

M5Stamp Pocket Guide

February 2023 Review Edition.

Written By
Adam Bryant

Copyright notice.

The Images and documentation found in this publication are a combination of the authors own work along with images and information provided by M5Stack with permission for use.

In order to reuse any image or part of this publication, please contact the copyright holders for permission before use.

If you find my guides helpful, you can find me on the M5Stack Community forums located @ <https://community.m5stack.com/>

On YouTube @ <https://www.youtube.com/channel/UCn5qzjLQdVz9K8SM0ojJAEA>

[hackster.io](https://www.hackster.io/AJB2K3) @ <https://www.hackster.io/AJB2K3>

And on Discord, Mastadon, Twitter and Facebook.

Thank you for reading.

Table Of Contents.

Copyright notice.	2
Chapter 1 Introduction to the M5Stamps.	1
M5Stamp Pico D4.	1
The M5Stamp C3 and M5Stamp C3U.	1
The M5Stamp S3	1
Chapter 2 M5Stamp Pinouts.	3
Power Pins.	6
UART Pins	6
ADC Pins	6
USB Pins	7
I2C Pins.	7
SPI Pins.	7
I2S Pins	8
LEDPWM/PWM/LEDC Pins	8
Capacitive Touch pins.	8
LCD/Camera Pads.	9
Addition functions.	10
Chapter 3 - Assembling and fitting connectors to the Stamps.	13
Chapter 4 - Programming the M5Stamps.	15
What is Micropython?	15
Introducing Thonny	16
Installing Micropython Firmware onto the M5Stamps.	18
Chapter 6 - Exploring the Micropython Modules	21
Chapter 7 - Connecting Things.	25
A simple Example.	25
Controlling the GPIO's in Micropython.	27
Machine/umachine Module	27
The Pin Class.	29
Example 1.0 - Turn on an LED with a button.	31

Example 1.1 - Blink an LED.	33
Time Class.	35
Experiment 1.2 - BiColour LED Blinker with 2 GPIO's.	37
Experiment 1.3 - LED Traffic Light	38
Neopixel Module and Class	41
Example 1.3 - RGB LED Strip	42
ADC Block Class	43
ADC Class	43
Example 1.4 - Analog Input	43
Pin as Interrupt Request (IRQ).	45
Example 1.5 - Interrupt Pin	45
Rotary Encoder	47
Bounce and Debounce	48
The PWM Class.	51
Example 1.6 - PWM Servo Control	52
Example 1.7 - PWM LED Control.	54
Example 1.8 - Play music with PWM.	55
One Wire Interface	58
Example 1.9 - DS18B20 One Wire Temperature.	58
Example 1.10 - L9110S H-Bridge Driver	60
Test 1.0 - Model Railway Semaphore Signal.	62
I2C Communication in Micropython.	63
Connecting I2C units to the M5Stamps.	63
I2C Class.	64
Communicating with the ENVIII Unit.	65
Converting the code into a Module/Library.	68
Giving up and using someone else's library.	69
I2C Output with OLED.	70
I2S Class.	71
SPI Interfacing	73
Example SPI 01 GC9A01 (Round Display)	73

Module.	77
WLAN Class.	78
M5Stamps and UIFlow2	79
Display images on the M5Dial/GC9A01 round Display	79
M5Stamp Accessories.	80
ESP32 Downloader	80
M5StampS3 Breakout Board	80
M5Stamp CatM Module	80
Stamp ISP	80
StampTimerPower	81
M5Stamp RS485	81
M5Stamp Extend I/O Module	81
I2C Communication part 2	82
M5Stack Devices fitted with the M5Stamp	84
Advanced Chapter 1 - Compiling fresh Micropython Firmware	85
Introduction	86
Installing CMake.	86
Installing ESP-IDF.	87
Copying Micropython and compiling into firmware.	92

Chapter 1

Introduction to the M5Stamps.

The M5Stamp is a family of Small Display-less controllers produced by M5Stack. As of writing, there are now four versions available to purchase and use in products:

- M5Stamp Pico D4,
- M5Stamp C3,
- M5Stamp C3U,
- M5Stamp S3.

M5Stamp Pico D4.

The M5Stamp Pico D4 is the smallest and lightest of the family consisting of only 12 GPIO's and a user programmable button and an RGB WS2812 (Neopixel compatible) LED. In order to save more PCB space it does not have an onboard USB to UART interface adapter requiring an external converter to program.

The M5Stamp uses two Xtensa® 32-bit LX6 microprocessor cores clocked 240MHz and has 4MB of flash memory.

The M5Stamp C3 and M5Stamp C3U.

The M5Stamp C3 and M5Stamp C3U are essentially the same device with the exception of the C3 uses a USB to UART adapter on the PCB where as the C3U uses the USB to UART (JTAG) programmer on the C3's chip.

The M5Stamp C3 and M5Stamp C3U are powered by a single core 32bit RISC-V processor clocked at 160 MHz and also have 4MB of flash memory.

The M5Stamp S3

The M5Stamp S3 is the most powerful model built around the Xtensa® 32-bit LX7 microprocessor cores clocked 240MHz but with 8MB of SPI flash memory. From what's available online it looks like the M5Stamp S3 also has its own internal USB to UART adapter on the chip.

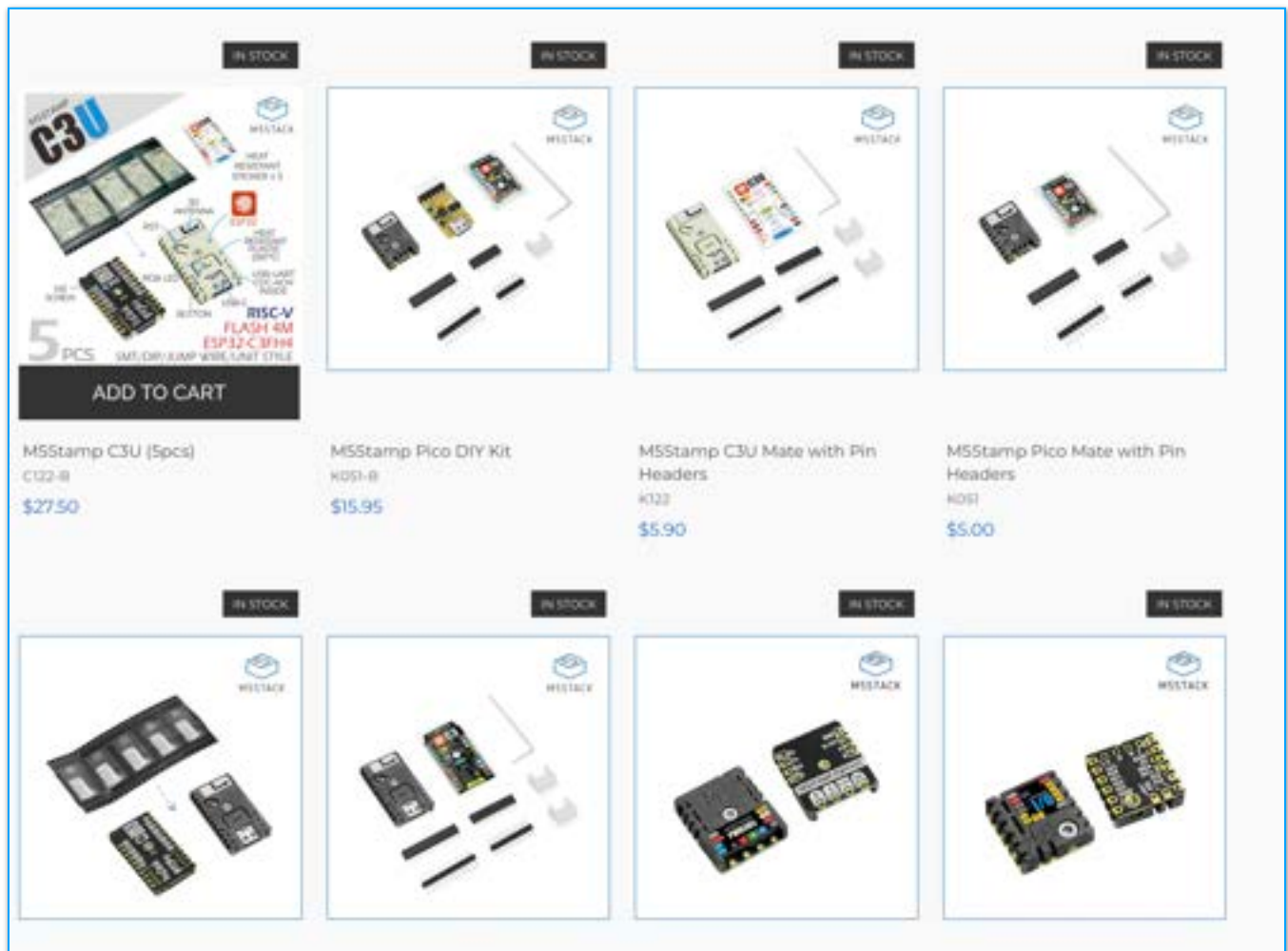
The M5Stamp S3 has 23 GPIO's and on the rear it has pads for direct connection to an LCD screen or a camera.

The next biggest difference between the M5Stamp S3 and its predecessors is that it has 15 ADC GPIO's of which 14 can be set as touch sensitive pins.

Another difference between the M5Stamp S3 and the previous M5Stamps is that not only can you install firmware using a bin package, you can now connect it as a USB drive and install firmware using the UF2 file format.

Because of the tiny 1.27mm pin spacing of the StampS3, there are two versions available. The Gray covered one come with standard size pin headers whereas the orange version come with 1.27mm header pins pre fitted.

When purchasing M5Stamps controllers there are three options available. You can buy the controllers as bare individuals, packs of five bare controllers or in a “Mate” pack which consists of the controller, headers, grove connectors and the small Allen (hex) ranch for removing the cover retaining screw.

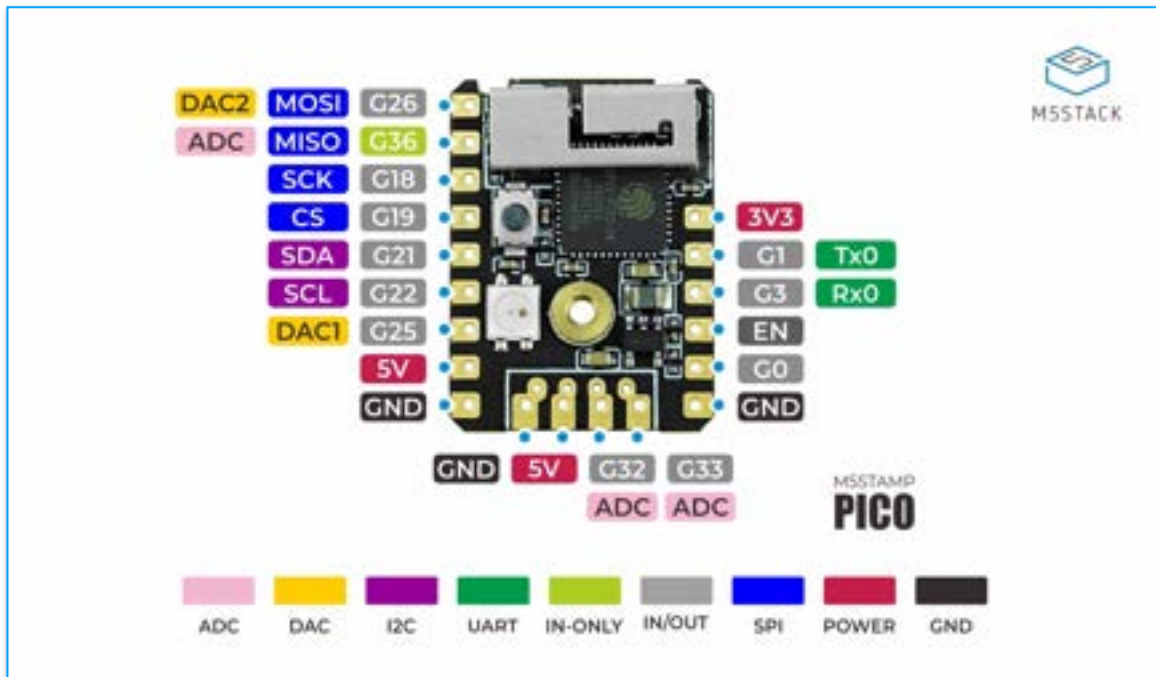


Screen shot of <https://shop.m5stack.com/pages/search-results-page?q=m5stamp> showing some of the products and purchase options.

Chapter 2

M5Stamp Pinouts.

While there are some pins on the stamps that are the same in terms of function, The M5Stamp C3 and C3U have more I/O (input and Output) pins that the M5Stamp Pico D4 but not as many as the M5Stamp S3. The following diagrams found on the M5Stack web site documents show what they are.

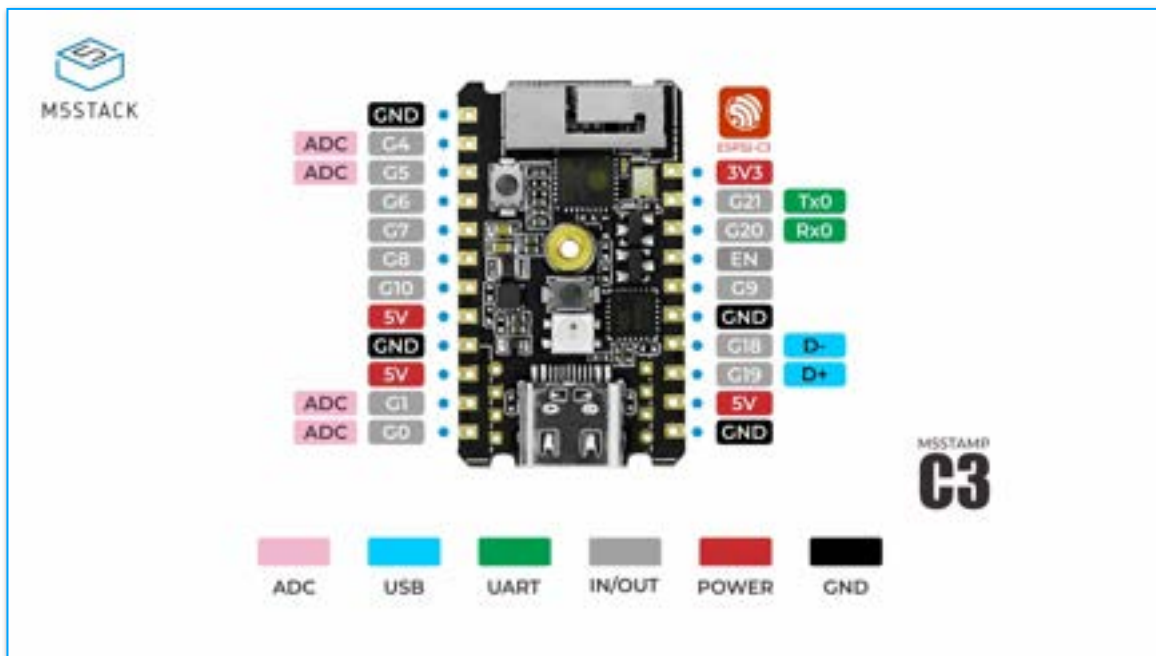


M5Stamp Pico D4 Pinout from https://docs.m5stack.com/en/core/stamp_pico?id=description

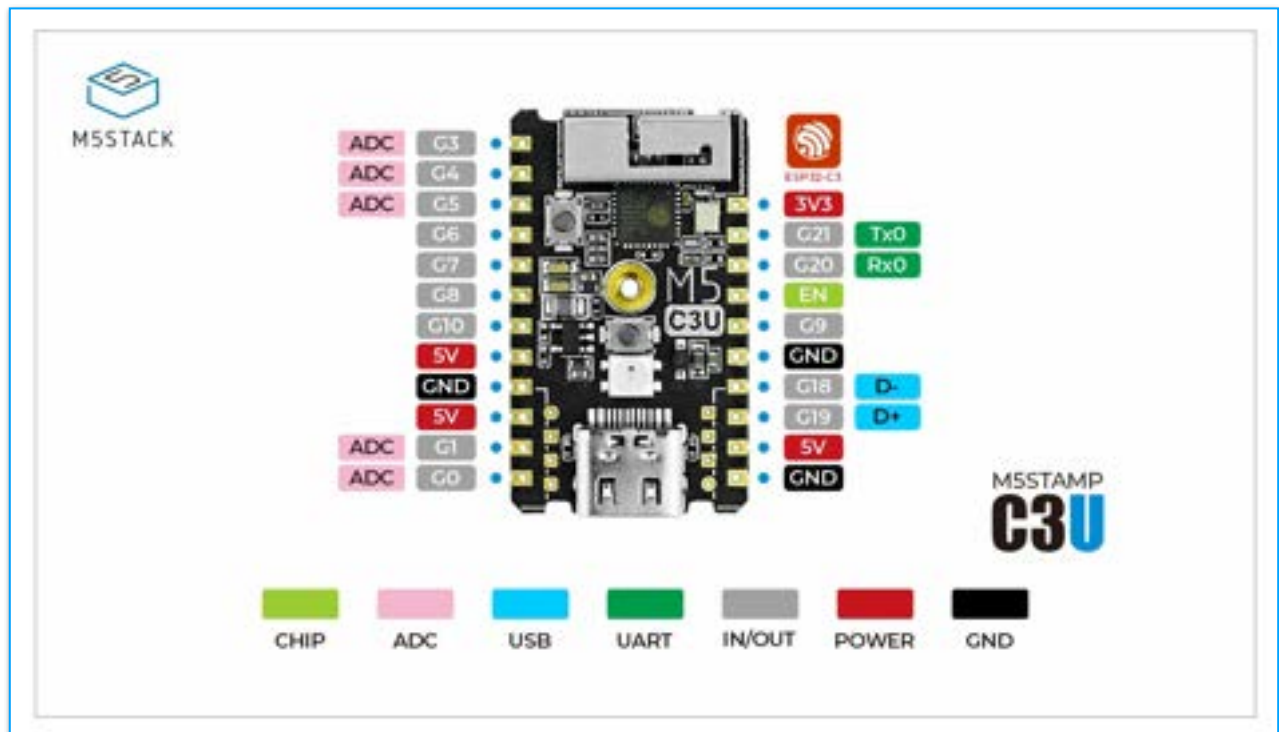
As you can see in the above pinout diagram of the M5Stamp Pico D4, several of the pins serve multiple functions as well as being GPIO (General Purpose Input / Output) pins.

On the right side of the M5Stamp Pico D4 is the Programming header which is where the programming adapter will connect. While these pins can be used for projects it is worth noting that it may cause problems when you come update or reprogram the M5Stamp Pico D4.

The Following image shows the pinout for the M5Stamp C3 and C3U. While most of the pins are the same, it is worth pointing out that the top left pin on the M5Stamp C3 is a ground pin where as on the M5Stamp C3U it is a GPIO pin.



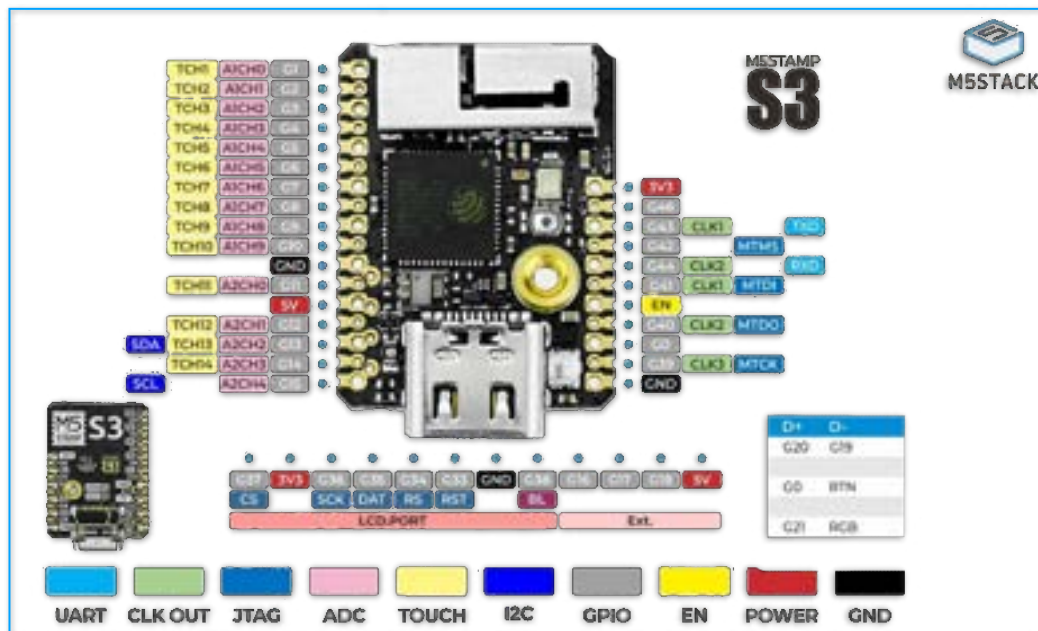
M5Stamp C3 Pinout from https://docs.m5stack.com/en/core/stamp_c3



M5Stamp C3U Pinout from https://docs.m5stack.com/en/core/stamp_c3u

While the M5Stamp C3 and C3U have their own USB adapters for programming, you will notice that they also have the Programming header broken out on the righthand side just like the M5Stamp Pico D4.

It is also worth noting that on the M5Stamp Pico C3 and C3U there is also the Pins D+ and D- broken out so that if the M5Stamp C3/C3U is used in a project where the USB connector is not accessible, an external USB connector can be added to the host circuit board and be connected via an internal connector or direct soldered connection.



M5Stamp S3 Pinout from <https://docs.m5stack.com/en/core/StampS3?ref=pfpqkvphmgr>

The M5Stamp S3 is the fourth member of the M5Stamp controller family and has the greatest amount of GPIO than the previous versions but also has pads on the rear for connection to an LCD display.

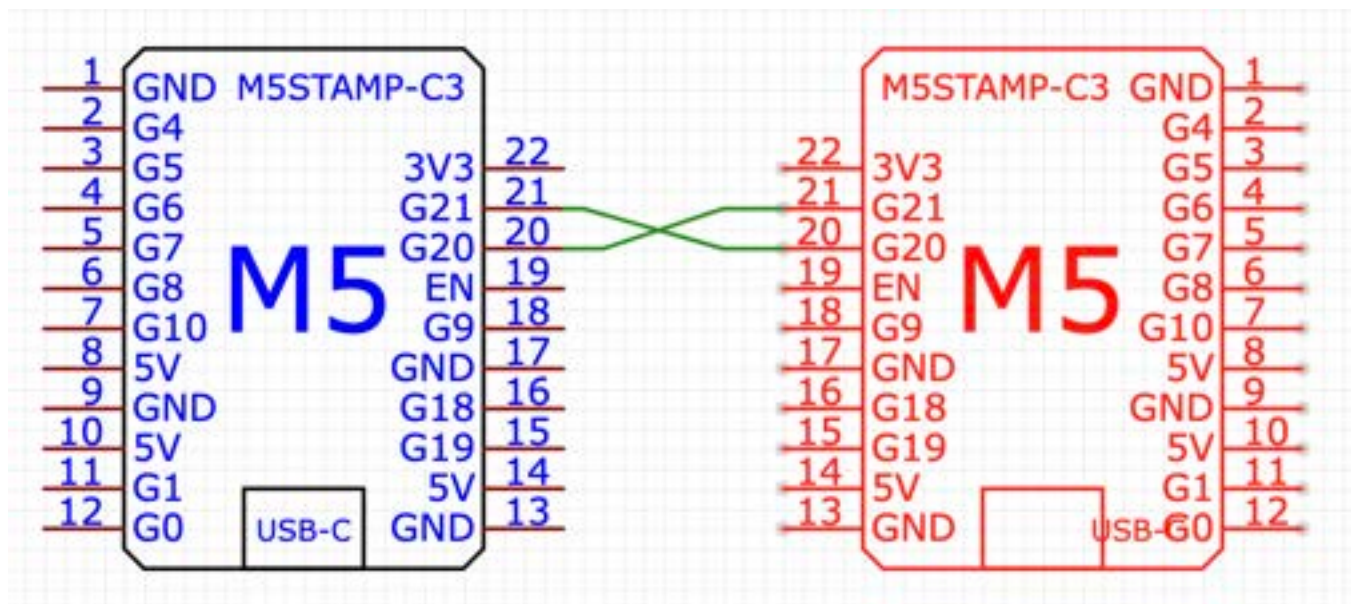
Power Pins.

The M5Stamps have both 5V and 3V3 power pins available for external connection. The M5stamps are 3.3 volt devices but have onboard 5 volt converters. Please make sure not to Put 5 volts on the 3V3 or GPIO pins or it could result in serious damage to the M5Stamps.

According to M5Stacks web site data, the M5Stamps need a power supply of 5V @ 500mA.

UART Pins

UART stands for universal asynchronous receiver-transmitter and is a universal two wire communication method used for direct communication between two devices. When connecting devices over UART, the TX pin of one device must be connected to the RX pin of the other device.



Example UART connection between two M5Stamp C3's.

ADC Pins

ADC stands for Analog to Digital Converter and is a set of pins connected to an internal ADC converter. The internal ADC converter consist of two 12-bit SAR registers giving up to 6 sampling channels split as follows.

For the M5Stamp C3 and C3U, ADC1 has five channels connected to GPIO pin 0 > 4 while ADC is connected to GPIO5.

For the M5Stamp Pico D4 GPIO's 0 > 5 are not broken out and so the GPIO's are mapped to GPIO 32 and 33.

The ADC's convert voltages from 0 > 3.3 volts into a value of 0 > 255.

USB Pins

As mentioned earlier, the M5Stamp C3 and C3U have an internal USB JTAG connection which is advisable on GPIO 18 and 19 (D+ and D-).

DAC Pins

Unlike the M5Stamp C3 and C3U, the M5Stamp Pico D4 has two DAC (Digital to Analog Converters) Channels which are tied to pins GPIO25 and 26.

The DAC channels convert digital values of 0 > 255 into an internal voltage of 0 to 1200mV.

I2C Pins.

I2C stands for Inter-Integrated Circuit and is a serial, synchronous, half-duplex communication protocol that allows co-existence of multiple masters and slaves on the same bus. The I2C bus consists of two lines: serial data line (SDA) and serial clock (SCL). Both lines require pull-up resistors.

With such advantages as simplicity and low manufacturing cost, I2C is mostly used for communication of low-speed peripheral devices over short distances (within one foot).

ESP32 has 2 I2C controller (also referred to as port), responsible for handling communications on the I2C bus. A single I2C controller can operate as master or slave.

On the M5Stamp Pico D4 the I2C pins are tied to GPIO's 21 and 22 however on the M5Stamp C3/C3U any pin can be used as I2C pins.

SPI Pins.

SPI stands for Serial Peripheral Interface (not spy) and just like the I2C pins, the SPI pins on the M5Stamp Pico D4 are dedicated to the pins in the diagram previously where as they can be assigned to any pins on the M5Stamp C3/C3U. There are two SPI channels available that allow transmissions up to 80Mhz.

I2S Pins

I2S stands for Inter IC Sound and is a serial, synchronous communication protocol that is usually used for transmitting audio data between two digital audio devices. ESP32 devices contain two I2S peripherals which can be configured to input and output sample data via the I2S driver.

I2S can work in either TDM Mode or PDM mode.

In TDM mode the I2S bus uses four wires to communicate with other I2S devices. The Pins in TDM mode are as follow:

- MCLK - Master Clock line,
- BCLK - Bit Clock Line,
- WS - Word (Slot) Select line,
- DIN/DOOUT - Serial Data Input/Output line.

In PDM mode, the I2S bus only requires two pins:

- CLK - Clock Line,
- DIN/DOOUT - Serial Data Input/Output line.

Each I2S controller has the following features that can be configured by the I2S driver:

- Operation as system master or slave
- Capable of acting as transmitter or receiver
- DMA controller that allows for streaming sample data without requiring the CPU to copy each data sample

Each controller can operate in simplex communication mode. Thus, the two controllers can be combined to establish full-duplex communication.

LEDPWM/PWM/LEDC Pins

The LEDPWM/PWM/LEDC pins are designed to be used to generate a PWM signal for controlling LEDs. While their primary function is to PWM control less, they can be used for other non LED PWM functions like low voltage/low current motors.

Capacitive Touch pins.

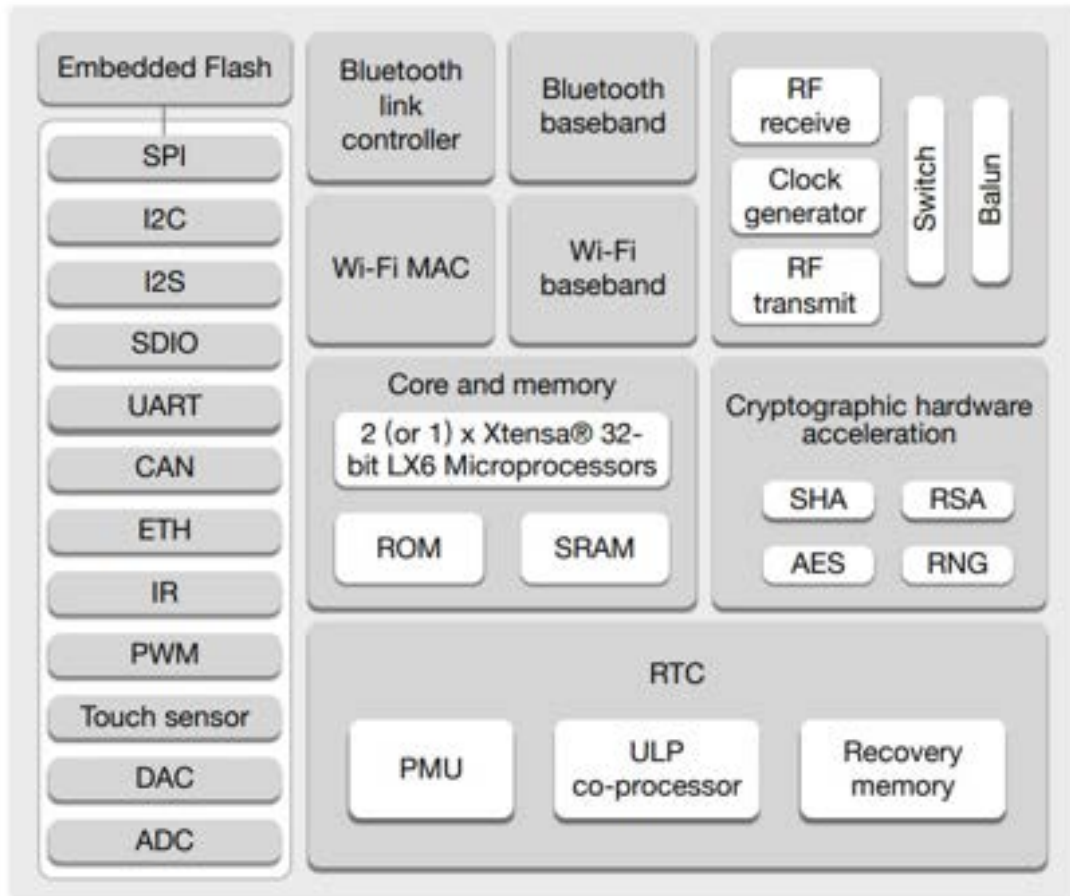
The M5Stamp S3 has a big difference over the C3 and Pico D4 in that most of its pins are ADC pins meaning that it now has 14 capacitive touch sensitive pins.

LCD/Camera Pads.

On the back of the M5Stamp S3 is a line of solder pads. These pads can be used to directly solder a camera or an TFT screen or for a connector that allows add and removal of cameras and displays without the need to solder and desolder the cables.

Addition functions.

As well as the GPIO pins, the M5Stamps have an array of additional function that are buried inside the little black chip. The functions can be found on the block diagrams shown below.

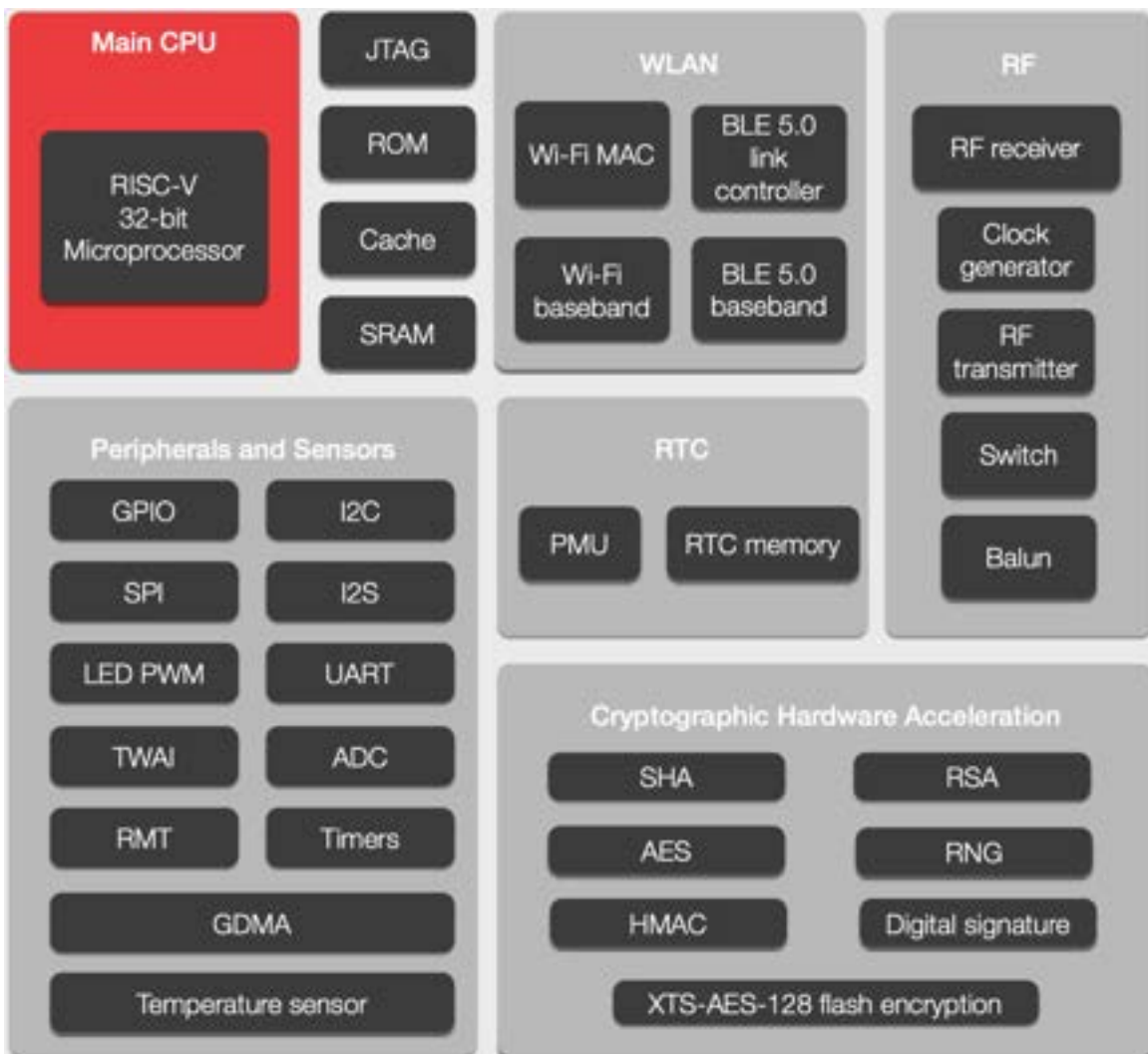


M5Stamp Pico D4 internal Block Diagram.

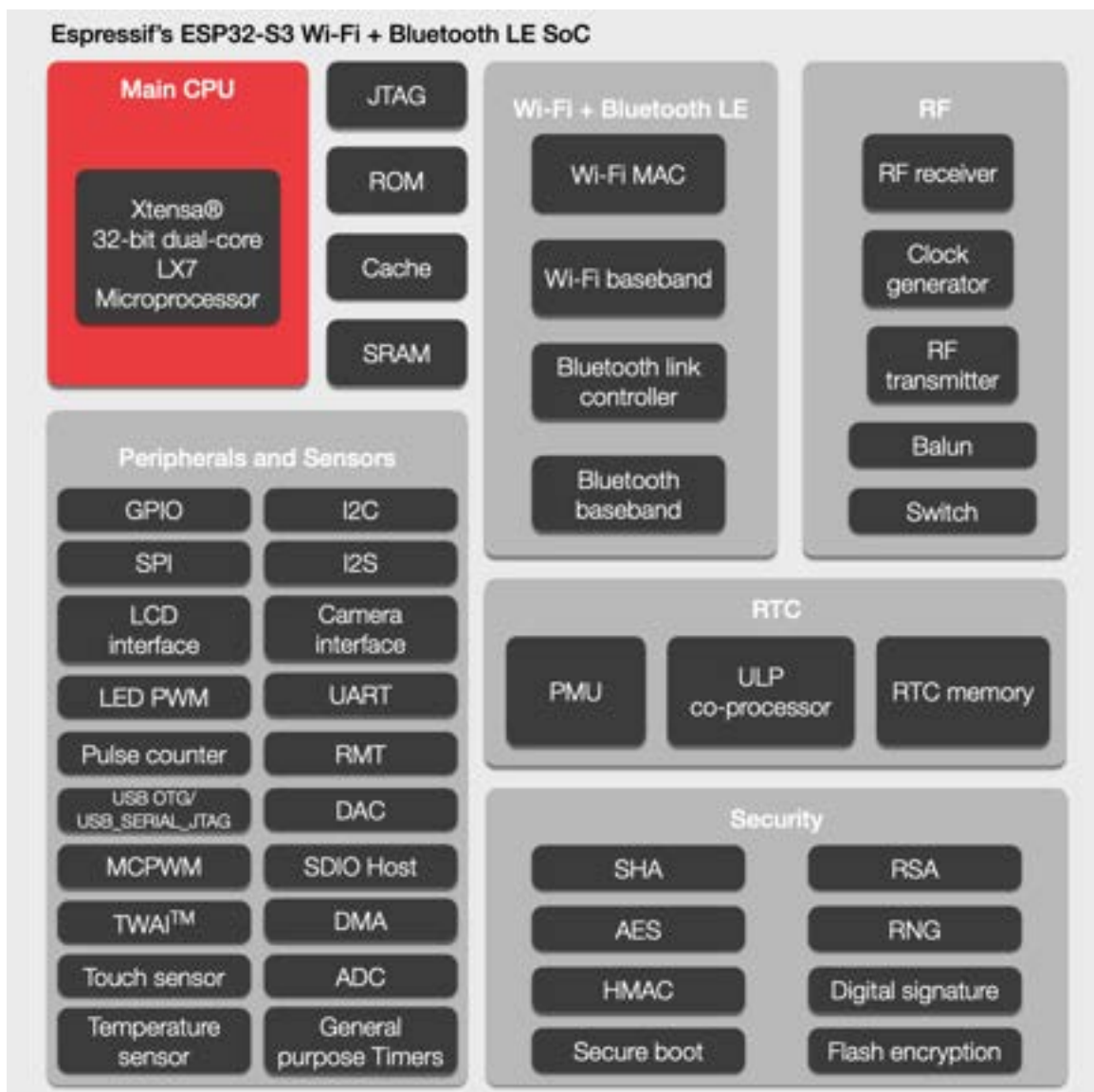
On the left hand side of the diagram you can see the I/O matrix. Unlike other microcontrollers, the M5Stamps do not really have dedicated function assigned to pins. The I/O matrix allows users and programmers to simply assign functions from the I/O matrix to specific pins for a projects needs.

While functions can be assigned to pins, there are some pins where this does work very well and it is advisable to use the pins assigned in the pinout diagrams shown on previous pages.

The following is the block diagram of the M5Stamp C3/C3U



M5Stamp C3/C3U Block Diagram.

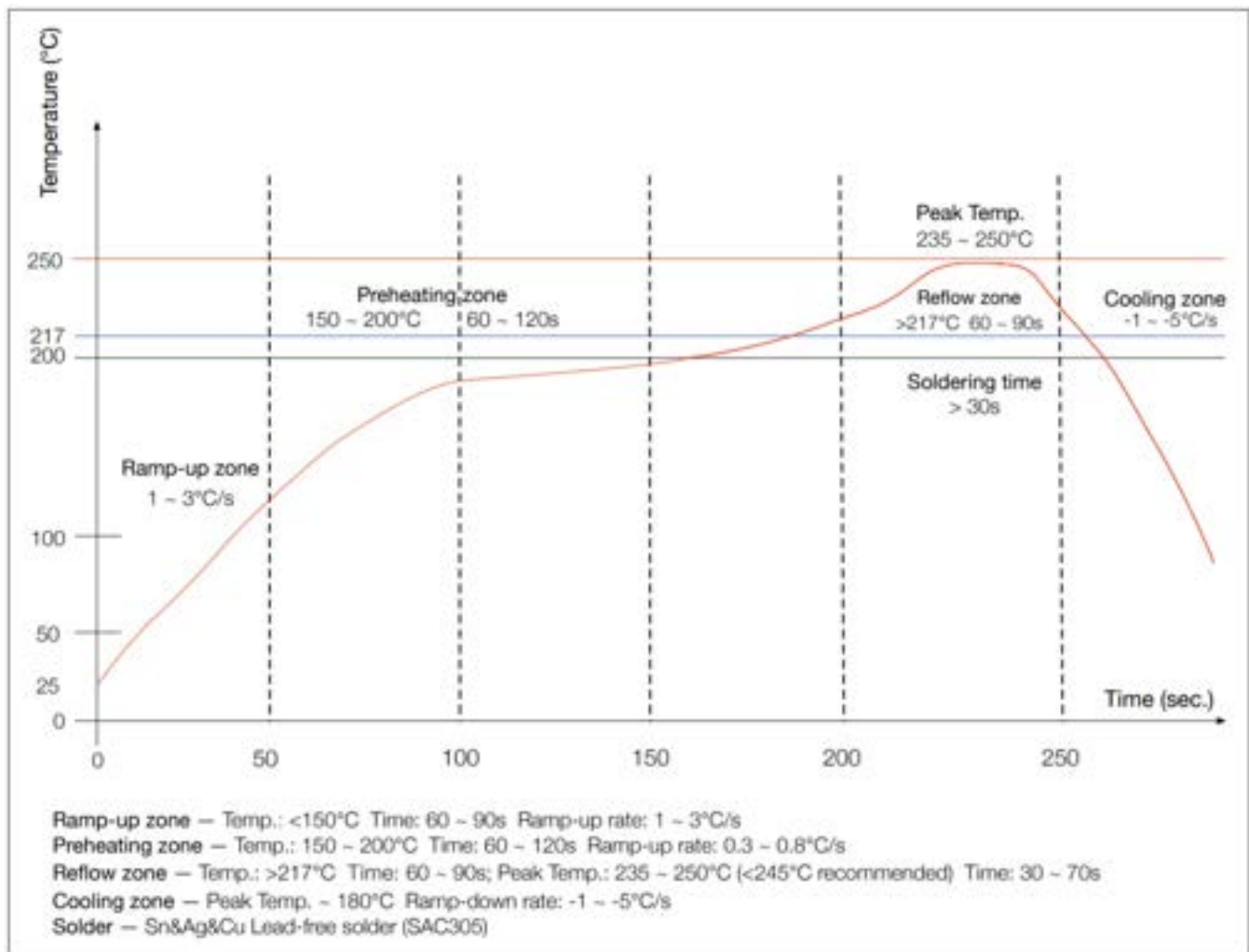


M5Stamp S3 Block Diagram.

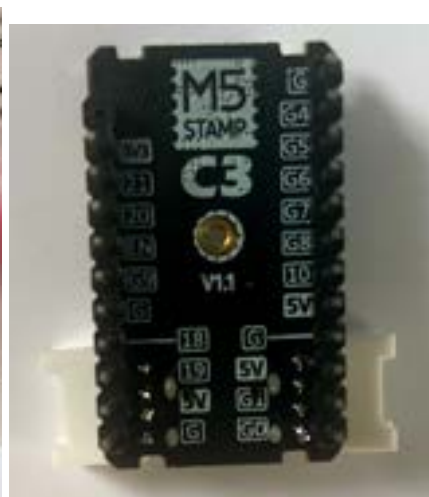
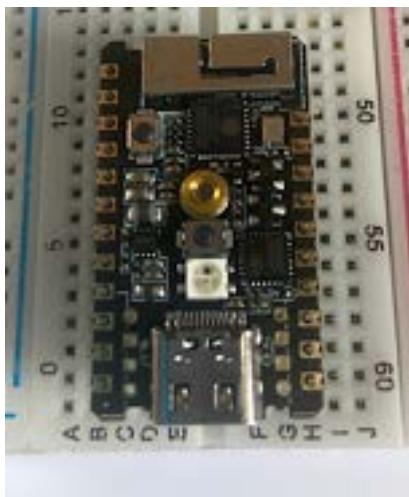
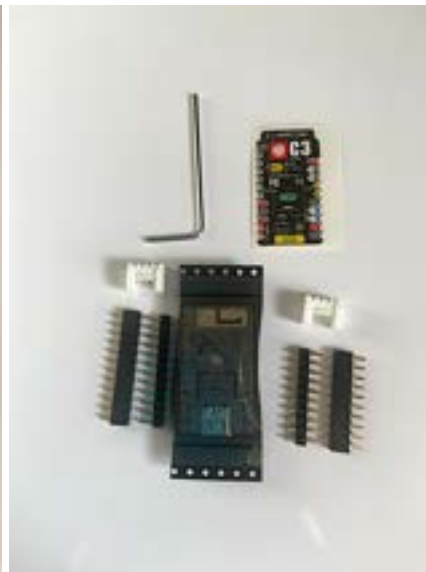
Chapter 3 - Assembling and fitting connectors to the Stamps.

In this chapter I will show you how to assemble and solder the headers and connectors to the various M5Stamps. The choice of what headers/ connectors you use will depend on your requirement but if you want to use headers and “Grove” connectors on the same M5Stamp, the headers must be soldered first or the contacts will be covered by the “Grove” connectors.

No matter which M5Stamp you use you first start by removing the cover screw that holds the plastic cover on. If you are surface mounting the M5Stamps then there is no need to remove the cover as it has been designed to survive soldering as long as the recommended reflow profile show below is used.



Reflow Profile for soldering with the plastic covers fitted to the M5Stamps.



The Above Images show the process of adding headers to the M5Stamp C3.

Image 1 shows how The M5Stamps C3 Mate arrives. Image 2 show the contents of the M5Stamps C3 Mate. Image 3 Shows the cover removed for soldering the underside headers. In image 4 I'm using a solder less breadboard to help aline the headers before soldering. Image 5 shows the Grove connectors soldered in place. Image 6 Shows the cover replaced after being trimmed to remove the brake of sections covering the Grove port.

Chapter 4 - Programming the M5Stamps.

As of writing, the current environments for programming M5Stamps consist of Arduino, ESP-IDF and Micropython/UIFlow.

As M5Stack products use the Micropython/UIFlow environment, this handbook will be based around programming with the Micropython language and by extension UIFlow.

What is Micropython?

From the Micropython website located at <https://micropython.org/> the description of Micropython is as follows:

MicroPython is a lean and efficient implementation of the Python 3 programming language that includes a small subset of the Python standard module and is optimised to run on microcontrollers and in constrained environments.

MicroPython is packed full of advanced features such as an interactive prompt, arbitrary precision integers, closures, list comprehension, generators, exception handling and more. Yet it is compact enough to fit and run within just 256k of code space and 16k of RAM.

MicroPython aims to be as compatible with normal Python as possible to allow you to transfer code with ease from the desktop to a microcontroller or embedded system.

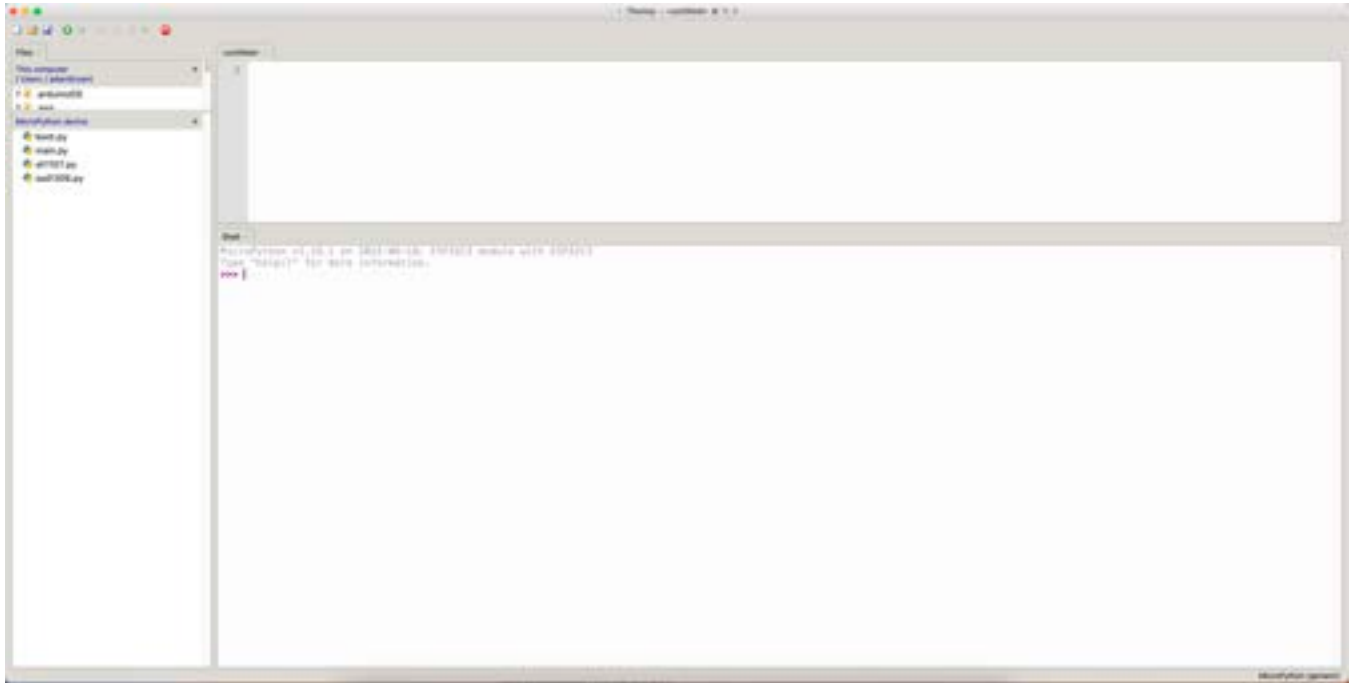
There are many version of Micropython available depending on which hardware platform it needs to be used on and which version a product is built around. During the writing of this handbook I used Micropython version 1.19.1 on the M5Stamp C3 and C3U from the mainstream branch and UIFlow which is built on the 1.12 version customised by M5Stack.

One thing that makes Micropython easier to understand is that when using the live internal REPL command, you can actually view the functions available inside the installed Micropython module.

Introducing Thonny

The way I look for these functions is to use a program called Thonny which can be downloaded from: <https://thonny.org/>

Once installed along with the device drivers, you just need to plug the M5StampC3/C3U into the USB port (or via the adapter for the M5Stamp Pico D4), open up Thonny:

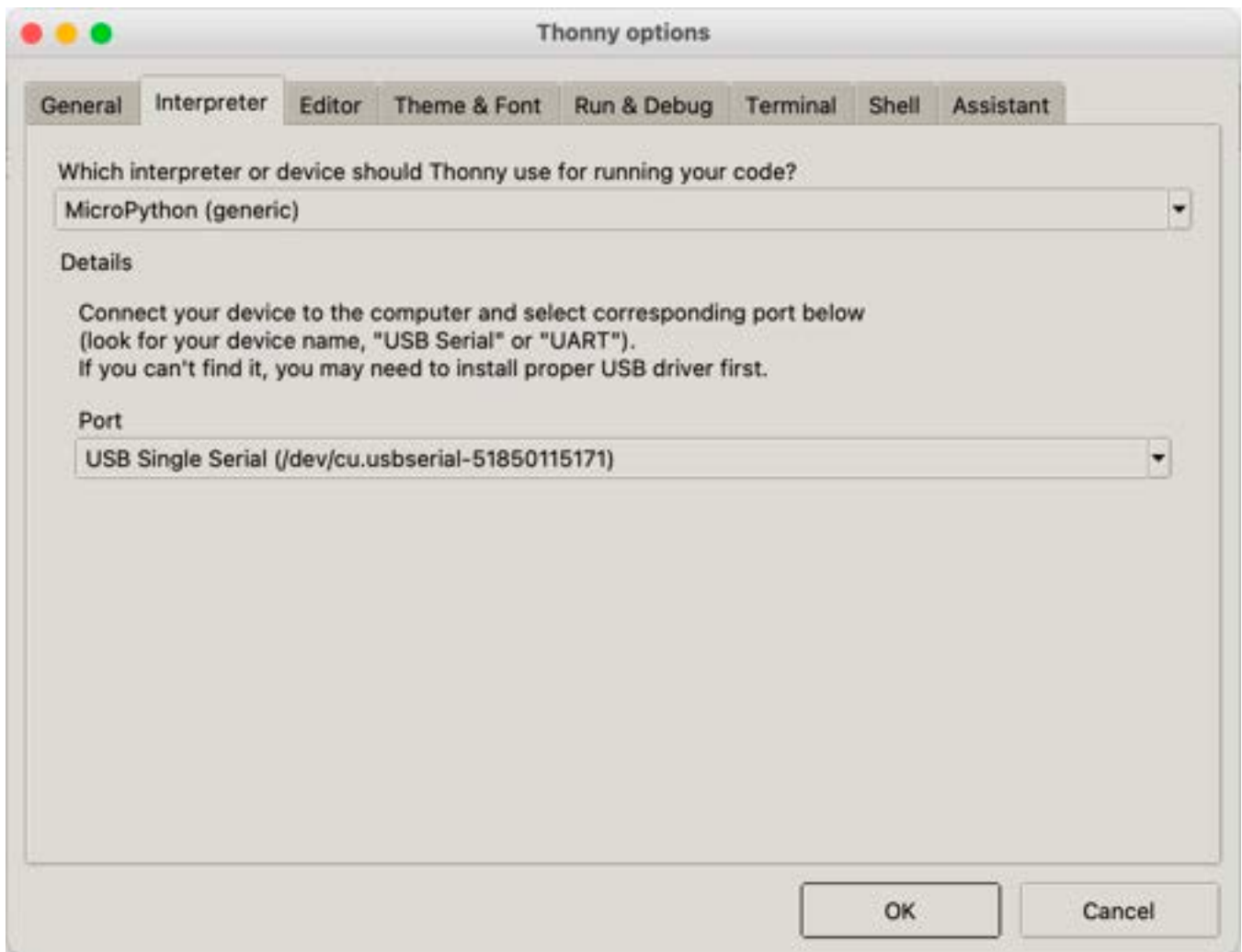


Thonny open and connected to an M5Stack C3.

And you should have a screen showing the above.

If you don't see the MicroPython message in the bottom box then the M5Stamp is not detected and will need MicroPython to be installed or, if you don't see the Shell window at the bottom, you need to click on the View menu and click on "Shell" to select it.

If the MicroPython firmware has been installed but not connecting, click on Thonny's "Run" menu then click on "Select Interpreter" to open the interpreter menu:



Thonny's Interpreter setup window.

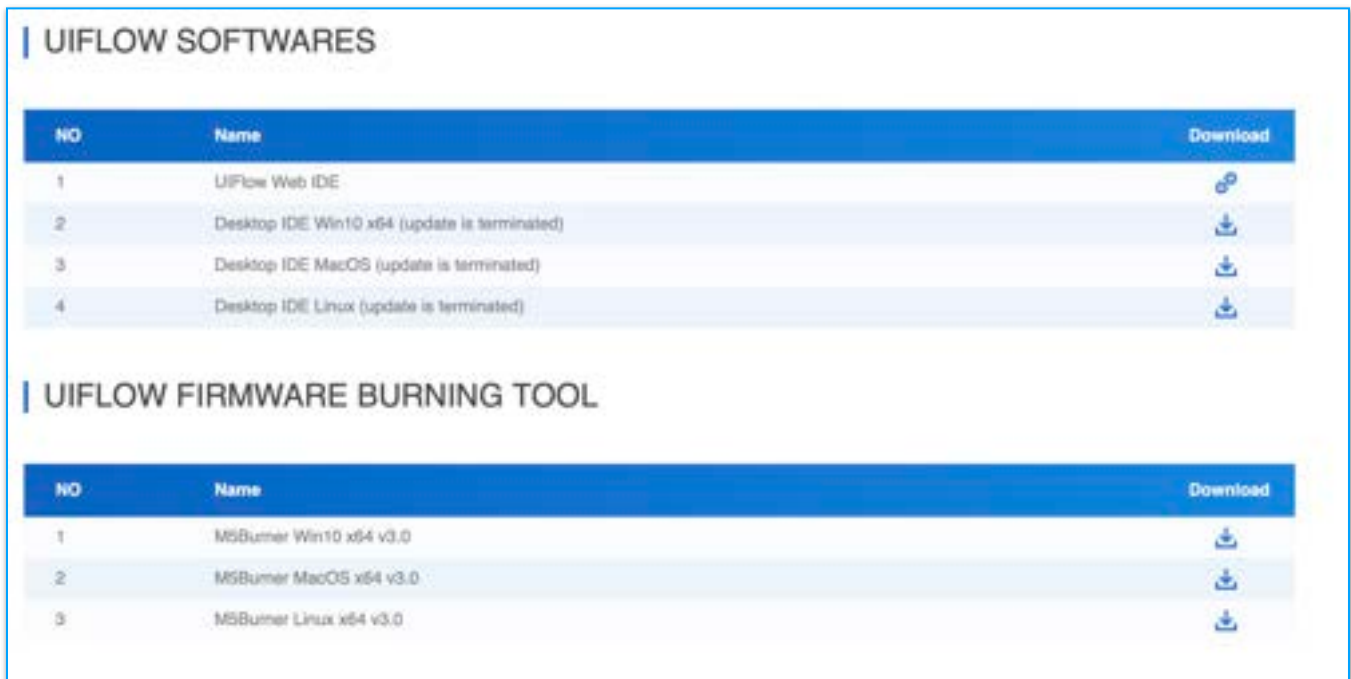
In the above screen shot you can see that I have installed a generic version of Micropython for the ESP32-C3 and the port that it has identified itself as to the computer.

If the port item is blank, click on the arrow to the far right of the box to open the drop down menu and select a port. This is the Linux/OSX version of Thonny and the Windows version will show a different port naming structure.





Once these setting have been configured to connect to the plugged in M5Stamp, press 'OK' to return to the main screen.




Installing Micropython Firmware onto the M5Stamps.

Normally M5Stamp firmware is installed onto the M5Stamps using the M5Burner firmware tool found on the M5Stack Web site here: <https://docs.m5stack.com/en/download>



The image shows two sections of a web page. The first section, titled 'UIFLOW SOFTWARES', contains a table with four rows of software downloads. The second section, titled 'UIFLOW FIRMWARE BURNING TOOL', contains a table with three rows of firmware burning tools. Both tables have columns for 'NO', 'Name', and 'Download'.

NO	Name	Download
1	UIFlow Web IDE	
2	Desktop IDE Win10 x64 (update is terminated)	
3	Desktop IDE MacOS (update is terminated)	
4	Desktop IDE Linux (update is terminated)	

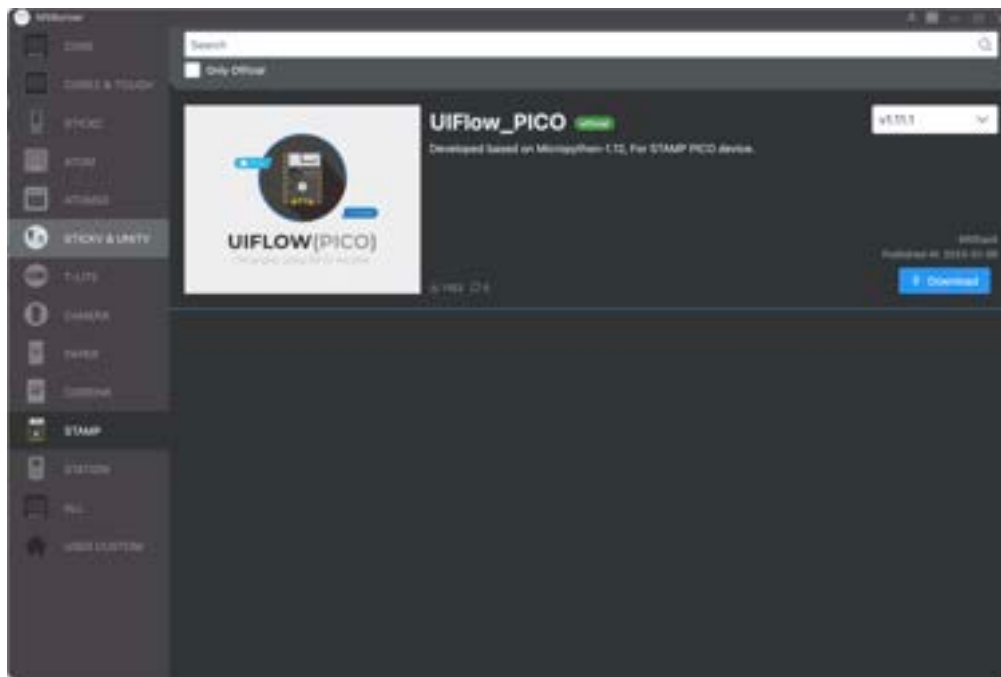
NO	Name	Download
1	M5Burner Win10 x64 v3.0	
2	M5Burner MacOS x64 v3.0	
3	M5Burner Linux x64 v3.0	

The development tools download page on the M5Stack web site.

You need to click on the M5Burner version for your OS to download and then save the program into the correct locations.

On OSX based computers, M5Burner must be placed in the “Applications folder”

Once M5Burner is installed you need to physically connect the controllers. If you are using the M5Stamp C3, C3U or S3 you just plug in a cable to the onboard USB port. If you are running the M5Stamp Pico D4 or have the M5Stamps installed in a way that prevents access to the USB port you can use the ESP downloader kit or include the Stamp ISP on the edge of a PCB as a programming adapter. and running you will see the following screen:



M5Burner firmware menu for the M5Stamp Pico.

Clicking the icons on the left allow you to select the available firmware for each M5Stack controller and install it to the controller. However, this only works for the M5Stamp Pico D4 and so for the M5Stamp C3 and M5Stamp C3U we have to install non M5Stack Micropython firmware in order to use the M5Stamps.

For this we need to resort to using command line tool in a terminal. The reason for using command line tools is due to the location of where the firmware is installed in the memory of the controller. Normally firmware is installed into the ESP32 at the address location of **0x1000** but if you try installing it to this location on the M5Stamp C3 and M5Stamp C3U which uses a RISC-V core, the M5Stamps will fail to boot. In order to install the firmware into the correct location, open a terminal (after connecting the M5Stamp C3 or C3U with a USB cable) and type the following:

```
esptool.py --chip esp32c3 --port /dev/ttyUSB0 erase_flash
```

To erase what ever is already in the M5Stamps memory and then:

```
esptool.py --chip esp32c3 --port /dev/ttyUSB0 --baud 460800 write_flash -z 0x0 esp32c3-20220117-v1.18.bin
```

To install the firmware to the correct address of 0x0.

Now that we have Micropython installed on the M5Stamps, we can now look at the installed modules and see how they work.

Chapter 6 - Exploring the Micropython Modules

Functions in Micropython are collected into classes which are then grouped into modules or libraries. In order to use functions you first have to import the module containing the class or directly import only the needed class from the container module.

Finding the internal modules requires issuing command into the **REPL (Read, Evaluate, Print and Loop)**. Looking at the bottom of Thonny at the moment the shell only has the following text:

```
MicroPython v1.19.1 on 2022-06-18; ESP32C3 module with ESP32C3
Type "help()" for more information.
>>>
```

After the 3 arrows (>>>) type in help() and hit return.

```
MicroPython v1.19.1 on 2022-06-18; ESP32C3 module with ESP32C3
Type "help()" for more information.
>>> help()
Welcome to MicroPython on the ESP32!
```

For generic online docs please visit <http://docs.micropython.org/>

For access to the hardware use the 'machine' module:

```
import machine
pin12 = machine.Pin(12, machine.Pin.OUT)
pin12.value(1)
pin13 = machine.Pin(13, machine.Pin.IN, machine.Pin.PULL_UP)
print(pin13.value())
i2c = machine.I2C(scl=machine.Pin(21), sda=machine.Pin(22))
i2c.scan()
i2c.writeto(addr, b'1234')
i2c.readfrom(addr, 4)
```

Basic WiFi configuration:

```
import network
sta_if = network.WLAN(network.STA_IF); sta_if.active(True)
sta_if.scan() # Scan for available access points
sta_if.connect("<AP_name>", "<password>") # Connect to an AP
sta_if.isconnected() # Check for successful connection
```

Control commands:

CTRL-A -- on a blank line, enter raw REPL mode
CTRL-B -- on a blank line, enter normal REPL mode
CTRL-C -- interrupt a running program
CTRL-D -- on a blank line, do a soft reset of the board
CTRL-E -- on a blank line, enter paste mode

For further help on a specific object, type help(obj)

For a list of available modules, type help('modules')

>>>

This give us a little bit of details to understand how the REPL/Shell works. The REPL/Shell is actually connected to the M5Stamp and directly running code on the physical M5Stamp.

In order to view the modules pre-compiled into the firmware type:

help('modules')

And hit return. Micropython is case sensitive so make sure the command is typed exactly as show.

>>> *help('modules')*

<i>_main_</i>	<i>framebuf</i>	<i>uasyncio/stream</i>	<i>uplatform</i>
<i>_boot</i>	<i>gc</i>	<i>binascii</i>	<i>urandom</i>
<i>_onewire</i>	<i>inisetup</i>	<i>ubluetooth</i>	<i>ure</i>
<i>_thread</i>	<i>math</i>	<i>ucollections</i>	<i>uselect</i>
<i>_uasyncio</i>	<i>micropython</i>	<i>ucryptolib</i>	<i>usocket</i>
<i>_webrepl</i>	<i>neopixel</i>	<i>uctypes</i>	<i>ussl</i>
<i>apa106</i>	<i>network</i>	<i>uerrno</i>	<i>ustruct</i>
<i>btree</i>	<i>ntptime</i>	<i>uhashlib</i>	<i>usys</i>
<i>builtins</i>	<i>onewire</i>	<i>uheapq</i>	<i>utime</i>
<i>cmath</i>	<i>uarray</i>	<i>uio</i>	<i>utimeq</i>
<i>dht</i>	<i>uasyncio/_init_</i>	<i>ujson</i>	<i>uwebsocket</i>
<i>ds18x20</i>	<i>uasyncio/core</i>	<i>umachine</i>	<i>uzlib</i>
<i>esp</i>	<i>uasyncio/event</i>	<i>uos</i>	<i>webrepl</i>
<i>esp32</i>	<i>uasyncio/funcs</i>	<i>upip</i>	<i>webrepl_setup</i>
<i>flashbdev</i>	<i>uasyncio/lock</i>	<i>upip_utarfile</i>	<i>websocket_helper</i>

Plus any modules on the filesystem

>>>

The list above shows us what modules are precompiled into the mainstream Micropython 1.19.1 firmware that I am using with the M5StampC3 and M5StampC3U. In order to view the functions in a module, you first need to import the module with:

>>> *import esp32*

And then use the `dir()` command to view the functions.

```
>>> dir(esp32)
['__class__', '__name__', 'HEAP_DATA', 'HEAP_EXEC', 'NVS', 'Partition', 'RMT',
'WAKEUP_ALL_LOW', 'WAKEUP_ANY_HIGH', 'gpio_deep_sleep_hold', 'idf_heap_info',
'wake_on_ext0', 'wake_on_ext1', 'wake_on_touch']
>>>
```

Most of that list is arguments which can be set in a function but more information on arguments will be covered later.

This is just a brief guide into exploring the various modules in the Micropython environment. In the next chapter I will show you a quick example program.

Chapter 7 - Connecting Things.

A simple Example.

In the previous pages I showed you how to connect an M5Stamp to the computer and access Micropython through Thonny. In this chapter I will show you how to light up the built in LED.

To light up the onboard RGB LED on the M5Stamp Pico D4 in UIFlow you use the following block code:



When you view the code in UIFlows Micropython view you will see the following.

```
from m5stack import *
from m5ui import *
from uiflow import *
import time

while True:
    rgb.setColorAll(0xff0000)
    wait(0.5)
    rgb.setColorAll(0x000000)
    wait(0.5)
    wait_ms(2)
```

However, on the M5Stamp C3 and M5Stamp C3U We don't have the m5stack, m5ui and uiflow modules. In Micropython, the code will almost be the same but, we need to use the neopixel and time module along with the machine module:

```
from machine import Pin
```

```
from neopixel import NeoPixel  
Import time
```

```
pin = Pin(2, Pin.OUT)  
np = NeoPixel(pin, 1)
```

```
while True:  
  np[0] = (255, 255, 255)  
  np.write()  
  wait(0.5)  
  np[0] = (0, 0 0)  
  np.write()  
  wait(0.5)  
  wait_ms(2)
```

And that's all it takes to blink the onboard RGB in a red colour.

You can view a Youtube short of the Blink program running on the M5Stamp C3 here:

Record and upload Youtube video!

Controlling the GPIO's in Micropython.

Machine/umachine Module

Now that the basic example is out of the way, it is time to start understanding how to work with the various classes and functions available. To start I will show you how to control the GPIO's in their basic mode first.

Under Micropython, the basic functions for accessing the GPIO's are found in Micropython's **Machine** module but, there is a problem. If you look at the list of modules on the previous pages you will see that there is no module named "machine", but there is a module named 'umachine' You can type machine or umachine and you will still get the same results due to the way the module has been created. If you import machine or umachine and use the dir() command you will see that both of the machine modules contain the same classes and functions.

```
>>> import machine
>>> dir(machine)
['_class__', '__name__', 'ADC', 'ADCBlock', 'DEEPSLEEP', 'DEEPSLEEP_RESET', 'EXT0_WAKE',
'EXT1_WAKE', 'HARD_RESET', 'I2C', 'PIN_WAKE', 'PWM', 'PWRON_RESET', 'Pin', 'RTC', 'SLEEP',
'SOFT_RESET', 'SPI', 'Signal', 'SoftI2C', 'SoftSPI', 'TIMER_WAKE', 'TOUCHPAD_WAKE', 'Timer',
'UART', 'ULP_WAKE', 'WDT', 'WDT_RESET', 'bitstream', 'deepsleep', 'disable_irq', 'enable_irq', 'freq',
'idle', 'lightsleep', 'mem16', 'mem32', 'mem8', 'reset', 'reset_cause', 'sleep', 'soft_reset', 'time_pulse_us',
'unique_id', 'wake_reason']
>>> import umachine
>>> dir(umachine)
['_class__', '__name__', 'ADC', 'ADCBlock', 'DEEPSLEEP', 'DEEPSLEEP_RESET', 'EXT0_WAKE',
'EXT1_WAKE', 'HARD_RESET', 'I2C', 'PIN_WAKE', 'PWM', 'PWRON_RESET', 'Pin', 'RTC', 'SLEEP',
'SOFT_RESET', 'SPI', 'Signal', 'SoftI2C', 'SoftSPI', 'TIMER_WAKE', 'TOUCHPAD_WAKE', 'Timer',
'UART', 'ULP_WAKE', 'WDT', 'WDT_RESET', 'bitstream', 'deepsleep', 'disable_irq', 'enable_irq', 'freq',
'idle', 'lightsleep', 'mem16', 'mem32', 'mem8', 'reset', 'reset_cause', 'sleep', 'soft_reset', 'time_pulse_us',
'unique_id', 'wake_reason']
>>>
```

The list is a bit hard to read on the screen and so I have rewritten the list below in an easier to read format along with what each class is to be used for.

- ADC - Used for controlled the built in Analog to Digital Converter.
- ADCBlock - Used for finer control over the ADC functions.
- DEEPSLEEP
- DEEPSLEEP_RESET
- EXT0_WAKE
- EXT1_WAKE
- HARD_RESET

- I2C - Used for I2C device communication on dedicated pins,
- PIN_WAKE
- PWM - Used for PWM control of motors and lighting,
- PWRON_RESET
- Pin - Used for Direct access to the pins/GPIO's,
- RTC - Used for controlling the Real Time Clock.
- SLEEP
- SOFT_RESET
- SPI - Used for SPI device communication on dedicated pins,
- Signal
- SoftI2C - Alternative non-pin dependent I2C functions,
- SoftSPI - Alternative non-pin dependent SPI functions,
- TIMER_WAKE
- TOUCHPAD_WAKE
- Timer
- UART
- ULP_WAKE
- WDT
- WDT_RESET
- bitstream
- deepsleep
- disable_irq
- enable_irq
- freq
- idle
- lightsleep
- mem16
- mem32
- mem8
- reset
- reset_cause
- sleep
- soft_reset
- time_pulse_us
- unique_id
- wake_reason

The Pin Class.

In order to control hardware connected to the physical pins of the M5Stamps we need to make use of the built in library of functions located in the Pin class which is found in Machine/Umachine in Mainstream Micropython firmware, M5 on UIFlow2 Firmware and Board in Adafruits Circuitpython version of Micropython. We can see from the list show previously that there are a lot of classes available in the **umachine** module but, for basic GPIO control we just need the **Pin** class. To access the **Pin** class we need type the following in to **REPL** or add it to the top of a program.

```
from umachine import Pin
```

Once the Pin class is imported we now have access to the following Pin functions.

- DRIVE_0 - Sets pin/GPIO sink/source current to 5mA,
- DRIVE_1 - Sets pin/GPIO sink/source current to 10mA,
- DRIVE_2 - Sets pin/GPIO sink/source current to 20mA,
- DRIVE_3 - Sets pin/GPIO sink/source current to 40mA,
- IN - Sets pin/GPIO mode as an Input,
- IRQ_FALLING - Sets the IRQ to trigger if a signal falls from high to low
- IRQ_RISING - Sets the IRQ to trigger if the signal increases from low to high.
- OPEN_DRAIN - Sets pin/GPIO mode as open drain,
- OUT - Sets pin/GPIO mode as an Output,
- PULL_DOWN - Sets the internal pulldown resistor to on,
- PULL_UP - Sets the internal pull-up resistor to on
- WAKE_HIGH - Triggers the IRQ when the pin gets a high signal
- WAKE_LOW - Triggers the IRQ when the pin gets a low signal
- init - Used to setup a pin/GPIO,
- irq - Used to configure pins as IRQ's,
- off - Sets the initial state of the pin/GPIO to off or 0
- on- Sets the initial state of the pin/GPIO to on or 1

The above list is a collections or arguments that can be passed to the Pin Class by placing them in between brackets '()' when defining the function of the pin. When creating a pin we first give it a name, call the class and then set the arguments for the pin to use, for example:

```
Pin4 = Pin(4, Pin.OUT)
```

pin0 is the pin name and is in lowercase because the Pin class is already created with a capital 'P'. After the '=' we have the class 'machine.Pin()' (shortened to just Pin) which states to use the Pin class from the umachine module.

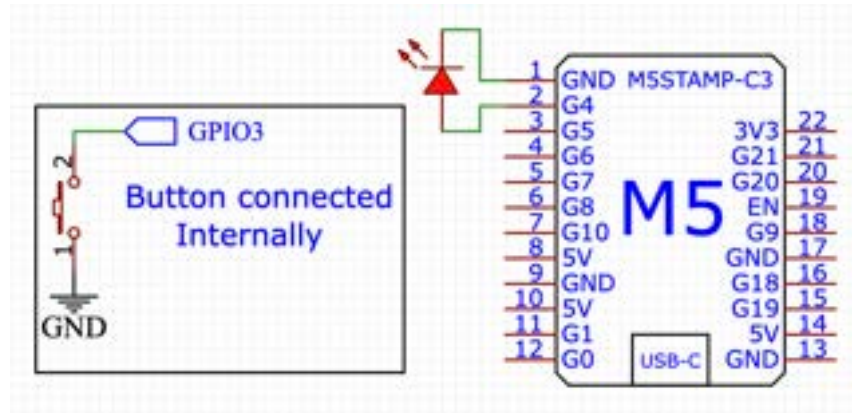
Next we have a selection of arguments starting with the pin number which is where the GPIO pin is declared followed by the mode (input or output) followed by the argument that declares if the internal

pull up/pull down resistor is to be used.

If you look back at the list above you will see that there are more arguments that can be added but for a basic example these are the basic arguments required to set up pin control.

Example 1.0 - Turn on an LED with a button.

For this first example I will show you how to use the basic pin function to turn on and turn off an LED. The LED will be connected to GPIO4 but for the button I will use the on board user programmable button connected to GPIO3.



Circuit diagram showing the externally connected led and how the button is connected internally.

While it is not recommended to use an LED without a resistor in this way, the M5Stamp C3's GPIO's can be limited to only output enough current to light up the LED as I will show later. To control the Led with the button we first need to set up the pins:

```
>>> from umachine import Pin
>>> led = Pin(4, Pin.OUT)
>>> button = Pin(3, Pin.IN, Pin.PULL_UP)
```

Here I have set the pull-up resistor but this may cause confusion because if we run the following:

```
>>> button = Pin(3, Pin.IN, Pin.PULL_UP)
>>> button.value()
1
>>> button.value()
0
>>>
```

Pin value returns a 1 when not pressed and a 0 when pressed because the internal pull-up resistor is pulling the pin unto 3.3v when not in use. The Pull up resistor has to be set high because if we look at the circuit diagram we see that the pin gets pulled low when the button is pressed.

Once the pins are defined we need some sort of logic to connect them together. For this we can use an If function inside a while loop. The loop function is used to make the inclosed code constantly repeat itself while the if function waits for an event to happen:

```
while True:  
    if button.value() == 0:  
        led.value(1)  
        led.value(0)
```

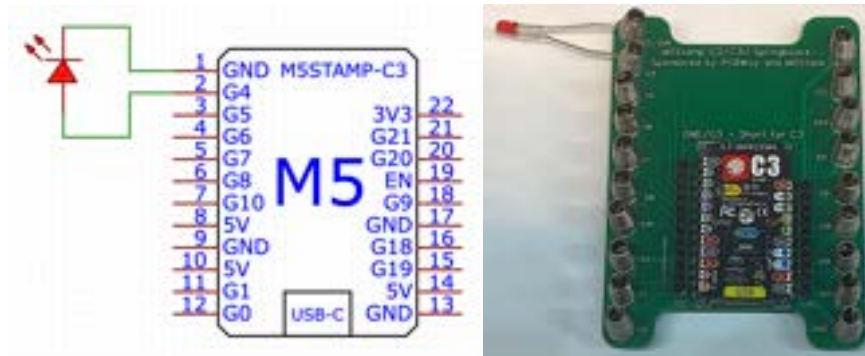
If you run the final code:

```
from umachine import Pin  
led = Pin(4, Pin.OUT)  
button = Pin(3, Pin.IN, Pin.PULL_UP)  
while True:  
    if button.value() == 0:  
        led.value(1)  
        led.value(0)
```

You won't see anything happen but when you press the button on the front of the M5Stamp C3 you will see the LED connected to GPIO4 light up and then go out when you release the button.

Example 1.1 - Blink an LED.

In the first example I showed you how to manually light up an LED connected to the M5Stamp C3 by pressing the onboard button. In this example I will show you how to make the LED come on and off “Blink” without any pressing of the buttons. Leave the LED connected to GPIO4 as shown in the circuit diagram.



Basic circuit diagram and photo of an LED connected to the M5Stamp C3's GPIO4 and GND. .

Type the following code into Thonny and save the file to the M5Stamp C3 as main.py (all lower case).

```
import machine
import time

pin4 = Pin(4, Pin.OUT, Pin.PULL_UP)
while True:
    pin0.on()
    wait(0.5)
    pin0.off()
    wait(0.5)
    wait_ms(2)
```

In the above code I imported all of the machine functions by by importing the whole machine module:

```
import machine
```

Were as on the previous page I used to only import the Pin class from the machine Module:

```
from machine import Pin
```

From a beginners point of view there is nothing much to be gained between the different import lines of code but when programs are starting to get memory intensive then importing just specific classes reduces how much memory a program will use.

In order to use a pin for GPIO use, we first need to define what pin is to be used. To define a pin, use the following line of code immediately after the import line:

```
pin4 = Pin(4, Pin.OUT)
```

For the sake of my sanity I called the pin pin4 as I'm using GPIO4 and I could have chosen something silly like elvis!

Next I call the pin class and then use the the argument 'Pin.OUT' to define the pin as an output. If I want to use the pin as an input, I would write the line as follows:

```
pin4 = Pin(4, Pin.IN, Pin.PULL_UP)
```

Which set the pin as an input and sets the internal "PULL UP" or "PULL DOWN" resistor.

Next we can use an additional argument to set the pins initial state as on with:

```
pin4 = Pin(4, Pin.OUT, value=1)
```

Which set the Pins initials value to on (0) or off (1). For some reason, the outputs of an ESP32 work backwards and uses 1 for off and 0 for on! If you look at the list of functions on the previous page you can see that there are the functions off() and On() that can be used in place of value(0) and value(1) respectively.

```
pin4 = Pin(4, Pin.OUT, drive=Pin.DRIVE_3)
```

The Pin.DRIVE_3 allows control of a pins source/sink currents. In Micropython there are four available options for drive currents and they are as follows:

- Pin.DRIVE_0: 5mA / 130 ohm
- Pin.DRIVE_1: 10mA / 60 ohm
- Pin.DRIVE_2: 20mA / 30 ohm (default strength if not configured)
- Pin.DRIVE_3: 40mA / 15 ohm

Once the GPIO/Pin has been defined, a program can be created to control the pin.

```
from machine import Pin
```

```
pin4 = Pin(4, Pin.OUT)
```

```
pin4.on()  
wait(0.5)
```



```
pin4.off()
wait(0.5)
```

Time Class.

Unfortunately when run, the program will only run once very fast and then end. To get the code to continuously run and flash the LED at a speed we can see, the code must be place in a while true loop and the time functions must be imported to allow timing control in the code and then the time.sleep() function added to pause the code between turning the LED connected to the pin on and off.

```
from machine import Pin
import time
```

```
pin4 = Pin(4, Pin.OUT)
while True:
    pin4.on()
    time.sleep(0.5)
    pin4.off()
    time.sleep(0.5)
    time.sleep_ms(2)
```

You can view a short video of this code in action here: <https://youtube.com/shorts/VjIZ2rRLnes?feature=share>

If we import the time module and run the dir(time) command, we see the following arguments for the time class.

- gmtime - Returns the current UTC time zone data.
- localtime - returns the local time zone data.
- mktime - returns the number of second that have passed since 1st jan 2000 and a specified time.
- sleep - used to add a pause in a program measured in seconds.
- sleep_ms - used to add a pause in a program measured in milliseconds.
- sleep_us - used to add a pause in a program measured in microseconds.
- ticks_add - Ad to tick count,
- ticks_cpu - Sets the tick rate to the cpu cycles,
- ticks_diff - Compares ticks to another timer,
- ticks_ms - Sets tick to milisecond,
- ticks_us- Set ticks to microseconds.
- time - Used to return the time in seconds since the last epoc.
- time_ns - Used to return the time in nanoseconds since the last epoc.

As you can see in the example above, you have to use the class name followed by the time function for example:

```
time.sleep()
```

The three main arguments we are interested in first is `sleep`, `sleep_ms` and, `sleep_us`.

- `sleep` is used when we want to define a time in seconds,
- `sleep_ms` is for when we want to define a time in milliseconds,
- `sleep_us` is for when we want to define a time in microseconds.

The `sleep()` functions are used to inset a pause between commands and functions being executed in programs.

The function `gmtime` is used to get the current system time. To use `gmtime()` you use the following code:

```
import time
```

Followed by:

```
time.gmtime()
```

When run in REPL `time.gettime` will return the system time of the host computer running Thonny as an eight tuple list:

```
>>> import time
>>> time.gmtime()
(2023, 1, 28, 14, 4, 50, 5, 28)
>>>
```

The format of the entries in the eight tuple list are as follows:

- year includes the century (for example 2023.
- month is 1-12
- mday is 1-31
- hour is 0-23
- minute is 0-59
- second is 0-59
- weekday is 0-6 for Mon-Sun
- yearday is 1-366

Along with `time.gmtime()` there is also `time.localtime()` which return the current time from the host computer. The `.gmtime()` should return the UTC time where as `.localtime` returns the local area timezone time.

The function `time.mktime` works differently to the previous two in that instead of returning a eight tuple value, `mktime` returns the number of seconds that have passed since the first of January 2000:

```
>>> import time
>>> time.gmtime()
>>> time.mktime((2023, 1, 28, 14, 12, 42, 5, 28))
728230362
>>>
```

The ticks functions are used for counters. The ticks values are not connected to any timing system but can be connected to use other timing functions,

`Ticks_cpu` is used to count the cycles of the microprocessors core `cpu`, `ticks_ms` is used to increase the ticks counter every millisecond and `tics_us` is used to increase the ticks counter every microsecond.

`Ticks_add` is used to offset the ticks value being added to the ticks counter independent of a time base and `ticks_diff` is used to calculate the difference in values between two ticks counters, for example:

```
print(time.ticks_ms())
```

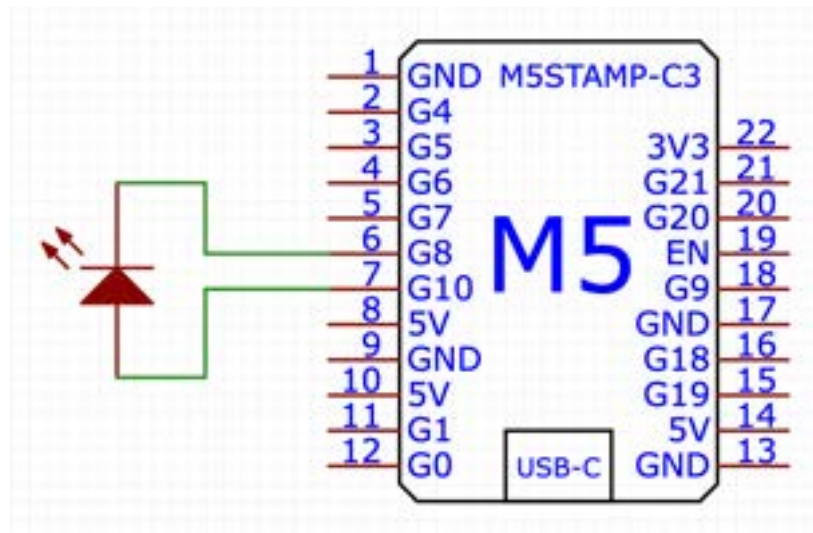
Returns the current ticks count of the millisecond counter running.

So far I have just discussed using the GPIO's as output pins but, they can also be used as input pins. When used as input pins, GPIO's can behave in one of two ways. The first is as an ordinary input GPIO which is queried during a programs process and the second is as an Interrupt Pin which interrupts the flow of a program Kind of like an emergency stop button on a production line).

`time()` and `time_ns()` are used to return seconds since a predefined epoch event. By default the epoch event is set to 1st January 1970 00:00:00 UTC.

Experiment 1.2 - BiColour LED Blinker with 2 GPIO's.

In the following example I will show you how to control a bicolour (Red/Green) LED using only two GPIO's. The LED is connected to GPIO8 and GPIO10 as shown in the circuit diagram below:



Circuit diagram of a Bicolour LED connected to GPIO8 and GPIO10

This works because both pins are able to both sink and source current to the LED. On paper it doesn't look like it should work because both pins are set as output. To make the LED light one pin is set as on which results in the pin "sourcing" current while when a pin is set to off the pin is "sinking" current.

The Code:

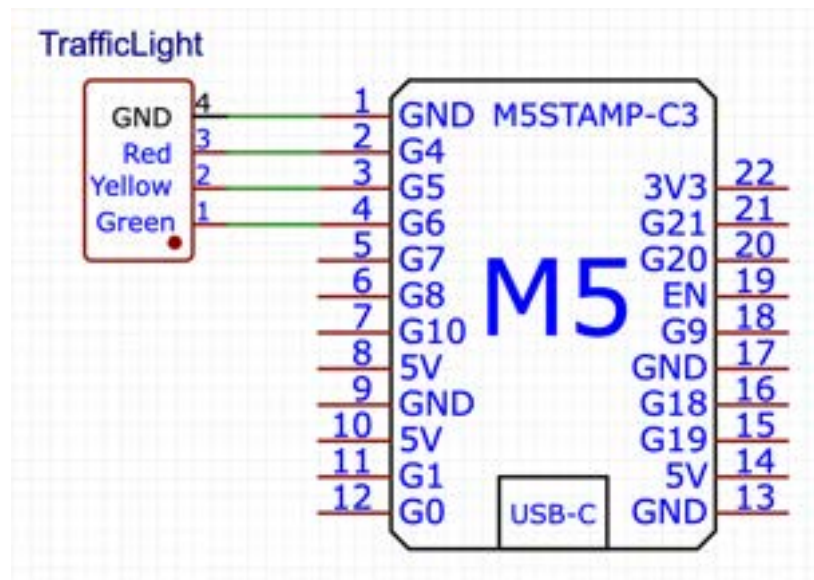
```
from umachine import Pin
import time
while True:
    leg2.off()
    leg1.on()
    time.sleep(0.5)
    leg1.off()
    leg2.on()
    time.sleep(0.5)
```

Experiment 1.3 - LED Traffic Light

In this experiment I will show you how to use one of the cheap LED traffic lights modules available in electronic starter kits. The particular module I used was purchased on Amazon from the seller AZDelevry: https://www.amazon.co.uk/dp/B086TRY65Y?ref=ppx_yo2ov_dt_b_product_details&th=1



These modules use only four pins to control them and need to be connected to the M5StampC3 as shown:



Circuit diagram of the Traffic Light modules connected to the M5StampC3.

For the M5StampC3 the traffic light can just be plugged in but, for other modules you will need to use jumper wires in order to connect them together.

The code for controlling the traffic light module is simple and just requires the Pin and Sleep modules:

```
from umachine import Pin
import time
```

We then define the three LEDs and which physical pin they are connected to along with what mode the pins need to use.

```
red = Pin(4, Pin.OUT)
yellow = Pin(5, Pin.OUT)
green = Pin(6, Pin.OUT)
```

And then create a simple while loop to turn the LEDs on or off in a pattern

```
while True:
    red.on()
    time.sleep(1)
    yellow.on()
    time.sleep(1)
    green.on()
    yellow.off()
    red.off()
    time.sleep(1)
    time.sleep(1)
    green.off()
    yellow.on()
    time.sleep(1)
    green.off()
    yellow.off()
    red.on()
    time.sleep(1)
```

This loop looks a bit strange to some readers but this is the pattern used by UK traffic lights.

Neopixel Module and Class

So far I have just showed how to light up a single LED connected to the GPIO's of the M5Stamp but the M5Stamps have a built in RGB led. To use the RGB LED we need to make use of the Neopixel Module which allows us to control WS2812/Neopixel based LED devices.

To access the neopixel module we type:

```
>>> from machine import Pin
>>> import neopixel
>>> dir(neopixel)
['__class__', '__name__', '__file__', 'bitstream', 'NeoPixel']
>>>
```

We can see that there isn't much to the module except for the classes "bitstream" and "NeoPixel" but, if type:

```
>>> from neopixel import NeoPixel
>>> dir(NeoPixel)
['__class__', '__getitem__', '__init__', '__len__', '__module__', '__name__', '__qualname__',
 '__setitem__', 'write', '__bases__', '__dict__', 'fill', 'ORDER']
>>>
```

We can see that the NeoPixel class has a few arguments. To use the built in RGB LED we initialise the pin that the RGB LED is connected to with:

```
RGBLed = NeoPixel(Pin(2), 1)
```

And then type the following to light up the RGB LED:

```
>>> RGBLed[0] = (255,0,0)
>>> RGBLed.write()
```

Which will turn on the RGB LED.

To blink the LED on or off we just need to use a while loop similar to what has been shown before:

```
from machine import Pin
from neopixel import NeoPixel
import time

RGBLed = NeoPixel(Pin(2), 1)
```

```

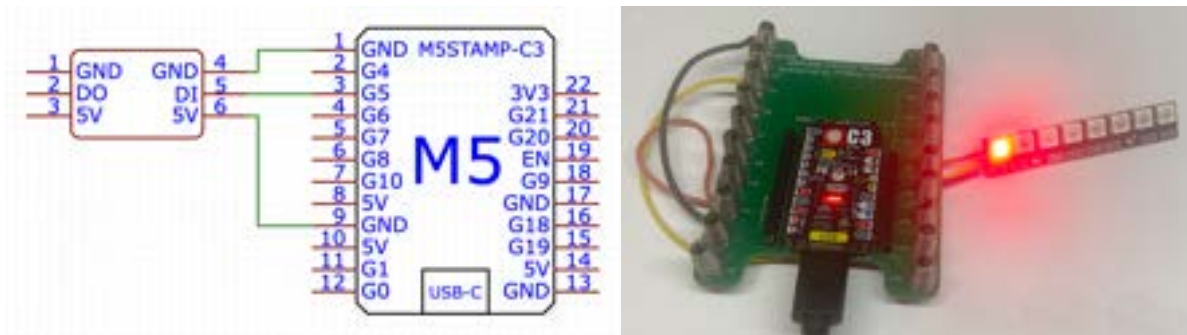
while True:
    RGBLed[0] = (255,0,0)
    RGBLed.write()
    time.sleep(0.5)
    RGBLed[0] = (0,0,0)
    RGBLed.write()
    time.sleep(0.5)

```

You can view a video of the RGB LED flashing on and off here:

Example 1.3 - RGB LED Strip

I have just showed you how to control the internal RGB LED but how do you control external RGB LED's?



An eight WS2812/Neopixel RGB LED string connected to GPIO5.

As long as they are of the WS2812/Neopixel type, control is just the same as I have shown. The only difference between controlling the Internal RGB LED and external RGB LED's is the GPIO specified in:

```

RGBLed = NeoPixel(Pin(5), 1)

```

Instead of using GPIO2 which is internally connected to the onboard RGB LED we can use any of the numbered pins around the outside of M5Stamps such as GPIO5 for example:

```

from machine import Pin
from neopixel import NeoPixel
import time

```

```

RGBLed = NeoPixel(Pin(5), 8)

```

```

while True:
    RGBLed[0] = (255,0,0)
    RGBLed.write()

```



```
time.sleep(0.5)
RGBLed[0] = (0,0,0)
RGBLed.write()
time.sleep(0.5)
```

You can view a video of the example running here: <https://youtube.com/shorts/2qxmaVj6Bmk?feature=share>

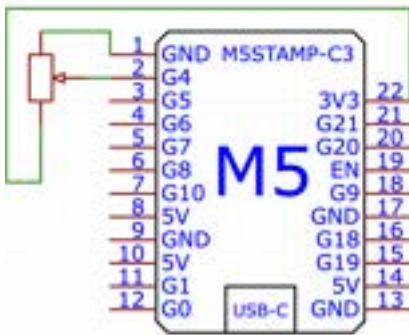
ADC Block Class

The ADC Block class is used to gain finer control of the ADC channels available on the M5Stamps. On the M5Stamp C3 And C3U the ADC pins are broken out out GPIO0, GPIO1, GPIO4 and GPIO5 which are connected to ADC Block 1. While there is a second ADC block, the second block is not broken out as its also used by wifi and any attempt to access it will break WIFI use.

ADC Class

Example 1.4 - Analog Input

In order to show how the ADC class works I will show you how to connect a potentiometer or variable resistor to an ADC pin in order for the M5Stamp to read the value and display the value on screen.



In example 1.0 I showed you how to read the internal button. The problem with the button is that it only returns a value of 0 or 1. If the GPIO is configured as an ADC we will get values of 0 to 4095 due to the 12bit resolution of the ADC. In order to configure the GPIO as an ADC we need to import the ADC class as well as the Pin class:

```
from umachine import ADC, Pin
```

And then we define the pin as an ADC input with:

```
ADCPin4 = ADC(Pin(4))
```

And to read the potentiometer connected to GPIO4 we just use:

```
print(ADCPin4.read_u16())
```

However if we adjust the potentiometer the reading will not change until we issue the print line again. In order to continuously read the potentiometer connected to GPIO4 we need to place the print function in a while true loop. And import the time class as we did previously.

```
from machine import ADC, Pin  
import time
```

```
ADCPin4 = ADC(Pin(4))  
while True:  
    print(ADCPin4.read_u16())  
    time.sleep_ms(100)
```

You can see a video of the example working here:

Pin as Interrupt Request (IRQ).

As well as using pins as inputs and outputs, you can also use pins as interrupts or IRQ (**I**nterrupt **R**e**Q**uest). When pins are set as an IRQ the code sits outside of the main code flow until the pin is pressed and then the code connected to the IRQ will interrupt the main program loop.

To use the pin as an IRQ you would use:

```
pin1 = Pin.irq(1, trigger=Pin.IRQ_RISING)
```

When the pin is set to act as an IRQ, the running code will be interrupted when the pin is triggered by a voltage that raises from 0 to 3.3v.

When used as an IRQ the Pin.irq class has the following Arguments that are used to trigger the IRQ pin:

- Pin.IRQ_FALLING - When the pin detects the pin voltage dropping,
- Pin.IRQ_RISING - When the pin detects the pin voltage rising,
- Pin.IRQ_LOW_LEVEL - When the pin is pulled low,
- Pin.IRQ_HIGH_LEVEL - when the pin gets pulled high,

Example 1.5 - Interrupt Pin

In order to show the use of an IRQ I will use the button on the bottom of the M5Stamp which is connected to GPIO3. We need to add the button as an IRQ:

```
from machine import Pin
import time

button_pressed_count = 0

def button1_pressed(change):
    global button_pressed_count
    button_pressed_count += 1

button1 = Pin(3 Pin.IN, Pin.PULL_DOWN)

button1.irq(handler=button1_pressed, trigger=Pin.IRQ_FALLING)

button_pressed_count_old = 0
while True:
    if button_pressed_count_old != button_pressed_count:
        print('Button 1 value:', button_pressed_count)
        button_pressed_count_old = button_pressed_count
```

If you run this example and look in the repel you will see:

```
>>> %Run -c $EDITOR_CONTENT  
Button 1 value: 1
```

With the Button 1 value increasing with each button press.

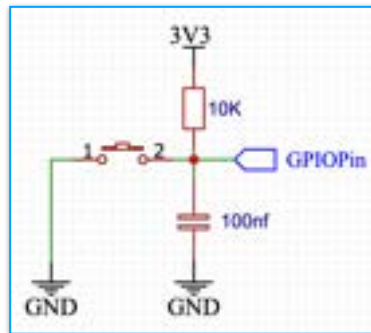
Rotary Encoder



In this section I will show you how to use a rotary encoder. We can not connect a raw rotary encoder directly as the readings will be non consistent due to contact bounce. M5Stack produce a Rotary encoder unit that has been assembled with the additional “debounce” circuitry built in (<https://shop.m5stack.com/products/encoder-unit>) or we can use the Encoder adapter unit for connecting raw encoders via the I2C bus with the additional “debounce” circuitry built in.

Bounce and Debounce

If you have tried reading an external button connected to the M5Stamps, you may have noticed a similar inconsistency read or false press just like we are seeing with the rotary encoder. To reduce the false triggers we need to debounce the pin readings. There are two ways we can debounce the readings, the first is to use a hardware debounce circuit like M5Stack are using consisting of nothing more then a resistor pulling the pin high and a capacitor connecting the pin to ground forming a resistor/capacitor filter.



Example of hardware debounced button used on M5Stack devices.

When the button is not pressed the capacitor is charged. When the button is pressed the capacitor is discharged through the button holding the pin high.

In UIFlow2 the Control blocks for the Rotary encoder and encoder unit only exist for the M5Dial which has a built in rotary encoder connected to the outer ring.



Below is the micropython code generated by UIFlow when the above blocks are used.

```
import os, sys, io
import M5
from M5 import *
from hardware import *

label0 = None
rotary = None

def setup():
    global label0, rotary
    M5.begin()
    label0 = Widgets.Label("label", 72, 88, 1.0, 0xffffffff, 0x222222, Widgets.FONTS.DejaVu18)

    rotary = Rotary()
    label0.setCursor(x=50, y=100)

def loop():
    global label0, rotary
    M5.update()
    label0.setText(str(str((rotary.get_rotary_value()))))
    label0.setVisible(True)

if __name__ == '__main__':
    try:
```

```
setup()
while True:
    loop()
except (Exception, KeyboardInterrupt) as e:
    try:
        from utility import print_error_msg
        print_error_msg(e)
    except ImportError:
        print("please update to latest firmware")
```

If we don't have the M5Dial, it doesn't mean we can't use the rotary encoder. To use the encoder we have to dive back into Micropython to read the encoder.

The PWM Class.

The PWM (Pulse Width Modulation) class is used to provide timing controls to better control LEDs, Motor, and servos. In the past the voltage or the current would be raised or lowered to increase or decrease a motors. Speed or a lights brightness. PWM works by providing full power to a device but altering how long the device is switched on for. In the case of a servo, altering the width of the pulse changes the position of a servos horn.

To access the PWM functions we import the PWM class from the umachine module with:

```
from umachine import Pin, PWM
```

And then we can view the class with the dir(PWM) command:

```
>>> from umachine import Pin, PWM
>>> dir(PWM)
['__class__', '__name__', '__bases__', '__dict__', 'deinit', 'duty', 'duty_ns', 'duty_u16', 'freq', 'init']
>>>
```

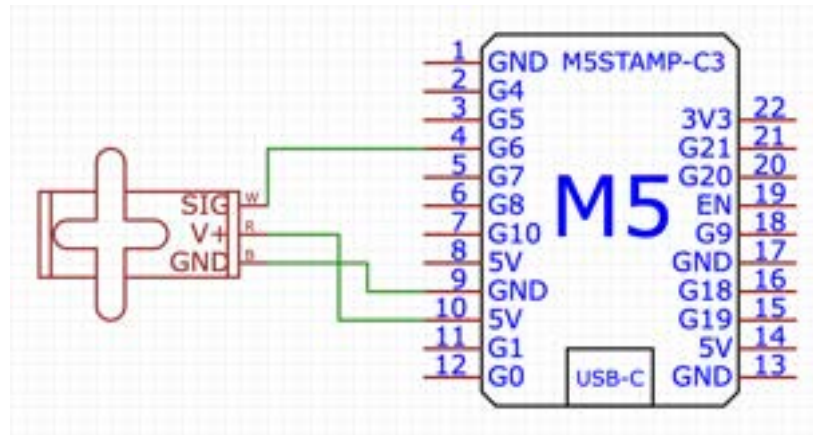
The PWM class consist of the following functions and arguments:

- `__class__`
- `__name__`
- `__bases__`
- `__dict__`
- `init` - Used to Setup a Pin for PWM use,
- `deinit` - Clear the pin of the PWM setting to allow it to be used elsewhere,
- `duty` - Set the PWM duty time in seconds,
- `duty_ns` - Sets the duty time in nanoseconds
- `duty_u16`
- `freq` - Set the signal frequency.

Example 1.6 - PWM Servo Control

In order to demonstrate the use of PWM I am using the 180 degree servo from M5Stack found here: <https://shop.m5stack.com/products/servo-kit-180> and the 360 degree servo from M5Stack found here: <https://shop.m5stack.com/products/servo-kit-360>

In example 1.4 the M5Stack Micro servo is connected to GPIO as shown below.



Circuit diagram showing the servo connected to GPIO6.

In order to use a pin for PWM control, we first need to import the Pin and PWM classes with:

```
from umachine import Pin, PWM
```

And then initialise the pin as a PWM pin with:

```
servo = PWM(Pin(6), 50)
```

Which initialises GPIO6 as a PWM pin with a base frequency of 50Hz. 50hz is the lowest frequency I can get the M5Stack servos to run before they start trying to self destruct. Next we set the initial servo position, through trial and error I have found that at 50Hz, the servos I am using more to the 0, 90 and 180 with the following duty times:

```
servo.freq(50)  
servo.duty(30) #Far right rotation  
servo.duty(75) #Center position  
servo.duty(135) #Far Left rotation
```

When I increase the servo.freq() to 100hz, the positions are as follows:

```
servo.freq(100)
servo.duty(55) #Far right rotation
servo.duty(150) #Center position
servo.duty(260) #Far left rotation
```

This shows that dependent on operating frequency the servo position will not use the same servo duty. Now we have are base level functions worked out, we can create the following program:

```
from umachine import Pin, PWM
import time
servo = PWM(Pin(6), 50)
while True:
    servo.duty(30)
    time.sleep(0.5)
    servo.duty(75)
    time.sleep(0.5)
    servo.duty(135)
    time.sleep(0.5)
    servo.duty(75)
    time.sleep(0.5)
    servo.duty(30)
    time.sleep(0.5)
```

If the program is then run you will see the servo move from one side to the other stopping in the middle.

You can see a video of this program running here: <https://youtube.com/shorts/xciQBikbtWw?feature=share>

Example 1.7 - PWM LED Control.

In examples 1.0 and 1.1 I mention earlier that You can use the PWM functions to control the brightness of an LED. In the following example I use the same circuit diagram from those earlier examples but use PWM to increase the brightness of the LED connected to GPIO4 until it reaches a set point and then decrease the LED brightness.

```
from machine import Pin, PWM  
from time import sleep
```

```
pwm = PWM(Pin(4))  
pwm.freq(1000)
```

```
while True:  
    for duty in range(65025):  
        pwm.duty_u16(duty)  
        sleep(0.0001)  
    for duty in range(65025, 0, -1):  
        pwm.duty_u16(duty)  
        sleep(0.0001)
```

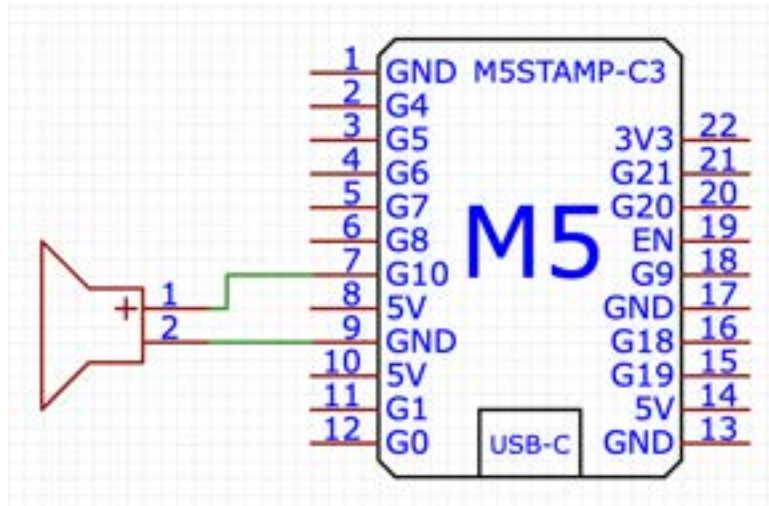
You can view a short video of this code running here: <https://youtube.com/shorts/uX469AqeJyI>

The code work by increasing the led brightness over 65025 steps and then when it reaches 65025 slowly decreases the brightness by the same amount of steps.

Example 1.8 - Play music with PWM.

Not only can PWM be used for controlling Servos LEDs and motors, PWM can also be used for producing sound. In this example I will use the PWM class and functions in order to play some sounds out of a buzzer.

For this example, the buzzer is connected to GPIO10 as shown below.



Buzzer connected to GPIO10.

In order to play music we need to use the sleep function as well and the Pin and PWM functions, so as before we start with:

```
from machine import Pin, PWM
from time import sleep
```

Next we define GPIO10 as the buzzer pin with:

```
BUZZER_PIN = 10
```

And then we create a PWM pin with:

```
buzzer = PWM(Pin(SPEAKER_PIN))
```

Once this is done we next set the buzzers duty cycle using:

```
buzzer.duty_u16(1000)
```

Which causes the buzzer to emit an annoying tone. Change the tone to a better sounding tone with:

```
buzzer.duty_u16(1000)
```

Followed by:

```
buzzer.freq(1000)
```

And to turn off the sound use:

```
buzzer.duty_u16(0)
```

This isn't very interesting until we place it in a loop:

```
while True:
    buzzer.duty_u16(1000)
    sleep(1)
    buzzer.duty_u16(0)
    sleep(1)
```

And now we have a little alarm like sound turning on and off.

Playing tone is not very interesting and so lets play some music:

```
from umachine import Pin, PWM
from utime import sleep
buzzer = PWM(Pin(10))

tones = {
    "B0": 31, "C1": 33, "CS1": 35, "D1": 37, "DS1": 39, "E1": 41, "F1": 44, "FS1": 46,
    "G1": 49, "GS1": 52, "A1": 55, "AS1": 58, "B1": 62, "C2": 65,
    "CS2": 69, "D2": 73, "DS2": 78, "E2": 82, "F2": 87, "FS2": 93, "G2": 98,
    "GS2": 104, "A2": 110, "AS2": 117, "B2": 123, "C3": 131, "CS3": 139,
    "D3": 147, "DS3": 156, "E3": 165, "F3": 175, "FS3": 185,
    "G3": 196, "GS3": 208, "A3": 220, "AS3": 233, "B3": 247, "C4": 262, "CS4": 277, "D4": 294, "DS4": 311,
    "E4": 330, "F4": 349, "FS4": 370, "G4": 392, "GS4": 415, "A4": 440, "AS4": 466, "B4": 494, "C5":
    523, "CS5": 554, "D5": 587, "DS5": 622, "E5": 659, "F5": 698,
    "FS5": 740, "G5": 784, "GS5": 831, "A5": 880, "AS5": 932, "B5": 988, "C6": 1047, "CS6": 1109, "D6":
    1175, "DS6": 1245, "E6": 1319, "F6": 1397, "FS6": 1480, "G6": 1568, "GS6": 1661,
    "A6": 1760, "AS6": 1865, "B6": 1976, "C7": 2093, "CS7": 2217, "D7": 2349, "DS7": 2489, "E7":
    2637, "F7": 2794, "FS7": 2960, "G7": 3136, "GS7": 3322, "A7": 3520,
    "AS7": 3729, "B7": 3951, "C8": 4186, "CS8": 4435, "D8": 4699, "DS8": 4978
}

song = ["E5", "G5", "A5", "P", "E5", "G5", "B5", "A5", "P", "E5", "G5", "A5", "P", "G5", "E5"]
mario = ["E7", "E7", 0, "E7", 0, "C7", "E7", 0, "G7", 0, 0, 0, "G6", 0, 0, 0, "C7", 0, 0, "G6",
    0, 0, "E6", 0, 0, "A6", 0, "B6", 0, "AS6", "A6", 0, "G6", "E7", 0, "G7", "A7", 0, "F7", "G7",
```

```
0, "E7", 0, "C7", "D7", "B6", 0, 0, "C7", 0, 0, "G6", 0, 0, "E6", 0, 0, "A6", 0, "B6", 0,  
"AS6", "A6", 0, "G6", "E7", 0, "G7", "A7", 0, "F7", "G7", 0, "E7", 0, "C7", "D7", "B6", 0, 0]
```

```
def playtone(frequency):  
    buzzer.duty_u16(1000)  
    buzzer.freq(frequency)
```

```
def bequiet():  
    buzzer.duty_u16(0)
```

```
def playsong(mysong):  
    for i in range(len(mysong)):  
        if (mysong[i] == "P" or mysong[i] == 0):  
            bequiet()  
        else:  
            playtone(tones[mysong[i]])  
            sleep(0.3)  
            bequiet()  
playsong(mario)
```

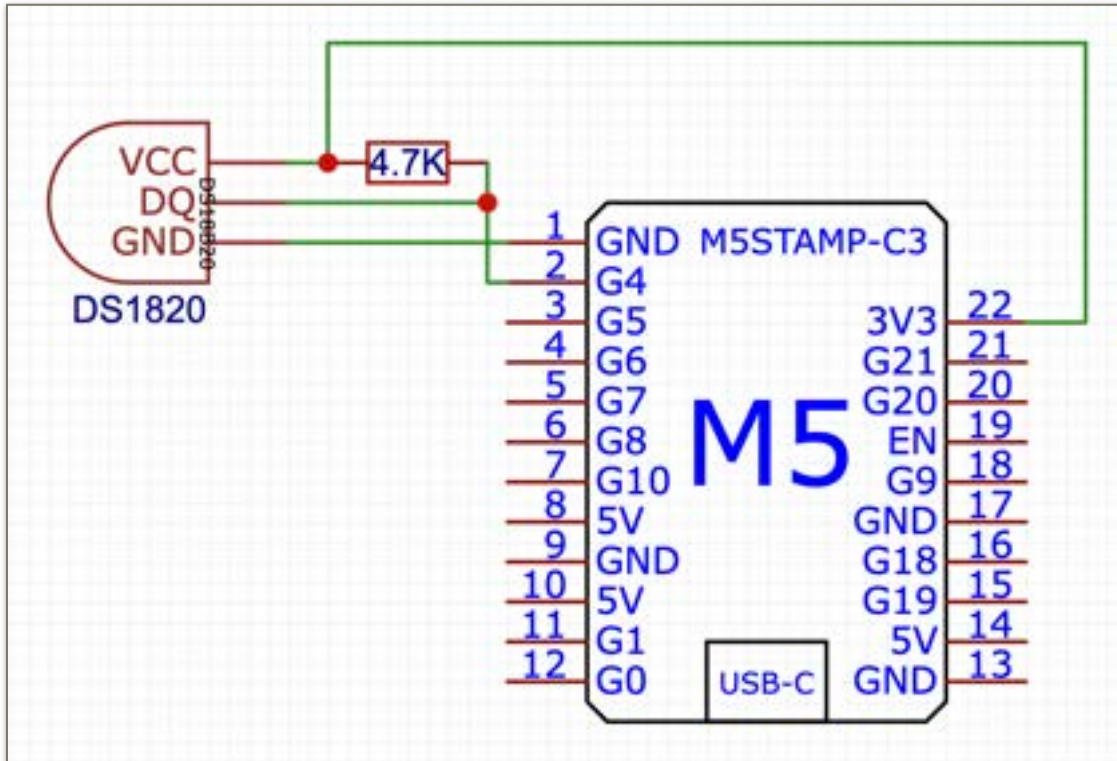
While these are just basic examples, by experimenting and testing these basic functions can be used to produce more elaborate sounds effects.

This isn't the only way to play music and we will explore another way in the I2S section.

One Wire Interface

Example 1.9 - DS18B20 One Wire Temperature.

In this example I have connected a DS18B20 which is a One Wire Temperature probe. These devices have only three wires (Positive, Negative and Signal) Positive is connected to 3V3, GND to any GND pin and the signal is connected to pin 4 with a resistor connected between 3V3 and Pin4



In order to use the DS18B20 we first import the needed libraries with:

```
import machine, onewire, ds18x20, time
```

We then define which pin the DS18B20 is connected to with:

```
ds_pin = machine.Pin(4)
```

We then define the device and attach it to the pin with:

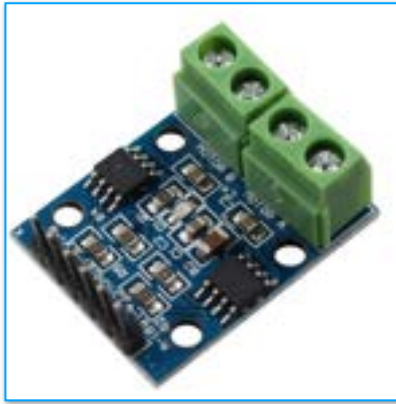
```
ds_sensor = ds18x20.DS18X20(owewire.OneWire(ds_pin))
```

We create a variable that will scan the interface and hold the DS18B20's addressing then we run a loop to get the temperature from the sensor and convert it to degrees Celsius with the following loop:


```
while True:
    ds_sensor.convert_temp()
    time.sleep_ms(750)
    for rom in roms:
        print(ds_sensor.read_temp(rom))
    time.sleep(5)
```

Example 1.10 - L9110S H-Bridge Driver

While it is possible to drive DC (2 Wire) motors with a max voltage of 3.3V from the M5Stamp pins, it is not advisable. If we just need simple On/Off motor control for motors we can use a cheap L9110S H Bridge driver module that can be purchased for only a few Pounds/Dollars.

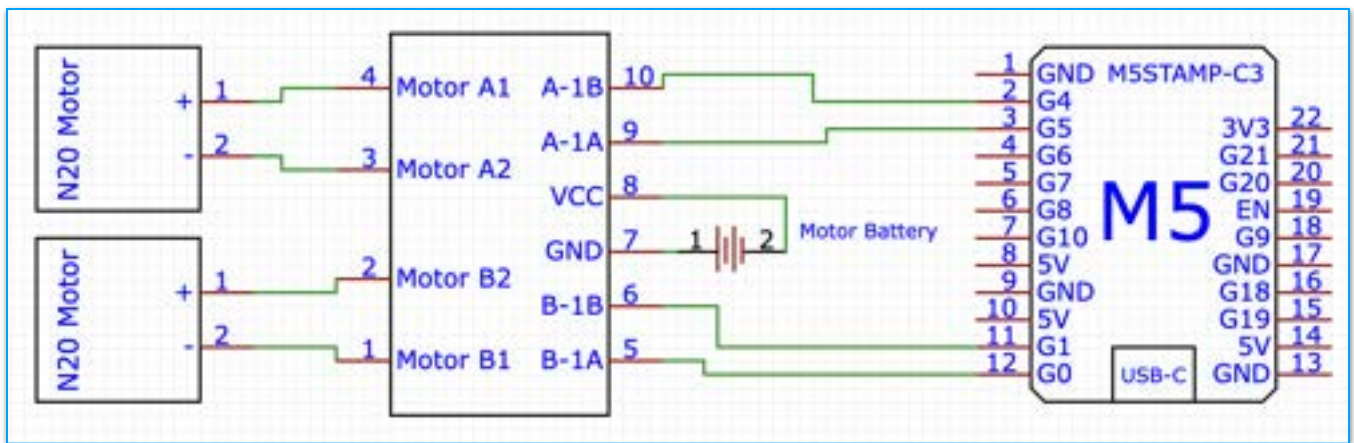


These modules actually contain two H-Bridge drivers and they are very easy to use. On One side we have the two 2-way connector blocks which are for connecting the motors, while on the other side we have a 6-way header which has two pins for connecting a battery for the motors and four control pins for controlling the two motors labeled A-1A, A-1B, B-1A, B-1B.

By default the Pins on the L9110s drivers are pulled high by the onboard resistors. To drive the motors the control pins must be pulled low. If both pins are pulled low at the same time then the drivers will sit in “Break” mode preventing the motor shafts from turning.

The simplicity of these drivers mean that we can even use a humble 555 timer to provide PWM speed control to the motors.

In the following schematic diagram I have connected the L1990s to the M5StampS3 with two N20 motors and a separate battery to power just the motors. Ideally there should also be a wire connecting the L1990s Negative to the M5Stamps Negative.



This example could be used for a simple robot chassis as long as the motors can run at 5V or under but, for motors that require higher voltages, a different H-Bridge rated for those voltages will be required and then you run the risk of the higher voltage being fed back through the control pins to the M5Stamp.

Test 1.0 - Model Railway Semaphore Signal.

I have on my Model railway layout a semaphore signal (the one with the arm) that needs automating. I should have by now given you enough information on guidance on how to control it.

Your test is to design for me the circuit diagram and Micropython program to control it from a push of a button!

I2C Communication in Micropython.

In order to demonstrate how to do I2C communication we will need some I2C units to connect to the M5Stamps. To demonstrate the use of I2C I will use the OLED unit from M5Stack: <https://shop.m5stack.com/collections/m5-sensor/products/oled-unit-1-3-128-64-display> along with the ENVIII sensor unit from M5Stack: <https://shop.m5stack.com/products/env-iii-unit-with-temperature-humidity-air-pressure-sensor-sht30-qmp6988>

In order for these to communicate with the M5Stamps in Micropython we need to use additional modules which will need to be downloaded and saved to the M5Stamps.

Connecting I2C units to the M5Stamps.

I2C can be run in Hardware I2C mode or Software I2C Mode. In Hardware mode, the I2C bus will use GPIO18 and GPIO19 (D+ and D-) which unfortunately is tied to the USB port that we use to communicate with Thonny on the host computer. To get around this hardware issue we need to use Software I2C mode which uses “Bit Banging” to force I2C communications on none I2C pins.

On the stamps are dedicated pins which can be used for connecting I2C units to the M5Stamps. I use the word ‘dedicated’ here because these pin locations can have a ‘Grove’ connector fitted to provide a simple connection to other M5Stack units. On the M5Stamp Pico D4 this connection is located on the bottom of the M5Stamp module whereas on the M5Stamp C3 and M5Stamp C3U it is located on the bottom left hand side.

The ‘Grove’ connector is provided in the M5Stamp C3 or M5Stamp C3U Mate kits and in order to solder the connector in place, you need to unscrew the cover, remove the “break off” section of the cover, solder the connectors to the M5Stamp and then refit the modified cover(see attached images).

TODO - Include images of fitting grove connector here !!!



M5Stamp C3 with Grove connector fitted.

I2C Class.

In order to access the I2C functions we need to import both the Pin functions and the I2C functions from the machine module with the following command:

```
from machine import Pin, I2C
```

When we run `dir(I2C)` the I2C functions we can use are as follow:

- `readinto` - Used to write to a specified device on the bus,
- `start` - Used to Manually trigger a Start condition,
- `stop` - Used to Manually trigger a Stop condition,
- `write` - Used to write the contents of a buffer to the I2C bus,
- `__bases__`
- `__dict__`
- `init` - Used to initialise the I2C Port,
- `readfrom` - Used to read bytes from a specific device on the bus,
- `readfrom_into` - Used to read bytes from a device into the buffer,
- `readfrom_mem` - Used to read from a devices stated memory address,
- `readfrom_mem_into` - Used to read from a devices stated memory address into the buffer,
- `scan` - Used to scan the I2C bus for connected to devices.
- `writeto` - Sends a byte array to a device.
- `writeto_mem` - Used to write to a devices specified memory location,
- `writeto` - writes the content of a vector to a device on the bus,

If we can't use the I2C class we can use the SoftI2C class with the following import line:

```
From machine import SoftI2C
```

If you run the `dir(SoftI2C)` function you will see the same functions and arguments as if using the I2C class:

```
>>> from machine import I2C, SoftI2C
>>> dir(I2C)
['__class__', '__name__', 'readinto', 'start', 'stop', 'write', '__bases__', '__dict__', 'init', 'readfrom',
'readfrom_into', 'readfrom_mem', 'readfrom_mem_into', 'scan', 'writeto', 'writeto_mem', 'writeto']
>>> dir(SoftI2C)
['__class__', '__name__', 'readinto', 'start', 'stop', 'write', '__bases__', '__dict__', 'init', 'readfrom',
'readfrom_into', 'readfrom_mem', 'readfrom_mem_into', 'scan', 'writeto', 'writeto_mem', 'writeto']
>>>
```

To set up GPIO's for I2C or SoftI2C, the code is the same:

```
From machine import Pin, I2C
i2c = I2C(0, scl=Pin(0), sda=Pin(1), freq=400000)
```

Or

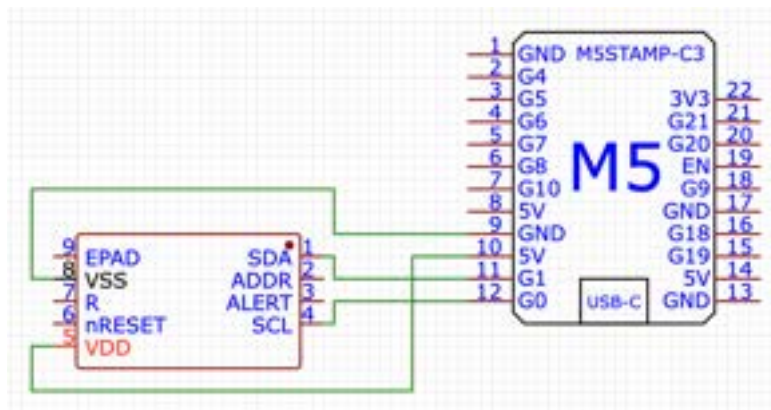
```
From machine import Pin, SoftI2C
i2c = SoftI2C(0, scl=Pin(0), sda=Pin(1), freq=400000)
```

For Soft I2C. GPIO0 and GPIO 1 are the pins assigned to the Grove connector on the bottom right of the M5Stamp C3 and M5Stamp C3U.

Earlier I said that you couldn't use hard I2C because it was tied to GPIO18 and GPIO19 (according to micropython.org but if we use the following code which reassigns the HardI2C to run on GPIO0 and GPIO1 followed by running a scan of the I2C bus, we can see the reported I2C address form the two sensors in the ENVIII unit.

Communicating with the ENVIII Unit.

The ENVIII Unit is connected to the M5Stamp C3 as shown in the following circuit diagram. Do not worry about the extra pins as they are connected inside the ENVIII unit.



Circuit diagram of the SHT30 connected to the M5Stamp C3.

Now that we can see that the I2C bus can see the I2C sensors, we need to find modules that will allow us to communicate and talk to the sensors. The two sensors in the ENVIII Unit consist of the SHT30 which reported its address as 68 and the QMP6988 which reported its address as 112. I won't worry about the QMP6988 at the moment and will just concentrate on the SHT30. While I could show you how to use a pre-made library for communication with the SHT30, I have had issues and so will take this time to work on a lower level I2C access method in order to show you how to understand I2C communication with the SHT30 sensor.

```
>>> from machine import Pin, I2C
>>> i2c=I2C(0,scl=Pin(0), sda=Pin(1), freq=400000)
>>> i2c.scan()
[68, 112]
>>>
```

If we run the same code but for SoftI2C, we get the same response:

```
>>> from machine import Pin, SoftI2C
>>> i2c=SoftI2C(scl=Pin(0), sda=Pin(1), freq=400000)
>>> i2c.scan()
[68, 112]
>>>
```

It is worth noting that when using Soft I2C, you do not need to declare the I2C channel number like you do with Hard I2C.

In Micropython, I2C address are reported as a decimal number. In order to get the Hex number you can use:

```
[hex(x) for x in i2c.scan()]
```

Which will return the addresses as a hexadecimal value:

```
>>> [hex(x) for x in i2c.scan()]
['0x44', '0x70']
```

After a lot of testing I found that I was having issue with the I2C class which were fixed when switched to the soft I2C class. With the change in class, the first block of code now looks like this:

```
>>> from umachine import Pin, SoftI2C,
>>> i2c=I2C(scl=Pin(0), sda=Pin(1), freq=400000)
>>> [hex(x) for x in i2c.scan()]
['0x44', '0x70']
>>>
```

Now that this section is working it is time to move on and wake up the sensor to start reading. For this step we use the *i2c.writeto* function. This function send a byte array over the I2C connection to a device. The Byte message that needs to be sent here is found on table 9. The Byte array consist of 0x2C (MSB) followed by 06 (LSB). The function that writes the buffer consists of the function *i2c.writeto* and then in the brackets we have the device address we retrieved from the above block of code followed by the byte array to send and a closing bracket.


```
>>> i2c.writeto(0x44, b'\x2C\x06')
2
```

When this line is run, you can see that it returns a 2, why 2, I have no idea!

The next step is to fetch the data from the SHT30 using the `i2c.readfrom_into` function. The following code specifies that the call must return six bytes and store it in the variable `result`:

```
result = bytearray(6)
i2c.readfrom_into(0x44, result)
result
```

If this section is run, the response in REPL will be as follows:

```
>>> result = bytearray(6)
>>> i2c.readfrom_into(0x44, result)
>>> result
bytearray(b'h\xdf\xc9i\xc7\xc7')
```

This is a raw output containing both the temperature and the humidity. To retrieve the temperature we need another block of code:

```
def temp_c(data):
    value = (((data[0] << 8 | data[1]) * 175) / 0xFFFF) - 45 ;
    temp = value
    return temp
```

Which preforms the maths when we call the `temp_c(result)` function:

```
>>> temp_c(result)
26.69032
>>>
```

For another example of reading and writing low level I2C devices without libraries, you can check out the later chapter on using the official M5Stamp add ons which don't have existing libraries written for them as of the time of writing this book.

Converting the code into a Module/Library.

In order to convert the code into a module/library, some changes need to be made.

Giving up and using someone else's library.

If the above is too confusing and you want to use a pre-made library, Robert Hammelrath from the Micropython Forums built the code to work with the ESP32's (apparently the old code was for the ESP8266). You can find the modified library module on his Github page here: <https://github.com/robert-hh/SHT30>

Download the file and save it to the M5Stamp C3 as sht30py and then you communicate with the SHT30 with the following:

```
from machine import I2C, Pin
import sht30

i2c=I2C(0, sda=Pin(4), scl=Pin(5))
sht=sht30.SHT30(i2c=i2c, i2c_address=68)

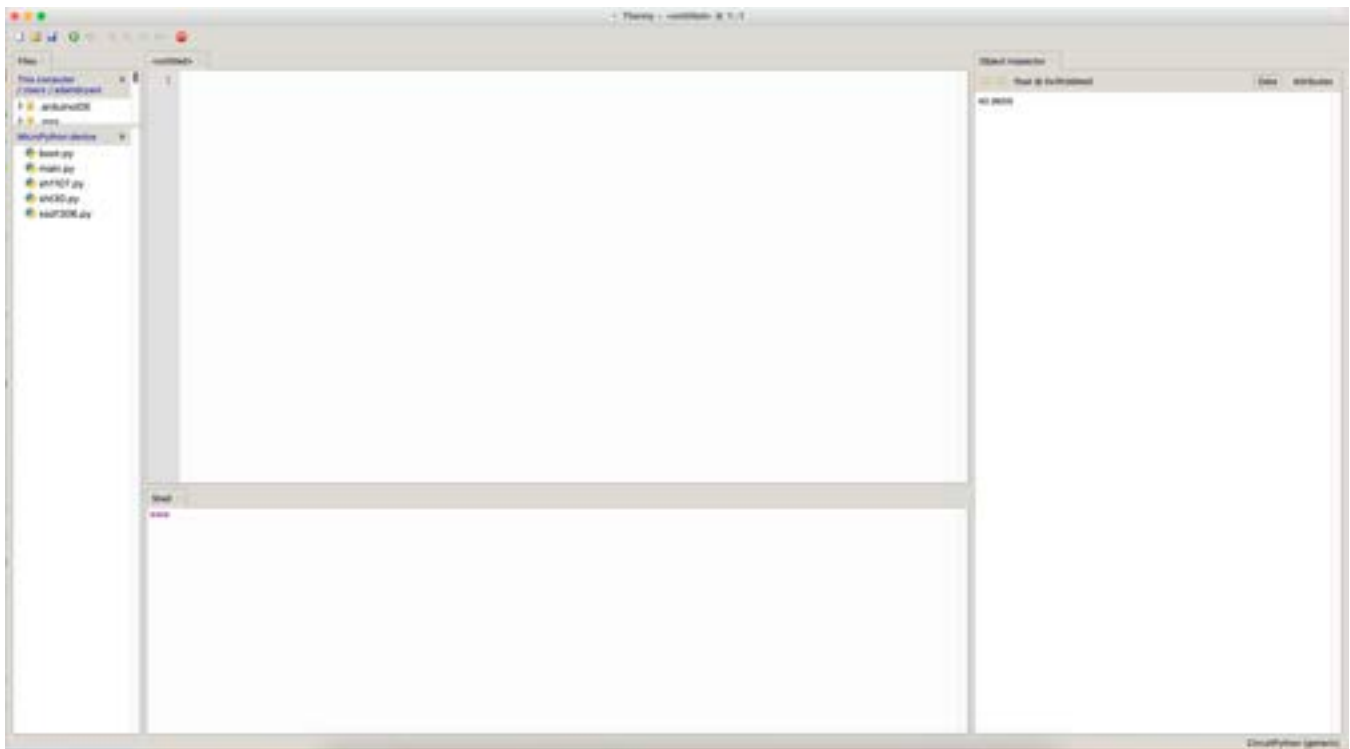
sht.measure()
```

I2C Output with OLED.

In the previous chapter I should you the basics of I2C communication by accessing the ENVIII unit with a pre-made driver in order to better understand the way the I2C function work. In this chapter I will access and control the M5Stack OLED unit with a pre-made library module. The OLED display is a SH1107 with an address of 0x3C and so we need a SH1107 display driver installed onto the M5Stamp C3 in order to communicate.

The Driver I used here is by Nemart69 on GitHub <https://github.com/nemart69/sh1107-micropython>

Click on the sh1107.py file, copy the contents into a new file on Thonny and save it to the M5Stamp C3's memory as sh1107.py (all in lower case).



In the screen shot above you can see that I have a few libraries/modules installed now. The two most important are the sh1107 and the ssd1306. The sh1107 is for the M5Stack OLED while the ssd1306 is for those small cheap OLEDs.

I2S Class.

By default I2S support in micropython is not available. In order to use I2S we need to Install the I2S module and class. **Unfortunately as of writing, I2S is only in a technical review on Micropython 1.19 and not included in mainstream precompiled firmwares but, it is available on the preinstalled M5Stack firmware on the M5Stamp Pico D4.**

M5Stacks I2S class contains the following functions:

- read
- start
- stop
- write
- __bases__
- __dict__
- CHANNEL_ALL_LEFT
- CHANNEL_ALL_RIGHT
- CHANNEL_ONLY_LEFT
- CHANNEL_ONLY_RIGHT
- CHANNEL_RIGHT_LEFT
- DAC_BOTH_EN
- DAC_DISABLE
- DAC_LEFT_EN
- DAC_RIGHT_EN
- FORMAT_I2S
- FORMAT_I2S_LSB
- FORMAT_I2S_MSB
- FORMAT_PCM
- FORMAT_PCM_LONG
- FORMAT_PCM_SHORT
- I2S_NUM_0
- I2S_NUM_1
- MODE_ADC_BUILT_IN
- MODE_DAC_BUILT_IN
- MODE_MASTER
- MODE_PDM
- MODE_RX
- MODE_SLAVE
- MODE_TX
- adc_enable
- bits
- deinit

- init
- nchannels
- sample_rate
- set_adc_pin
- set_dac_mode
- set_pin
- volume

In order to demo strait the I2S Class I will be using the PDM Microphone unit from M5Stack: <https://shop.m5stack.com/products/pdm-microphone-unit-spm1423?ref=pfpqkvphmgr>



M5Stack PDM MIMMS I2S Microphone Unit.

SPI Interfacing

Example SPI 01 GC9A01 (Round Display)

Round displays are more commonly available than they used to be and are now available to the hobbyist. The most common of these displays is those powered by the GC9A01 controller. In this example I will show you how to get started with the GC9A01 display from WaveShare and the M5Stamp C3. This section is designed as an introduction to the basics and the New M5Dial uses the GC9A01 display connected to the M5StampS3.

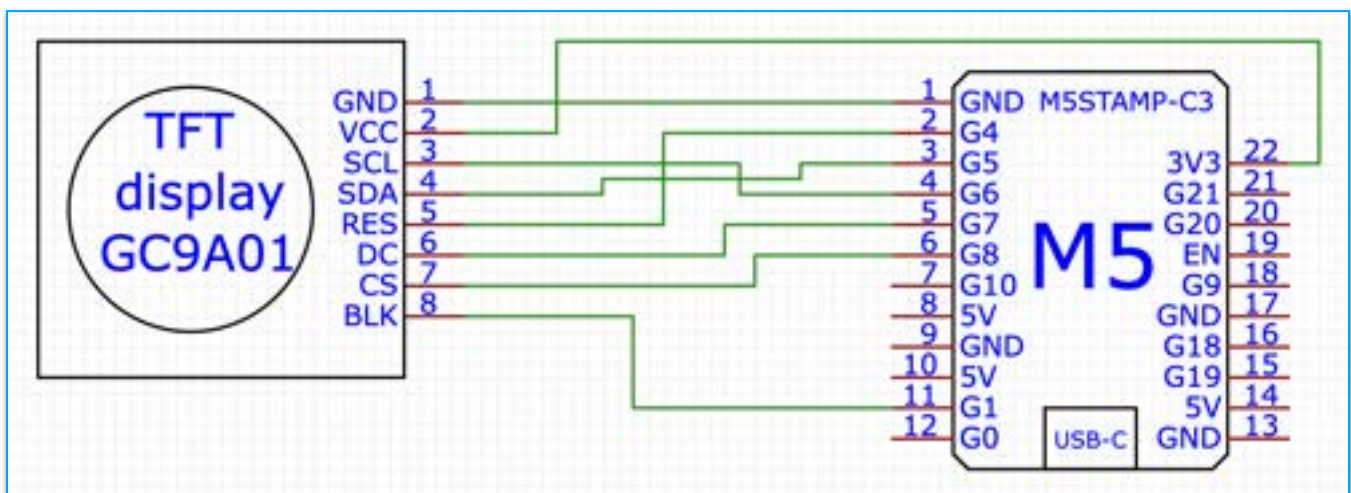
To control the GC9A01 display we first need a driver module. The driver module I am using is written by Russ Hughes and can be found on the Github page here: <https://github.com/russhughes/gc9a01py>

After downloading a copy of the GitHub files, use Thonny to copy the file gc9a01py.py and the fonts folder to the memory of the M5Stamp C3.

Next create a new file and type in the following Imports:

```
from machine import Pin, SPI,
import time
import gc9a01 as gc9a01
from fonts import vga2_bold_16x16 as font
```

The next step is to configure the SPI interface and the pins used for controlling the display. I have connected the GC9A01 display to the M5StampC3 as shown in the schematic below.



The SPI and Pins are configured with the following lines:

```
spi = SPI(1, baudrate=60000000, sck=Pin(6), mosi=Pin(5))
tft = gc9a01.GC9A01(
    spi,
    dc=Pin(7, Pin.OUT),
    cs=Pin(8, Pin.OUT),
    reset=Pin(4, Pin.OUT),
    backlight=Pin(1, Pin.OUT),
    rotation=0)
```

Once the Setup is complete, only a few lines are required to display text on the screen.

```
while True:
    tft.fill(0)
    tft.text(font, "Hello World", 50, 100)
    time.sleep(1)
```

The following is the complete listing for the M5StampC3.

```
import random
from machine import Pin, SPI,
import time
import gc9a01 as gc9a01
from fonts import vga2_bold_16x16 as font

spi = SPI(1, baudrate=60000000, sck=Pin(6), mosi=Pin(5))
tft = gc9a01.GC9A01(
    spi,
    dc=Pin(7, Pin.OUT),
    cs=Pin(8, Pin.OUT),
    reset=Pin(4, Pin.OUT),
    backlight=Pin(1, Pin.OUT),
    rotation=0)

while True:
    tft.fill(0)
    tft.text(font, "Hello World", 20, 20)
    time.sleep(1)
```


To create the same in UIFlow 2 for the M5StampS3 based controllers, The M5Dial must be selected as the controller and the following code can be used:

```
import os, sys, io
import M5
from M5 import *

label0 = None

def setup():
    global label0

    M5.begin()
    label0 = Widgets.Label("label", 50, 100, 1.0, 0xffffffff, 0x222222, Widgets.FONTS.DejaVu18)

    label0.setCursor(x=50, y=100)
    label0.setText(str('Hello World'))
    label0.setVisible(True)

def loop():
    global label0
    M5.update()

if __name__ == '__main__':
    try:
        setup()
        while True:
            loop()
    except (Exception, KeyboardInterrupt) as e:
        try:
            from utility import print_error_msg
            print_error_msg(e)
        except ImportError:
            print("please update to latest firmware")
```

In UIFLows Blockly view, we use the following blocks:



Module.

The Network module contains the classes and functions needed for connecting to networks. On the M5Stamp Pico D4 this function consists of wired lan and wireless lan whereas the M5Stamp C3 and M5Stamp C3U only have wireless lan. If we import the network module and use the dir() command, we see the following Classes and arguments.

```
>>> import network
>>> dir(network)
['__class__', '__init__', '__name__', 'AP_IF', 'AUTH_MAX', 'AUTH_OPEN', 'AUTH_WAPI_PSK',
'AUTH_WEP', 'AUTH_WPA2_ENTERPRISE', 'AUTH_WPA2_PSK', 'AUTH_WPA2_WPA3_PSK',
'AUTH_WPA3_PSK', 'AUTH_WPA_PSK', 'AUTH_WPA_WPA2_PSK', 'MODE_11B', 'MODE_11G',
'MODE_11N', 'PPP', 'STAT_ASSOC_FAIL', 'STAT_BEACON_TIMEOUT', 'STAT_CONNECTING',
'STAT_GOT_IP', 'STAT_HANDSHAKE_TIMEOUT', 'STAT_IDLE', 'STAT_NO_AP_FOUND',
'STAT_WRONG_PASSWORD', 'STA_IF', 'WLAN', 'phy_mode']
>>>
```

The network module contains the following classes and arguments:

- AP_IF - Creates an Access Point,
- AUTH_MAX,
- AUTH_OPEN,
- AUTH_WAPI_PSK,
- AUTH_WEP,
- AUTH_WPA2_ENTERPRISE,
- AUTH_WPA2_PSK,
- AUTH_WPA2_WPA3_PSK,
- AUTH_WPA3_PSK,
- AUTH_WPA_PSK,
- AUTH_WPA_WPA2_PSK,
- MODE_11B,
- MODE_11G,
- MODE_11N,
- PPP,
- STAT_ASSOC_FAIL,
- STAT_BEACON_TIMEOUT,
- STAT_IDLE,
- STAT_NO_AP_FOUND,
- STAT_WRONG_PASSWORD,
- STA_IF Creates a Station Point,
- WLAN - The wireless lan class,
- phy_mode'

WLAN Class.

In order to create a wireless network connection, we first have to configure the M5Stamp as either an access point or a station. To do that we use the following code:

```
import network  
wlan = network.WLAN(network.STA_IF)
```

For station mode or:

```
import network  
wlan = network.WLAN(network.AP_IF)
```

For access point mode.

To make the networks active, we use:

```
Wlan.active(True)
```

M5Stamps and UIFlow2

Display images on the M5Dial/GC9A01 round Display

In an earlier chapter I showed you how to run the GC9A01 round display on the M5StampC3. The M5Dial is a UIFlow2 compatible M5StampS3 which uses the same GC9A01 round display.

In order to display images on the display we have to create these image to a specific format and size before they can be uploaded to M5Stamps limited memory.



M5Stamp Accessories.

As well as the Family of M5Stamp controllers, M5Stack has produce a range of M5Stamp accessories that are designed to be used in the same way as the M5Stamps. In the following sections I will introduce you to the current accessories and show you how to use them with M5Stamp controllers.

ESP32 Downloader

M5StampS3 Breakout Board



M5Stamp CatM Module

Stamp ISP



The Stamp ISP is a stamp factor USB to UART adapter built around the CH9102 and can be soldered directly to a pcb or used with jumpers to link to other controllers.

StampTimerPower

StampTimerPower is a low-power control module with built-in Real Time Clock, wake-up in a STAMP series package, with manual wake-up+manual sleep+RTC timer wake-up+battery charging+5V boost output+3V3 power output and other functions. The RTC is built around the BM8563, The overflow time can be set to wake up the module in order to provide power. To start up an connected M5Stamp. The module is available in a STAMP package and can be used as a power supply part for low-power products by using SMD, DIP, and flying wires.

M5Stamp RS485

The Stamp RS485 is an RS485 adapter module built around the SP485EEN-L/TR which adapted the Normal RS485 12/24V bus used on commercial equipment to the 3v3 voltage bus that the M5Stamps use.

M5Stamp Extend I/O Module



The Stamp IO module is built around an STM32F030 and connects to the I2C port to provide an additional eight usable Input and output pins which can be used for analog and digital input and output, PWM servo control and RGB LED control. You may feel a bit of Déjà vu with the module as it is a miniaturised SMD version of the EXTI/O2 Unit:



ExtI/O2 Unit

I2C Communication part 2

Setting up and access the the StampIO module consist of the following Micropython code:

```
>>> from umachine import Pin, SoftI2C
>>> i2c=SoftI2C(scl=Pin(0), sda=Pin(1), freq=400000)
>>> i2c.scan()
[69]
>>>
```

Which you can see returns 69 (0x45) for the Stamp Io's i2c address letting us know that it is communication with the host controller. In order to configure the eight I/O channels for use we first have to configure them using the `i2c.writeto()` function There are dedicated registers for setting the channel mode and they are found at address' 0x00 to 0x07.

The values that need writing to these registers are as follows:

- DIGITAL_INPUT_MODE=0
- DIGITAL_OUTPUT_MODE=1
- ADC_INPUT_MODE=2
- SERVO_CTL_MODE=3
- RGB_LED_MODE=4

When the registers have been set to digital mouse you can use `i2.read()` to get data from the registers 0x20 to 0x27 which will return a 1 or a 0, Or `i2c.writeto()` to send a 1 or a 0 to set the output registers 0x10 to 0x17 high or low.

In analogue mode you can send 0 to 255 to the output registers 0x30 to 0x37 or read values of 0 to 4095 from input registers 0x40, 0x42, 0x44, 0x46, 0x48, 0x4A, 0x4C, 0x4E.

In servo mode you can control servos by sending values 0 to 180 to registers 0x50 to 0x57. In servo Pulse mode you send values of 500 to 2000ms to the register 0x60, 0x62, 0x64, 0x66, 0x68, 0x6A, 0x6C, 0x6E.

To control LEDS you need to send 3 byte values representing the color. Values to the registers 0x70, 0x73, 0x76, 0x79, 0x7C, 0x7F, 0x82, 0x85.

In order to set the pins modes you need to use:

i2c.writeto_mem(0x45, 0x00, b'\x00') for Pin 0

i2c.writeto_mem(0x45, 0x01, b'\x00') for Pin 1

And so forth up to

i2c.writeto_mem(0x45, 0x07, b'\x00') for Pin 7

The important part of the code is in the brackets. 0x45 is the ext I/O's I2C address returned from the I2C scan but the following 0x00 to 0x07 refer to the pin mode registers. The mode is set by adding the following:

b'\x00'

This is a byte array which holds the mode values. Which need to be translated as the following:

- *b'\x00'* for digital input mode,
- *b'\x01'* for digital output mode
- *b'\x02'* for ADC input mode,
- *b'\x03'* for servo control mode,
- *b'\x04'* for RGB LED control mode.

And so to set the pins into digital outputs you use:

i2c.writeto_mem(0x45, 0x00, b'\x01')

And then to set the pin high use

i2c.writeto_mem(0x45, 0x10, b'\x01')

And set the pin low with:

i2c.writeto_mem(0x45, 0x01, b'\x00')

M5Stack Devices fitted with the M5Stamp

Currently there are now devices that come with the StampS3 installed. These devices are the M5Dial, AirQ, The M5Stamp Capsule and the StampS3 Cardputer.

Advanced Chapter 1 -
Compiling fresh Micropython Firmware



Warning!

This chapter is very dangerous to both your physical and mental wellbeing.

Do not consider following this chapter if you are nervous or easily angered.

You have been warned!

Introduction

Compiling a fresh clean version of micropython is extremely painful and stressful if you have never tried this before due to the numerous packages and dependencies required. Another big issue with compiling from source is that tutorials often don't work because something in the package has been changed since the last time the tutorial was written. For me this issue turned out to be a file called `esp_adc_cal` and left me crying for help.

In order to compile Micropython firmware you first need to install CMake which just consist of downloading the installer from <https://cmake.org/> and then just running the installer to install Cmake. Next you need to install ESP-IDF which requires using the command line to download files from GitHub and then compiling them. Once these two are installed, you need to use the command line again to download and install the Micropython source.

Installing CMake.

As mentioned earlier, Installing Make consists of visiting make.org, going to the download page, downloading the make package for your operation system and installing it. In OSX this just requires copying the CMake.app package to OSX's apps folder. There is another step in OSX that need to be performed that is hard to discover, the patch to the make app needs to be added to a file but where is this file? The profile file is found in the computer users folder in OSX. To get to it you open a command line shell which open in the appropriate location.



Then you type in:

nano ~/.zprofile

And the terminal will display the following



```
adambyrant - nano ~/.zprofile - 80x24
UW PIC0 5.09 File: /Users/adambryant/.zprofile

# Setting PATH for Python 3.10
# The original version is saved in .zprofile.py save
PATH="/Library/Frameworks/Python.framework/Versions/3.10/bin:${PATH}"
export PATH

#setting path for cmake for espressif
export PATH=$PATH:/Applications/CMake.app/Contents/bin/

^G Get Help ^O WriteOut ^R Read File ^Y Prev Pg ^K Cut Text ^C Cur Pos
^X Exit ^J Justify ^W Where is ^V Next Pg ^L UnCut Text ^T To Spell
```

If it doesn't show the same as the screen shoot above, you need to paste the following lines in and then close and save by pressing CTRL+X.

```
#setting path for cmake for espressif
export PATH=$PATH:/Applications/CMake.app/Contents/bin/
```

Once this is done you can move on to installing ESP-IDF.

Installing ESP-IDF.

Reopen the command line shell (hence forth referred to as ZSH) and type in the following to create a dedicated folder for development using esp-idf:

```
mkdir -p ~/esp
```

Enter the folder by typing:

```
cd ~/esp
```

And type:

```
git clone --recursive https://github.com/espressif/esp-idf.git
```

To download esp-idf to the esp folder. This process make take a while as it will download lots of files and folders but, when its done to need to enter the folder by typing:

```
cd ~/esp/esp-idf
```

And then typing:

```
./install.sh esp32s3, esp32, esp32c3
```

This installs and configures esp-idf for the ESP32, ESP32C3 and ESP32S3 used in the stamps. If you only install one of the variants, you can re-run this command to add the other variants. This command only needs running once to configure the esp-idf version but, every time you want to use esp-idf you need to run:

```
. $HOME/esp/esp-idf/export.sh
```

Which creates the path commands needed for esp-idf to find certain files and folders. Once this has finished you next need to close the terminal and open a new terminal for the path change to be read. NOTE: if you restart the computer, the path will be lost and you will need to re-run the previous command. If you intend to do a lot of esp32 dev then you can add the following to the terminal's profile file and it will be permanently set.

```
alias get_idf='. $HOME/esp/esp-idf/export.sh'
```

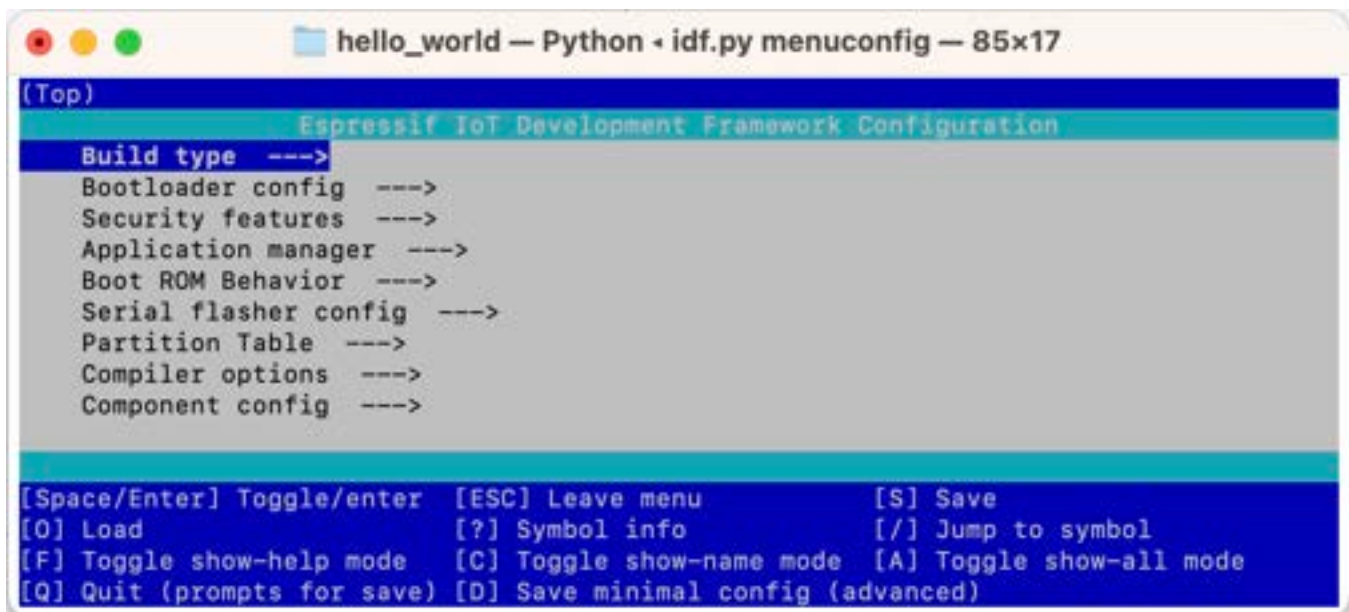
In order to test that everything is working so far we need to copy the Hello_world project to the esp directory using:

```
cp -r $IDF_PATH/examples/get-started/hello_world .
```

Next enter the folder, set the target to the esp version (in my case the esp32c3) and run idf.py's menuconfig:

```
cd ~/esp/hello_world  
idf.py set-target esp32s3  
idf.py menuconfig
```

If everything worked out then you should see the following menu on the screen:



There are lots of options available for configuring projects but, for this basic hello world example, we don't need to alter anything and so can just close the menuconfig by pressing ESC.

To build the project you need to run:

idf.py build

Which will show you the progress of the build process before giving you the command to flash the hello world files to the device.

```

Bootloader binary size 0x4fff0 bytes, 0x3010 bytes (38%) free.
[100%] Built target bootloader_check_size
[100%] Built target app
[ 98%] No install step for 'bootloader'
[ 98%] Completed 'bootloader'
[100%] Built target bootloader
[100%] Built target __ldgen_output_sections.ld
[100%] Linking CXX executable hello_world.elf
[100%] Built target hello_world.elf
[100%] Generating binary image from built executable
esptool.py v4.5
Creating esp32c3 image...
Merged 1 ELF section
Successfully created esp32c3 image.
Generated /Users/adambryant/esp/esp-idf/hello_world/build/hello_world.bin
[100%] Built target gen_project_binary
hello_world.bin binary size 0x28f80 bytes, Smallest app partition is 0x100000 bytes, 0xd7080 bytes (84%) free.
[100%] Built target app_check_size
[100%] Built target app

Project build complete. To flash, run this command:
/Users/adambryant/.espressif/python_env/idf5.1_py3.10_env/bin/python ../components/esptool_py/esptool/esptool.py -p {PORT} -b 460800 --before d
efault_reset --after hard_reset --chip esp32c3 write_flash --flash_mode dio --flash_size 2MB --flash_freq 80m 0x0 build/bootloader/bootloader.
bin 0x8000 build/partition_table/partition-table.bin 0x10000 build/hello_world.bin
or run 'idf.py -p {PORT} flash'

```

If the compile completed without errors then you can move on to flashing hello world to the M5Stamp C3/C3U or S3 using the following command:

idf.py -p PORT flash

The word “PORT” has to be replaced with the actual devices usb port which on windows will be COM followed by a number but on *nix and OSX it will look something like this:

```
/dev/cu.usbserial-54F70157991
```

So in my case it looks like the following:

```
idf.py -p /dev/cu.usbserial-54F70157991 flash
```

You will see lots of text flash up the screen as the code gets written to the M5StampC3 and if everything worked fine you will see the following at the end of the process.

```
Flash will be erased from 0x00000000 to 0x00004fff...
Flash will be erased from 0x00010000 to 0x0003ffff...
Flash will be erased from 0x00080000 to 0x00083fff...
Compressed 20464 bytes to 12601...
Writing at 0x00000000... (100 %)
Wrote 20464 bytes (12601 compressed) at 0x00000000 in 0.6 seconds (effective 257.4 kbit/s)...
Hash of data verified.
Compressed 167808 bytes to 88885...
Writing at 0x00010000... (16 %)
Writing at 0x0001a484... (33 %)
Writing at 0x000207e0... (50 %)
Writing at 0x00028340... (66 %)
Writing at 0x0002ec6c... (83 %)
Writing at 0x00035d9e... (100 %)
Wrote 167808 bytes (88885 compressed) at 0x00010000 in 3.1 seconds (effective 430.0 kbit/s)...
Hash of data verified.
Compressed 3872 bytes to 183...
Writing at 0x00080000... (100 %)
Wrote 3872 bytes (183 compressed) at 0x00080000 in 0.1 seconds (effective 418.1 kbit/s)...
Hash of data verified.
Leaving...
Hard resetting via RTS pin...
[100%] Built target flash
Done
adam@bryant@Adams-iMac hello_world %
```

In order to see what the example does you need to open a monitor with the following command:

```
idf.py -p /dev/cu.usbserial-54F70157991 monitor
```

And if things works the shell will show the following with a countdown before resetting and starting again.

```
I (311) cpu_start: Max chip rev: v0.99
I (316) cpu_start: Chip rev: v0.3
I (320) heap_init: Initializing. RAM available for dynamic allocation:
I (328) heap_init: At 3fcd7218 len 0004ffff (319 KiB): DRAM
I (334) heap_init: At 3fcdc718 len 00029500 (10 KiB): STACK/DRAM
I (341) heap_init: At 50000020 len 00001f00 (7 KiB): RTCRAM
I (348) spi_flash: detected chip: generic
I (351) spi_flash: flash id: d10
W (355) spi_flash: Detected size(4496KiB) larger than the size in the binary image header(2048KiB). Using the size in the binary image header.
I (369) sleep: Configure to isolate all GPIO pins in sleep state
I (376) sleep: Enable automatic switching of GPIO sleep configuration
I (383) app_start: Starting scheduler on CPU0
I (387) main_task: Started on CPU0
I (387) main_task: Calling app_main()
Hello world!
This is esp32c3 chip with 1 CPU core(s), WiFi/BLE, silicon revision v0.3, 2MB external flash
Minimum free heap size: 338024 bytes
Restarting in 10 seconds...
Restarting in 9 seconds...
Restarting in 8 seconds...
Restarting in 7 seconds...
Restarting in 6 seconds...
Restarting in 5 seconds...
Restarting in 4 seconds...
Restarting in 3 seconds...
```


Copying Micropython and compiling into firmware.

git clone <https://github.com/micropython/micropython>

Again this will download lots more files and folders creating a separate Micropython directory.

Enter the directory and run the following to update the files and folders:

git submodule update --init --recursive

Once complete find the mpy-cross folder, enter it and type make to compile the mpycross compiler.