MicroPython Workshop Documentation

Release 1.0

Matt Trentini

Jan 17, 2020

Contents

Contents:

CHAPTER 1

Introduction

1.1 MicroPython

MicroPython is a lean and efficient implementation of the Python 3 programming language that includes a small subset of the Python standard library 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.

1.2 Workshop

This workshop is intended to get you started experimenting with hardware *as quickly as possible*! It's our experience that one of the best ways to get started with MicroPython is to simply *use it on a device*.

It's based around using *Shields*. Shields allow peripherals to be easily connected to the microcontroller to extend functionality in interesting ways such as by illuminating LEDs, connecting sensors and buttons for input, or outputting to displays.

The shields are built for a developer board called the *Wemos D1 Mini*, originally built around an ESP8266 microcontroller. These boards became very popular and so shields for them are readily available and inexpensive. Instead of the original ESP8266 board, a much more powerful microcontroller - an ESP32 - was chosen for this workshop. The development board is known as the *TTGO Mini 32* board and it matches the pin layout of the Wemos D1 Mini so all the existing shields can be used.

Shields are *perfect* for a starter workshop since there's no need to solder components together or even use a breadboard. Just plug the shield in to the microcontroller board and get going! Many shields are available inexpensively online.

In general, each page that describes a shield will walk through the basics of how to interact with it and then propose exercises to work through. Generally the exercises increase in difficulty.

Let's get to it!

CHAPTER 2

Setup

2.1 Prerequisites

To participate in the workshop, you will need the following:

- A laptop with Linux, Mac OS or Windows and at least one free USB port.
  - A USB Type A to Micro B cable is provided as part of the workshop kit, however if your laptop only has USB-C ports then bring along either a USB-C to Micro B cable or a USB-C to USB Type A socket dongle.
- If it's Windows or Mac OS, make sure to install drivers for the CH340 USB to Serial chip. MacOS El Capitan may require disabling "kext signing" to install it.
- If your OS is Linux-based then, depending on the distribution, you may need to configure a *user* to have *permission* to access the serial port. This is usually performed by adding a user to a *group* that controls access to the serial ports; on many distributions this is the *dialout* group:

```
sudo adduser [user] dialout
```

  Replace `[user]` with the name of the user that should have access to the serial port. It's typically necessary to log out and then back in again.

- Mu installed on your laptop. This will be used for writing code, transferring code to the device, and even running an interactive terminal directly on the microcontroller.
  - You will need to install the alpha of the next version of Mu (found in the box at the top of the download page) in order to work with the microcontroller we're using in the workshop, so make sure you grab this one! Unfortunately they don't provide prebuilt binaries of this for Linux distros, so if that's your weapon of choice you will have to build from source - condensed set of instructions:

```
git clone https://github.com/mu-editor/mu.git
cd mu
python -m venv env
source env/bin/activate
pip install -e ".[dev]"
python run.py
```

In addition, at the workshop, you will receive:

- "TTGO MINI 32" development board (with an ESP32 at the heart of it)
- RGB LED Shield

Other Shields will be available for use during the workshop (but at lower numbers, sharing is caring!).

The firmware that is flashed on the boards is also available at https://micropython.org/download#esp32

2.2 Development Board

The board we are using is called a "TTGO MINI 32" which has an ESP32 module on it, which we will be programming. It comes with the latest version of MicroPython already setup on it, together with all the drivers we are going to use.

Note: The numbers printed next to the pins on the bottom of the board are different from what we're going to be using - this is because the shields we're using have a different pin numbering scheme (which can be seen printed on the shields next to the pins). We're going to use a module to map these, so we can just use the pin names the shields use.

On top it has a micro-USB socket, for connecting to the computer. On the side is a reset button. Then on each side of the board are two rows of pins - the inside row of which we will be connecting the shields to.

There are many symbols next to the pins on the underside of the board. The numbers are pin numbers we can use to control those particular pins. Some of the other important symbols are as follows:

- `3V3` - this is a fancy way to write 3.3V, which is the voltage that the board runs on internally. You can think about this pin like the plus side of a battery. There are several pins like this, they are all connected inside.
- `GND` - this is the ground. Think about it like the minus side of the battery. They are also all connected.
- `5V` - this pin is connected with the 5V from your computer. You can also use it to power your board with a battery when it's not connected to the computer. The voltage applied here will be internally converted to the 3.3V that the board needs.
- `RXD` / `TXD` - these are connected to the UART port used for device communications. This UART port is the one also used by the USB to ommunicate with the device from your PC, so don't connect anything to these pins or your USB communications might have problems!
- `RST` - this is a reset pin (connected to the corresponding RESET button).

2.3 Connecting

The board you got should already have MicroPython with all the needed libraries flashed on it - so let's get started, and open up Mu (which hopefully you already have installed from the Prerequisites section, if not, get it now!). The first thing that should appear is a window asking what type of code or device we're using - select the ESP MicroPython option. If you can't find this option, you may not have the alpha version necessary! Make sure the top bar of the window shows the version as Mu 1.1.0.alpha (or later). If you've selected another option (or used Mu for something else previously) then press the Mode button to bring the selection menu up again.

Now plug your TTGO MINI 32 into your laptop via the Micro USB cable, and you should see a "Detected new ESP MicroPython device" message in the bottom left and corner of the Mu window (note: this could take a couple of minutes if it is the first time you're plugging the device in, especially if you're on Windows). Once you see the message, press the REPL button at the top of the Mu window - a terminal should appear at the bottom of the Mu window with a messsage about MicroPython.

If you instead get "Could not find an attached device." message box, review your connections and make sure you've got the driver installed before finally unplugging and replugging the device. Hopefully one of these things identify the issue at hand!

2.3.1 Hello world!

Once you have your terminal to your microcontroller, click in the terminal and press "enter" and you should see the MicroPython interactive terminal (or REPL) prompt, that looks like this:

```
>>>
```

It's traditional to start with a "Hello world!" program, so type this and press "enter":

```
print("Hello world!")
```

If you see "Hello world!" displayed in the next line, then congratulations! You got it working.

2.4 Running Scripts

The MicroPython REPL is very powerful for running specific commands, but for repeatedly running commands it can get pretty messy. Mu makes life easy in this regard, by providing the ability to write scripts directly in the editor, and then simply press a button to run the script on the device. If you instead wrote `print("Hello Mu!")` under the `# Write your code here :-)` message in the editor, then you can simply press the Run button to run the code on the device - you should see *Hello Mu!* appear in the terminal from your script running.

If a script is to be run whenever the device is powered however, it likely makes more sense to put the script into a file on the MicroPython internal file system. On startup, A MicroPython device will search for a file named `boot.py` and run it if it is found. Following this, the same will be done for `main.py`. Upon completion of both of these files (successfully or otherwise), the REPL will begin.

2.4.1 File Transfer

In order for the device to run your script on startup, or to enable importing of modules into the MicroPython workspace, you will need to put the appropriate files on the device.

In order to access the file browser in Mu, click the REPL button to close it. This enables the Files button - if you now press that you will see the files on the device, and the files in the Mu folder on your computer (likely empty). You can't edit files directly on the device, but if you drag a file from the device box to your computer box it will copy if from the device to your computer, and then you can right click on it and "Open in Mu" to edit it.

Note that you can either see the REPL *or* the File browser, not both at the same time - if the button for what you want is disabled, something is probably already open and taking up the real estate.

For an example of file browser utility, if you retrieve and open the `d1_mini.py` file that we're going to use during the workshop for shield interaction, you will see that there is no magic there, just mapping numbers to more human-comprehensible names.

We can use this process to go the other way - if you create a new file in Mu, add the line `print("MicroPython is pretty neat")` to it, save it as `main.py` and then drag it from your computer onto your device, then every time the device resets, it will now print your message on startup.

2.5 Official Documentation and Support

The official documentation for this port of MicroPython is available at http://docs.micropython.org/en/latest/esp32/quickref.html.

There is a also a forum on which you can ask questions and get help, located at http://forum.micropython.org/.

Finally, there is a MicroPython Slack channel that you can join at https://slack-micropython.herokuapp.com/, where people chat in real time.

CHAPTER 3

Getting Started

This section describes how to start utilising some of the MicroPython specific features of your board by itself, before we start adding more hardware into the mix in the coming pages.

3.1 LED Basics

The traditional first program for hobby electronics is a blinking light - so let's stick with tradition and try to build that!

The boards you have actually have an LED built-in, so we can use that. It actually has three in fact - a red "power" LED, a blue "battery charge" LED, and a green LED that we can control (it defaults to off so it may not be easy to find until you enable it!).

One side of the green LED is connected to the `GND` (0 volts) pin internally, and the other side is connected to `D1`. We should be able to make that LED shine with our program by making `D1` behave like the `3V3` (3.3 volt) pins. We need to "set the `D1` pin high", or in other words, make it connected to `3V3`. Let's try that:

```
from machine import Pin
import d1_mini

led = Pin(d1_mini.LED, Pin.OUT)
led.on()
```

The first line "imports" the "Pin" function from the "machine" module. In Python, to use any libraries, you first have to import them. The "machine" module contains most of the hardware-specific functions in Micropython.

The second line imports a helper "d1_mini" module that provides the pin mappings to easily interact with specific pins on the board. Note that that is the number one after the letter `d`, not a lower case L! Each of the digital pins (`Dx`) on the board can be found in this module, as well as some hardware-role-specific pins (such as those used for *I2C* or SPI communications). Note that this module is specifically for D1 Mini form factor boards, such as the Wemos D1 Mini and the TTGO MINI 32 - if you were to use a different board, you would likely need a different helper module!

Once we have the "Pin" function imported, we use it to create a pin object, with the first parameter telling it to use the `LED` value from our helper module, and the second parameter telling it to switch it into output mode (instead of the input mode it would default to otherwise). Once created, the pin is assigned to the variable we called "led".

Finally, we set the pin high, by calling the "on" method on the "led" variable. At this point the LED should start shining - exciting!

Now, how to make the LED stop shining? There are two ways. We could switch it back into "input" mode, where the pin is not connected to anything. Or we could turn the microcontroller pin "off". If we do that, both ends of the LED will be connected to `GND`, and the current won't flow. We do that with:

```
led.off()
```

3.2 LED Blinking

Now, how can we make the LED blink 10 times? We could of course type `led.on()` and `led.off()` ten times quickly, but that's a lot of work and we have computers to do that for us. We can repeat a command or a set of commands using the "for" loop:

```
for i in range(10):
    led.on()
    led.off()
```

What happened? Nothing interesting, the LED just shines like it did. That's because the program blinked that LED as fast as it could – so fast, that we didn't even see it. We need to make it wait a little before the blinks, and for that we are going to use the "time" module. First we need to import it:

```
import time
```

And then we will repeat our program, but with the waiting included:

```
for i in range(10):
    led.on()
    time.sleep(0.5)
    led.off()
    time.sleep(0.5)
```

Now the LED should turn on and off every half second!

3.3 Networking

One of the exciting features of the ESP32 microcontroller (the heart of the TTGO MINI 32) is built-in Wi-Fi. While Wi-Fi itself may be a complicated beast, luckily for us MicroPython makes it simple to use! First of all lets connect to the network:

```
import network
sta_if = network.WLAN(network.STA_IF)
sta_if.active(True)
sta_if.connect('<your SSID>', '<your password>')
```

You will have to replace `<your SSID>` and `<your password>` with the relevant login details for your network.

This creates a reference to the STAtion InterFace of the board (type of Wi-Fi connection that connects to an Access Point), enables it, and then attempts to connect to the defined network. The success of this can be checked with:

```
sta_if.isconnected()
```

Note that `isconnected()` will immediately return the current status of the network (whether it has connected or not) - if you want your code to pause until the network is connected, then it is common to have a `while` loop that will do nothing until the network connects. Safer implementations will include a timeout in the check in case of a missing network!

Once a connection is established, the details of this can be checked with:

```
sta_if.ifconfig()
```

Which provides the device IP, device netmask, default gateway, and DNS server.

More network control commands can be found in the MicroPython WLAN documentation. There is also an Access Point interface (`network.AP_IF`) which allows other devices to connect to your device. This can be very useful, but for the moment we're just going to focus on connecting to another network - as this allows us to reach out to harness the power of the internet!

3.4 This Jen, is The Internet

Now that we've got a network connection (and that network extends out to the World Wide Web), it's a relatively simple matter to make web requests, utilising the `urequests` library. This is a MicroPython implementation of the Python requests library. It's had some features gutted to make it more microcontroller friendly, but it is still powerful!

To test it out, lets retrieve a random activity from the Bored API:

```
import urequests
req = urequests.get('https://www.boredapi.com/api/activity/')
```

And with that, we should now have the response to our activity request request! The text of the response can be found in `req.text` – check it out!

This is a JSON API, and so we can see the text of our request result is a string encoded JSON response. Turning a JSON string into a Python `dict` is pretty easy in Python (and MicroPython), and even easier when dealing with a request, as we can simply call the `.json()` method on it:

```
>>> req_dict = req.json()
>>> print(req_dict['activity'])
'Make homemade ice cream'
```

As simply as that, we can now harness information from the internet, and the myriad of public APIs out there (like those on this list of public APIs I found). Not only that, by using query strings we can pass information to websites, either for storage or for a customised response. We also have access to PUT requests, not just GET requests. I won't go into that here, but be aware that it is a simple thing to do if you need to!

Now that we've got the basic functions of the board under control, lets get some more hardware involved!

CHAPTER 4

RGB LED Shield

Fig. 1: RGB LED Shield, with LED indices annotated

The RGB LED Shield is a basic shield, featuring 7 independently-controllable RGB (Red, Green, Blue) LEDs. With the combination of the red, green, and blue channels of these LEDs, they can each to be set to any colour on the spectrum.

There are many types of RGB LEDs out there - these ones happen to be NeoPixels, which conveniently are natively supported in MicroPython!

4.1 Light Those LEDs

In order to start working with the LEDs, we'll need to connect the shield to your TTGO board! But first...

> **Warning:** While it is *possible* to plug shields in to the device while it is powered (plugged in to your computer), it is **not recommended**! As such, please remember to unplug the USB from your board *prior* to connecting or disconnecting any shields, or else you risk damaging the shield or the board.

> **Warning:** The LEDs on this shield are incredibly bright at full brightness! Do not look directly at the LEDs if they're at a higher brightness, or you don't know how bright they are! The `boot.py` script on your board sets these to off on initialisation, but it is still better to avoid staring at the board when initially powered in case they are enabled on startup.
>
> When setting the values of these LEDs, take this into account and scale down - you're unlikely to need the whole 255 range, clamping it at a maximum of 50 is more than likely plenty.

Now we've got that out of the way, let's plug it together! It's important to pay attention to the orientation of the shield - the "LOLIN" label should be over the USB port of the main board. Then simply align the 8 pins on either side with the sockets on the main board and push them together!

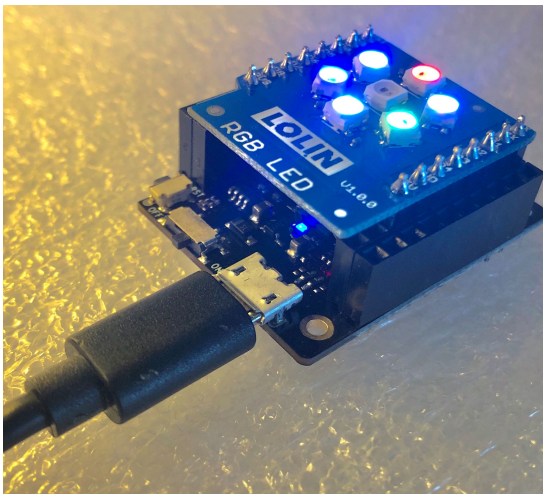


Fig. 2: This image was taken with maximum values for the LEDs capped at 25 - 10% of the actual maximum! Note the `LOLIN` text over the USB connection.

Now we've got the shield on the board, connect to the board with your USB again, and get into Mu. Now let's run the following commands to get these LEDs lit:

```
import machine
import d1_mini
import neopixel
np = neopixel.NeoPixel(machine.Pin(d1_mini.D4), 7)
np.fill((25,25,25))  # This just fills the memory of the np object
                     # The LEDs will not have changed colour yet
np.write()  # This is what actually changes the colour of the LEDs
```

Now all of your LEDs should be illuminated white! Now lets run down what we just did:

- Imported our `d1_mini` module – we need this to set the pin that is used for communicating with the LEDs
- Imported the `neopixel` module – this is the driver that knows how to talk to the NeoPixels
- Created an `np` object that represents our set of 7 NeoPixels. This constructor took two argument: a MicroPython Pin object that can be used for communicating with the LEDs, and the number of LEDs in our LEDs "strip"
- Filled our `np` objects buffer with a single colour. Note that this only took a single argument, which was an RGB tuple (but we set them all to the same value, which made white!). Also note that this didn't actually make the LEDs change, it just changed the buffer in the `np` object.
- Wrote out our `np` buffer values to the NeoPixels themselves – this is what made the lights light!

Okay that's great, but having 7 LEDs with the same colour isn't super useful (it might be if they were dim, but if that was the case we wouldn't have needed a warning earlier!). Luckily, modifying the LED object buffer is just as easy, by directly indexing and setting colours in the `np` object:

```
np[0] = ( 0, 0, 0)  # Sets the 0 index (centre) LED to black (off)
np[1] = (25, 0, 0)  # Sets the 1 index LED to red
np[3] = ( 0,25, 0)  # Sets the 3 index LED to green
np[5] = ( 0, 0,25)  # Sets the 5 index LED to blue
np.write()          # Writes the buffer out to the LEDs
```

Similarly to the `fill()` command, indexing just modifies the buffer - we still need to `write()` our changes out to the LEDs for them to change colour.

You should now have a centre LED that is off, and alternating colours around your LED circle - exciting! You now have the full extent of control over these LEDs to bend them to your will.

4.2 Exercises

Time to take those concepts and put them into action! The following subsections detail different exercises that can be accomplished using the techniques covered so far.

4.2.1 Spin Cycle

Make one LED at a time light up around the circle of LEDs to make a spinning animation!

Hint: You can use `time.sleep()` to add delays to control the speed of the spin!

Extension: Make the LED fade through the colours of the rainbow while it spins!

4.2.2 Digital Dice

Rapidly cycle through LED combinations representing the six sides of a (six sided) dice, before slowing down, and ultimately "landing" on one of the "sides".

Hint: MicroPython has a cut-down version of the `random` module built-in! On the ESP32 (the microcontroller on the TTGO 32 Mini) we have access to the Functions for integers from BigPython!

Extension: Add a signal to show when the face has stopped changing – maybe a colour change, or a sequence of flashing (or whatever else takes your fancy!).

CHAPTER 5

LED Matrix Shield

Fig. 1: LED Matrix Shield, with row and bit ordering annotated

The LED Matrix Shield is a D1 Mini form factor shield, featuring an 8x8 LED Matrix of red LEDs.

The shield has a TM1640 LED matrix driver on it that means we don't need to worry about how to make the matrix turn each LED on, we just send it a command with the LEDs we want enabled and it makes it happen!

Further to this, there's a MicroPython TM1640 driver that has already been developed, and so we can use this to easily control the matrix in MicroPython. This `tmp1640.py` file from this should already be loaded onto your TTGO board, so you should be set to get started!

5.1 Plugging in

> **Warning:** While it is *possible* to plug shields in to the device while it is powered (plugged in to your computer), it is **not recommended**! As such, please remember to unplug the USB from your board *prior* to connecting or disconnecting any shields, or else you risk damaging the shield or the board.

In order to start working with the LED matrix, we'll need to connect the shield to your TTGO board. If there is already a shield connected to your board (such as the RGB LED shield from the previous section), then first remove this. Then plug the LED Matrix shield into the board - the "LOLIN" label should be over the USB port of the main board. Then simply align the 8 pins on either side with the sockets on the main board and push them together!

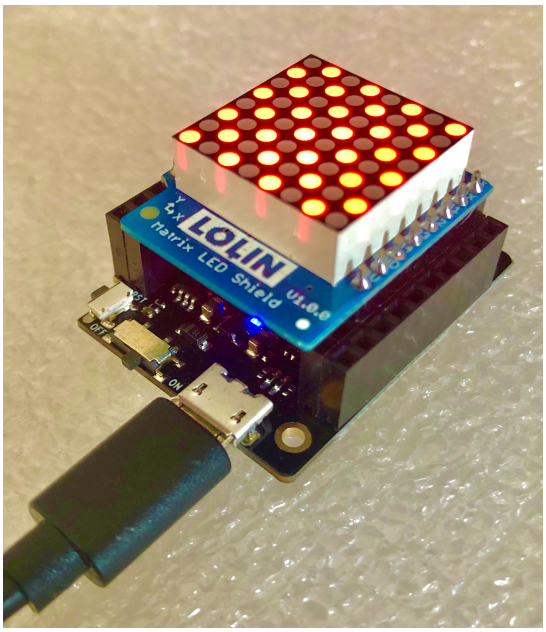


Fig. 2: Note the `LOLIN` text over the USB connection

5.2 Enter the Matrix

Now we've got the shield on the board, connect to the board with your USB again, and get into the REPL (by connecting to your device in your serial terminal software of choice). Now let's run the following commands to get these LEDs lit:

```
import machine
import d1_mini
import tm1640
tm = tm1640.TM1640(clk=machine.Pin(d1_mini.D5), dio=machine.Pin(d1_mini.D7))
tm.write([255,255,255,255,255,255,255,255])  # 255 = 0b11111111
```

Now all LEDs on the LED Matrix should be illuminated! Now lets run through what we just did:

- Imported the MicroPython `machine` module – we need this to configure our pins before passing them to our `tm1640` driver

- Imported our `d1_mini` module – we need this to get the pin information to then configure the correct pins for communicating with the LED Matrix driver chip
- Imported the `tm1640` module – this is the MicroPython driver that will provide us easy control over the `tm1640` LED matrix driver chip on the shield
- Created a `tm` object that represents our LED Matrix driver. This took two parameters, the two pins that are used for communicating with the `tm1640` chip (a clock pin and a data in/out pin)
- Wrote an 8-long list filled with 255's to our `tm` object – this is what turns the LED Matrix LEDs on!

The format of the list that we wrote to the `tm` object is that each element of the list represents a row, and each bit in that element represents one of the LEDs in that row, with 1 being illuminated and 0 being off. As `255 == 0b11111111`, and we wrote 255 to all 8 elements of the list, we illuminated all 8 LEDs, on all 8 rows.

Our driver library also gives us access to a `brightness()` method on our `tm` object, if we wanted to reduce the brightness of the LED matrix. This takes a single argument of an integer from 0 to 7, where 0 is the minimum brightness, and 7 is the maximum (this is the default). So if we wanted to reduce brightness by a bit but not all the way we could do:

```
tm.brightness(3)
```

If you wanted to (slightly) improve visualisation of what you are writing to the matrix in your code, you could format it like so:

```
tm.write([
    0b00000000,
    0b01100110,
    0b01100110,
    0b00000000,
    0b00000000,
    0b10000001,
    0b01111110,
    0b00000000,
])
```

5.3 Blue pill, red pill

Now that you know how to light up LEDs individually, it's time to learn about some convenience functions that can help display *text*.

Let's display a letter on the matrix:

```
tm1640.display_letter(tm, "X")
```

And, for the pièce de résistance:

```
tm1640.scroll_text(tm, "Scrolling for days...")
```

These are enabled by using a *FrameBuffer*, a module built-in to MicroPython that provides a general - and efficient! - way to draw onto an in-memory 'canvas'.

5.4 Advanced: FrameBuffer

A flexible way to control the LEDs in the matrix is by using a MicroPython frame buffer. This is done like so:

```
# Instantiate our 8x8 frame buffer
import framebuf  # Bring in the frame buffer library
buf = bytearray(8)  # Reserve 8 bytes of memory for the frame buffer
fb = framebuf.FrameBuffer(buf, 8, 8, framebuf.MONO_HMSB)

# Draw things into our frame buffer
fb.text('!', 0, 0, 1)  # Draw an !
fb.hline(3, 7, 2, 1)  # Supplement the bottom of the ! as the font is 7x7
fb.vline(0, 0, 8, 1)  # Draw line down left side
fb.vline(7, 0, 8, 1)  # Draw line down right side
fb.pixel(1, 0, 1)  # Extend end of lines
fb.pixel(1, 7, 1)
fb.pixel(6, 0, 1)
fb.pixel(6, 7, 1)

# Draw the buffer of the frame buffer to the "display"
tm.write_hmsb(buf)  # Note that this takes buf, not fb
```

By using this we have a powerful set of tools for drawing whatever we want to the matrix (including text) without knowing the specific set of bits corresponding to our image!

5.5 Exercises

Time to take those concepts and put them into action! The following subsections detail different exercises that can be accomplished using the techniques covered so far.

5.5.1 Exercise 1: Wake up, Neo

Implement a simple countdown timer.

Ask the user for a duration in seconds. Count down from that time, scrolling the number past until 0 is reached, then display an asterix and *invert* it every half second to indicate an alarm is occurring.

Extension: Also use the buzzer and button shields - beep with each passing second, buzz when 0 is reached and use the button to stop the alarm.

5.5.2 Exercise 2: Be still my beating heart

Display an image of a heart on the LED matrix.

Now, *animate* it, by displaying different sized hearts in rapid succession.

Extension: Use a buzzer shield (with a 2UP board) to beep in time with the heart.

5.5.3 Exercise 3: Lo-fi Charting

Render a *simple* chart.

Use the following data:

```
data = [100, 130, 160, 160, 250, 180, 150, 100]
```

Scale it appropriately (so the maximum data is display with the topmost LED).

Extension: Provide an option to fill all the LEDs below (more like a bar chart).

CHAPTER 6

Button Shield

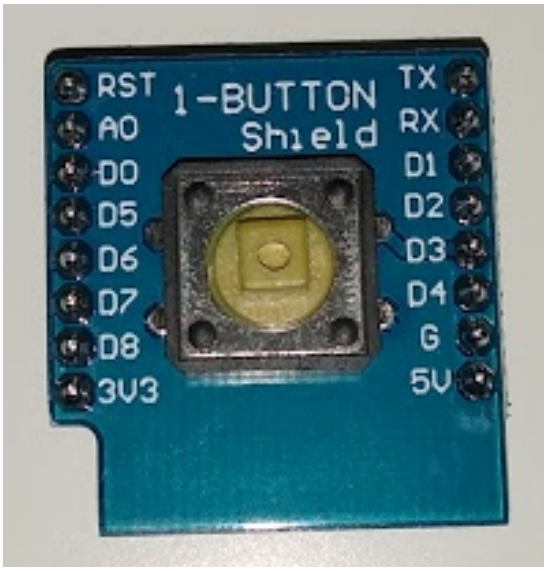


Fig. 1: Button Shield

The Button Shield is a D1 Mini form factor shield, featuring a button. That's it! The button is wired directly to the microcontroller, and can be used as a way of providing user input to our code.

6.1 Plugging in

Warning: While it is *possible* to plug shields in to the device while it is powered (plugged in to your computer), it is **not recommended**! As such, please remember to unplug the USB from your board *prior* to connecting or disconnecting any shields, or else you risk damaging the shield or the board.

In order to start working with the button, we'll need to connect the shield to your TTGO board. If there is already a shield connected to your board, then first remove this. Then plug the button shield into the inner row of pins on the TTGO board, aligning the top edge of the shield (that says "1-BUTTON Shield") with the antenna of the TTGO board. The shield should *not* hang over the USB port!

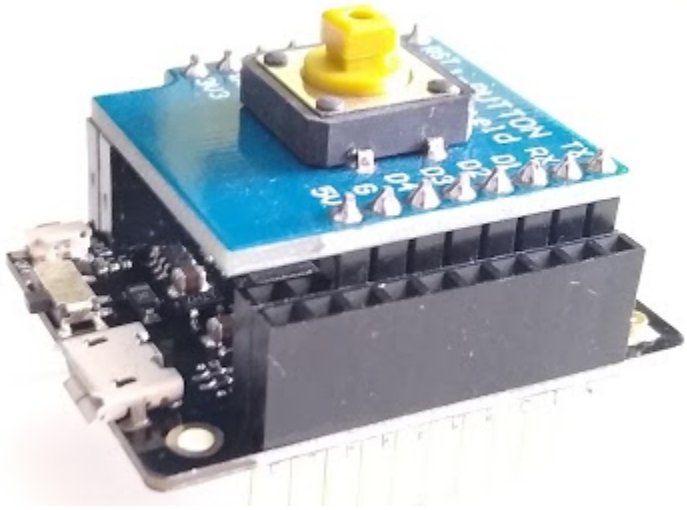


Fig. 2: Note the two empty pin sockets in the headers closer to the USB!

6.2 Human Machine Interface

Now we've got the shield on the board, let's run the following commands to test our button:

```
from machine import Pin
import d1_mini
button = Pin(d1_mini.D3, Pin.IN, Pin.PULL_UP)
button.value()
# Hold the button down and then run that line again
button.value()
```

Now all LEDs on the LED Matrix should be illuminated! Now lets run through what we just did:

- Imported the `Pin` module from the `machine` module – we need this to configure our pin to interface with the button
- Imported our `d1_mini` module – we need this to get the pin information to then configure the correct pin to connect to the button
- Created a `button` object that represents our button. This sets up our relevant pin as an input (`Pin.IN`), and enables a pull-up resistor on the pin (`Pin.PULL_UP`)
- Checked what the current value of the button is - this result should be a 1 when the button is left alone, and a 0 when the button is pressed

We now have the ability to alter the result of our code by pressing a button! That is pretty exciting, as with this simple input capability we can now make gadgets that do things on command.

The only piece of magic we used to do this was the pull-up resistor. This is used because one side of our button is connected to `GND` (0 volts), and the other side of our button is connected to the microcontroller pin. When we press the button, these two sides of the button are connected together, but otherwise they are not connected. This means that we know that when the button is pressed, the pin should return the value 0, as it is connected to `GND`. But what

about when the button is *not* pressed? By default, when the button isn't pressed it is "floating" - it is not connected to a voltage source from which it could get a voltage. As such, we can not determine what it's voltage might be. By enabling the pull-up resistor, then the pin is connected to 3.3 volts through a resistor. This means that when the button is not pressed, it is connected to 3.3 volts, and so reads as a 1. It is connected to the voltage through a resistor, meaning that it is a "weak" connection, which is then overridden by our direct connection to 0 volts when we press the button. SparkFun has a tutorial on this topic if this doesn't make any sense, or you would like to know more!

6.3 Make Something Happen

The most obvious use for a button, is to make something happen when you press the button! So lets do that:

```
while button.value():
    pass  # Ignore the button while it isn't pressed
print("The button has been pressed!")
```

Now we can make things happen on our device from the outside world!

6.4 Exercises

Time to take those concepts and put them into action! The following subsections detail different exercises that can improve our usage of the button.

6.4.1 Exercise 1: Make things happen again and again and again

Expand on the *Make Something Happen* code to make it so that the message is printed every time the button is pressed, not just the first time.

The message should only print once each time the button is pressed, no matter how long it is held.

Extension: Debounce the button. When a button is pressed or released, it opens and closes many times very quickly, which can cause undesirable behaviour (such as your code thinking it was pressed when you actually released it!). Expand your code to account for this bouncing, to correctly detect only real button presses.

6.4.2 Exercise 2: Bored Button

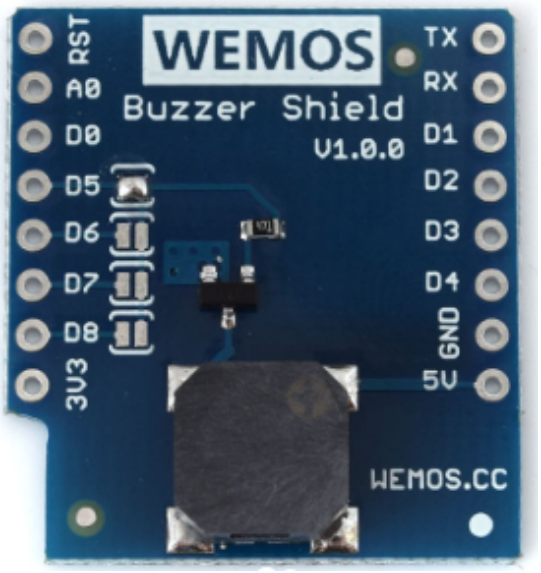
Hook the button up to the retrieval of an activity from the *Bored API* to make a button that the user can press when bored to get an idea of something they could do!

If the user has pressed the button too many times too quickly, re-list the recent options provided and suggest they give one a shot, or wait a certain period if they've earnestly considered them and decided they want a new option. You will need a way of tracking time, such as `time.ticks_ms()`.

Extension: Allow the user to hold the button down to override the wait time for the next activity if they hit the limit.

CHAPTER 7

Buzzer Shield



The Buzzer Shield is a D1 Mini form factor shield, featuring a buzzer which can be used for generating sounds.

The buzzer is controlled by simply toggling the appropriate pin at the specific frequency that we want to hear. We could do this by repeatedly setting our pin high, waiting for a period, and then setting it low again, however this would be very laborious. Instead, we're going to use a technique called Pulse-Width Modulation, (PWM). This is a feature of our hardware (so it's very fast and stable, instead of relying on a software implementation), and we can set it up and control it using the MicroPython PWM function.

7.1 Make Some Noise

Warning: While it is *possible* to plug shields in to the device while it is powered (plugged in to your computer), it is **not recommended**! As such, please remember to unplug the USB from your board *prior* to connecting or disconnecting any shields, or else you risk damaging the shield or the board.

In order to start working with the buzzer, we'll need to connect the shield to our TTGO board. If there is already a shield connected to your board (such as the LED Matrix shield from the previous section), then first remove this. Then plug the buzzer shield into the board – the large, black component (the buzzer!) should be over the USB port of the main board. Then simply align the 8 pins on either side with the sockets on the main board and push them together!

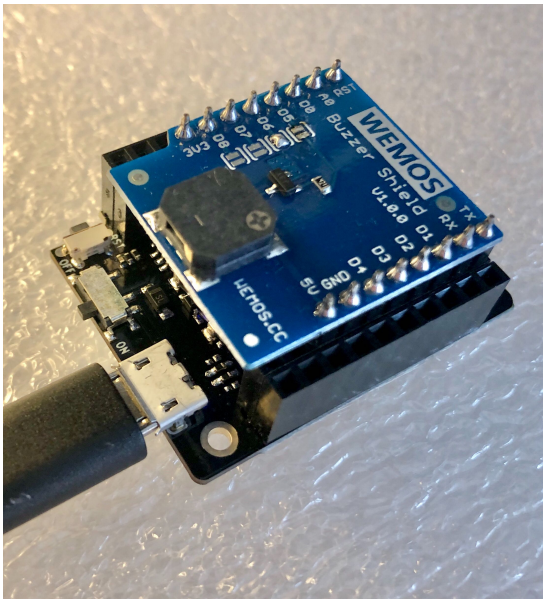


Fig. 1: Note the large black component over the USB connection

Now we've got the shield on the board, connect to the board with your USB again, and get into Mu. Now before we start using the buzzer...

Warning: When we use the buzzer, we tell it to produce a set frequency - it will keep doing this indefinitely until we tell it to stop (or you unplug it)! This may be loud / unintended – as such, always make sure you know how to disable the buzzer (will be in the code below), and where possible have your scripts copy + pasted in or running from a file (instead of typing the commands into the REPL) with a the buzzer disabling at the end of the script (after a delay) so that we're only getting sounds when we want them!

Now that we've got that out of the way, lets make some noise:

```
import machine
import d1_mini
import time
buzzer_pin = machine.Pin(d1_mini.D6, machine.Pin.OUT)
buzzer = machine.PWM(buzzer_pin)
buzzer.freq(1047)
buzzer.duty(50)
```

(continues on next page)

(continued from previous page)

```
time.sleep(1)
buzzer.duty(0)
buzzer.deinit()
```

If all went well, your buzzer should have made a (1 kHz) beep for 1 second, and then stopped! Now lets run through what we just did:

- Imported the MicroPython `machine` module – we need this to configure our pin to control the buzzer
- Imported our `d1_mini` module – we need this to get the pin information to then configure the correct pin to use for the buzzer
- Imported the MicroPython `time` module – we're going to use this to add delays to the code
- Created an object that controls pin connected to the buzzer, and set it as an output (as we will be *driving* the buzzer)
- Created a new object that gives us PWM control over the buzzer pin
- Set the PWM frequency to 1047 hertz – note that this does not generate any noise by itself, as the PWM defaults to having a duty cycle of 0 (that is, it is enabled 0% of each cycle)
- Set the PWM duty cycle to 50 – the duty cycle is a 10 bit number (has a maximum of 1023) so by setting it to 50, it will be enabled roughly 5% of the time. This is what makes it buzz!
- Delays code execution by 1 second – this gives us an opportunity to hear it buzzing
- Sets the PWM duty cycle to 0 – stops the buzzer producing any noise (as it is once again enabled 0% of the time)
- Deinitialises the buzzer – this should be done once the buzzer is not being used, and the buzzer reinitialised (via `buzzer = machine.PWM(buzzer_pin)`) if we want to use it again. **NOTE**: If we don't do this and then we change the `buzzer_pin` variable in any way (such as running the above code again, re-setting `buzzer_pin`), the buzzer will stop working and we will be required to unplug and re-plug the board to get it to work again!

A couple of the numbers in here may seem a tad arbitrary, so lets explain them a bit better.

We used a frequency of 1047 because this is a nice "C" note and the buzzer has a peak frequency response between 1 and 3 kHz – this is the area where it provides the best results. It will still work outside this range however!

We used a duty cycle of 50 to reduce the volume of the sound output. Peak volume is at 50% duty cycle (setting of ~512), however due to the logarithmic nature of sound, we only need a small amount of this to make a relatively loud buzz. Because the buzzer generates sounds by alternating the voltage over the diaphragm, moving closer to 100% duty cycle has the same effect as being close to 0% duty cycle. Feel free to try buzzing at 50% duty cycle to see why we reduced the output!

Being able to make *a* sound is nice, but it would be even nicer to make a *nice* sound. First of all, lets define the frequencies of some specific notes:

```
C6  = 1047
CS6 = 1109
D6  = 1175
DS6 = 1245
E6  = 1319
F6  = 1397
FS6 = 1480
G6  = 1568
GS6 = 1661
A6  = 1760
```

(continues on next page)

(continued from previous page)

```
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
```

These are taken from the Pyboard "Play Tone" page – you will see that there are more notes on that page. We're not defining the lower range as two octaves covering our peak frequency response will serve us fine.

Now lets create a function that will allow us to play a song by passing it a buzzer object, a list of notes, the delay between each note, and an optional duty cycle to use when playing a note:

```
def play(buz_pin, notes, delay, active_duty=50):
    buz = machine.PWM(buz_pin)
    for note in notes:
        if note == 0:  # Special case for silence
            buz.duty(0)
        else:
            buz.freq(note)
            buz.duty(active_duty)
        time.sleep(delay)
    buz.duty(0)
    buz.deinit()
```

To put it into action, lets create a song by defining a list of notes, and then `play()` it:

```
song = [
     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,
]
play(buzzer_pin, song, 0.15)
```

With any luck we should have heard a recognisable little tune! We've now set up a framework to allow us to play arbitrary songs – neat!

7.2 Exercises

Time to take those concepts and put them into action! The following subsections detail different exercises that can be accomplished using the techniques covered so far.

7.2.1 Alerts

Set up a `success()` function that you could easily put into a future project that utilises the buzzer to play a success notification (the audio equivalent of a green tick). What that sounds like is up to your imagination!

Extension: Make a `failure()` function for when things don't quite go as planned.

CHAPTER 8

OLED Shield

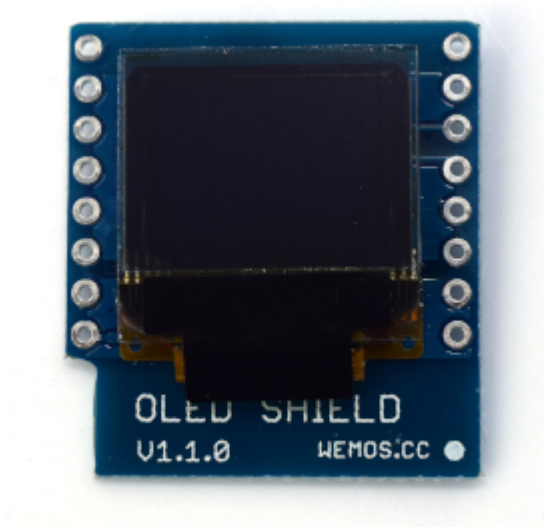


Fig. 1: OLED Shield

The OLED Shield is a D1 Mini form factor shield that hosts a *tiny* 17mm 64x48 monochrome OLED display.

Note: What is an *OLED*? It stand for Organic Light-Emitting Diode. It describes the technology used to turn on or off pixels - dots - on the display. The term is largely unimportant, just think of it like a tiny little TV where you can control every pixel.

Although small, this shield packs a punch! You can draw pixels, render lines and display text.

The *integrated circuit* that makes the magic happen is an SSD1306. It sits between the microcontroller and the raw display and allows us to send commands to make stuff appear. The SSD1306 is a common chip and so a driver for it is built-in to MicroPython.

Let's see how it works!

8.1 Plug me in

Warning: While it is *possible* to plug shields in to the device while it is powered (plugged in to your computer), it is **not recommended**! As such, please remember to unplug the USB from your board *prior* to connecting or disconnecting any shields, or else you risk damaging the shield or the board.

As with all the other shields, the first thing to do is plug the OLED shield into the microcontroller. Take care with orientation:

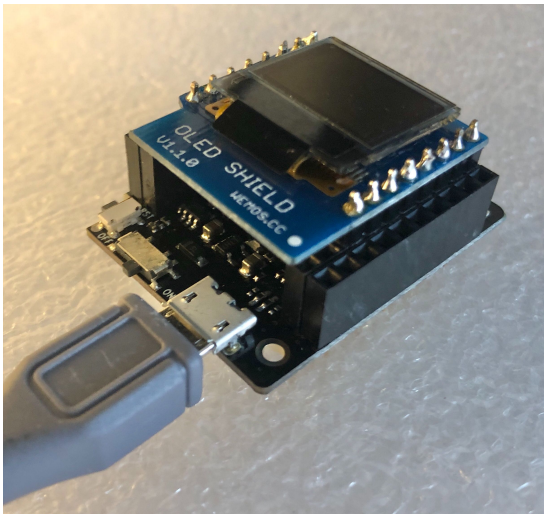


Fig. 2: OLED Shield, plugged in correctly

8.2 Techy details, I squared C?

The SSD1306 is controlled by sending *I2C data* at it. What's I2C? It's a communications protocol, but it's not critical to know more than that to use the OLED Shield. If you would like to know more, take a look at *I2C*.

8.3 Drawing

The SSD1306 driver provides a handful of functions to display pixels on the OLED. First though, the display needs to be initialised. Let's work through an example:

```
1 from machine import Pin, I2C
2 import d1_mini
3 import ssd1306
4
5 i2c = I2C(-1, scl=Pin(d1_mini.SCL), sda=Pin(d1_mini.SDA))
6
```

(continues on next page)

(continued from previous page)

```
width, height = 64, 48
oled = ssd1306.SSD1306_I2C(width, height, i2c)
```

Here we initialise `oled` so it's using microcontroller pins that are configured to use I2C. We also take care to set the width and height of our display - the SSD1306 can work with a number of differently sized displays but it's critical to configure it appropriately.

It's worth noting that the display's 'origin' - where `x` and `y` are both zero - is in the *top left*.:

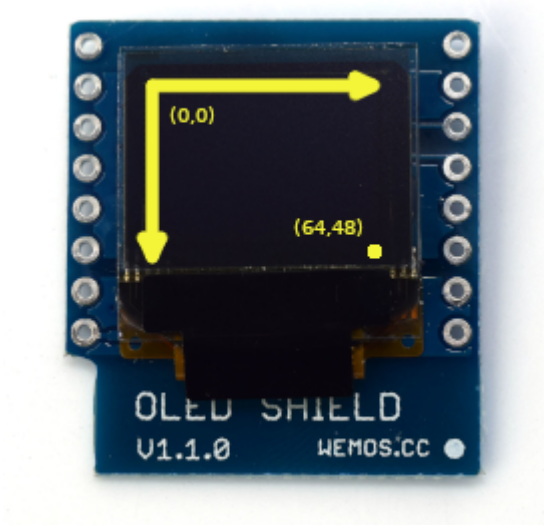


Fig. 3: OLED Shield, annotated with row and column numbering

Now we can draw things!:

```
# Text is easy!
oled.text('Hello,', 0, 0, 1)
oled.text('World!', 0, 10, 1)

# Draw a triangle and it's centroid
v1, v2, v3 = (2, 24), (2, 46), (60, 46)
oled.vline(v1[0], v1[1], v2[1] - v1[1], 1)
oled.hline(v2[0], v2[1], v3[0] - v2[0], 1)
oled.line(v1[0], v1[1], v3[0], v3[1], 1)
oled.pixel((v1[0] + v2[0] + v3[0]) // 3, (v1[1] + v2[1] + v3[1]) // 3, 1)

oled.show()
```

:

The drawing commands are defined in FrameBuffer which the SSD1306 driver uses internally. `text`, `pixel`, `hline`, `vline` and `line` are fairly clearly named - you can probably guess what they do! - but see the FrameBuffer docs if you'd like more details.

Note that the display is monochrome so there's only two *colour values* (the last parameter in the drawing methods) that make sense: 0 (black) or 1 (white).

Fig. 4: We can draw!

8.4 Exercises

8.4.1 Exercise 1: Spirals for days

Render a square-edged spiral using `hline` and `vline`:

8.4.2 Exercise 2: Animate the spiral

Render the same spiral using `pixel` but use `show` after each pixel is drawn so that the sprial appears to draw from the centre to the outside.

Bonus points: Make the animation *loop forever* by giving the spiral a *maximum length* so the 'oldest' pixel is erased when the spiral becomes too long. It should look like the old snake game!

8.4.3 Exercise 3: Bouncy, bouncy [Hard]

Render a pixel near the centre of the display. It's a bouncy ball! Give it a *velocity* and *direction* and render it moving about the screen, bouncing off the edges of the screen

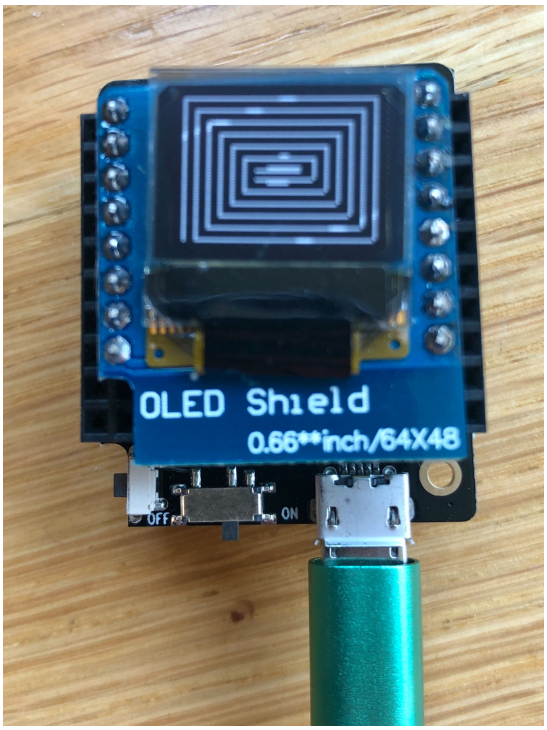


Fig. 5: Spiral

CHAPTER 9

PIR Shield

Fig. 1: PIR Shield

The PIR (Passive InfraRed) Shield is a D1 Mini form factor shield, featuring a PIR sensor. PIR sensors detect motion in a wide area around them, and are commonly used in security applications (for detecting movement in places nothing should be moving). It works very simply - it outputs 0 volts (logic low) when no movement is detected, and outputs 3.3 volts (logic high) when it detects movement. As such all we need to do is hook it up to an input on our microcontroller and we're good to go!

9.1 Plugging in

> **Warning:** While it is *possible* to plug shields in to the device while it is powered (plugged in to your computer), it is **not recommended**! As such, please remember to unplug the USB from your board *prior* to connecting or disconnecting any shields, or else you risk damaging the shield or the board.

In order to start working with the PIR sensor, we'll need to connect the shield to your TTGO board. If there is already a shield connected to your board, then first remove this. Then plug the button shield into the inner row of pins on the TTGO board, aligning the top edge of the shield (that says "LOLIN") with the antenna of the TTGO board. The shield should *not* hang over the USB port!

9.2 Going Through the Motions

Now we've got the shield on the board, let's run the following commands to test our PIR sensor:

```
from machine import Pin
import d1_mini
pir = Pin(d1_mini.D3, Pin.IN)
pir.value()
# Move your hand in front of the sensor and then run that line again
pir.value()
```

As simply as that, we can now detect movement! Now lets run through what we just did:

- Imported the `Pin` module from the `machine` module – we need this to configure our pin to interface with the PIR sensor
- Imported our `d1_mini` module – we need this to get the pin information to then configure the correct pin to connect to the PIR sensor
- Created a `pir` object that represents our PIR sensor. This sets up our relevant pin as an input (`Pin.IN`)
- Checked what the current value of the PIR sensor is - this result should be a 0 when the PIR sensor doesn't detect anything, and a 1 when movement is detected

We now have the ability to alter the result of our code by sensing movement! That is pretty exciting, as with this simple input capability we can now make gadgets that do things based on activity in the real world.

9.3 Make Something Happen

The most obvious use for a motion detector, is to make something happen when motion is detected! So lets do that:

```
while not pir.value():
    pass  # Ignore the PIR sensor when no motion is detected
print("Motion detected!")
```

Now we can make things happen on our device from the outside world!

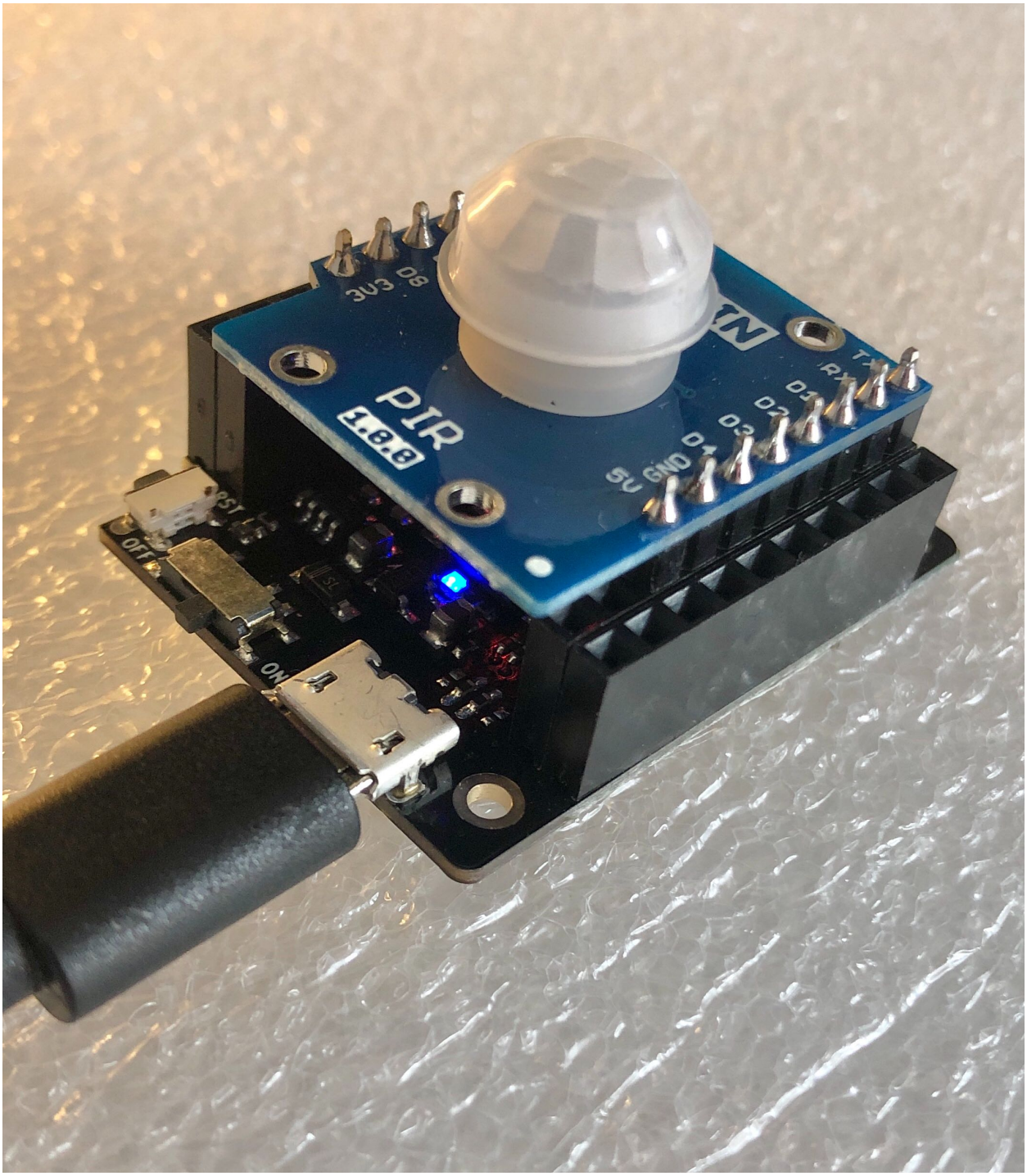


Fig. 2: Note the two empty pin sockets in the headers closer to the USB!

9.4 Exercises

Time to take those concepts and put them into action! The following subsections detail different exercises that can improve our usage of the button.

9.4.1 Exercise 1: Make things happen again and again and again

Expand on the *Make Something Happen* code to make it so that the message is printed every time motion is detected, not just the first time. The message should only print once each time motion is detected, and once the motion stops being detected, it should print how long it was detected for. For this, you may want to use `time.ticks_ms()`.

CHAPTER 10

Other Shields

There are a number of other shields that are not yet integrated into the workshop documentation but may be encountered.

Warning: While it is *possible* to plug shields in to the device while it is powered (plugged in to your computer), it is **not recommended**! As such, please remember to unplug the USB from your board *prior* to connecting or disconnecting any shields, or else you risk damaging the shield or the board.

10.1 Ambient Light

The Ambient Light Sensor Shield uses a BH1750 chip to detect the amount of ambient light present. It uses *I2C* to communicate with the microcontroller.

10.2 IR controller

The IR Controller has four infrared emitters and a receiver to allow bi-directional infrared communications. Two GPIO pins are used to send and receive data.

10.3 Temperature Sensor (SHT30)

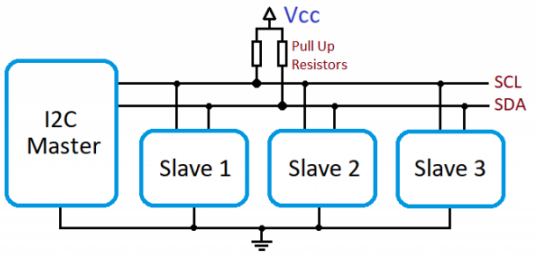
There are a number of temperature sensors but one of the more common models in the Wemos form factor is the SHT30 Shield. Named after the SHT30 temperature and humidity sensor it uses *I2C* to communicate with the microcontroller.

CHAPTER 11

MQTT

CHAPTER 12

I2C



- The MicroPython I2C docs explain how to send I2C commands
- Wikipedia has a thorough explanation of I2C
- Sparkfun has a good ground-up explanation too

CHAPTER 13

Indices and tables

- genindex
- modindex
- search