Let's try using the `sprintf` C++ function. It stores the data into a C string buffer pointed by the `str` character pointer. We won't get into pointers in this chapter.

```
1    int sprintf ( char * str, const char * format, ... );
```

Let us create a character buffer which can hold 100 characters. We store our string into the buffer. Notice `%d` (decimal) and `%f` (float) which are **format specifiers**. When used the format specifiers are replaced by the values of the subsequent arguments. Note that order of the arguments matter!

```
    char buffer[100];
    sprintf (buffer, "num: %d and f_num: %f \n", num, f_num);
    Serial.print (buffer);
    delay(2000);
```

And there we go! The output is **num: 10** and **f_num: 10.123450**. This certainly makes formatting things a lot easier. Just be mindful of your buffer size!

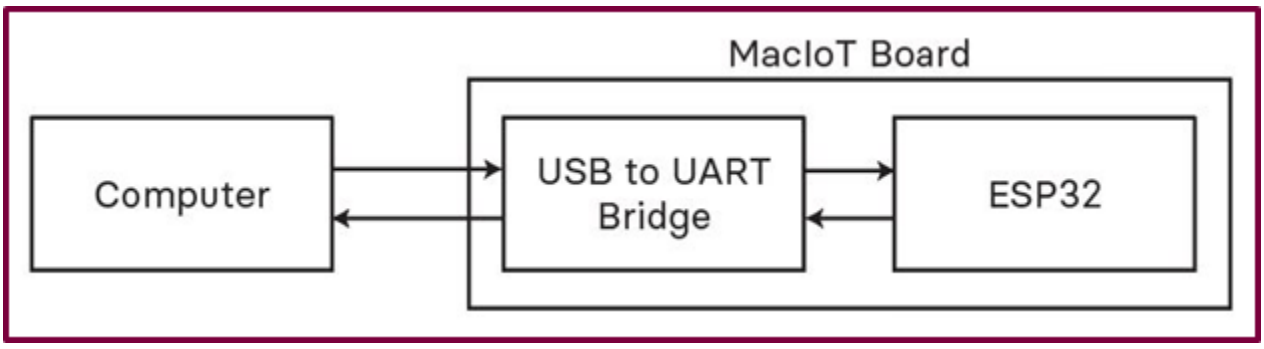
### Changing the Serial Baud Rate

How do we go about changing the Serial monitor Baud Rate? This is very simple. Simply go to your **platformio.ini** file and add: `monitor_speed = 115200` . It'll look something like this:

```
    [env:esp32dev]
    platform = espressif32
    board = esp32dev
    framework = arduino
    monitor_speed = 115200
```

### How Serial Information is Passed

Let's look at a simple diagram of how information is passed. On the MacIoT board, there is a **USB to UART bridge** (a chip) which handles converting the **Serial transmission** (bits per second) into **UART communication** (TX, RX). From there, the TX and RX pins of the bridge are connected to the TX and RX pins of the ESP32.




## Inputs

Now that we've gone through the Getting Started - mainly the **Blinking an LED** - we should have a better understanding moving forward. In this section, we'll be fine-tuning the concept of GPIOs to solidify our understanding.

## Buttons and Switches

On the MacIoT board, there are four switches, and four buttons. The switches and buttons are on **Port B** of the MCP IO Expander. For the MCP pin values, they are as follows (referencing the labels on the MacIoT board):

```
Switches:
SW3; GPB0; MCP Pin Value 8
SW4; GPB1; MCP Pin Value 9
SW5; GPB2; MCP Pin Value 10
SW6; GBP3; MCP Pin Value 11
------------
Buttons:
SW7; GBP4; MCP Pin Value 12
SW8; GBP5; MCP Pin Value 13
SW9; GBP6; MCP Pin Value 14
SW10; GPB7; MCP Pin Value 15
```

With this information, let's try writing an extremely simple code that will print a message to the terminal screen when we interact with a switch and the button. Let's use the switch **SW3** and the button **SW7** labeled on your MacIoT board. I'll assume you have the gist of how to properly write your `Setup()` function, and I won't bore you with the details.

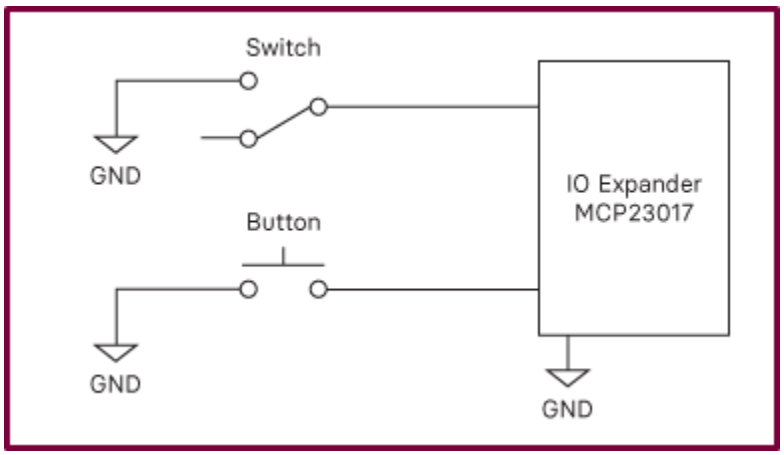
So this should be pretty straight forward, it should look something like this:

```
1  if (mcp.digitalRead(8) == 0) {
2      Serial.print("Switch On\n");
3  }
4
5  if (mcp.digitalRead(12) == 0) {
6      Serial.print("Button Pressed\n");
7  }
```

If we read a logic high on the respective pins, that means the switch or button has been pressed, and therefore our code works. Let's give it a shot and see what happens. Some of you might be wondering why we're checking if the pin reads 0 rather than 1. Checking the header file, you should've noticed:

```
/*
* Beware!
* On Arduino platform, INPUT = 0, OUTPUT = 1, which is the inverse
* of the MCP23017 definition where a pin is an input if its IODIR bit
* is set to 1.
* This library pinMode function behaves like Arduino's standard
* pinMode for consistency.
* [ OUTPUT | INPUT | INPUT_PULLUP ]
*/
```

Maybe some of you are unconvinced. Let's take another angle of approach. Below is a very simplified diagram that shows how buttons and switches are connected to the IO expander.




When the switch or button closes the circuit, the signal gets pulled to GND (ground), which will cause the value to be 0.

## Outputs

Now that we've covered the inputs, let's cover simple outputs: LEDs. Seeing as we have already gotten some exposure in our previous section - **Blinking an LED** - this section will be very brief and will just go a little more in detail.

There are eight LEDs that we can control, and are all on **Port A** of the MCP23017 IO expander.

```
LEDs:
D3; GPA0; MCP Pin Value 0
D4; GPA1; MCP Pin Value 1
D5; GPA2; MCP Pin Value 2
D6; GBA3; MCP Pin Value 3
D7; GBA4; MCP Pin Value 4
D8; GBA5; MCP Pin Value 5
D9; GBA6; MCP Pin Value 6
D10; GPA7; MCP Pin Value 7
```

Now that we know how to control inputs, let's use a button to toggle an LED, kind of like a switch. There's a catch though. We'll need to use latches so that the LED will still "latch on" to the value (whether it's a logic HIGH or LOW). That way, **we don't** have to constantly be holding down on the button in order for the LED to light. Looking at the header, there is an option for latching.

```
OLAT_A = 0x14 ///< Provides access to the port A output latches.
OLAT_B = 0x15 ///< Provides access to the port B output latches.
```

So, let's go ahead and enable latching on **Port A** outputs. In `Setup()` add:

```
mcp.writeRegister(MCP23017Register::OLAT_B, 0xFF);
```

Once that's done, we're ready to write implement our code. There's many ways of going about doing this, so don't be concerned if your answer looks slightly different. First let's write the condition statement to see if the button has been pressed to turn ON the LED.

```
if (mcp.digitalRead(12) == 0 && flag == 0) {
    Serial.print("Turn LED On\n");
    mcp.digitalWrite(0, 1);
    flag = 1;
    delay(250);
}
```

Notice the **flag** variable. This is to check the current state of the LED. If `flag == 0` , that means the LED is currently OFF. So when we read an INPUT value on MCP PIN 12 (GPB4 Button) and the state of the LED is OFF, we enter the `if statement` . We'll print a message to the serial monitor to let us know that we're currently inside the if statement, and we call `mcp.digitalWrite` to turn the LED (GPA0) on. Then, we change the state of the LED by setting `flag = 1` . Finally, there's a `delay(250)` of a quarter second. Why is this? This is because the `loop()` function is moving too fast, and there won't be proper latching shown. Try it without the delay and with the delay to see the difference.

```
else if (mcp.digitalRead(12) == 0 && flag == 1) {
    Serial.print("Turn LED Off\n");
    mcp.digitalWrite(0, 0);
    flag = 0;
    delay(250);
}
```

Now, the LED will still be on even if we release the button. So how do we turn it off now? So once again, we check if the button has been pressed. As well, we check to see if the status of the LED is currently on, `flag == 1` . If it is, we can enter the statement and turn the LED off, making sure to reset our flag, and the whole process restarts.

## Example Input and Output

```
#include <Arduino.h>
#include <MCP23017.h>
#include <Wire.h>

```

```
#define MCP23017_ADDR 0x20
uint8_t flag = 0;

MCP23017 mcp = MCP23017(MCP23017_ADDR);

void setup() {
  // put your setup code here, to run once:
  Wire.begin();
  Serial.begin(9600);
  mcp.init();

  mcp.portMode(MCP23017Port::A, 0x00);
  mcp.portMode(MCP23017Port::B, 0xFF);

  mcp.writeRegister(MCP23017Register::GPIO_A, 0x00);
  mcp.writeRegister(MCP23017Register::GPIO_B, 0x00);
  //Enables latching so the button doesn't have to be conitunally pressed
  mcp.writeRegister(MCP23017Register::OLAT_A, 0xFF);

  mcp.writePort(MCP23017Port::A, 0x00);
}

void loop() {
  // put your main code here, to run repeatedly:

  if (mcp.digitalRead(8) == 0) {
    Serial.print("Switch On\n");
    delay(250);
  }

  if (mcp.digitalRead(12) == 0 && flag == 0) {
    Serial.print("Turn LED On\n");
    mcp.digitalWrite(0, 1);
    flag = 1;
    delay(250);
  } else if (mcp.digitalRead(12) == 0 && flag == 1) {
    Serial.print("Turn LED Off\n");
    mcp.digitalWrite(0, 0);
    flag = 0;
    delay(250);
  }
}
```

# MODULE 4: SENSORS

## General Overview

Module 4 will cover I2C communication and the various sensors that are included on the MacIoT board. These sensors include the 9-Axis IMU, temperature and humidity sensors, and the light sensor. As well as how to create libraries.

## I2C Basics

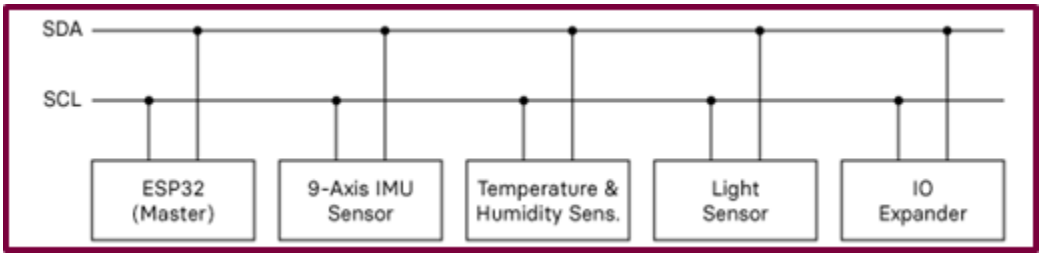
### How Does I2C Work?

I2C is a simple, serial protocol for low-speed devices. An I2C bus consists of a "leader", and other "follower" devices. The two wires that I2C use are SCL (serial clock) and SDA (serial data). Each follower device on the bus will have a unique address. When running an I2C scanner program on the MacIoT, the addresses of the devices can be found.

Excluding the LED Screen, when we run an I2C scanner script, we are able to determine the addresses of the devices on the I2C bus.

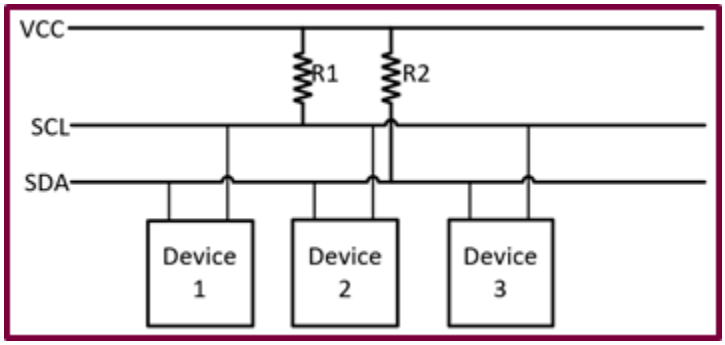
```
Scanning...
I2C device found at address 0x1C  !      9-Axis IMU Magnetometer
I2C device found at address 0x20  !      Input/Output Expander
I2C device found at address 0x40  !      Temperature and Humidity Sensor
I2C device found at address 0x52  !      Light Sensor
I2C device found at address 0x6A  !      9-Axis IMU
done
```

When we run `Wire.begin()`, we join the I2C bus as the leader. Knowing the addresses of these sensors allow us to interface and communicate with them. For the following sensors in this chapter, we'll explore each sensor in greater detail, and you will come to see how I2C communication is involved in each sensor.

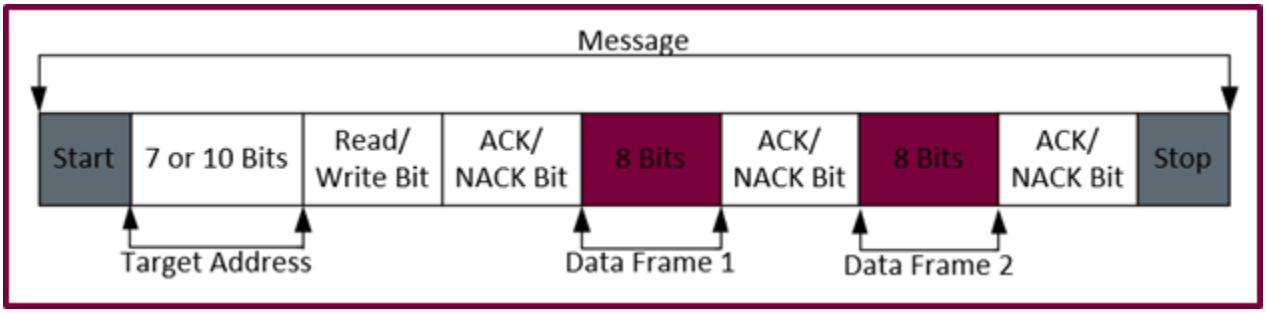


## The Hardware

There are two **open-drain** lines for I2C these include the serial data line (SDA) and the serial clock line (SCL). The SDA carries data in the form of bits and the SCL provides the clock signal. Open-drain (or open-collector) means the output pin is driven by a single transistor. The output pin voltage of the SDA and SCL line is either on or off. When the device is "off" the pin is left floating and hence requires a pull-up resistor to set its value (usually 5V or 3.3V). SDA and SCL lines are active LOW which means that to enable the pin it must have a logic 0 (0V) applied while logic 1 (5V as an example) will disable the pin.



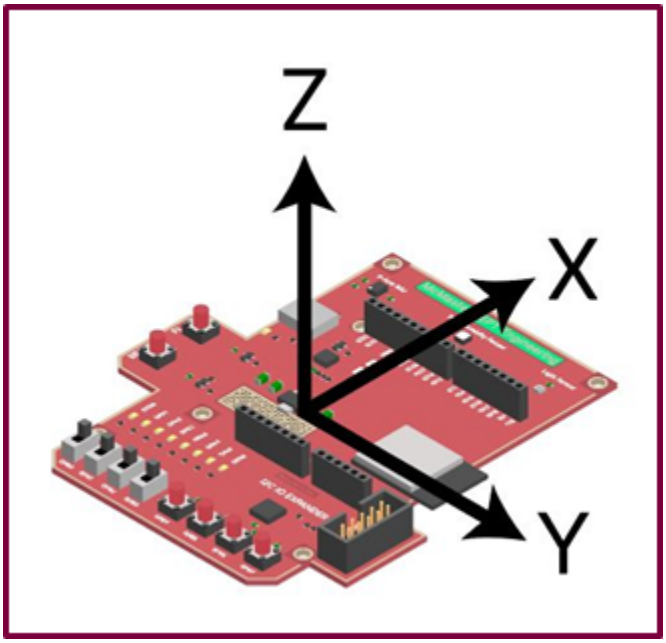

Data is sent in the form of bits. These are bits include start, address, acknowledge, read/write, data, and stop bits.




## 9-AXIS IMU Magnetometer

Onboard the MacIoT, there is a 9-Axis Inertial Measurement Unit (IMU) chip - the **LSM9DS1** (library). It has a 3-axis accelerometer, 3-axis gyroscope, and 3-axis magnetometer, hence the 9-axis name. But what exactly does this all mean? We'll explain a high level concept of each of them so that we gain a better understanding. IMU data is extremely important in our everyday life, so familiarizing ourselves with it is key!

According to the manufacturer's datasheet of the IMU chip, we're able to determine the axes for our board. This way, we can later verify with the data that we display later on. Once we know the axes, we can "map" certain movements and orientations of the board with applications, such as controlling a robot by tilting our board forward or reverse (a sneak peek into the distant future...).




## Setup

Let's look at the relevant codes. The library has simplified the amount of coding greatly, and thus the code itself is not that complicated. In the library source file **LSM9DS1.cpp**, the **IMU** object is instantiated with class `LSM9DS1Class`, looking something like this: `LSM9DS1Class IMU(Wire);`. The header file **LSM9DS1.h** has an extern call extern `LSM9DS1Class IMU;`. The extern keyword let's us know that our class will be externally linked, and it can be used in other files within the project scope.

```
#include <Arduino.h>
#include <Arduino_LSM9DS1.h>
void setup() {
    Serial.begin(9600);
    while (!Serial);
    Serial.println("Started");
    if (!IMU.begin()) {
        Serial.println("Failed to initialize IMU!");
        while (1);
    }
}
```

Our `setup()`code is very standard. After including the libraries, we'll test to see if we are able to communicate to the IMU sensor. If not, simply print that it fails to initialize.

## Accelerometer

As the name suggests, accelerometers are used to measure acceleration forces. The IMU will display the acceleration in the X, Y, and Z directions. First we have to check if there's even data available. Using `int LSM9DS1Class::accelerationAvailable()`, we're able to determine if there's any data.

The library function to get acceleration data is:
`int LSM9DS1Class::readAcceleration(float& x, float& y, float& z)`.

With three float parameters for X, Y, Z directions. Essentially, the registers containing the acceleration data is stored into an integer `data[3]` buffer, where the accelerations in all axes can be stored. Notice that the X, Y, Z are passed by reference with `float&`. This means that the address of these parameters are passed in, not the actual values of them.

After initializing our own X, Y, Z float parameters, the rest of the code is very straightforward.

```
1  if (IMU.accelerationAvailable()) {
2      IMU.readAcceleration(x, y, z);
3      Serial.print("Acceleration\n");
4      Serial.print(x);
5      Serial.print('\t');
6      Serial.print(y);
7      Serial.print('\t');
8      Serial.println(z);
9  }
```

Now all that's left to do is just print the values out. The `\t` character is an ASCII-encoded tab character. Which simply prints a tab. You may be wondering what the current values being outputted represent, especially if the board is stationary. So why are there values being outputted? Well, that's because even when the board isn't moving, there is always gravity affecting it! For example, if we were to tilt the board vertically, there would be very little gravity acting on the Z-axis.

You can also set the range of the accelerometer, which in turns will affect the sensitivity of your measurement. Higher ranges will have less precision but can measure larger movements. The function to set the range is `void LSM9DS1Class::setAccelerationRange(uint8_t range)`. The following ranges are:

```
#define    LSM9DS1_ACCELRANGE_2G  (0b00 << 3)
#define    LSM9DS1_ACCELRANGE_16G (0b01 << 3)
#define    LSM9DS1_ACCELRANGE_4G  (0b10 << 3)
#define    LSM9DS1_ACCELRANGE_8G  (0b11 << 3)
```

## Magnetometer

The magnetometer measures the strength and direction of magnetic fields near the MacIoT. The library function works the same way as the accelerometer, reading data from registers and storing them in buffers.

The function is:
`int LSM9DS1Class::readMagneticField(float& x, float& y, float& z)`

Of course we have to also check if any magnetometer data is available with:
`int LSM9DS1Class::magneticFieldAvailable()`.

```
    if (IMU.magneticFieldAvailable()) {
        IMU.readMagneticField(x, y, z);
        Serial.print("Magnetic Field\n");
        Serial.print(x);
        Serial.print('\t');
        Serial.print(y);
        Serial.print('\t');
        Serial.println(z);
    }
```

## Gyroscope

Finally, the gyroscope, which measures the rate of rotation over the axes. This helps with determining the orientation.

Check if gyroscope data is available: `int LSM9DS1Class::gyroscopeAvailable()`
Interface and read data: `int LSM9DS1Class::readGyroscope(float& x, float& y, float& z)`

```
    if (IMU.gyroscopeAvailable()) {
        IMU.readGyroscope(x, y, z);
        Serial.print("Gyroscope\n");
        Serial.print(x);
        Serial.print('\t');
        Serial.print(y);
        Serial.print('\t');
        Serial.println(z);
    }
```

## Flashing LEDs with the Accelerometer

Alright now all that's fine and dandy, but what can we actually do with this information? Well, one of the biggest things accelerometers are used for are airbags. That sudden negative acceleration will trigger the accelerometer reading and cause it to deploy the airbags in your cars. But we don't have airbags, so we'll work with what we have: LEDs!

So, let's do something a little bit different. Let's turn on the **Port A** LEDs by tilting the MacIoT board about the **X-axis**, so the readings of gravity on the **Z-axis** will vary. Regarding positive or negative rotations, the axes all follow the right-hand rule, where the direction of curl represents positive turns.

After checking if the accelerometer is available, let's check the Z-axis values. We are not concerned with the measurements in the X or Y axes. Go ahead and test the values to see the range. Depending on the amount we rotate, the values we read will be smaller or greater. All that's left is to just map an LED to a certain range. The values in the 1G range isn't much to work with, so I will use the 2G range: `IMU.setAccelerationRange(LSM9DS1_ACCELRANGE_2G);`.

```
    if (z > 0 && z <= 0.25) {
        mcp.digitalWrite(0, 1);
    } else if (z >= 0.26 && z <= 0.50) {
        mcp.digitalWrite(1, 1);
    } else if (z >= 0.51 && z <= 0.75) {
        mcp.digitalWrite(2, 1);
    } else if (z >= 0.76 && z <= 1) {
        mcp.digitalWrite(3, 1);
    } else if (z >= 1.01 && z <= 1.25) {
        mcp.digitalWrite(4, 1);
    } else if (z >= 1.26 && z <= 1.50) {
        mcp.digitalWrite(5, 1);
    } else if (z >= 1.51 && z <= 1.75) {
        mcp.digitalWrite(6, 1);
    } else if (z >= 1.76 && z <= 1.90) {
        mcp.digitalWrite(7, 1);
    } else if (z >= 1.91) {
        // Turn off LEDs.
    }
```

## Temperature and Humidity

So, let's get started! We'll be interfacing with the on-board **Si7020-A20** temperature and humidity sensor. Afterwards, we'll learn how to create a library with our own functions.

### Setup

As usual, we must include our standard Arduino libraries, `IMU.setAccelerationRange(#include <Arduino.h>);` and `#include <Wire.h>`, as we'll be working with I2C communication. The address of the sensor is **0x40**.

So in our setup, let us call `Wire.begin()`to join the I2C bus, as well as `Serial.begin()`at a baud rate of **9600** (note: make sure the .ini file matches this baud rate).

```
1   void setup() {
2       Wire.begin();
3       Serial.begin(9600);
4       // Start I2C Transmission
5       Wire.beginTransmission(ADDR);
6       Serial.print("Transmitting\n");
7       // Stop I2C Transmission
8       Wire.endTransmission();
9       delay(300);
10  }
```

We'll start transmitting to the **"follower"** sensor, with `Wire.beginTransmission` at the device's address. Once the serial message is printed, we'll know we're succesfully transmitting.

### Getting Humidity and Temperature

In our main loop, we will send the humidity measurement command **0xF5**. This information is given by the datasheet, so don't worry too much about it. We'll then request 2 bytes of data from the sensor. `Wire.available()`checks if there are two bytes available, and if they are, `Wire.read()` and store them in an **unsigned int** array, which I have initialized globally as `data[]`. Finally we'll convert the data as per the manufacturer's datasheet, to get the actual humidity in % RH.

```
1   Wire.beginTransmission(ADDR);
2   Wire.write(0xF5);                    // Send humidity measurement
3   Wire.endTransmission();
4   delay(500);
5
6   // Request 2 bytes of data
7   Wire.requestFrom(ADDR, 2);
8
```

```
    // Read 2 bytes of data; humidity MSB, humidity LSB
    if(Wire.available() == 2)
    {
        data[0] = Wire.read();
        data[1] = Wire.read();
    }

    // Convert the data
    float humidity  = ((data[0] * 256.0) + data[1]);
    humidity = ((125 * humidity) / 65536.0) - 6;
```

To get **temperature** data, it's the same process as getting humidity data, except we'll be passing in a different address command with `Wire.write()`.

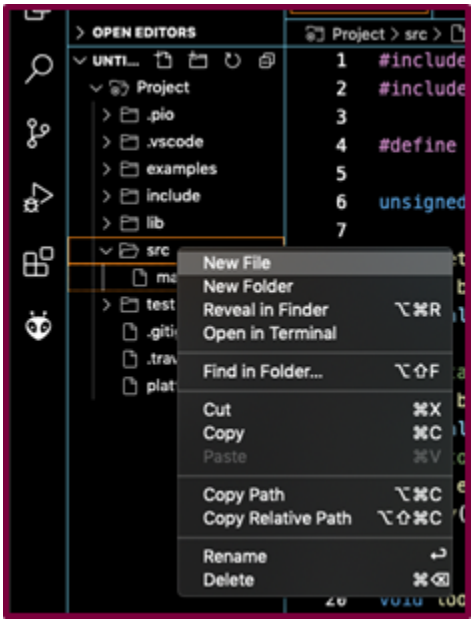
```
    Wire.beginTransmission(ADDR);
    Wire.write(0xF3); // Send temperature measurement
    Wire.endTransmission();
    delay(500);

    // Request 2 bytes of data
    Wire.requestFrom(ADDR, 2);

    // Read 2 bytes of data; temp MSB, temp LSB
    if(Wire.available() == 2)
    {
        data[0] = Wire.read();
        data[1] = Wire.read();
    }

    // Convert the data
    float temp  = ((data[0] * 256.0) + data[1]);
    float ctemp = ((175.72 * temp) / 65536.0) - 46.85;
    float ftemp = ctemp * 1.8 + 32;
```

Finally, we'll convert the data to Celsius and Fahrenheit according to the manufacturer's given equation. And that's pretty much it. Pretty straight forward.

## Creating a library

So, our temperature and humidity code looks unnecessarily long, especially with all the functions inside. So far as we've seen, we've been using libraries to help improve our coding process. We're going to go ahead and create a library for the functions above that we just wrote. This section is optional and for those who are interested.

To make things simple, first let us create the header and source files of our library within our current project `src`directory. Later, we will move it properly into the project's `lib`directory, as we have been doing with other libraries we are using. Let us call this library **TempHumid**, with a source file **TempHumid.cpp** and header file **TempHumid.h**.




After creating both files, create your library folder **TempHumid** under the `lib`directory of the main project. Within this library, create another directory called `src`which will contain **TempHumid.h** and **TempHumid.cpp** within. It should look something like this:

```
project_dir
├── include
│       └── README
├── lib
│    └── TempHumid
│            └── src
│                  ├── TempHumid.h
│                  └── TempHumid.cpp
├── platform.ini
├── src
│    └── main.cpp
└── test
       └── README
```

## TempHumid.h

So before we begin writing our header file, we must check and ensure that we do not have a reference to this library already. Otherwise we will compile it twice and run into some issues. So we will check to see if the library has been defined.

```
#ifndef TempHumid_H // If not defined
#define TempHumid_H // Define it

>> Header Code

#endif
```

Now, we are ready to get to the main part of the header file. We will create a class, called `TempHumid`. A class is a user-defined type or data structure that has its own data and functions (which are called members). So you can have things such as member variables or member functions. Access to these members can be private, protected, or public.

```
class class_name {
    access_specifier_1:
        member1;
    access_specifier_2:
        member2;
    ...
} object_names;
```

So for private members, only the class has access to these attributes. The public (anything that isn't the class itself) cannot access these members. So for our purposes, it'll look something like this:

```
class TempHumid {
    private:
        unsigned int data_[2];
        int deviceAddress_;
    public:
        // Constructor
        TempHumid();
        // Methods
        float getCelsius();
        ...
};
```

One thing we'll need for sure is a **constructor**. In C++, the constructor is a special method that will be be created when an object of the class is created. Now let us think about what other

methods we'll use, now that we have the full code already. Well we need information such as the temperatures in Celsius and Fahrenheit and Humidity. So something like `float getCelsius()`. See if you can come up with other methods you want to use in your own library.

I have initialized two private variables `unsigned int data_[2]`and `int deviceAddress_`. Note that the **underscore** _ is a convention for data members of classes.

## TempHumid.cpp

When writing our source code for the library, we'll include the relevant `#include <Wire.h>` library. In our constructor, we set `deviceAddress_`to be the address that is passed through to the constructor. Notice that when including the header **TempHumid.h** we are using just quotes, and not wrapping it brackets $< >$. That is because the files are in the same directory.

```
#include "TempHumid.h"
#include <Arduino.h>

TempHumid::TempHumid(int address) {
    deviceAddress_ = address;
}

float TempHumid::getCelsius() {
    ...
    return ctemp;
}
```

Finally, for writing our own functions, we can simply reference the code earlier. How you want to create our library is up to you. In other words, "This problem is left as an exercise for the reader". If you are ever stuck, reference other libraries that you have previously come across.

## main.cpp

So now your main source code should look something like this. I have my object `th` of **class TempHumid** with `TempHumid th = TempHumid(ADDR)`, passing in the device I2C address. I had three global variables for **humidity, fahrenheit, and celsius** values. The rest is up to how you setup your library.

```
#include <Arduino.h>
#include <Wire.h>
#include <TempHumid.h>
#define ADDR 0x40

float    humidity,
        fahrenheit,
```

```
            celsius;

    TempHumid th = TempHumid(ADDR);
    void setup(void) {
        ...
    }

    void loop(void) {
        ...
    }
```

## Light

Onboard the MacIoT sensor there is the **APDS9306** (library) sensor. Like the other sensors, it uses I2C interface. It is a digital ambient light sensor that converts light intensity to digital signals.

### Global Setup

As always, we link our libraries. I'll be using the asynchronous one, `#include <AsyncADPS9306.h>`. In short, the difference between the two is that with **synchronous** code, you must wait for it to finish before moving on, whereas **asynchronous** allows you to move on to another task before it finishes. There are situations when one is preferred over the other, but in our cases right now, it shouldn't matter which one.

Let's take a look at the **typedef enum** declaration in the library header. With the way they declared their enumeration, we can call `APDS9306_ALS_GAIN_t my_type`, where **my_type** is of enumerated type `APDS9306_ALS_GAIN_t`and can take on any of the enumerators listed as its value.

```
    typedef enum
    {
        APDS9306_ALS_GAIN_1 = 0x00,    // Enumerators
        APDS9306_ALS_GAIN_3 = 0x01,
        APDS9306_ALS_GAIN_6 = 0x02,
        APDS9306_ALS_GAIN_9 = 0x03,
        APDS9306_ALS_GAIN_18 = 0x04
    }
    APDS9306_ALS_GAIN_t;
```

So when we want to set the gain of the sensor, we initialize a variable of enumerated type `APDS9306_ALS_GAIN_t` and set the value to be one of the enumerators, such as `APDS9036_ALS_GAIN_1`, which will set our gain to be a value of one.

Now that we know how to work with enumerated types, let's go ahead and set the gain and time.

```
#include <AsyncAPDS9306.h>

AsyncAPDS9306 sensor;
const APDS9306_ALS_GAIN_t again = APDS9306_ALS_GAIN_1;
const APDS9306_ALS_MEAS_RES_t atime = APDS9306_ALS_MEAS_RES_16BIT_25MS;
```

We instantiate an object **sensor** of class `AsyncAPDS9306`. Looking through the library, we realize there are two things we must set: time and gain. Now for some of us, we might've came across something unfamilar, which is the **typedef enum**, or enumeration declaration. An enumeration type is one whose value is constrained to a certain range of values, which may include named constants.

## Setup

In our setup, all we're seeing is if the sensor can be located.

```
void setup() {
    if (sensor.begin(again, atime)) {
        Serial.print("Sensor found !");
    } else {
        Serial.print("Sensor not found !");
    }
}
```

Notice the class method `sensor.begin()`. The parameters that are passed are the gain and the time. Looking at the source **AsyncAPDS9306.cpp** file, we see that `AsyncAPDS9306::begin()` begins I2C transmission and calls `Wire.begin()`. It then sets the gain and time, using the following function:

```
int AsyncAPDS9306::_writeValue(uint8_t reg, uint8_t value, bool i2cStopMessage) {
    Wire.beginTransmission(APDS9306_I2C_ADDRESS);
    Wire.write(reg);
    Wire.write(value);
    return Wire.endTransmission(i2cStopMessage);
}
```

As you can see, all we're doing is passing the register command and sending it through the I2C to the sensor, using `Wire.beginTransmission()`.

## Getting Luminosity

In our for loop, we begin polling for measurements. **startTime** and **duration** are for determining the amount of time it took to get the sensor data. The `millis()` function is an Arduino function that gets the current time in milliseconds. Take the difference between the time at the start of the measurement and the end of the measurement will result in the amount of time passed.

```
    unsigned long startTime;
    unsigned long duration;
    Serial.println("Starting Luminosity Measurement");
    startTime = millis();
    sensor.startLuminosityMeasurement();   //Start sensor measurement
    while (!sensor.isMeasurementReady()){} //If sensor measurement is not ready,continue
    duration = millis() - startTime;           //Measurement time
```

Finally we instantiate an object **data** of class `AsyncAPDS9036Data`, in which we are able to calculate the luminosity.

```
    AsyncAPDS9306Data data = sensor.getLuminosityMeasurement(); //Get sensor measurement
    Serial.print(duration);
    Serial.print("ms (following datasheet, should be ");
    Serial.print(data._integrationTime(atime));
    Serial.println("ms");
    Serial.print(F("Raw: ")); Serial.println(data.raw);
    float lux = data.calculateLux();
    Serial.print("Luminosity : ");
    Serial.print(lux, 2);
    Serial.println("lux");
    delay(2000);
```

# MODULE 5: CAN

## General Overview

Module 5 will introduce CAN communciation. Controller Area Network (CAN) provides communication between various devices. One main application of CAN is within the automotive industry to send information from various sensors, ECUs, and other electronic devices.

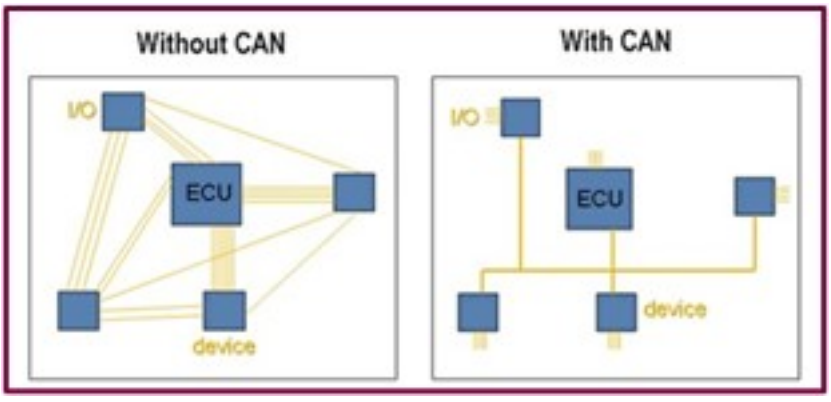
## Introduction to CAN

In this chapter, we will be focusing on the CAN (Controller Area Network) protocol. CAN is used extensively in many industries, most notably in automotive! It allows many different MCUs (microcontrollers) to communicate with each other. These MCUs can be thought of as **nodes** in a CAN bus.

To establish our own very simple CAN bus, we will need to download the library from here.

### Overview

In a typical implementation of a CAN bus, there are many different nodes connected. These nodes can share information to each other, all through the bus. Because there is so much information on the bus, nodes can decide which information is relevant to them, and accept or reject it. CAN interconnects all the nodes using only two wires. Without CAN, there would be way more wires to deal with.




SOURCE: NATIONAL INSTRUMENTS - CONTROLLER AREA NETWORK (CAN) OVERVIEW

CAN is multimaster, which means that any node on the bus can send a message. As well, it's multicast, which means all the nodes can receive the message, as we touched upon previously. Any node that sends a message is called a transmitter. And of course, we will have receiver nodes

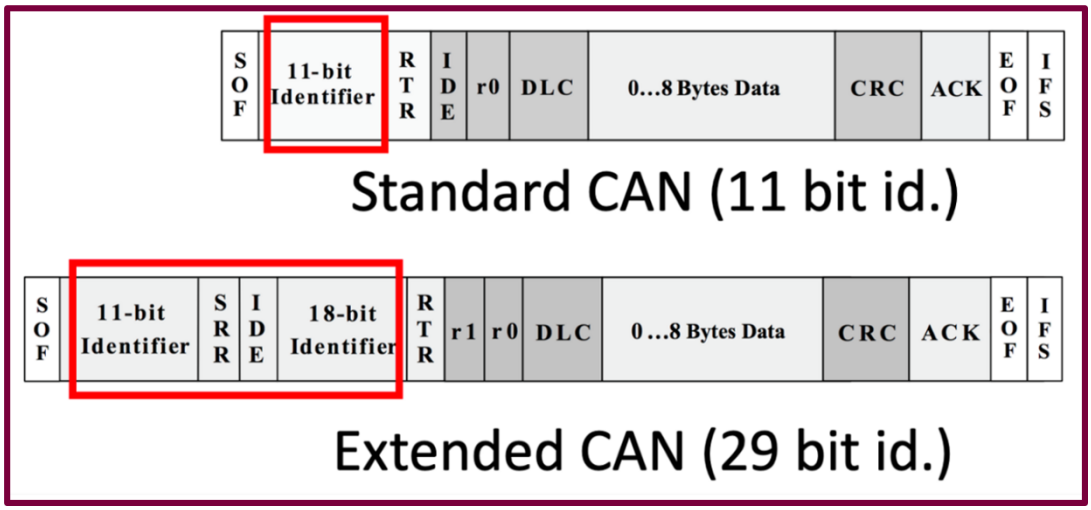
that receives the message.

But with every node sending messages, and the bus being a **serial** bus, how does the messages get differentiated? Each message will have its own priority assigned to it. For example, you would want messages regarding the engine or airbags of a car to have higher priority than the blinkers, right?

## CAN Frames

Let us explore CAN frames (messages). There are two main types of CAN frames:

- CAN 2.0A (Standard) - 11 bit Message Identifier
- CAN 2.0B (Extended) - 29 bit Message Identifier




SOURCE: TEXAS INSTRUMENTS - INTRODUCTION TO THE CONTROLLER AREA NETWORK

| Abbreviation | Abbreviation Meaning | Description |
|---|---|---|
| SOF | Start of Frame | Bit noting the start of a message |
| Identifier | Identifier | Establishes the priority of the message |
| SRR | Substitute Remote Request | (ext) Replaces the RTR bit in a standard message |
| IDE | Identifier Extensions | (ext) Indicates that there are more identifier bits |
| RTR | Remote Transmission Request | Bit is dominant when the information is required from another node |
| r1 | Reserved Bit | Reserved Bit |
| r0 | Reserved Bit | 6 Reserved Bit |
| DLC | Data Length Code | The number of bits that are in the payload |
| Data | Payload | Payload / data of up to 64 bits (8 bytes) |

| Abbreviation | Abbreviation Meaning | Description |
|---|---|---|
| CRC | Cyclic Redundancy Check | Error detection |
| ACK | Acknowledgment | Acknowledges if data received is good or bad |
| EOF | End of Frame | Notes the end of the message frame |
| IFS | Interframe Space | Time required by controller to move a correct frame to its buffer area |

## CAN Transmitter

Now we will be going over how to setup and write the CAN transmitter code so you can send CAN messages to other devices.

```
#include <CAN.h>

void setup() {
    Serial.begin(115200);
    while (!Serial);

    Serial.println("CAN Sender");

    if (!CAN.begin(500E3)) {
        Serial.println("Starting CAN failed!");
        while (1);
    }
}
```

The setup code seems pretty straight forward. We print "Starting CAN failed!" if we fail to connect to the CAN bus. The ! in front of `CAN.begin(500E3)` negates, so it's saying "If CAN doesn't begin at the specified speed". Here, the speed is 500 kbps is written in bps. The 500E3 is just the alternative way of writing 500,000 bps. Also something to note is that due to the limitations of the ESP32 hardware, the **BaudRate cannot be lower than 50 kbps**!

If we do go through the header file **ESP32SJA1000.h**, CAN is a member of class `ESP32SJA1000Class`. One of the public functions for this class is the `int ESP32SJA1000Class::begin(long baudRate)` function, which takes care of the lower level hardware configurations for our TX (transmit) and RX (receive) pins. Speaking of those pins, they are defined as:

```
#define DEFAULT_CAN_RX_PIN GPIO_NUM_4
#define DEFAULT_CAN_TX_PIN GPIO_NUM_5
```

Which corresponds exactly to how they are on the CAN bus shield, as those are the pins on the ESP32 chip! Now let's take a look at our main loop. We'll be focusing on the major parts.

If we check the **CANController.h** header file, we can see the following public definitions:

```
class CANControllerClass : public Stream {

public:
virtual int begin(long baudRate);
virtual void end();

int beginPacket(int id, int dlc = -1, bool rtr = false);
int beginExtendedPacket(long id, int dlc = -1, bool rtr = false);
```

So when we call `CAN.beginPacket(0x12)`, we are giving the message identifier a value of 0x12.

```
// send packet: id is 11 bits, packet can contain up to 8 bytes of data
Serial.print("Sending packet ... ");

CAN.beginPacket(0x12);
CAN.write('h');
CAN.write('e');
CAN.write('l');
CAN.write('l');
CAN.write('o');
CAN.endPacket();
```

Similarly for the the extended frame:

```
// send extended packet: id is 29 bits, packet can contain up to 8 bytes of data
Serial.print("Sending extended packet ... ");

CAN.beginExtendedPacket(0xabcdef);
CAN.write('w');
CAN.write('o');
CAN.write('r');
CAN.write('l');
CAN.write('d');
CAN.endPacket();
```

Each char we write with `CAN.write` has a size of 1 byte. Our packets can only send up to 8 bytes of data!

## CAN Receiver

Now that one of our nodes is transmitting data, let's set up another node to receive this message.

```
#include <CAN.h>

void setup() {
    Serial.begin(115200);
    while (!Serial);

    Serial.println("CAN Receiver");

    if (!CAN.begin(500E3)) {
        Serial.println("Starting CAN failed!");
        while (1);
    }
}
```

As you might've noticed, the setup is the exact same as in the previous page for the **CAN Transmitter**. Again, we are setting our BaudRate at 500 kbps.

In our main loop, we will now be parsing the data packet.

```
int packetSize = CAN.parsePacket();
```

We check if the packet we received is standard or an extended frame.

```
if (packetSize) {
// received a packet
Serial.print("Received ");

if (CAN.packetExtended()) {
    Serial.print("extended ");
}

if (CAN.packetRtr()) {
    // Remote transmission request, packet contains no data
    Serial.print("RTR ");
}

Serial.print("packet with id 0x");
Serial.print(CAN.packetId(), HEX);

if (CAN.packetRtr()) {
```

```
18          Serial.print(" and requested length ");
19          Serial.println(CAN.packetDlc());
20      } else {
21          Serial.print(" and length ");
22          Serial.println(packetSize);
23
24          // only print packet data for non-RTR packets
25          while (CAN.available()) {
26          Serial.print((char)CAN.read());
27          }
28          Serial.println();
29      }
30
31      Serial.println();
32      }
```

The rest of the logic is pretty straight forward. Prints the frame ID, and the length of the payload or data. Finally, it'll print the actual data for non-RTR packets.

# MODULE 6: WiFi and MQTT

## General Overview

Module 6 will go over the networking WiFi and MQTT. The module will also go over JSON which will be used to transmit the data.

## JSON Files

In these next few labs, you will learn the basics of connecting to WiFi and sending data to MQTT (MQ Telemetry Transport). As well, you will learn about JSON (JavaScript Object Notation) files, which store simple data structures and is a standard data interchange format. With most data that we move around, we want to keep it as a JSON, so that it may be easily translated by other programs if necessary. As we are not concerned with efficiency, JSON covers many applications that we'll be using.

### JavaScript Object Notation (JSON)

JSON files can house strings, numbers, arrays, booleans, etc.

JSON has the following format:

- Data is in key/value pairs
- Data is sperated by commas
- Curley braces hold objects
- Square braces hold arrays

JSON formats look something like this:

```
var person = {
    firstname: "John",
    lastname: "Doe",
    age: 22,
    city: "Toronto"
};
```

Or in our cases, something like this:

```
var sensordata = {
    temperature: 1234 ,
    humidity: 1234,
    light: 1234};
```

## Serializing JSON

So let's print a JSON object to the Serial Monitor! We'll be building off the Temperature and Humidity sensor code we wrote previously. We need to instantiate our **JsonDocument** object. This JSON document can either be of type `StaticJsonDocument` (recommended for documents smaller than 1KB) or `DynamicJsonDocument` (recommended for documents larger than 1KB).

When calculating the size of our JSON object, we can use the ArduinoJSON Assistant. The second way is to compute the capacity with macros. If we know the exact layout of the document we'll be working with, you can easily compute the required size. The two macros that you can use are `JSON_OBJECT_SIZE(n)` (returns size of JSON object with n members) and `JSON_ARRAY_SIZE(n)` (returns the size of JSON array with n elements). So if we know that we have a JSON document: `{"values":[1,2,3]}`. We know that there is one object, which is an array with three elements. Therefore the capacity will be:

```
const size_t capacity = JSON_OBJECT_SIZE(1) + JSON_ARRAY_SIZE(3) + 7;
```

Static declaration will use a template parameter, where size is defined:

```
StaticJsonDocument<bytes> doc;
```

Dynamic declaration will use the constructor argument:

```
DynamicJsonDocument doc(2048);
```

We'll globally declare our `DynamicJsonDocument doc(2048)`. Now let's store the sensor readings into our JSON object. We can do this by

```
doc["humidity"] = humidity;
doc["fahrenheit"] = fahrenheit;
doc["celsius"] = celsius;
```

Now all that's left is to output our data!
With the function: `size_t serializeJson(const JsonDocument& doc, String& output):`

```
    serializeJson(doc, Serial);
    Serial.println();
```

Giving us the following output:

```
    {
        "humidity":23.11758,
        "fahrenheit":79.21547,
        "celsius":26.24154
    }
```

## Deserializing JSON

Let's look at how we deserialize JSON data, with an array example. We must declare a character array that will house the JSON data.

```
char json[] = "{
    \"sensor\":\"TempHumid\",
    \"humidity\":23.11758,
    \"temperature\":[79.21547,23.11758]
    }";
```

Notice the use of the back slash \". This is to let the compiler know that we're using the escape sequence \" symbol for special characters within string literals and character literals.

Now all we need to do is assign variables to the values we need:

```
    const char* sensor = doc["sensor"];
    float humidity = doc["humidity"];
    float fahrenheit = doc["temperature"][0];
    float celsius = doc["temperature"][1];
```

Printing the variables give us:

```
TempHumid
23.12
79.22
23.12
```

## Connecting to WiFi

Now, we'll be connecting to the internet! The MacIoT wouldn't be an IoT board if we couldn't connect to the great beyond! We'll be using Arduino's `#include<WiFi.h>` library to connect. An important thing is to keep in mind is that the ESP32 can only connect to **2.4 GHz** internet, and not the fancy **5 GHz**.

We'll declare two `const char* char` pointers, the **const** telling the compiler these values will not change. **SSID** (Service Set Identifier) is the name of the 2.4 GHz WiFi network we're connecting to, and **PASSWORD** is self explanatory, the password of said network.

```
const char* SSID = "SSID";
const char* PASSWORD = "PASSWORD";
```

The function `WiFi.begin(ssid, pass)` initializes the WiFi library's network settings. The function returns `WL_CONNECTED` if successfully connected to the internet. Thus, we are checking if `WiFi.status()` has successfully connected, `WL_CONNECTED`. If not, we will print **.** periods to signify connecting.

```
void setup() {
    Serial.begin(115200);
    WiFi.begin(SSID, PASSWORD);
    Serial.print("Connecting to "); Serial.print(SSID);

    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial.print('.');
    }

    Serial.println('\n');
    Serial.println("Connection established!");
    Serial.print("IP address:\t");
    Serial.println(WiFi.localIP());
}
```

Finally, if we succesfully connected, we'll print the IP address of the MacIoT board, using `WiFi.localIP()`, giving us the following output:

```
Connection established!
IP address:     192.168.1.65
```

## JSON Files from the Internet

Now that we know how to connect to the internet and work with JSON files, let's combine the two and grab weather data from the internet!

## OpenWeatherMap

We're going to register on OpenWeatherMap to get an API (Application Programming Interface) key. API keys are a unique code that is passed in to an API to identify the calling application or user. The API for the current weather in a city is given by one of the following all ending in `&appid={your api key} (one example shown)`:

```
api.openweathermap.org/data/2.5/weather?id={city id}&appid={your api key}
api.openweathermap.org/data/2.5/weather?lat={lat}&lon={lon}
api.openweathermap.org/data/2.5/weather?zip={zip code},{country code}
api.openweathermap.org/data/2.5/weather?q={city name}
api.openweathermap.org/data/2.5/weather?q={city name},{state}
api.openweathermap.org/data/2.5/weather?q={city name},{state},{country code}
```

I'll use Toronto as an example. The City ID according to the website `https://openweathermap.org/city/6167865` is **6167865**. Filling in the information for the API call, I get the following data back:


```
{
    "coord": {
      "lon": -79.42,
      "lat": 43.7
    },
    "weather": [
      {
        "id": 800,
        "main": "Clear",
        "description": "clear sky",
        "icon": "01d"
      }
    ],
    "base": "stations",
    "main": {
      "temp": 280.57,
      "feels_like": 272.3,
      "temp_min": 279.82,
      "temp_max": 281.48,
      "pressure": 1019,
      "humidity": 33
    },
    "visibility": 14484,
    "wind": {
      "speed": 7.7,
      "deg": 270
    },
    "clouds": {
```

```
    "all": 1
  },
  "dt": 1589299821,
  "sys": {
    "type": 1,
    "id": 941,
    "country": "CA",
    "sunrise": 1589277315,
    "sunset": 1589329976
  },
  "timezone": -14400,
  "id": 6167865,
  "name": "Toronto",
  "cod": 200
}
```


## Using the API in Code

We'll need to include the `#include <ArduinoJson.h>`, `#include <SPI.h>`, and `#include <WiFi.h>` libraries. We need to declare our **SSID** and **PASSWORD** character pointers to connect to WiFi, like we did in the **Connecting to the Internet section**.

As well, we'll create a dynamic JSON document. Let us create strings **APIKey** and **cityID** that will hold our API key and the ID of the city on OpenWeatherMap. There are actually many different ways to do this, depending on how you are getting your weather data. I will be using the method where you pass in the cityID.

```
const char* SSID = "SSID"
const char* PASSWORD = "PASSWORD"
const char* SERVER = "api.openweathermap.org";
const String APIKey= "95df79396d5682651a5dbf2fd593fb65";
const String cityID = "6167865";
DynamicJsonDocument doc(1024);
WiFiClient client;
```

Finally we declare a **client** object of class `WiFiClient`. This creates a client that will let us connect to specified internet addresses.

### getWeather()

Now let's connect to OpenWeatherMap to make our API calls. We must first check if we are able to connect to the website, using `client.connect(IP/URL, PORT)`. **Port 80** is the default port for HTTP connection. Using `client.print`, we are able to make a **HTTP GET** request. The `client.print` prints data to the server that we are connected to. This tells the server that we are requesting to GET data. Which data specifically? The string of text after the URL:

`http://api.openweathermap.org/data/2.5/weather?id={cityID}&appid={APIKey}`

So therefore we specify the parameters that we want in accordance to how OpenWeatherMap uses their API keys.

```
Serial.println("\nStarting connection to server...");
    if (client.connect(SERVER, 80)) {
        Serial.println("Connected to server");

        // Make a HTTP request:
        client.print("GET /data/2.5/forecast?");
        client.print("id="+cityID);
        client.print("&appid="+APIKey);
        client.println("&units=metric");
        client.println("Host: api.openweathermap.org");
        client.println("Connection: close");
        client.println();
    } else {
        Serial.println("Unable to connect");
    }
```

Finally, now that we've established connection, all we need to do is parse, or deserialize the data. There are many ways of parsing your JSON data. I will use the simplest method for now. We can use the ArduinoJson Assistant to help us breakdown the JSON data that we received, or we can use a JSON formatter to help understand exactly what we're looking at. Either way works.

The function `readStringUntil()` lets us read the data JSON data until we reach the `\n` terminator character. Once we have the JSON data from the website, we can deserialize using the `deserializeJson()` function, and retrieve the data we're interested in as we did in the JSON Files section.

# MQTT

Now we are ready to send data using MQTT (MQ Telemetry Transport). The MQTT protocol is a publish / subscribe messaging system that's meant for machine to machine (M2M) telemetry. It is an extremely popular protocol for IoT development.

## Publiush/Subscribe

So what exactly do we mean by publish and subscribe? There are **topics** that get **published and subscribed**. The devices, or **clients** connected to the **broker** will subscribe to the topics that they need. The clients themselves can also publish to the broker, publishing messages to topics. Many devices can subscribe to a topic.

So, let's use this lab as an example. The MacIoT board (device / client) will publish sensor data

(temperature / humidity) to their respective topics esp32/Temperature and esp32/Humidity. A subscription could be an exact topic (word for word), in which case only messages to that topic will be received. However, it can also be wildcards which are + or #.

```
device/sensors/sensorType/reading
+  /sensors/sensorType/reading
device/   +   /sensorType/reading
device/sensors/     +    /reading
+  /   +   /     +    /    +
```

We'll be using the `#include <PubSubClient.h>` library. This library is used for simple publish and subscribing with an MQTT server. The other libraries we'll use are `#include <SPI.h>`, `#include <WiFi.h>`, and finally the optional library depending on whether you decided to write your own library: `#include <TempHumid.h>`.

## MQTT Code

Aside from the libraries and our TempHumid variables, we'll go through some of the global variables, and what they're used for. First off, we have the `uint64_t chipid`. This is for storing the ESP32s MAC address, which has a length of six bytes. Each ESP32 has their own unique MAC address, which you can actually change.

The next lines are for connecting to the internet and the MQTT broker. TCP/IP port 1883 is registered with IANA (Internet Assigned Numbers Authority). Port 8883 is for SSL (Secure Sockets Layer).

We are calling creating an object client of class `PubSubClient`, passing in the **espClient**. Finally, the string variables and colorsubsectionclr`char` buffers are for sending and receiving data.

```
1   const char* SSID = "SSID"
2   const char* PASS = "PASS"
3   const char* MQTT_SERVER = "149.248.57.163";
4   const int MQTT_PORT = 1883;
5
6   WiFiClient espClient;
7   PubSubClient client(espClient);
8
9   String tempString;
10  String humidString;
11  String strTest;
12
13  char tempBuffer[20];
14  char humidBuffer[20];
15  char receiveBuffer[100];
```

The callback function is called when a message arrives for subscription. Essentially, this is the function that displays any messages it receives from subscription. It will print the payload from the topic.

```
void callback(char* topic, byte* payload, unsigned int length) {
    Serial.print("Message arrived [");
    Serial.print(topic);
    Serial.print("] ");

    for (int i=0; i<length; i++) {
        Serial.print((char) payload[i]);
    }
    Serial.println();
}
```

The function `wifiConnect()` was explained in the **Connecting to WiFi chapter**. It functions exactly the same way, connecting to WiFi through the **SSID** and **PASS** we assigned above. Keep in mind that that your WiFi must be 2.4 GHz. Otherwise, it will not connect.

```
void wifiConnect() {
    Serial.begin(115200);
    delay(50);

    WiFi.begin(SSID, PASS);
    Serial.print("Connecting to "); Serial.print(SSID);
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial.print('.');
    }

    Serial.println('\n');
    Serial.println("Connection established!");
    Serial.print("IP address:\t");
    Serial.println(WiFi.localIP());
}
```

As the name implies, the `publish()` function let's us publish our data to the MQTT broker. Here, we are defining the name of our client to be the string "MacIoT-" and the Client ID of the ESP32. In my case, it is `MacIoT-384db3bf713c`. If `client.connect()` checks if the MQTT client has connected to the broker. We are passing the name of our Client `clientId.c_str()` to the broker. The function `c_str()` converts the string to a null terminated array of bytes. If it connects, we will publish a hello message, using the function `int publish (topic, payload)`. As well, we are subscribing using the function `boolean subscribe (topic, [qos])`, and we are subscribing to a wildcard, thus any topic of one word form. If we do not connect to the

broker, we retry the connection.

```
void publish() {
    while (!client.connected()) {
        Serial.print("Attempting MQTT connection...");
        String clientId = "MacIoT-";
        clientId += strTest;

        if (client.connect(clientId.c_str())) {
            Serial.println("Connected");
            // Once connected, publish an announcement...
            client.publish("testConnect", "Hello from ESP32!");
            client.subscribe("+");
        } else {
            Serial.print("failed, rc=");
            Serial.print(client.state());
            Serial.println(" try again in 5 seconds");
            // Wait 5 seconds before retrying
            delay(5000);
        }
    }
}
```

The function `sendTempHumid()` uses the library we built in our previous chapter to publish the sensor readings. The variables **tempString** and **humidString** are objects of the class String. It constructs a String from the float results in a string that contains the ASCII representation of that reading. The default is base ten. In order to send the data to the MQTT broker using the publish function, the values that we send must be in the form of character arrays. Thus, we will use the function `stringVariable.toCharArray(buf, len)`. We have initialized the character buffers globally. The length of the the character array is the length of the string + 1 for the terminating character.

```
void sendTempHumid() {
    humidity = th.getHumidity();
    celsius = th.getCelsius();
    tempString = String(celsius);
    humidString = String(humidity);
    tempString.toCharArray(tempBuffer, tempString.length() + 1);
    humidString.toCharArray(humidBuffer, humidString.length() + 1);
    client.publish("esp32/Temperature", tempBuffer);
    client.publish("esp32/Humidity", humidBuffer);
}
```

In our setup, we initialize our **TempHumid** class and begin serial communication with the Serial

Monitor. The function `ESP.getEfuseMac()` gets the Chip ID of the ESP32. When printing the upper two bytes (2 bytes = 16 bits), we use a bitwise operator >> to shift the value by 32. Two buffers are created to store the values of the upper two bytes and lower four bytes. The String variable `strTest` is set to the sum of the two strings. Finally at the end of our setup, we use the function `setServer(server, port)` to set the broker details. The `setCallback(callback)` function sets the message callback function which we defined above.

```
void setup() {
    th.init();
    Serial.begin(115200);
    //The chip ID is essentially its MAC address (length: 6 bytes).
    chipid = ESP.getEfuseMac();
    // Print Upper 2 Bytes
    Serial.printf("ESP32 Chip ID = %04X", (uint16_t)(chipid>>32));
    // Print Upper 2 Bytes
    Serial.printf("%08X\n", (uint32_t)chipid);

    char buffer1[10]; char buffer2[10];
    char *intStr1 = itoa((uint16_t)(chipid>>32), buffer1, 16);
    char *intStr2 = itoa((uint32_t)chipid, buffer2, 16);

    strTest = String(intStr1) + String(intStr2);
    wifiConnect();
    client.setServer(MQTT_SERVER, MQTT_PORT);
    client.setCallback(callback);
}
```

Finally in our main loop, we test the connection to the server. If we're not connected, we attempt to connect with our `publish()` function, to publish a hello message and subscribe to the wildcard topic. The `loop()` function is regularly called to allow the ESP32 to process incoming messages and maintain its connection to the server. We call `sendTempHumid()` function to send the temperature and humidity data to the broker. And that's it!

```
void loop() {
    if (!client.connected()) {
        publish();
    }
    client.loop();
    sendTempHumid();
    delay(5000);
}
```

# MODULE 7: LoRa

## General Overview

Module 7 will go LoRa and how to setup senders and receivers.

## Introduction to LoRa

In these labs, we'll be looking into LoRa, or **Lo**ng **Ra**nge technology. LoRa is very popular in the field of IoT (Internet of Things), having devices communicate with each other wirelessly through frequency. As well, there is LoRaWAN (LoRa Wide Area Network), and while they're related, they are different from each other. So don't get them mixed up thinking they're the safe!

But what really separates a LoRa device and your standard devices? A simple of way of looking at it is that LoRa devices are able to connect to a network, where many other devices are also connected to the same network. Just think about all the "smart" devices, and how you're able to control them wirelessly. There's a standard that all LoRa devices have when connecting to networks.

Since LoRa is **low power**, it uses low bandwidths, where the tradeoff is then greater range distance communication. As a comparison, Wi-Fi would be high bandwidth and low distance range.

The library we'll be using for these labs is located here.

## Reciever

We'll be setting up two devices, one as a receiver, and one as a sender. As usual, we'll set up Serial communication at a **BaudRate of 115200**. We connect to the LoRa shield using `SPI.begin(14, 12, 13, 27)`, defined as follows:

```
1    void SPIClass::begin(int8_t sck, int8_t miso, int8_t mosi, int8_t ss)
```

So if you look at the pinouts from the ESP32 on the MacIoT board, we'll be able to determine the appropriate pins used.

```
1    #include <Arduino.h>
2    #include <SPI.h>
3    #include <LoRa.h>
4
5    void setup() {
```

```
    Serial.begin(115200);
    while (!Serial);

    Serial.println("LoRa Receiver");
    SPI.begin(14, 12, 13, 27);
    LoRa.setPins(27, 25, 26);
    if (!LoRa.begin(868E6)) {
        Serial.println("Starting LoRa failed!");
        while (1);
    }
}
```

The function `LoRaClass::setPins` sets the pins. Note that the slave select ss is the same as defined in the SPI definition.

```
void LoRaClass::setPins(int ss, int reset, int dio0)
{
    _ss = ss;
    _reset = reset;
    _dio0 = dio0;
}
```

In our for loop, we parse the packet using `LoRa.parsePacket()`. If packetSize is not null, that means we've received a packet. The function `LoRa.read()` prints the actual message packet:

```
int LoRaClass::read()
{
    if (!available()) {
        return -1;
    }

    _packetIndex++;
    return readRegister(REG_FIFO);
}
```

Notice the name of the register, indicating that it's a FIFO, which stands for First-In, First-Out. As the name implies, messages that are received first in the queue, will be removed / read first.

```
void loop() {
    // try to parse packet
    int packetSize = LoRa.parsePacket();
    if (packetSize) {
    // received a packet
```

```
        Serial.print("Received packet '");

        // read packet
        while (LoRa.available()) {
            Serial.print((char)LoRa.read());
        }

        // print RSSI of packet
        Serial.print("' with RSSI ");
        Serial.println(LoRa.packetRssi());
        }
    }
```

Finally, we'll print the RSSI of the packet. RSSI stands for **R**eceived **S**ignal **S**trength **I**ndicator which determines the signal strength received from a router, or in our case, our sender.

## Sender

Our sender code will look somewhat similar to our Receiver code. Make sure you have gone through the Receiver page for a more in-depth explanation with regards to some aspects.

We globally initialize a variable called counter which will increment the message number we have sent.

```
#include <Arduino.h>
#include <Wire.h>
#include <SPI.h>
#include <LoRa.h>

int counter = 0;

void setup(void) {
    pinMode(14, OUTPUT);
    digitalWrite(14, LOW);

    Wire.begin(21, 22);

    SPI.begin(14, 12, 13, 27);
    LoRa.setPins(27, 25, 26);
    if (!LoRa.begin(868E6)) {
        while (1);
    }
}
```

In our loop code, we send our packet "hello" as a string, as well as the message number counter. Data packets that we send are wrapped with the functions `LoRaClass::beginPacket()` and `LoRaClass::endPacket()` to signal the start and end of message transmission.

```
1   void loop(void) {
2       LoRa.setTxPower(0);
3       LoRa.beginPacket();
4       LoRa.print("hello ");
5       LoRa.print(counter);
6       LoRa.endPacket();
7
8       counter++;
9       delay(1000);
10  }
```

# MODULE 8: SENSOR VISUALIZATION

## General Overview

As we have discussed the MacIOT board houses the **temperature** and **humidity** sensor (**Si7020-A20**) pairing this with python and matplotlib we can develop a program that is able to plot live temperatures and humidity. This can be used in a variety of ways such as a temperature sensor withing a battery pack that can show the endpoint user the live temperature to the humidity sensor monitoring safety critical components. We can also use the **IMU** to visualize in 3D the rotation of the board.

## main.cpp

The start of this program is with the main.cpp file. This will take the I2C data from both the IMU and temperature and humidity sensors and send them serially to the python program that will run all our visualizations.

### setup()

We will start from importation the IMU library from (here) and putting `#include <LSM9DS1.h>` and `#include <Wire.h>` at the top. We will also set a global int data[2] which will be used to store the MSB and LSB of the LSM9DS1 readings. We also need to define a variable called declination which is the disparity between the true north angle vs magnetic north `#define DECLINATION -10.25` (you can calculate yours here)

```
    Wire.begin();
    Serial.begin(9600);
    // Start I2C Transmission
    Wire.beginTransmission(0x40);
    Serial.print("Transmitting\n");
    // Stop I2C Transmission
    Wire.endTransmission();

    while (!Serial);
        //Serial.println("Started");
        if (!IMU.begin()) {
            Serial.println("Failed to initialize IMU!");
            while (1);
        }
    delay(300);
```

## printData()

This function will print the **heading, pitch, roll, temperature (in Celsius) and humidity**. Pitch and roll calculations are found here as well as the heading calculation found here.

The function is defined as:

```
void printData(float ax, float ay, float az, float mx, float my, float mz){}
```

Inside the function we pass in the sensor readings from the IMU and perform the calculations based on the documentation that was linked.

```
//IMU DATA
float roll = atan2(ay, az);
float pitch = atan2(-ax, sqrt(ay * ay + az * az));

float heading;
if (my == 0)
heading = (mx < 0) ? PI : 0;
else
heading = atan2(mx, my);

heading -= DECLINATION * PI / 180;

if (heading > PI) heading -= (2 * PI);
else if (heading < -PI) heading += (2 * PI);

// Convert everything from radians to degrees:
heading *= 180.0 / PI;
pitch *= 180.0 / PI;
roll  *= 180.0 / PI;
```

To receive the temperature the same code is used from **Module 4**.

```
//TEMP DATA
Wire.beginTransmission(0x40);
Wire.write(0xF3); // Send temperature measurement
Wire.endTransmission();
delay(100);
// Request 2 bytes of data
Wire.requestFrom(0x40, 2);
// Read 2 bytes of data; temp MSB, temp LSB
if(Wire.available() == 2)
{
```

```
    data[0] = Wire.read();
    data[1] = Wire.read();
    }

    // Convert the data
    float temp  = ((data[0] * 256.0) + data[1]);
    float ctemp = ((175.72 * temp) / 65536.0) - 46.85;
```

. . . and the humidity data.

```
    //HUMID DATA
    Wire.beginTransmission(0x40);
    Wire.write(0xF5);  // Send humidity measurement
    Wire.endTransmission();
    delay(100);
    // Request 2 bytes of data
    Wire.requestFrom(0x40, 2);
    // Read 2 bytes of data; humidity MSB, humidity LSB
    if(Wire.available() == 2)
    {
    data[0] = Wire.read();
    data[1] = Wire.read();
    }
    // Convert the data
    float humidity  = ((data[0] * 256.0) + data[1]);
    humidity = ((125 * humidity) / 65536.0) - 6;
```

Using the `Serial.print()` we can format the serial code to be easily extractable in python by separating each data point with commas. This will allow us to use the .split function in python (more on this later).

```
    Serial.print(heading);
    Serial.print(',');
    Serial.print(pitch);
    Serial.print(',');
    Serial.print(roll);
    Serial.print(',');
    Serial.print(ctemp);
    Serial.print(',');
    Serial.println(humidity);

    delay(1);
```

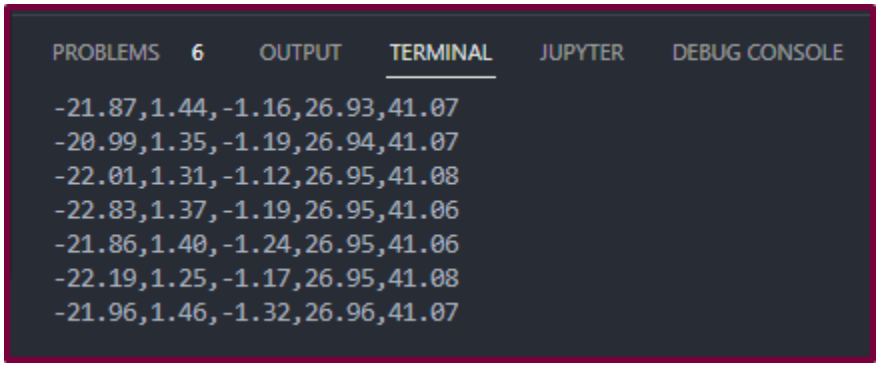
## loop()

Heading over to the `void loop()` part of the code this part is simple. All that is done is **defining float variables** to store the **accelerometer data and the magnetometer data**. Calling the IMU to read those and store them in said variable. And finally passing the values to the `printData()` function.

```
    float ax, ay, az, mx, my, mz;

    IMU.readAcceleration(ax, ay, az);
    IMU.readMagneticField(mx, my, mz);

    printData(ax, ay, az, mx, my, mz);
```

## Running the Code

Running the code and by opening the **serial monitor** you should see something similar to below showing the heading, pitch, and roll. Followed by the temperature and humidity:

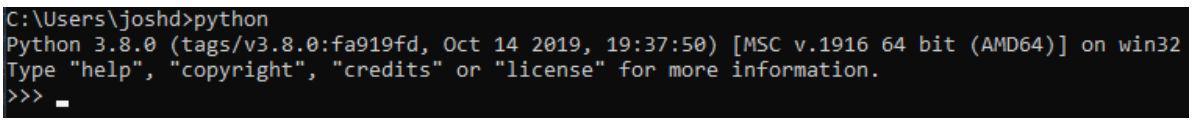



# Temperature and Humidity Visualization

This section will go through the python code to graph live the temperature and humidity data.

## Do you have Python?

To check if you have python open the command prompt/command terminal by using your operating systems search function. Type **python** and hit **enter**. You should see your python version. . . close the command window (example below).



If you don't have python installed head to this python website and download the latest version for your operating system. Follow the installation instructions and run the **.exe** file to finish the installation process.

### Python Libraries

To successfully run this application, there are a couple libraries that you need. The first is matplotlib which is a visualization library (like MATLAB). You will also need the NumPy library which adds support for large multidimensional arrays. You will also need the pySerial library which allows for serial communication (our main form from the .cpp program). Finally, the time library is also required (but this is already installed by default).

To install matplotlib, NumPy, and pySerial head once again to your command prompt and type (one at a time):

- pip install matplotlib
- pip install numpy
- pip install pyserial

## TEMP_HUMIDITY.py

Create a python file called `TEMP_HUMIDITY.py` and start by importing the above libraries.

```
1    import matplotlib.pyplot as plt
2    import numpy as np
3    import time
4    import serial
```

The import will import the library and using the as keyword we can call the library functions by a shortform term and creates an alias.

We will start by defining the class. A class is a way to initialize multiple objects with similar properties and helps with code reusability.

```
1    class Visualizer:
```

We are now going to define the `__init__()` (like the constructor in C) function which always executes on object initialization. We are going to call `serial.Serial()` to set the COM port, baud rate, and timeout. Using the **self** parameter will allow us to reference the current instance of the class.

Other than this we have almost completed the `__init__()` function and will just define empty variables that we will use later.

```
    def __init__(self):
            #Change port and baudrate
            self.esp32 = serial.Serial(port = "COM4", baudrate = 9600, timeout=.1)
            self.temp_array = [] #empty array for temperature data
            self.humid_array = [] #empty array for humidity data
            self.time = [] #empty array for time data
            self.start_time = 0 #Variable to set the start_time
            self.timeout = 0 #Timeout used to determine how long to take data points
            self.counter = 0 #Used for the serial flush
```

## Humidity Graph

Now we are going to setup a function within our class. Start by defining the class called `graph_humidity()` , with parameters **self**, **time_amount**, and **keep_alive = false**. The time_amount will set our how long to graph for and keep_alive will make sure the graph doesn't close after completion.

This function is simple. The `for i in range(10):` will flush the serial port making sure that our Serial data is setup correctly and will clear the buffers. After this we will fill the variable **start_time** using `time.time()` which will get the current time. `self.timeout` we will set that to the **start_time + time_amount** to find at what time the program should stop.

The **while** loop will act as follows: While the current time is less than the `self.timeout` we will set the title of the plot using `plt.title()` and start reading the lines of the serial port and decoding. Using the python `.split(',')` and knowing that the **.split()** creates a list of values we can access the humidity by accessing the index of the humidity values (which will be index 4).

Since the while loop is faster than the serial transfer there is an **if statement** that if there is no data ignore that data, if there is data append the time array, and humidity array, as well as plot both using `plt.plot()` .

Another one more if statement to determine whether to keep the plot alive after the timeout is finished.

```
    #Graph the humidity
    def graph_humid(self, time_amount, keep_alive = False):
        #Flushes the serial
        for i in range(10):
            esp32_bytes = self.esp32.readline()
            self.counter = self.counter + 1
            print(self.counter)
        #Set start time to the current time
        self.start_time = time.time()
```

```
        #Set timeout to the amount of time to gather data
        self.timeout = time.time() + time_amount

        while(time.time() < self.timeout):
            plt.title("Humidity vs Time")

            esp32_bytes = self.esp32.readline()
            esp32_decoded = esp32_bytes.decode()

            data = esp32_decoded.split(',')

            humidity = data[4]

            #Avoid no valued lines (while loop runs faster than the serial data)
            if esp32_decoded != '':
                self.time.append(time.time() - self.start_time)
                self.humid_array.append(float(humidity))
                plt.plot(self.time,self.humid_array,'o-',color = 'b')
                plt.pause(0.25)

        if keep_alive:
            plt.show()

        self.esp32.close()
```

## Temperature Graph

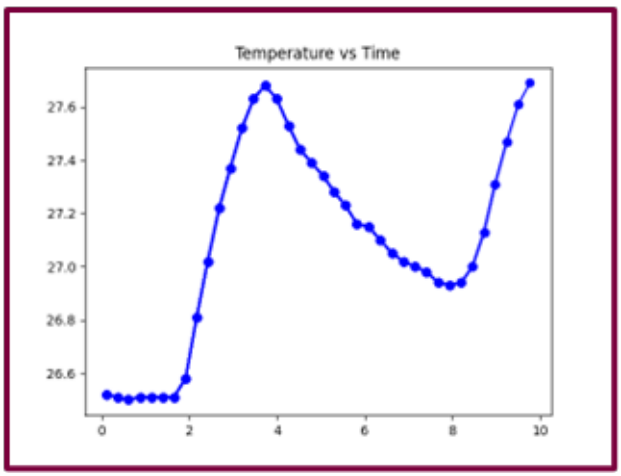
The **temperature graph** is the same as the humidity except for the graph title and `self.humid\_array` → `self.temp\_array` . As well as the index (which will be 3) in order to grab the temperature.

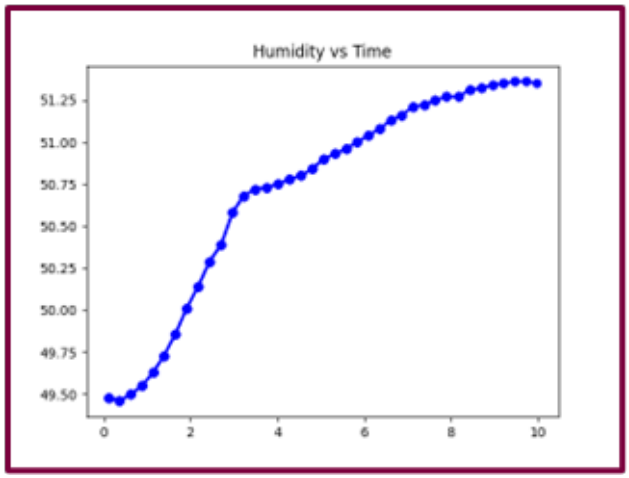
```
    #Graph the temperature
    def graph_temp(self, time_amount, keep_alive = False):
        #Flushes the serial
        for i in range(10):
            esp32_bytes = self.esp32.readline()
            self.counter = self.counter + 1
            print(self.counter)

        self.start_time = time.time()
        self.timeout = time.time() + time_amount

        while(time.time() < self.timeout):
```

```
            plt.title("Temperature vs Time")

            esp32_bytes = self.esp32.readline()
            esp32_decoded = esp32_bytes.decode()

            data = esp32_decoded.split(',')

            temperature = data[3]

            if esp32_decoded != '':
                self.time.append(time.time() - self.start_time)
                self.temp_array.append(float(temperature))
                plt.plot(self.time,self.temp_array,'o-',color = 'b')
                plt.pause(0.25)
        if keep_alive:
            plt.show()

        self.esp32.close()
```

We will now create a view object and call `view.graph_humid()` or `view.graph_temp()` in order to show the graphs.

And running the python script . . .




## IMU Visualization

This section will cover how to get a 3D view that follows the rotations of the MacIoT Board.

### IOT_LSM9DSL.py

Create a new file called **IOT_LSM9DSL.py**. This will house the class for the IMU as well as functions to `publishData()` and `start()` the serial communication. Import the following

libraries: serial, math , and time. We will then create a class called class IMUvisual: as well as creating the `__init__()` function with the parameter self. This init will only initialize the counter for the buffer and nothing else.

```
import serial
import math
import time

class IMUvisual:


    def __init__(self):
        #Fluch counter set
        self.counter = 0
```

Now define a function called **start** and initialize the esp32 to start serial communication. This will be the same code as above so give it a try.

```
#When called start serial communication with desired port and baudrate
def start(self, comName, baudSpeed):
    self.esp32 = serial.Serial(port=str(comName),baudrate=int(baudSpeed),timeout=.1)
```

You may have noticed above that I have input variables that will be passed into the function. This is so that you do not **hard code** the COM port and baud rate which will be handy very soon.

Now create a function called `publishData(self)` . This function will have the same serial flush as the **temperature and humidity visualization** except it will run 20 times to start at the same time as the OpenGL visualization. It will then perform the same split method to extract the data but this time taking values from index 0, 1, and 2. These are the heading, pitch, and rolll. It will then take those values and return them as floats.

```
#Publishes the data to be used in 3D visualization
def publishData(self):
    while True:
        #Flushes the serial
        if self.counter < 20:
            for i in range(20):
                esp32_bytes = self.esp32.readline()
                self.counter = self.counter + 1
                print(self.counter)
        else:
            esp32_bytes = self.esp32.readline()
            esp32_decoded = esp32_bytes.decode()
```

```
                data = esp32_decoded.split(',')

                if esp32_decoded != '':

                    yaw = data[0];
                    pitch = data[1];
                    roll = data[2];

                    #yaw,pitch,roll
                    #print(yaw,pitch,roll)
                    return float(yaw),float(pitch),-float(roll)

    def cleanup(self):
        pass
```

## IMU_VISUAL.py

Create a new file called **IMU_VISUAL.py**.

Once again, there are going to be required libraries that will need to be installed. Using the same installation process these libraries are pygame and OpenGL. Like before, perform the following in the command prombt/terminal.

- pip install pygame
- pip install PyOpenGL PyOpenGL_accelerate
- pip install pyopengl

Perform the following imports for **pygame, OpenGL, figure, tkinter, time and threading**. As well as the previously created **IOT_LSM9DS1** import the IMUvisual class.

```
import pygame
from pygame.locals import *
from OpenGL.GL import *
from OpenGL.GLU import *
from figure import *
from tkinter import *
from tkinter import ttk
import time
import threading
from IOT_LSM9DS1 import IMUvisual
```

Note: `import *` means that the imported modules will not need to be prefixed with the modules name. More info found here.

Create two variables `ApplicationGL` and `imu_visual`. ApplicationGL will hold the state of the application (**false = not running, true = running**) and imu_visual will initialize the `IMUvisual()` class. Also create a class called IMU which will have variables **Roll, Pitch, and Yaw**. Initialize `myimu = IMU`.

```
ApplicationGL = False

imu_visual = IMUvisual()

class IMU:
    Roll = 0
    Pitch = 0
    Yaw = 0

myimu = IMU
```

Create a new function called `RunApplication()`. Inside this application we will call the global variable ApplicationGL, and create new variables `myportName` and `myportSpeed`. We will also call `imu_visual.start()` which will call the start function from before and set the COM port and baud rate. This will also set **ApplicationGL = True** and thus will continue operation.

```
def RunApplication():
    global ApplicationGL
    myportName = Port_entry.get()
    myportSpeed = Baud_entry.get()
    imu_visual.start(myportName, myportSpeed)
    ApplicationGL = True
    ConfWindw.destroy()
```

Note: `Port_entry.get()` and `Baud_entry.get()` methods will return the values that are entered by the user and `ConfWindw.destroy()` will remove the window.

Next, we will setup the serial port window where the user can define the COM port as well as the baudrate. This is done through **tkinter**. The following code is commented and very self explanatory. Extra documentation can be found here.

```
#Creates an instance of the class Tk()
ConfWindw = Tk()
#Sets the window title
ConfWindw.title("Configure Serial Port")
#Sets the background color
```

```
    ConfWindw.configure(bg = "#2E2D40")
    #Configures the windows size
    ConfWindw.geometry('300x150')
    #Can the window be resized by the user?
    ConfWindw.resizable(width=False, height=False)
    #Where the window is located relative to the right hand side
    positionRight = int(ConfWindw.winfo_screenwidth()/2 - 300/2)
    #Where the window is located relative to the bottom
    positionDown = int(ConfWindw.winfo_screenheight()/2 - 150/2)
    #Sets the geometry
    ConfWindw.geometry("+{}+{}".format(positionRight, positionDown))
```

This next part is all about setting up the buttons, labels, and entry fields in the serial port window. This part is also self explanatory and will be described by the comments. Both the port and baud fields are generally the same. The ok button simply runs the application when it is pressed.

```
    #Creates the label and sets the properties
    Port_label = Label(text = "Port:",font =("",12),justify= "right",bg = "#2E2D40",
    fg = "#FFFFFF")
    #Places the label in a specific location
    Port_label.place(x = 50, y =30,anchor = "center")
    #Creates the COM port entry field and stores the value in Port_entry
    Port_entry = Entry(width = 20,bg = "#37364D", fg = "#FFFFFF", justify = "center")
    #Places the labe in a specific location
    Port_entry.place(x = 180, y = 30,anchor = "center")

    Baud_label = Label(text = "Speed:",font =("",12),justify= "right",bg = "#2E2D40",
    fg = "#FFFFFF")
    Baud_label.place(x = 50, y =80,anchor = "center")
    Baud_entry = Entry(width = 20,bg = "#37364D", fg = "#FFFFFF", justify = "center")
    Baud_entry.place(x = 180, y = 80,anchor = "center")

    #When the ok_button is pressed it will call the RunApplication
    ok_button = Button(text = "Ok",width = 8,command = RunApplication,bg="#135EF2",
    fg ="#FFFFFF")
    ok_button.place(x = 150, y = 120,anchor="center")
```

Next we will the function to initialize pygame.

```
    def InitPygame():
        global display
        #Initializes pygame
        pygame.init()
        #Sets the window size
```

```
        display = (640,480)
        #Initializes a window or screen for display
        pygame.display.set_mode(display, DOUBLEBUF|OPENGL)
        #Sets the caption
        pygame.display.set_caption('IMU visualizer   (Press Esc to exit)')
```

Note: DOUBLEBUF|OPENGL → | is combining both the DOUBLEBUF and OPENGL. More info on what DOUBLEBUF and OPENGL are can be found here.

Next, we will be initializing openGL. This will be done by defining another function called `InitGL()`.

```
    def InitGL():
        #Specify clear values for the color buffers
        glClearColor((1.0/255*46),(1.0/255*45),(1.0/255*64),1)
        #Enable or disable server-side GL capabilities
        glEnable(GL_DEPTH_TEST)
        #Specify the value used for depth buffer comparisons
        glDepthFunc(GL_LEQUAL)
        #Specify implementation-specific hints
        glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST)
        #Set up a perspective projection matrix
        gluPerspective(100, (display[0]/display[1]), 0.1, 50.0)
        #Multiply the current matrix by a translation matrix
        glTranslatef(0.0,0.0, -5)
```

This section of the code will be setting up the text that will appear at the bottom of the screen and will be defined as `DrawText()`.

```
    def DrawText(textString):
        #Sets the font
        font = pygame.font.SysFont ("Courier New",25, True)
        #Draw text on a new Surface
        textSurface = font.render(textString, True, (255,255,0), (46,45,64,255))
        #Transfer image to string buffer
        textData = pygame.image.tostring(textSurface, "RGBA", True)
        #Write a block of pixels to the frame buffer
        glDrawPixels(textSurface.get_width(), textSurface.get_height(), GL_RGBA,
        GL_UNSIGNED_BYTE, textData)
```

Now we will create a function to draw the board. It will be called `DrawBoard()`.

```
    def DrawBoard():
        #Delimit the vertices of a primitive or a group of like primitives
        glBegin(GL_QUADS)
        x = 0
        for surface in surfaces:
            for vertex in surface:
                #Sets the current color from an already existing array of color values.
                glColor3fv((colors[x]))
                #A pointer to an array of three elements.
                #The elements are the x, y, and z coordinates of a vertex.
                glVertex3fv(vertices[vertex])
            x += 1
        glEnd()
```

We will now define a function called `DrawGL()` . which will perform rotations and the 3D object manipulation.

```
    def DrawGL():
        #Clear buffers to preset values
        glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)
        #Replace the current matrix with the identity matrix
        glLoadIdentity()
        #Set up a perspective projection matrix
        gluPerspective(90, (display[0]/display[1]), 0.1, 50.0)
        #Multiply the current matrix by a translation matrix
        glTranslatef(0.0,0.0, -5)

        glRotatef(round(myimu.Pitch,1), 0, 0, 1) #Rotation
        glRotatef(round(myimu.Roll,1), -1, 0, 0) #Rotation

        DrawText("Yaw: {}°  Pitch: {}°  Roll: {}°".format(
            round(myimu.Yaw,1), round(myimu.Pitch,1),round(myimu.Roll,1))
            ) #Edits the text
        DrawBoard()
        pygame.display.flip() #Update the full display Surface to the screen
```

The next function is `ReadData()` and is very simple. It stores the `publishData()` in three variables **Roll, Yaw, and Pitch**.

```
    def ReadData():
        while True:
            myimu.Yaw,myimu.Pitch,myimu.Roll = imu_visual.publishData()
```

```
            time.sleep(0.03)
```

To use the `DrawBoard()` function you may have noticed `colors[]` and `vertices[]` this is defined in a separate file called **figure.py**. Go ahead and create this file and add:

```
    ##Vertices 0-7  (x,y,z)
    vertices= (
        (3, -.2, -1),
        (3, .2, -1),
        (-3, .2, -1),
        (-3, -.2, -1),
        (3, -.2, 1),
        (3, .2, 1),
        (-3, .2, 1),
        (-3, -.2, 1)
        )

    edges = (
        (0,1),
        (0,3),
        (0,4),
        (2,1),
        (2,3),
        (2,6),
        (5,1),
        (5,4),
        (5,6),
        (7,3),
        (7,4),
        (7,6)
    )

    surfaces = (
        (0,1,2,3),
        (4,5,6,7),
        (1,5,4,0),
        (3,2,6,7),
        (1,2,6,5),
        (0,3,7,4)
    )

    colors = (
        ((1.0/255*41),(1.0/255*217),(1.0/255*152)),
        ((1.0/255*41),(1.0/255*217),(1.0/255*152)),
```

```
        ((1.0/255*242),(1.0/255*66),(1.0/255*128)),
        ((1.0/255*242),(1.0/255*66),(1.0/255*128)),
        ((1.0/255*19),(1.0/255*94),(1.0/255*242)),
        ((1.0/255*242),(1.0/255*66),(1.0/255*128)),
    )

```

Finally, the **main()** function. This function incorporates threading. This is basically a way to have multiple processes occurring at once and can help speed up the application. If you would like to learn more, you can find it here.

```
def main():
    ConfWindw.mainloop() #Puts everything on display
    if ApplicationGL == True: #If the ApplicationGL is true then start the rest
        InitPygame()
        InitGL()

        try:
            myThread1 = threading.Thread(target = ReadData) #Start the ReadData function
            myThread1.daemon = True #Start the myThread1 in the background
            myThread1.start()
            time.sleep(2)
            while True:
                event = pygame.event.poll() #Get a single event from the queue
                if event.type == QUIT or (event.type == KEYDOWN and event.key == K_ESCAPE):
                    pygame.quit()
                    break

                DrawGL()
                pygame.time.wait(10)

        except:
            glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)
            DrawText("Sorry, something is wrong :c")
            pygame.display.flip()
            time.sleep(5)

if __name__ == '__main__': main()
```

# APPENDIX A

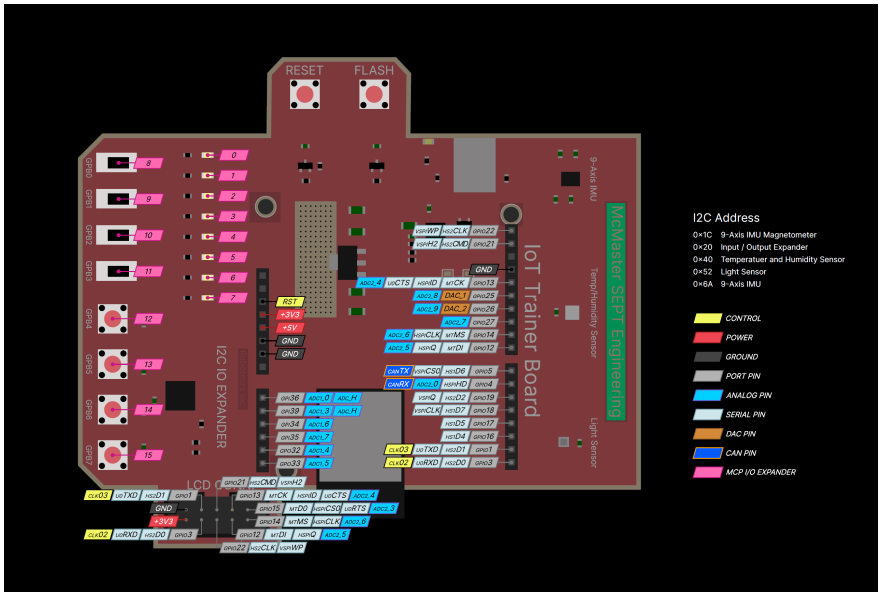



Figure 1: MacIoT Pinout

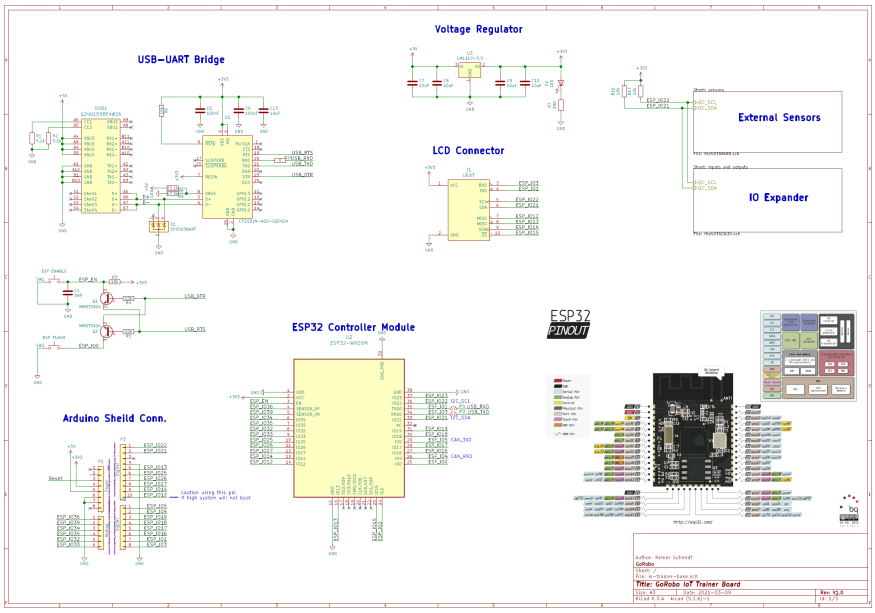



Figure 2: MacIoT Schematic Part 1

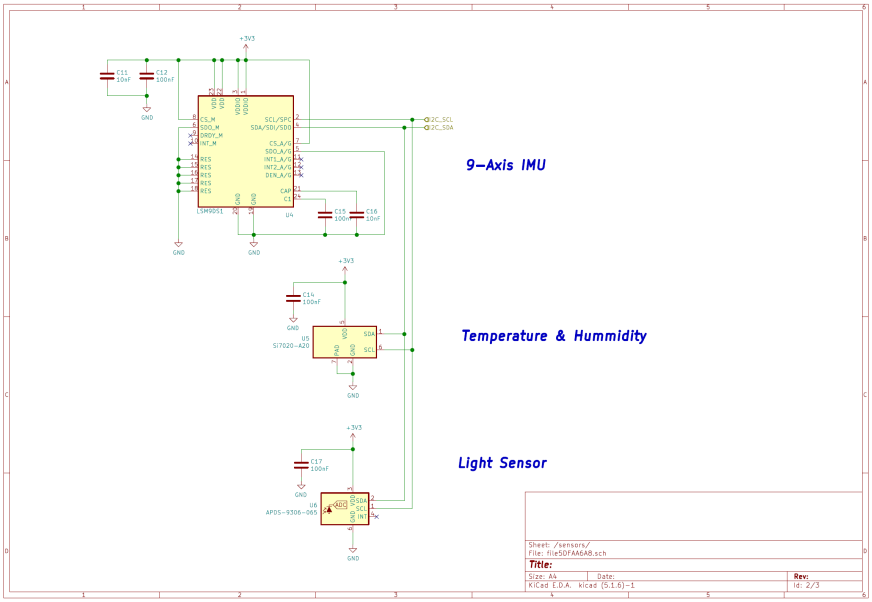



Figure 3: MacIoT Schematic Part 2

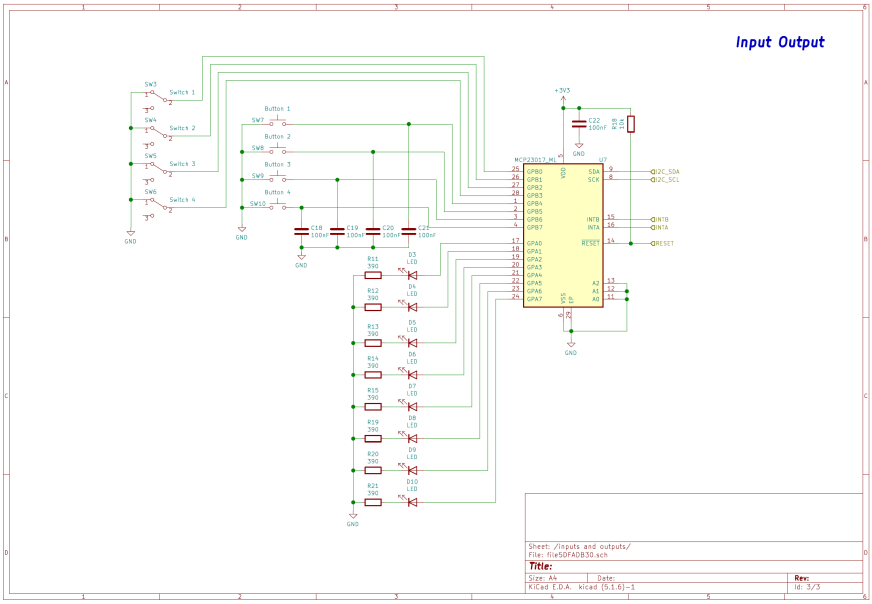



Figure 4: MacIoT Schematic Part 3

# REVISION LOG

The revision log will record the changes made to this document.

**2022-08-23**: MacIoT Manual issued with Modules 1 - 8 . . . . . Revision 1.0