Micropython on ESP8266 Workshop Documentation

Release 1.0

Radomir Dopieralski

Jul 20, 2017

Contents

1 Setup **3**
- 1.1 Prerequisites . . . 3
- 1.2 Development Board . . . 3
- 1.3 Connecting . . . 5
- 1.4 Official Documentation and Support . . . 5

2 Basics **7**
- 2.1 Blink . . . 7
- 2.2 External Components . . . 8
- 2.3 Pulse Width Modulation . . . 9
- 2.4 Buttons . . . 10
- 2.5 Servomechanisms . . . 12
- 2.6 Beepers . . . 13
- 2.7 Network . . . 14
- 2.8 WebREPL . . . 15
- 2.9 Filesystem . . . 15
- 2.10 Uploading Files . . . 16
- 2.11 HTTP Requests . . . 16

3 Shields **19**
- 3.1 Button . . . 19
- 3.2 DHT and DHT Pro . . . 19
- 3.3 Neopixel . . . 20
- 3.4 Relay . . . 20
- 3.5 OLED . . . 20
- 3.6 Motor . . . 21
- 3.7 Micro SD . . . 21
- 3.8 Battery . . . 21
- 3.9 Servo (Custom) . . . 21
- 3.10 TFT Screen (Custom) . . . 22

4 Advanced **23**
- 4.1 Schematics . . . 23
- 4.2 Analog to Digital Converter . . . 24
- 4.3 Communication Protocols . . . 25
- 4.4 Neopixels . . . 25
- 4.5 Temperature and Humidity . . . 26

4.6 LED Matrix and 7-segment Displays 27
4.7 TFT LCD Display 28

5 Indices and tables **31**

Contents:

CHAPTER 1

Setup

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.
- If it's Windows or Mac OS, make sure to install drivers for the CH340 UBS2TTL chip.
- A micro-USB cable with data lines that fits your USB port.
- You will need a terminal application installed. For Linux and Mac you can use `screen`, which is installed by default. For Windows we recommend PuTTy or CoolTerm.
- Please note that the workshop will be in English.

In addition, at the workshop, you will receive:

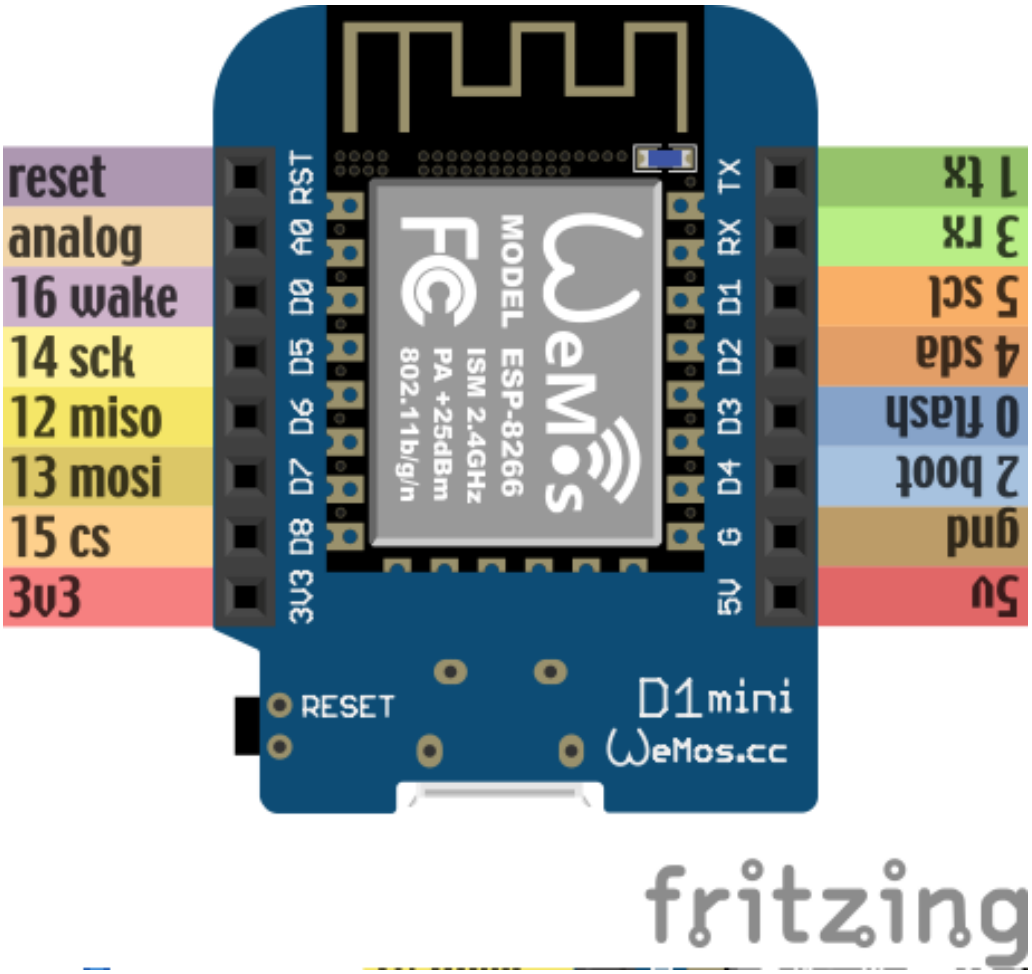
- WeMos D1 Mini development board with ESP8266 on it,
- Red LED,
- 100Ω resistor,
- SG90 microservo,
- Female-female and Male-female dupont cables,
- Piezoelectric speaker.

The firmware that is flashed on the boards is also available at https://bitbucket.org/thesheep/d1workshop/downloads/firmware-combined.bin

Development Board

The board we are using is called "WeMos D1 Mini" and has an ESP8266 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 D0, D1, D2, ... numbers printed on the board are different from what Micropython uses – because originally those boards were made for a different software. Make sure to refer to the image below to determine which pins are which.



It has a micro-USB socket for connecting to the computer. On the side is a button for reseting the board. Along the sides of the board are two rows of pins, to which we will be connecting cables.

The symbols meaning is 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.
- `gnd`, `G` - this is the ground. Think about it like the minus side of the battery.
- `gpioXX` - "gpio" stands for "general purpose input output". Those are the pins we will be using for sending and receiving signals to and from various devices that we will connect to them. They can act as output – pretty much like a switch that you can connect to plus or to minus with your program. Or they can act as input, telling your program whether they are connected to plus or minus.
- `a0` - this is the analog pin. It can measure the voltage that is applied to it, but it can only handle up to 3.3V.
- `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.

- `rst` - this is a reset button (and a corresponding pin, to which you can connect external button).

Many of the gpio pins have an additional function, we will cover them separately.

Connecting

The board you got should already have MicroPython with all the needed libraries flashed on it. In order to access its console, you will need to connect it to your computer with the micro-USB cable, and access the serial interface that appears with a terminal program.

Linux and MacOS

Simply open a terminal and run the following commands. On Linux:

```
screen /dev/ttyUSB0 115200
```

On MacOS:

```
screen /dev/tty.SLAB_USBtoUART 115200
```

To exit screen, press ctrl+A and then capital K.

Windows

For the serial interface to appear in your system, you will need to install the drivers for CH340. Once you have that, you can use either Hyper Terminal, PuTTy or CoolTerm to connect to it, following this guide.

The parameters for the connection are: 115200 baud rate, 8 data bits, no parity, 1 stop bit, no flow control.

Hello world!

Once you are connected, press "enter" and you should see the Micropython 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.

Official Documentation and Support

The official documentation for this port of Micropython is available at http://micropython.org/resources/docs/en/latest/esp8266/. There is a also a forum on which you can ask questions and get help, located at http://forum.micropython.org/. Finally, there are `#esp8266` and `#micropython` channels on http://freenode.net IRC network, where people chat in real time. Remember that all people there are just users like you, but possibly more experienced, and not employees who get paid to help you.

CHAPTER 2

Basics

Blink

The traditional first program for hobby electronics is a blinking light. We will try to build that.

The boards you have actually have a light built-in, so we can use that. There is a LED (light-emitting diode) near the antenna (the golden zig-zag). The plus side of that LED is connected to the `3v3` pins internally, and the minus side is connected to `gpio2`. So we should be able to make that LED shine with our program by making `gpio2` behave like the gnd pins. We need to "bring the `gpio2` low", or in other words, make it connected to `gnd`. Let's try that:

```
from machine import Pin

led = Pin(2, Pin.OUT)
led(0)
```

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.

Once we have the "Pin" function imported, we use it to create a pin object, with the first parameter telling it to use `gpio2`, and the second parameter telling it to switch it into output mode. Once created, the pin is assigned to the variable we called "led".

Finally, we bring the pin low, by calling the "led" variable with value 0. At this point the LED should start shining. In fact, it may have started shining a line earlier, because once we switched the pin into output mode, its default state is "low".

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 bring it "high". If we do that, both ends of the LED will be connected to "plus", and the current won't flow. We do that with:

```
led(1)
```

Now, how can we make the LED blink 10 times? We could of course type `led(0)` and `led(1)` 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(1)
    led(0)
```

Note, that when you are typing this, it will look more like:

```
>>> for i in range(10):
...     led(1)
...     led(0)
...
...
>>>
```

That's because the console automatically understands that when you indent a line, you mean it to be a block of code inside the "for" loop. You have to un-indent the last line (by removing the spaces with backspace) to finish this command. You can avoid that by using "paste mode" – press `ctrl+E`, paste your code, and then press `ctrl+D` to have it executed.

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
```

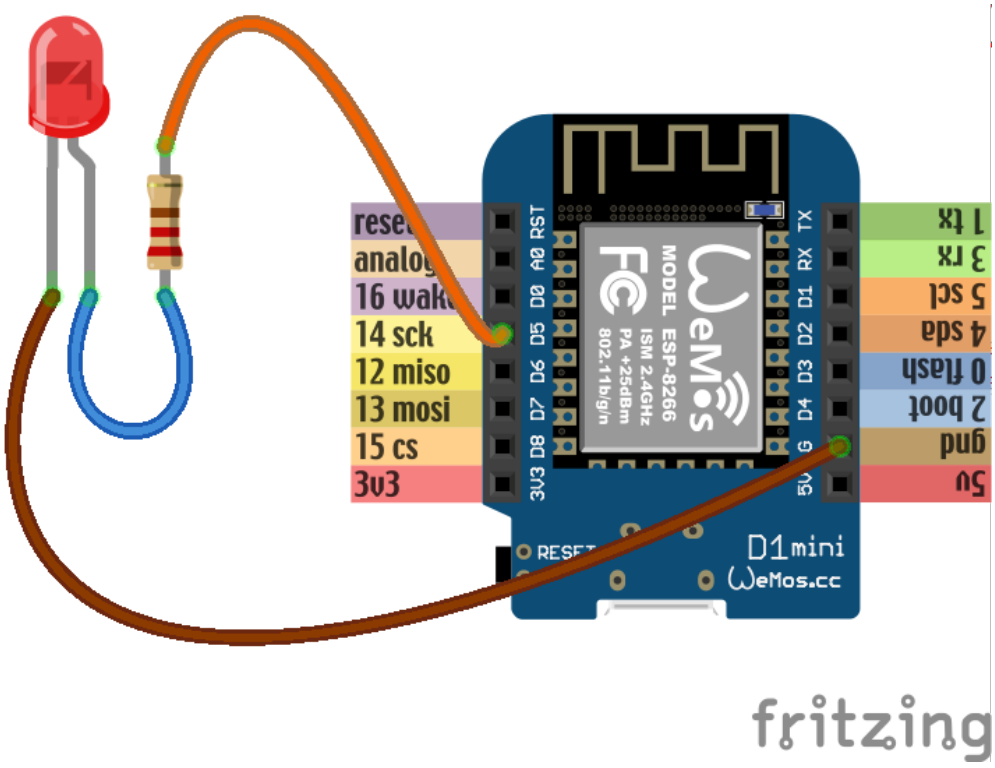
And then we will repeat our program, but with the waiting included:

```
for i in range(10):
    led(1)
    time.sleep(0.5)
    led(0)
    time.sleep(0.5)
```

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

External Components

Now let's try the same, but not with the build-in LED – let's connect an external LED and try to use that. The connection should look like this:



Note how one leg of the LED is a little bit longer, and the other had a flattening on the plastic of the LED next to it. The long leg should go to the plus, and the short one to the minus. We are connecting the LED in opposite way than the internal one is connected – between the pin and `gnd`. That means that it will shine when the pin is high, and be dark when it's low.

Also note how we added a resistor in there. That is necessary to limit the amount of current that is going to flow through the LED, and with it, its brightness. Without the resistor, the LED would shine very bright for a short moment, until either it, or the board, would overheat and break. We don't want that.

Now, let's try the code:

```
from machine import Pin
import time

led = Pin(14, Pin.OUT)
for i in range(10):
    led(1)
    time.sleep_ms(500)
    led(0)
    time.sleep_ms(500)
```

Again, you should see the LED blink 10 times, half a second for each blink.

This time we used `time.sleep_ms()` instead of `time.sleep()` – it does the same thing, but takes the number of milliseconds instead od seconds as the parameter, so we don't have to use fractions.

Pulse Width Modulation

Wouldn't it be neat if instead of blinking, the LED slowly became brighter and then fade out again? Can we do this somehow?

The brightness of the LED depends on the voltage being supplied to it. Unfortunately, our GPIO pins only have a simple switch functionality – we can turn them on or off, but we can't fluently change the voltage (there are pins that could do that, called DAC, for "digital to analog converter", but our board doesn't have those). But there is another way. Remember when we first tried to blink the LED without any delay, and it happened too fast to see?

Turns out we can blink the LED very fast, and by varying the time it is on and off change how bright it seems to be to the human eye. The longer it is on and the shorter it is off, the brighter it will seem.

Now, we could do that with a simple loop and some very small delays, but it would keep our board busy and prevent it from doing anything else, and also wouldn't be very accurate or terribly fast. But the ESP8266 has special hardware dedicated just for blinking, and we can use that! This hardware is called PWM (for Pulse Width Modulation), and you can use it like this:

```
from machine import Pin, PWM
import time

pwm = PWM(Pin(2))
pwm.duty(896)
time.sleep(1)
pwm.duty(512)
time.sleep(1)
pwm.duty(0)
```

If you run this, you should see the blue led on `gpio2` change brightness. The possible range is from 1023 (100% duty cycle, the LED is off) to 0 (0% duty cycle, the LED is on full brightness). Why is 0 full brightness? Remember, that the LED on the `gpio2` is reversed – it shines when the pin is off, and the duty cycle tells how much the pin is on.

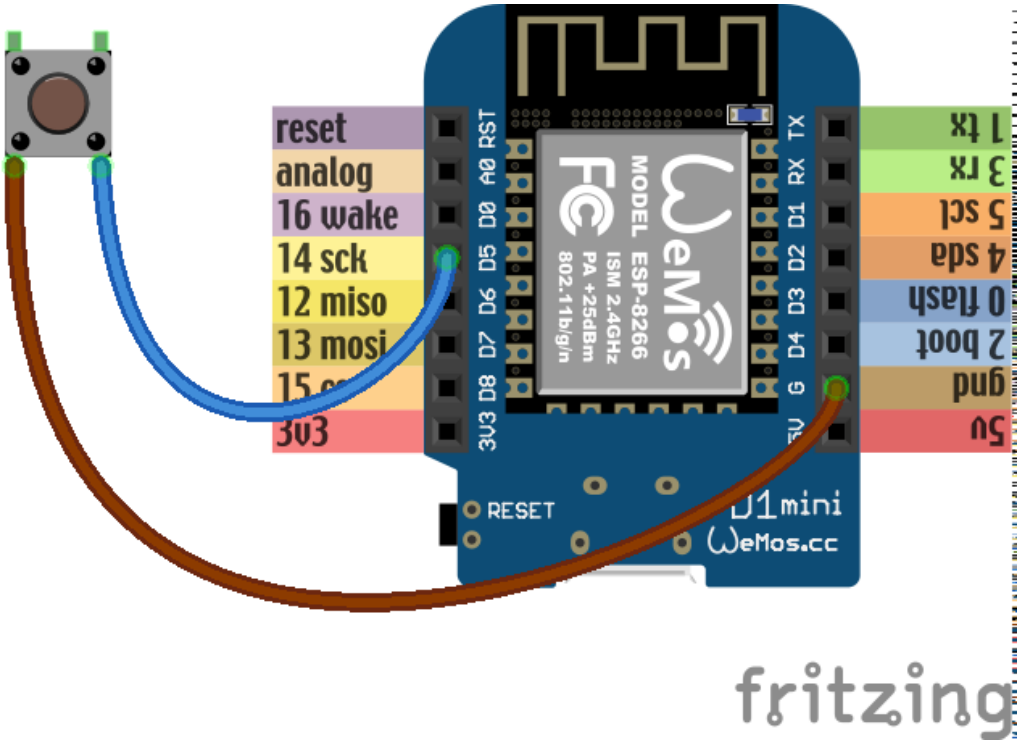
You can also change the frequency of the blinking. Try this:

```
pwm.freq(1)
```

That should blink the LED with frequency of 1Hz, so once per second – we are basically back to our initial program, except the LED blinks "in the background" controlled by dedicated hardware, while your program can do other things!

Buttons

We don't have a button in our kit, but we can simulate one by just using two wires, one with a male plug, and one with female. Connect them like so:



Now we will write some code that will switch the LED on and off each time the wires are put together:

```
from machine import Pin
led = Pin(2, Pin.OUT)
button = Pin(14, Pin.IN, Pin.PULL_UP)
while True:
    if not button():
        led(not led())
        while not button():
            pass
```

We have used `Pin.IN` because we want to use `gpio14` as an input pin, on which we will read the voltage. We also added `Pin.PULL_UP` – that means that there is a special internal resistor enabled between that pin and the `3V3` pins. The effect of this is that when the pin is not connected to anything (we say it's "floating"), it will return 1. If we didn't do that, it would return random values depending on its environment. Of course when you connect the pin to `GND`, it will return 0.

However, when you try this example, you will see that it doesn't work reliably. The LED will blink, and sometimes stay off, sometimes switch on again, randomly. Why is that?

That's because your hands are shaking. A mechanical switch has a spring inside that would shake and vibrate too. That means that each time you touch the wires (or close the switch), there are in reality multiple signals sent, not just one. This is called "bouncing", because the signal bounces several times.

To fix this issue, we will do something that is called "de-bouncing". There are several ways to do it, but the easiest is to just wait some time for the signal to stabilize:

```
import time
from machine import Pin
led = Pin(2, Pin.OUT)
button = Pin(14, Pin.IN, Pin.PULL_UP)
while True:
    if not button.value():
```

```
        led(not led())
        time.sleep_ms(300)
        while not button():
            pass
```

Here we wait 3/10 of a second – too fast for a human to notice, but enough for the signal to stabilize. The exact time for this is usually determined experimentally, or by measuring the signal from the switch and analyzing it.

Servomechanisms

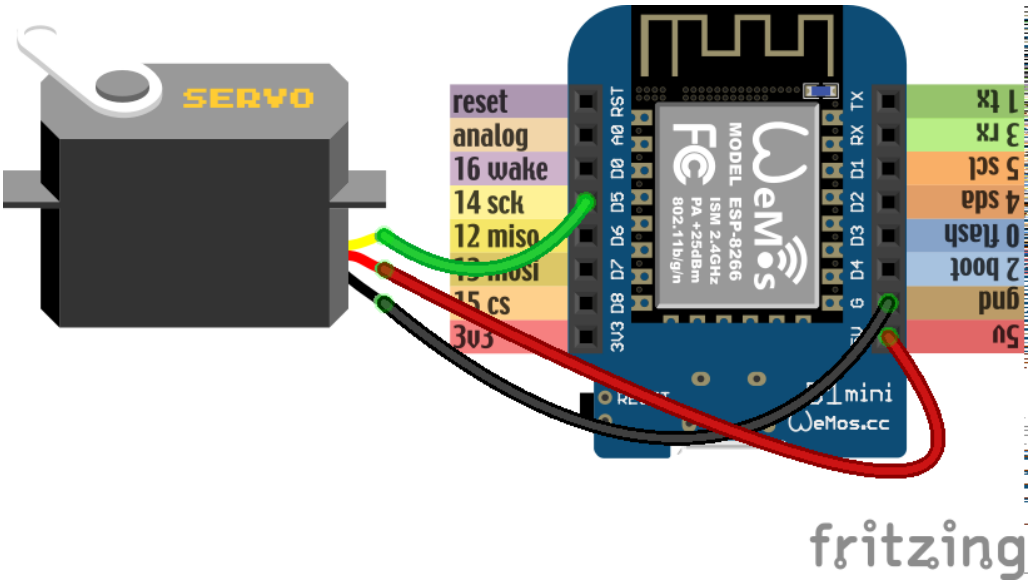
Time to actually physically move something. If you plan on building a robot, there are three main ways of moving things from the microcontroller:

- a servomechanism (servo for short),
- an H-bridge and a DC motor,
- a stepper or brushless motor with a driver.

We are going to focus on the servo first, because I think this is the easiest and cheapest way. We are going to use a cheap "hobby" servo, the kind that is used in toys – it's not particularly strong, but it's enough for most use cases.

Warning: Don't try to force the movement of the servo arms with your hand, you are risking breaking the delicate plastic gears inside.

A hobby servo has three wires: brown or black `gnd`, red or orange `vcc`, and white or yellow `signal`. The `gnd` should of course be connected to the `gnd` of our board. The `vcc` is the power source for the servo, and we are going to connect it to the `vin` pin of our board – this way it is connected directly to the USB port, and not powered through the board.



Caution: Servos and motors usually require a lot of current, more then your board can supply, and often even more than than you can get from USB. Don't connect them to the `3v3` pins of your board, and if you need two or more, power them from a battery (preferably rechargeable).

The third wire, `signal` tells the servo what position it should move to, using a 50Hz PWM signal. The center is at around 77, and the exact range varies with the servo model, but should be somewhere between 30 and 122, which corresponds to about 180° of movement. Note that if you send the servo a signal that is outside of the range, it will still obediently try to move there – hitting a mechanical stop and buzzing loudly. If you leave it like this for longer, you can damage your servo, your board or your battery, so please be careful.

So now we are ready to try and move it to the center position:

```
from machine import Pin, PWM
servo = PWM(Pin(14), freq=50, duty=77)
```

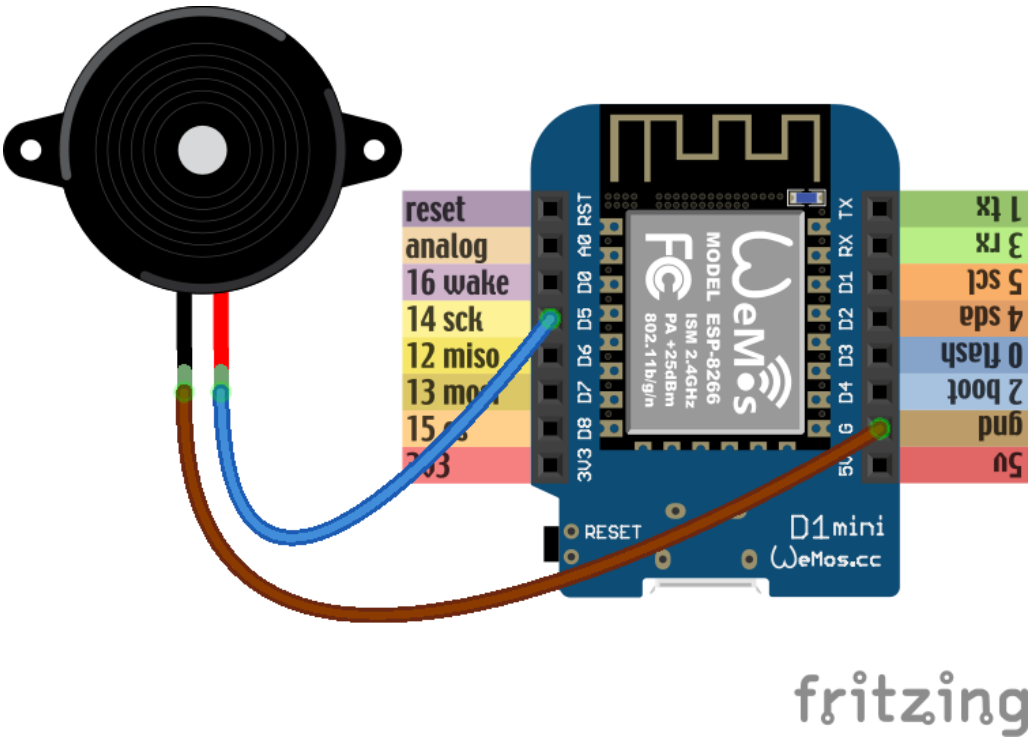
Then we can see where the limits of its movement are:

```
servo.duty(30)
servo.duty(122)
```

There also exist "continuous rotation" servos, which don't move to the specified position, but instead rotate with specified speed. Those are suitable for building simple wheeled robots. It's possible to modify a normal servo into a continuous rotation servo.

Beepers

When I wrote that PWM has a frequency, did you immediately think about sound? Yes, electric signals can be similar to sound, and we can turn them into sound by using speakers. Or small piezoelectric beepers, like in our case.



The piezoelectric speaker doesn't use any external source of power – it will be powered directly from the GPIO pin – that's why it can be pretty quiet. Still, let's try it:

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

```
beeper = PWM(Pin(14), freq=440, duty=512)
time.sleep(0.5)
beeper.deinit()
```

We can even play melodies! For instance, here's the musical scale:

```
from machine import Pin, PWM
import time
tempo = 5
tones = {
    'c': 262,
    'd': 294,
    'e': 330,
    'f': 349,
    'g': 392,
    'a': 440,
    'b': 494,
    'C': 523,
    ' ': 0,
}
beeper = PWM(Pin(14, Pin.OUT), freq=440, duty=512)
melody = 'cdefgabC'
rhythm = [8, 8, 8, 8, 8, 8, 8, 8]

for tone, length in zip(melody, rhythm):
    beeper.freq(tones[tone])
    time.sleep(tempo/length)
beeper.deinit()
```

Unfortunately, the maximum frequency of PWM is currently 1000Hz, so you can't play any notes higher than that.

It's possible to make the sounds louder by using a better speaker and possibly an audio amplifier.

Network

The ESP8266 has wireless networking support. It can act as a WiFi access point to which you can connect, and it can also connect to the Internet.

To configure it as an access point, run code like this (use your own name and password):

```
import network
ap = network.WLAN(network.AP_IF)
ap.active(True)
ap.config(essid="network-name", authmode=network.AUTH_WPA_WPA2_PSK, password=
↪"abcdabcdabcd")
```

To scan for available networks (and also get additional information about their signal strength and details), use:

```
import network
sta = network.WLAN(network.STA_IF)
sta.active(True)
print(sta.scan())
```

To connect to an existing network, use:

```
import network
sta = network.WLAN(network.STA_IF)
sta.active(True)
sta.connect("network-name", "password")
```

Once the board connects to a network, it will remember it and reconnect every time. To get details about connection, use:

```
sta.ifconfig()
sta.status()
sta.isconnected()
```

WebREPL

The command console in which you are typing all the code is called "REPL" – an acronym of "read-evaluate-print-loop". It works over a serial connection over USB. However, once you have your board connected to network, you can use the command console in your browser, over network. That is called WebREPL.

First, you will need to download the web page for the WebREPL to your computer. Get the file from https://github.com/micropython/webrepl/archive/master.zip and unpack it somewhere on your computer, then click on the `webrepl.html` file to open it in the browser.

In order to connect to your board, you have to know its address. If the board works in access point mode, it uses the default address. If it's connected to WiFi, you can check it with this code:

```
import network
sta = network.WLAN(network.STA_IF)
print(sta.ifconfig())
```

You will see something like `XXX.XXX.XXX.XXX` – that's the IP address. Enter it in the WebREPL's address box at the top like this `ws://XXX.XXX.XXX.XXX:8266/`.

To connect to your board, you first have to start the server on it. You do it with this code:

```
import webrepl
webrepl.start()
```

Now you can go back to the browser and click "connect". On the first connection, you will be asked to setup a password – later you will use that password to connect to your board.

Filesystem

Writing in the console is all fine for experimenting, but when you actually build something, you want the code to stay on the board, so that you don't have to connect to it and type the code every time. For that purpose, there is a file storage on your board, where you can put your code and store data.

You can see the list of files in that storage with this code:

```
import os
print(os.listdir())
```

You should see something like `['boot.py']` – that's a list with just one file name in it. `boot.py` and later `main.py` are two special files that are executed when the board starts. `boot.py` is for configuration, and you can put your own code in `main.py`.

You can create, write to and read from files like you would with normal Python:

```
with open("myfile.txt", "w") as f:
    f.write("Hello world!")
print(os.listdir())
with open("myfile.txt", "r") as f:
    print(f.read())
```

Please note that since the board doesn't have much memory, you can put large files on it.

Uploading Files

You can use the WebREPL to upload files to the board from your computer. To do that, you need to open a terminal in the directory where you unpacked the WebREPL files, and run the command:

```
python webrepl_cli.py yourfile.xxx XXX.XXX.XXX.XXX:
```

Where `yourfile.xxx` is the file you want to send, and `XXX.XXX.XXX.XXX` is the address of your board.

Note: You have to have Python installed on your computer for this to work.

This requires you to setup a network connection on your board first. However, you can also upload files to your board using the same serial connection that you use for the interactive console. You just need to install a small utility program:

```
pip install adafruit-ampy
```

And then you can use it to copy files to your board:

```
ampy --port=/dev/ttyUSB0 put yourfile.xxx
```

Warning: The serial connection can be only used by a single program at a time. Make sure that your console is discobbected while you use ampy, otherwise you may get a cryptic error about it not having the access rights.

HTTP Requests

Once you are connected to network, you can talk to servers and interact with web services. The easiest way is to just do a HTTP request – what your web browser does to get the content of web pages:

```
import urequests
r = urequests.get("http://duckduckgo.com/?q=micropython&format=json").json()
print(r)
print(r['AbstractText'])
```

You can use that to get information from websites, such as weather forecasts:

```
import json
import urequests
r = urequests.get("http://api.openweathermap.org/data/2.5/weather?q=Zurich&appid=XXX
↪").json()
```

```
print(r["weather"][0]["description"])
print(r["main"]["temp"] - 273.15)
```

It's also possible to make more advanced requests, adding special headers to them, changing the HTTP method and so on. However, keep in mind that our board has very little memory for storing the answer, and you can easily get a `MemoryError`.

CHAPTER 3

Shields

There is a number of ready to use "shields" – add-on boards – for the WeMos D1 Mini, containing useful components together with all the necessary connections and possible additional components. All you have to do is plug such a "shield" on top or bottom of the WeMos D1 Mini board, and load the right code to use the components on it.

Button

This is a very basic shield that contains a single pushbutton. The button is connected to pin `gpio0` and to `gnd`, so you can read its state with this code:

```
from machine import Pin
button = Pin(0)
if button.value():
    print("The button is not pressed.")
else:
    print("The button is pressed.")
```

Of course everything we learned about buttons and debouncing applies here as well.

DHT and DHT Pro

Those two shield have temperature and humidity sensors on them. The first one, DHT, has DHT11 sensor, the second one, DHT Pro, has DHT22, which is more accurate and has better precision.

In both cases the sensors are available on the pin `gpio2`, and you can access them with code like this:

```
from machine import Pin
import dht
sensor = dht.DHT11(Pin(2))
sensor.measure()
print(sensor.temperature())
print(sensor.humidity())
```

(Use `DHT22` class for the DHT Pro shield.)

It is recommended to use this shield with the "dual base", so that the temperature sensor is not right above or below the ESP8266 module, which tends to become warm during work and can affect temperature measurements.

Neopixel

That shield has a single addressable RGB LED on it, connected to pin `gpio4`. Unfortunately, that means that this shield conflicts with any other shield that uses the I^2C protocol, such as the OLED shield or the motor shield. You can use it with code lik this:

```
form machine import Pin
import neopixel
pixels = neopixel.NeoPixel(Pin(4, Pin.OUT), 1)
pixels[0] = (0xff, 0x00, 0x22)
pixels.write()
```

Relay

This shield contains a relay switch, together with a transistor and a couple of other components required to reliably connect it to the board. It uses pin `gpio5`, which unfortunately makes it incompatible with any other shields using the I^2C protocol, such as the OLED shield or the motor shield. You can control the relay with the following code:

```
from machine import Pin
relay = Pin(5, Pin.OUT)
relay.low() # Switch off
relay.high() # Switch on
```

OLED

A small, 64×48 monochrome display. It uses pins `gpio4` and `gpio5` to talk with the board with the I^2C protocol. It will conflict with any other shield that uses those pins, but doesn't use I^2C, like the neopixel shield or the relay shield. It can coexist with other shields that use I^2C, like the motor shield.

Up to two such displays can be connected at the same time, provided they have different addresses set using the jumper on the back.

You can control the display using the `ssd1306` library:

```
import ssd1306
from machine import I2C, Pin
i2c = I2C(-1, Pin(5), Pin(4))
display = ssd1306.SSD1306_I2C(64, 48, i2c)
display.fill(0)
display.text("Hello", 0, 0)
display.text("world!", 0, 8)
display.pixel(20, 20, 1)
display.show()
```

Motor

The motor shield contains a H-bridge) and a PWM chip, and it's able to drive up to two small DC motors. You can control it using I^{2}C on pins `gpio4` and `gpio5`. It will conflict with any shields that use those pins but don't use I^{2}C, such as the relay shield and the neopixel shield. It will work well together with other shields using I^{2}C.

Up to four such shields can be connected at the same time, provided they have different addresses selected using the jumpers at their backs.

In order to use this shield, use the `d1motor` library:

```
import d1motor
from machine import I2C, Pin
i2c = I2C(-1, Pin(5), Pin(4), freq=10000)
m0 = d1motor.Motor(0, i2c)
m1 = d1motor.Motor(1, i2c)
m0.speed(5000)
```

Micro SD

This shield lets you connect a micro SD card to your board. It connects to pins `gpio12`, `gpio13`, `gpio14` and `gpio15` and uses SPI protocol. It can be used together with other devices using the SPI protocol, as long as they don't use pin `gpio15` as CS.

You can mount an SD card in place of the internal filesystem using the following code:

```
import os
from machine import SPI, Pin
import sdcard
sd = sdcard.SDCard(SPI(1), Pin(15))
os.umount()
os.VfsFat(sd, "")
```

Afterwards you can use `os.listdir()`, `open()` and all other normal file functions to manipulate the files on the SD card. In order to mount the internal filesystem back, use the following code:

```
import flashbdev
os.umount()
os.VfsFat(flashbdev.bdev, "")
```

Battery

This shield lets you power your board from a single-cell LiPo battery. It connects to the `5V` pin, and doesn't require any communication from your board to work. You can simply plug it in and use it.

Servo (Custom)

There is an experimental 18-channel servo shield. It uses the I^{2}C protocol on pins `gpio4` and `gpio5` and is compatible with other I^{2}C shields.

In order to power the servos, you need to either provide external power to the pin marked with + next to the 5V pin, or connect it with the 5V pin to make the servos share power with the board.

You can set the servo positions using the following code:

```
from servo import Servos
from machine import I2C, Pin
i2c = I2C(-1, Pin(5), Pin(4))
servos = Servos(i2c)
servos.position(0, degrees=45)
```

TFT Screen (Custom)

There is an experimental breakout board for the ST7735 TFT screen. It uses the SPI interface on pins gpio12, gpio13, gpio14, and gpio15.

You can use it with the following example code:

```
from machine import Pin, SPI
import st7735

display = st7735.ST7735(SPI(1), dc=Pin(12), cs=None, rst=Pin(15))
display.fill(0x7521)
display.pixel(64, 64, 0)
```

If you have a display with a red tab, you need to use a different initialization:

```
display = st7735.ST7735R(SPI(1, baudrate=40000000), dc=Pin(12), cs=None, rst=Pin(15))
```

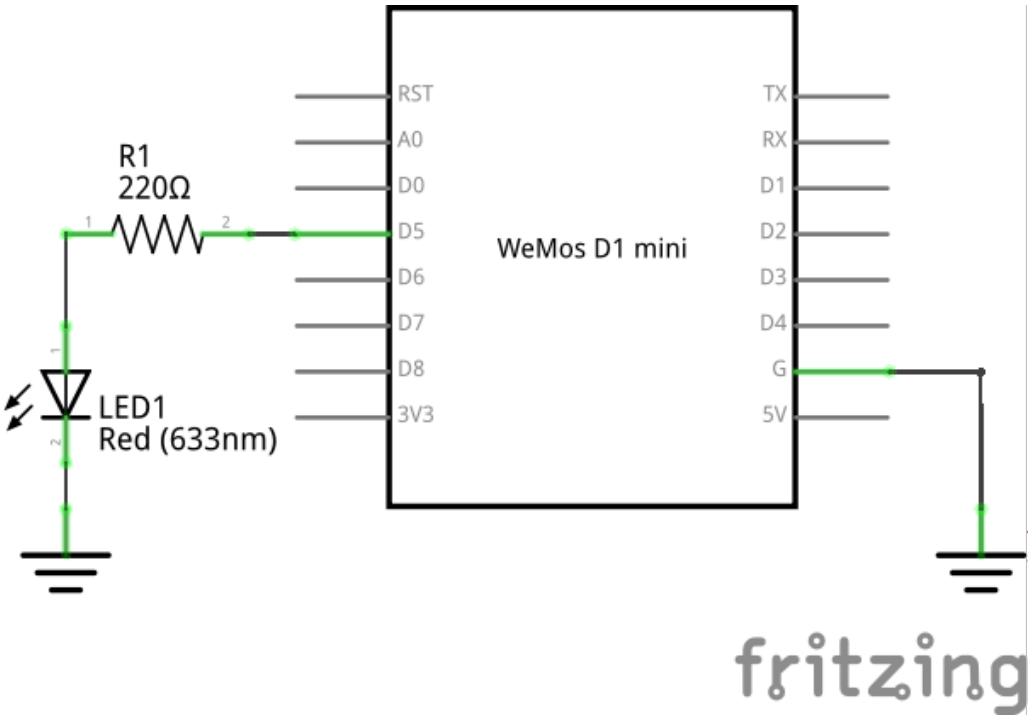
CHAPTER 4

Advanced

Schematics

The pretty colorful pictures that we have been using so far are not very useful in practical projects. You can't really draw them by hand, different components may look very similar, and it's hard to see what is going on when there are a lot of connections. That's why engineers prefer to use more symbolic representation of connection, a schematic.

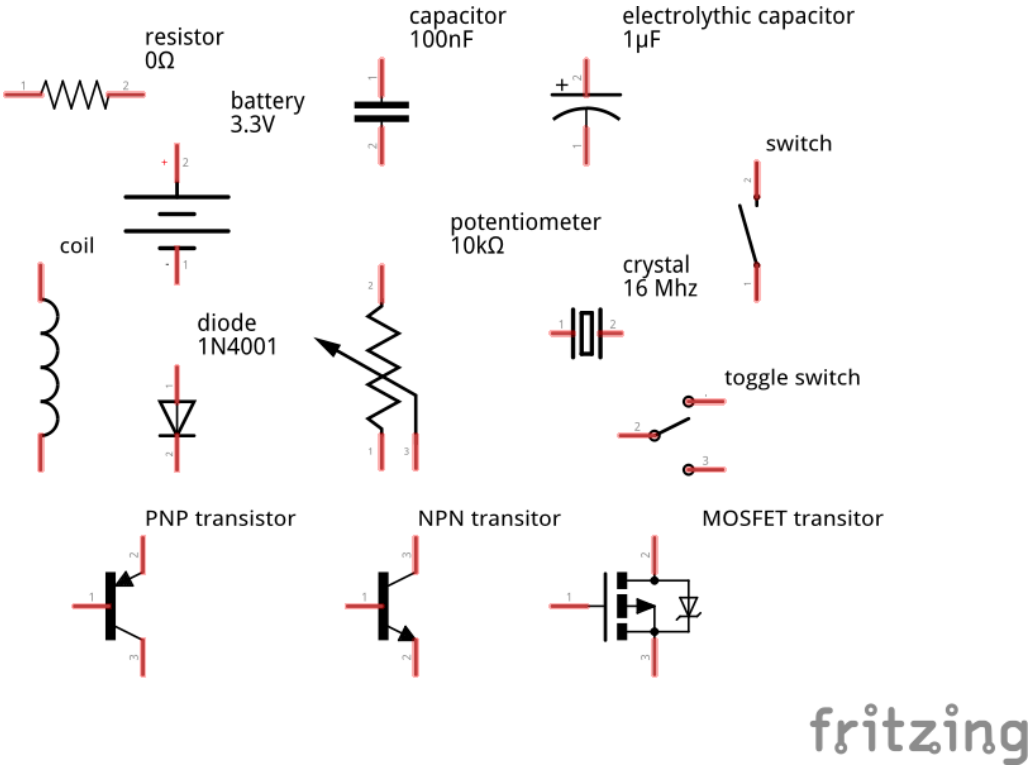
A schematic doesn't care how the parts actually look like, or how their pins are arranged. Instead they use simple symbols. For instance, here's a schematic of our experiment with the external LED:



The resistor is symbolized by a zig-zag. The LED is marked by a diode symbol (a triangle with a bar), with additional two arrows showing that it's a light emitting diode. The board itself doesn't have a special symbol – instead it's symbolized by a rectangle with the board's name written in it.

There is also a symbol for "ground" – the three horizontal lines. Since a lot of components need to be usually connected to the ground, instead of drawing all those wires, it's easier to simply use that symbol.

Here are some more symbols:

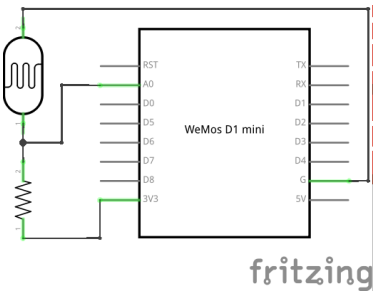


It's important to learn to read and draw electric schematics, because anything more advanced is going to use them, and you will also need them when asking for help on the Internet.

Analog to Digital Converter

Our board has only one "analog" pin, `A0`. That pin is connected to an ADC, or "analog to digital converter" – basically an electronic voltmeter, which can tell you what voltage is on the pin. The one we have can only measure from 0 to 1V, and would be damaged if it got more than 1V, so we have to be careful.

We will connect a photo-resistor to it. It's a special kind of a resistor that changes its resistance depending on how much light shines on it. But to make this work, we will need a second, fixed, resistor to make a "volatge divider". This way the voltage will change depending on the resistance of our photo-resistor.



Now, we will just read the values in our program, and print them in a loop:

```
from machine import ADC
adc = ADC(0)
while True:
    print(adc.read())
```

You should see a column of numbers changing depending on how much light the photo-resistor has. Try to cover it or point it toward a window or lamp. The values are from 0 for 0V, to 1024 for 1V. Ours will be somewhere in between.

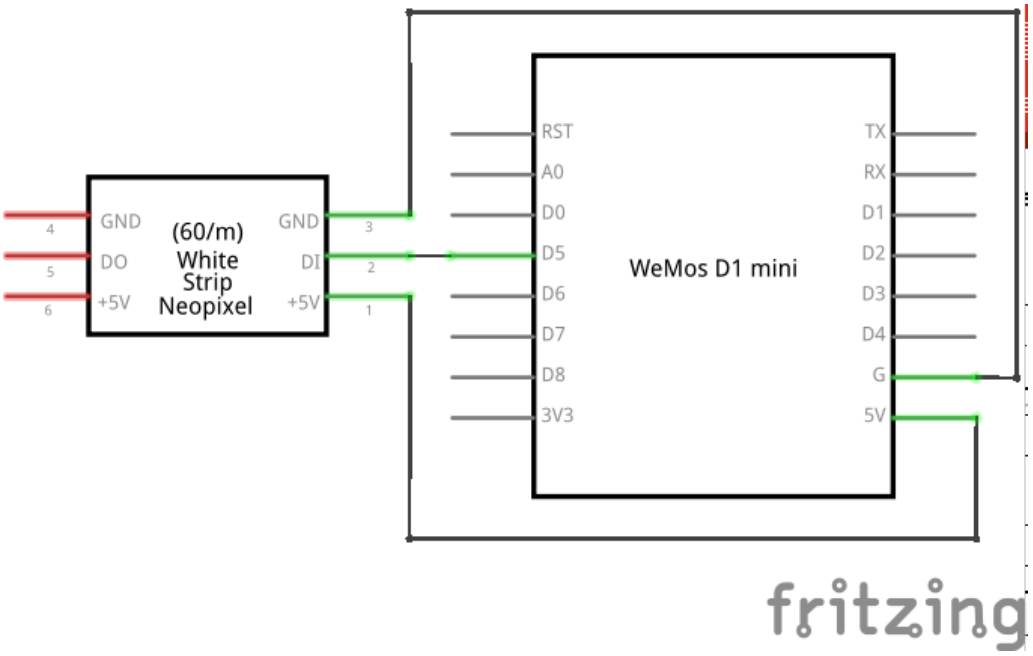
Communication Protocols

So far all devices we connected to the board were relatively simple and only required a single pin. More sophisticated devices are controlled with multiple pins, and often have very elaborate ways in which you have to change the pins to make them do something, like sending a character to them, or retrieving a value. Those ways are often standardized, and already implemented for you, so that you don't have to care about all the gory details – you just call high-level commands, and the libraries and/or hardware in your board handles it all for you.

Among the most popular protocols are UART, I^2C and SPI. We are going to have examples of each of them, but we are not going to get into details of how they work internally. It's enough to know that they let you send bytes to the device, and receive bytes in response.

Neopixels

Those are actually WS2812B addressable RGB LEDs, but they are commonly known as "neopixels". You can control individually the brightness and color of each of the LEDs in a string (or matrix, or ring). The connection is simple:



And the code for driving them is not very complex either, because the library for generating the signal is included in Micropython:

```
form machine import Pin
import neopixel
pixels = neopixel.Neopixel(Pin(14, Pin.OUT), 8)
pixels[0] = (0xff, 0x00, 0x00)
pixels.write()
```

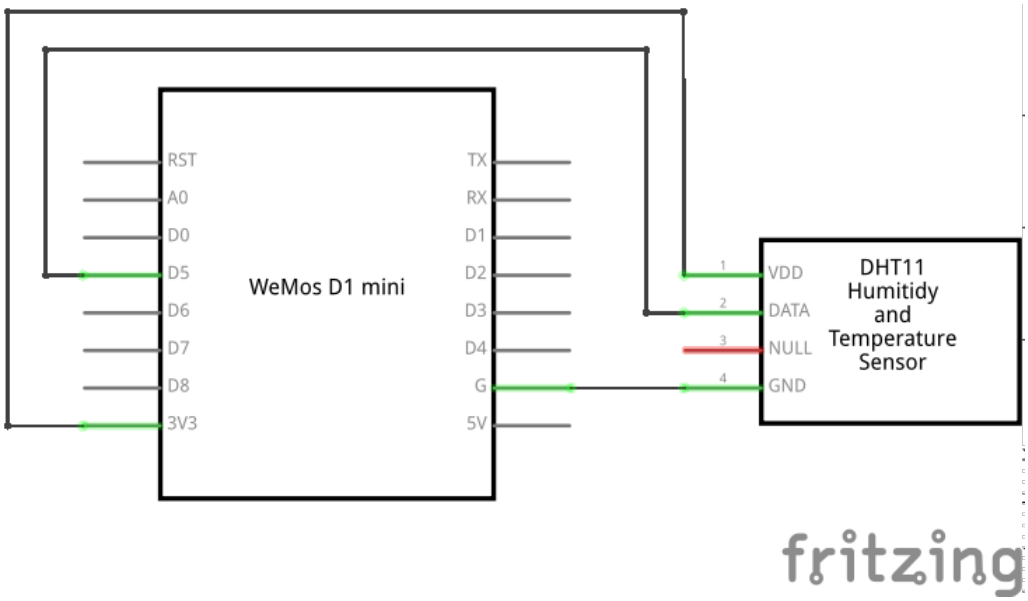
Where `8` is the number of LEDs in a chain. You can create all sorts of animations, rainbows and pretty effects with those.

Temperature and Humidity

The DHT11 and DHT22 sensors are quite popular for all sorts of weather stations. They use a single-wire protocol for communication. MicroPython on ESP9266 has that covered:

```
from machine import Pin
import dht
sensor = dht.DHT11(Pin(14))
sensor.measure()
print(sensor.temperature())
print(sensor.humidity())
```

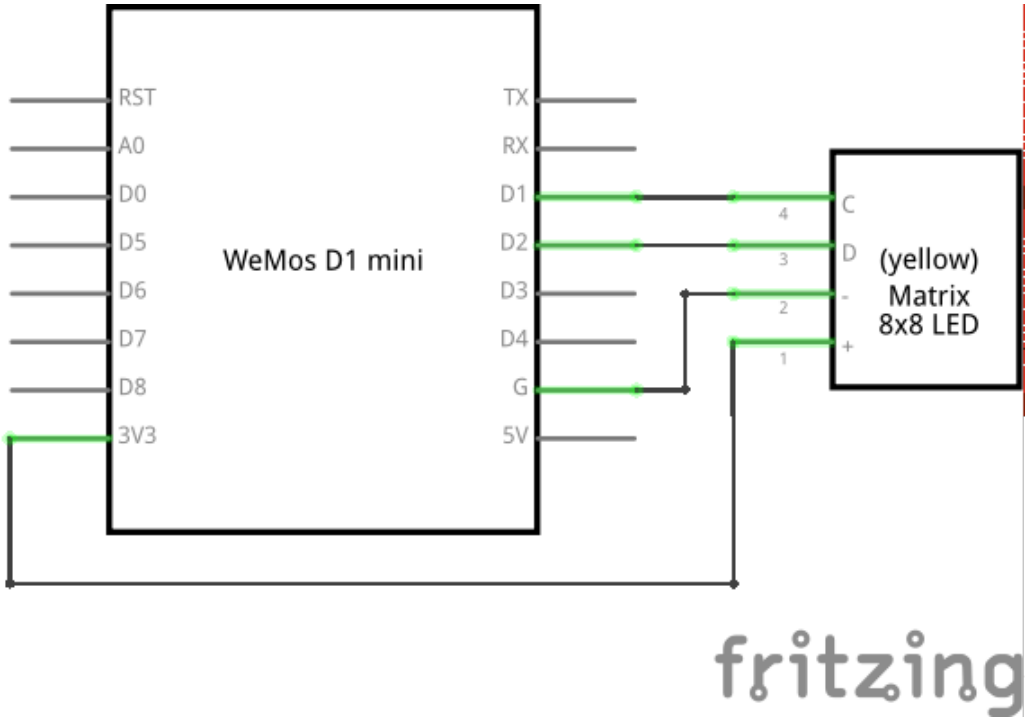
The connections are simple:



LED Matrix and 7-segment Displays

Adafruit sells a lot of "backpacks" with 7- or 14-segment displays or LED matrices, that we can control easily over I^2C. They use a HT16K33 chip, so that we don't have to switch on and off the individual LEDs – we just tell the chip what to do, and it takes care of the rest.

The schematic for connecting any I^2C device will be almost always the same:



Note: The two resistors on the schematic are needed for the protocol to work reliably with longer wires. For our

experiments, it's enough to rely on the pull-up resistors that are built into the board we are using.

The communication with the backpack is relatively simple, but I wrote two libraries for making it more convenient. For the matrix:

```
from machine import I2C, Pin
from ht16k33_matrix import Matrix8x8
i2c = I2C(sda=Pin(4), scl=Pin(5))
display = Matrix8x8(i2c)
display.brightness(8)
display.blink_rate(2)
display.fill(True)
display.pixel(0, 0, False)
display.pixel(7, 0, False)
display.pixel(0, 7, False)
display.pixel(7, 7, False)
display.show()
```

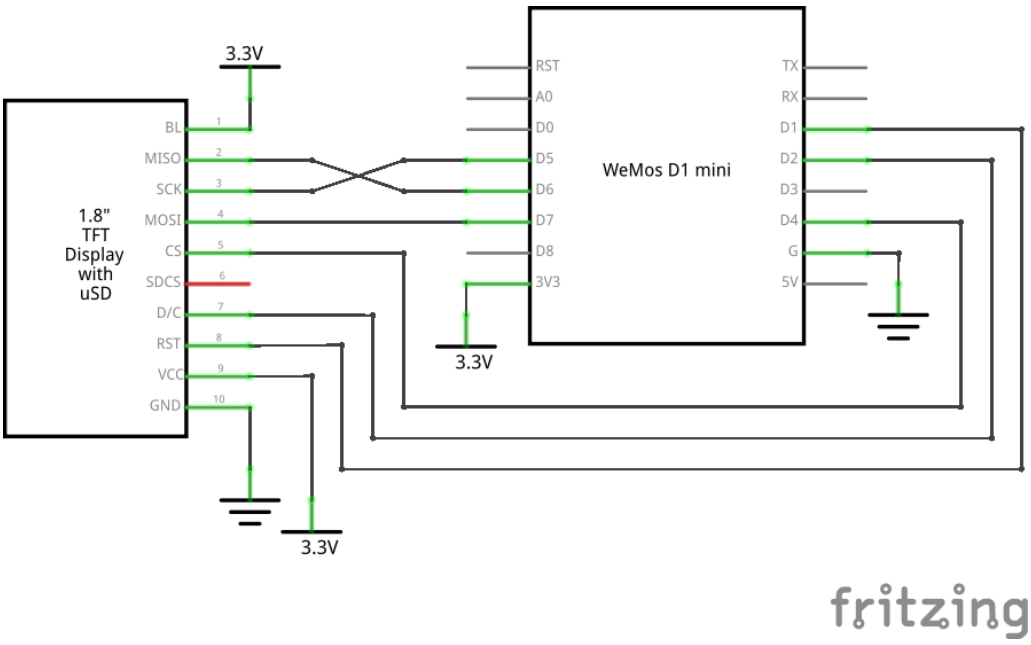
and for the 7- and 14-segment displays:

```
from machine import I2C, Pin
from ht16k33_seg import Seg7x4
i2c = I2C(sda=Pin(4), scl=Pin(5))
display = Seg7x4(i2c)
display.push("8.0:0.8")
display.show()
```

TFT LCD Display

The I^{2}C protocol is nice and simple, but not very fast, so it's only good when you have a few pixels to switch. With larger displays, it's much better to use SPI, which can be much faster.

Here is an example on how to connect an ILI9340 display:



And here is a simple library that lets you draw on that display:

```
from machine import Pin, SPI
import ili9341
spi = SPI(miso=Pin(12), mosi=Pin(13), sck=Pin(14))
display = ili9341.ILI9341(spi, cs=Pin(2), dc=Pin(4), rst=Pin(5))
display.fill(ili9341.color565(0xff, 0x11, 0x22))
display.pixel(120, 160, 0)
```

As you can see, the display is still quite slow – there are a lot of bytes to send, and we are using software SPI implementation here. The speed will greatly improve when Micropython adds hardware SPI support.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search