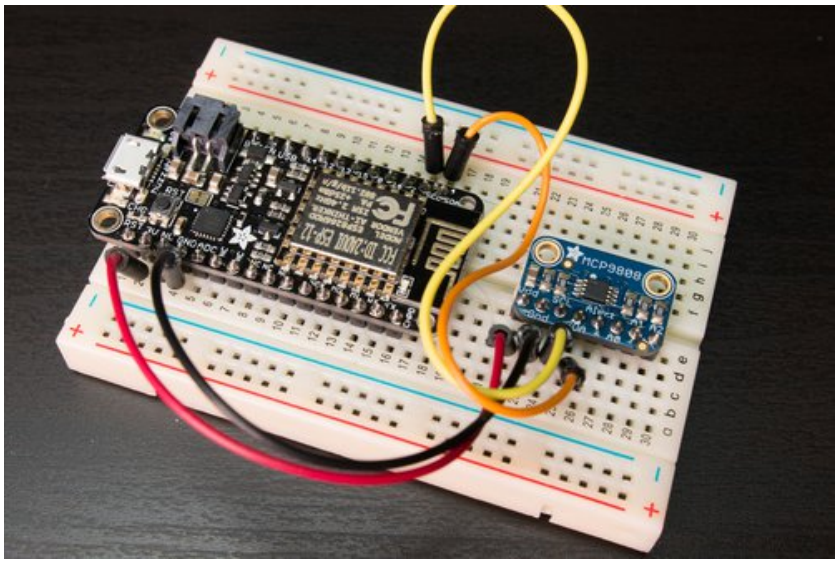
MicroPython Hardware: I2C Devices

Created by Tony DiCola

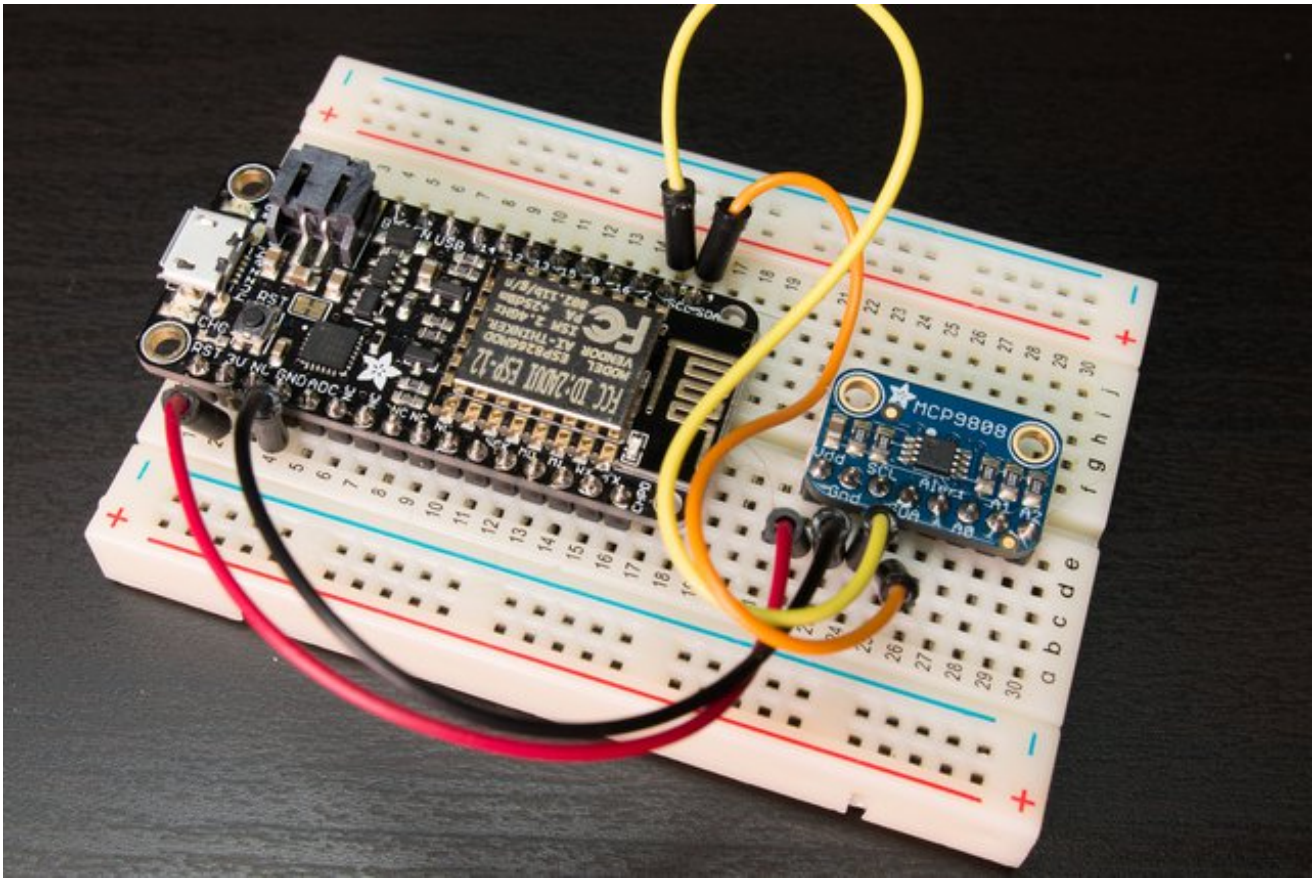


Last updated on 2017-01-27 10:52:59 AM UTC

Guide Contents

Overview

Note this guide was written for MicroPython.org firmware and not Adafruit CircuitPython firmware.



The I2C protocol (http://adafru.it/u2a) is a way for multiple devices to talk to each other using just two wires, a clock and data line. Typically you use I2C to talk to devices like sensors, small displays, PWM or motor drivers, and other devices. Each device shares the same clock and data line and the I2C protocol allows them to talk to each other without interferance. With MicroPython you can talk to I2C devices with a few lines of Python code!

This guide will explore how to read data from aMCP9808 I2C temperature sensor (http://adafru.it/e06) using MicroPython. You'll learn about basic I2C operations and how to perform them with MicroPython. Using this knowledge you can talk to any I2C device and build interesting MicroPython-powered projects!

Before you get started you'll want to be familiar with the basics of using MicroPython by reading these guides:

- MicroPython Basics: What is MicroPython? (http://adafru.it/pMb)
- MicroPython Basics: How to Load MicroPython on a Board (http://adafru.it/pNB)
- MicroPython Basics: Blink a LED (http://adafru.it/q2d)
- MicroPython Basics: Load Files & Run Code (http://adafru.it/r2B)

I2C Master

The I2C protocol is a way for multiple devices to communicate with each other using just two wires, a clock and data line. Unlike SPI or similar protocols there is no single 'master' which controls the clock and communication with each device. Instead any device can be the master that takes control of the I2C clock and data lines to communicate with other devices. Each I2C device is assigned a unique address that's used to identify it during read and write operations. When a device sees its address sent on the I2C bus it responds to the request, and when it sees a different address it ignores it. Using unique addresses many devices can share the same I2C bus without interferance.

With MicroPython you can interact with devices on an I2C bus using a few Python functions. In most cases you'll act as an I2C 'master' that reads and writes data with other I2C devices. However on some boards like the pyboard you can also act as an I2C 'slave' or peripheral that's assigned an address and can listen for and respond to requests from other I2C devices.

Be aware with MicroPython there are some differences in how each board implements the I2C API. As always consult each board's documentation:

- MicroPython ESP8266 I2C documentation (http://adafru.it/roB)
- MicroPython pyboard I2C documentation (http://adafru.it/roC)
    - The pyboard currently uses an older pyb module instead of the machine module for the I2C interface. You also need to explicitly tell the I2C API that you're creating a I2C master/client device bus. The functions for sending and receiving data have slightly different names and signatures compared to the other boards (like mem_read vs. readfrom_mem).
- MicroPython WiPy I2C documentation (http://adafru.it/roD)
    - The WiPy I2C API also requires you tell it you're creating a I2C master/client device bus.
- MicroPython micro:bit I2C documentation (http://adafru.it/roE)
    - The micro:bit I2C API is much simpler than other boards and doesn't support as many operations as described in this guide.

Note I2C protocol support across MicroPython boards is under development and might change over time. Be sure to check your board's documentation to find the latest information on I2C usage!

Hardware

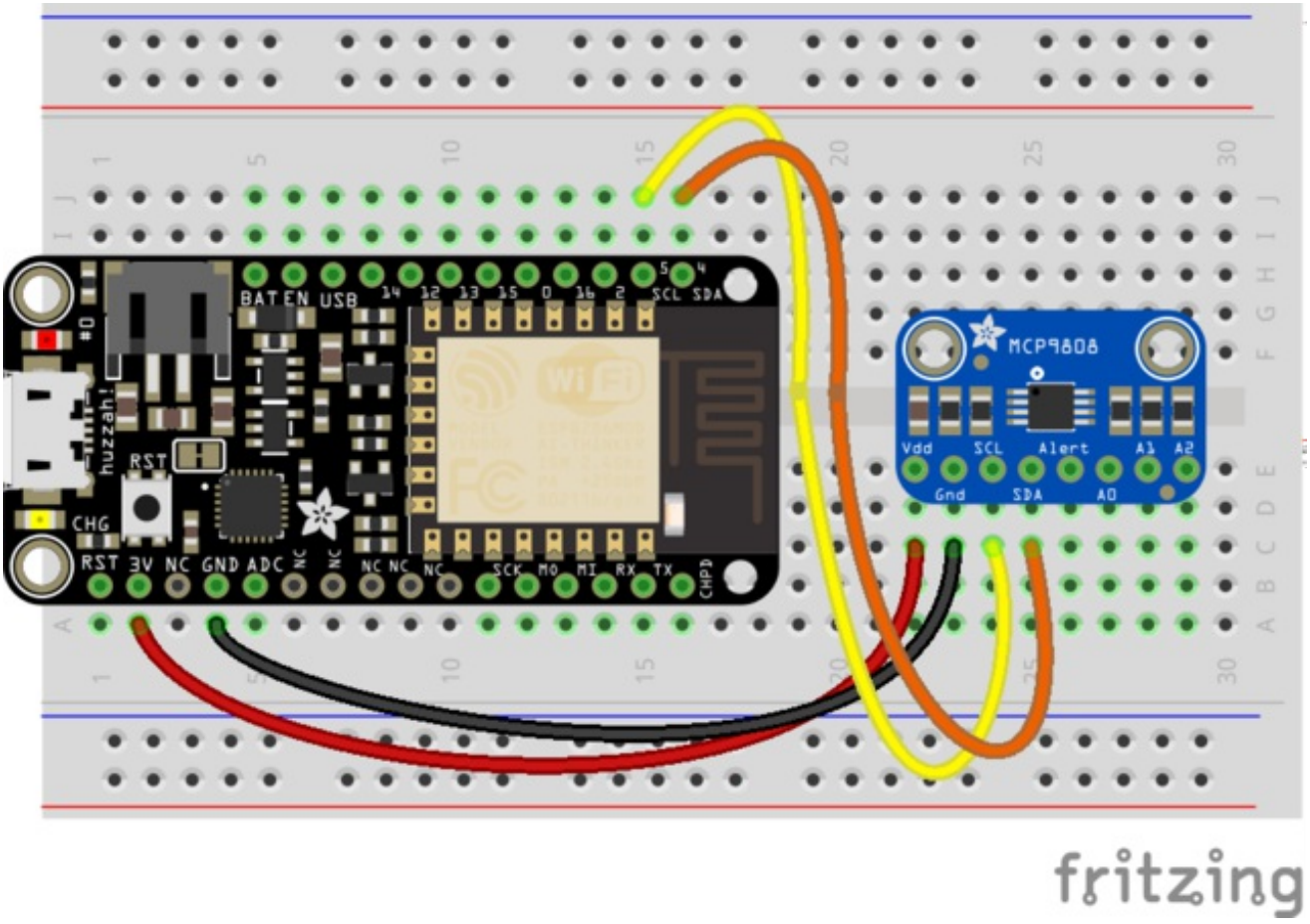
To demonstrate acting as an I2C master this guide will look at how to read an I2C temperature sensor, the MCP9808 (http://adafru.it/e06), using MicroPython. This sensor has a simple I2C interface that is similar to many other I2C devices and will help demonstrate basic I2C master usage.

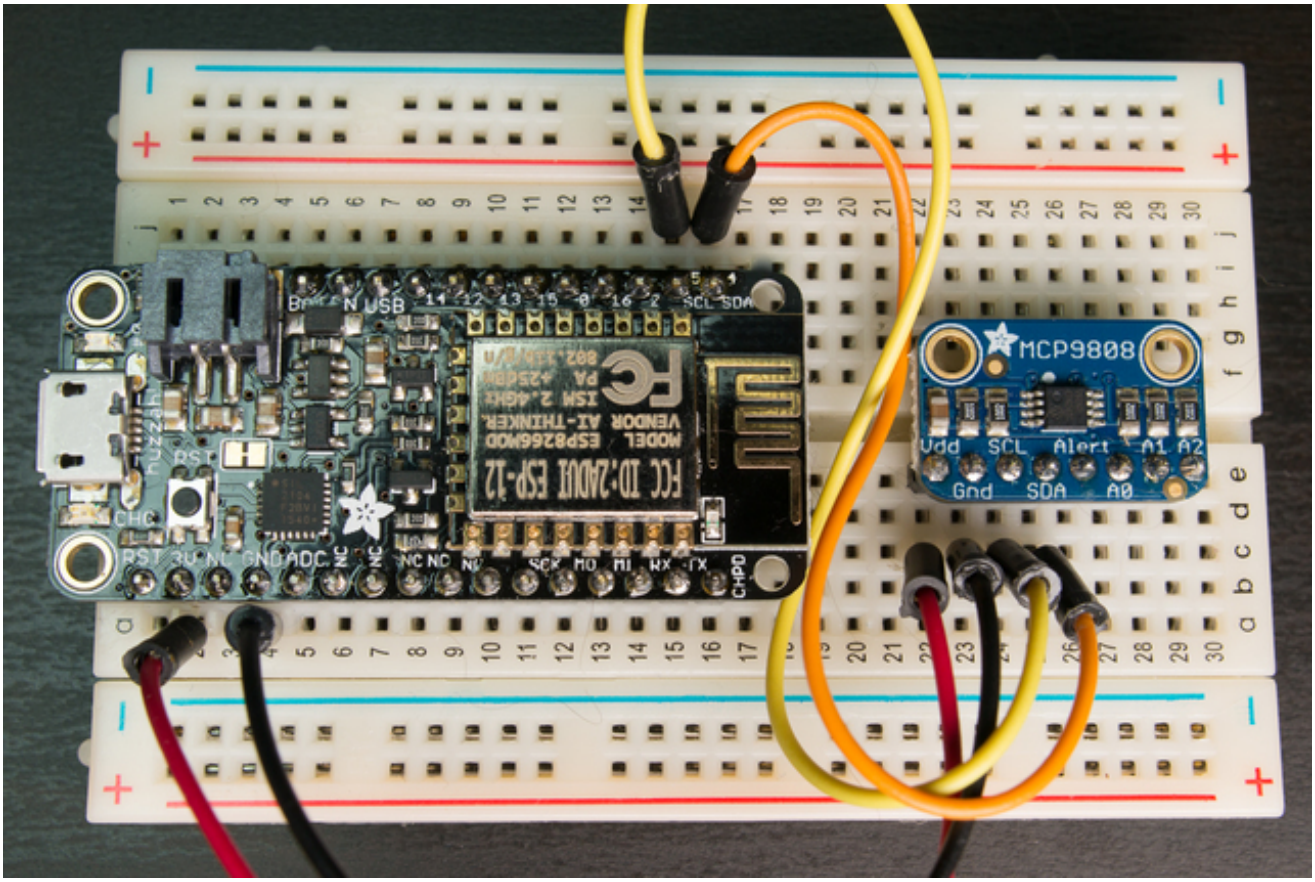
This guide will also focus on the ESP8266 MicroPython port, but you should be able to use similar code on other MicroPython boards. Where other boards differ look for notes in bold that call out the differences.

To follow this guide you'll need the following hardware:

- **ESP8266 board running MicroPython**, like the Feather HUZZAH ESP8266 (http://adafru.it/n6A). Other boards should work too but be sure to check your board's documentation for any differences in its I2C usage.
- **MCP9808 high accuracy I2C temperature sensor (http://adafru.it/e06).**
- **Breadboard (http://adafru.it/keP)** and **jumper wires (http://adafru.it/153)**.

Connect the components as shown below:





- **Board 3.3V power** to **MCP9808 Vdd**.
- **Board ground** to **MCP9808 ground**.
- **Board SCL** / **GPIO #5** to **MCP9808 SCL**.
- **Board SDA** / **GPIO #4** to **MCP9808 SDA**.

With the ESP8266 it implements I2C using software 'bit-banging' so you can use other GPIO pins instead of the #5 and #4 pins above. Other boards might not have this flexibility so be sure to check your board's documentation (http://adafru.it/pXc).

I2C Setup

To use I2C with MicroPython first you'll need to initialize access to the I2C bus. Connect to the board's serial or other REPL and run the following commands:

```
import machine
i2c = machine.I2C(machine.Pin(5), machine.Pin(4))
```

```
>>> import machine
>>> i2c = machine.I2C(machine.Pin(5), machine.Pin(4))
>>>
```

These commands will import the machine module which contains the API for hardware access. Then an instance of the I2C class is created by passing to its initializer the following parameters:

- **SCL pin**
- **SDA pin**

Remember to pass each pin as an instance of the machine Pin class, not as a number!

pyboard note: The pyboard currently uses an older pyb module instead of the machine module for I2C access, be sure to read the pyboard documentation (http://adafru.it/roC) for more details.

pyboard & WiPy note: The pyboard and WiPy need an extra parameter to explicitly specify I2C master/client mode. In addition these boards need to be told the hardware I2C bus ID number instead of explicit SCL & SDA pins (since these boards don't support software I2C on arbitrary GPIO pins). Be sure to read each board's documentation for more details.

micro:bit note: The micro:bit I2C interface is a bit simplified and doesn't need to be initialized in the same manner as other boards. Check out the micro:bit I2C documentation (http://adafru.it/roE) for more details.

Once the I2C bus is setup you can run a command that will scan for any devices connected to it and return their address. This is useful to confirm the devices you expect are able to talk to the board. To scan for devices call the **scan** function like:

```
i2c.scan()
```

```
>>> i2c.scan()
[24]
>>>
```

The **scan** function will return a list of addresses for devices that were found on the I2C bus. With the MCP9808 connected as above you should see just one device with the address value 24.

If you don't see anything returned double check your wiring, solder connections, and the power to the MCP9808 board.

Memory/Register Operations

For many I2C sensors and devices they expose data using memory or register addresses. This means you interact with a device using both its I2C address, and the address of a

register or memory location in the device. Check your device's datasheet to understand exactly how it exposes data and the memory or register addresses to use.

For the MCP9808 first define a few variables that will hold the device and register address values used in this guide. Creating variables for these values can help simplify and make your code more readable. These values come from the MCP9808 datasheet (http://adafru.it/roF):

```
address = 24
temp_reg = 5
res_reg = 8
```

```
>>> address = 24
>>> temp_reg = 5
>>> res_reg = 8
>>>
```

To read the temperature you need to get the value of the 16-bit temperature register (address 5). With both the device address (24) and register address (5) you can read the register value by calling the **readfrom_mem** function:

```
data = i2c.readfrom_mem(address, temp_reg, 2)
print(data)
```

```
>>> data = i2c.readfrom_mem(address, temp_reg, 2)
>>> print(data)
b'\xc1\x83'
```

This function takes the following parameters:

- **Device address**, this is the address of the I2C device.
- **Memory/register address**, this is the address of the memory or register to read on the device.
- **Number of bytes to read**, this is how many bytes from the register to read and return.

The **readfrom_mem** function will return a byte string with the data retrieved from the device. The length of the byte string should match the number of requested bytes.

Note the I2C protocol acknowledges each byte is received so an operation might fail with an exception (I2C bus error) if the requested device doesn't exist or never responds!

pyboard note: The pyboard currently uses a slightly different function for reading from memory, **mem_read**. Check the pyboard documentation (http://adafru.it/roC) for more details.

Another way to read data is into a buffer using the **readfrom_mem_into** function, for example:

```
data = bytearray(2)
i2c.readfrom_mem_into(address, temp_reg, data)
print(data)
```

```
>>> data = bytearray(2)
>>> i2c.readfrom_mem_into(address, temp_reg, data)
>>> print(data)
bytearray(b'\xc1\x82')
>>>
```

This function is similar to **readfrom_mem** and takes the address and register value as the first two parameters. However the third parameter is a buffer or bytearray object which will be filled with the data received from the device. You don't need to pass in the number of bytes to read since the buffer you create knows its size and will fill itself with data.

Using the **readfrom_mem_into** function can help you save memory and increase performance by creating a buffer to receive data ahead of time. This is useful if you're reading I2C data in a loop that's constantly calling read functions. If you're just using the **readfrom_mem** function in a loop you'll actually be allocating memory to hold the results and that can impact the performance of your program. Instead try creating a buffer once before the loop and use the **readfrom_mem_into** function to fill the buffer with data as needed. This can help prevent the need to 'garbage collect' unused memory (which can be very slow) inside the loop.

Once you have the temperature register value you can pull out the signed 12-bit value in degrees Celsius with a simple Python function:

```
def temp_c(data):
    value = data[0] << 8 | data[1]
    temp = (value & 0xFFF) / 16.0
    if value & 0x1000:
        temp -= 256.0
    return temp

temp_c(data)
```

```
>>> def temp_c(data):
...     value = (data[0] << 8) | data[1]
...     temp = (value & 0xFFF) / 16.0
...     if value & 0x1000:
...         temp -= 256.0
...     return temp
...
>>> temp_c(data)
24.125
>>>
```

Notice the temperature in degrees Celsius is printed!

Try holding your finger on the MCP9808 to heat it up and then read and print the temperature again:

```
temp_c(i2c.readfrom_mem(address, temp_reg, 2))
```

```
>>> temp_c(i2c.readfrom_mem(address, register, 2))
28.125
>>>
```

You should see the temperature increase a few degrees!

In addition to reading data from memory or registers you can also write data to using the **writeto_mem** function. For example the MCP9808 has a resolution register that lets you configure the accuracy and speed of temperature measurements. By default the device uses the most accurate but also slowest measurement resolution. You can tell the device to use faster but less accurate measurements by writing a 0 value to the resolution register (address 8):

```
i2c.writeto_mem(address, res_reg, b'\x00')
```

The **writeto_mem** function is similar to **readfrom_mem** and takes the device address and memory/register address as the first two parameters. The third parameter is a byte string with the bytes to send to the device, in this case the hex value 0x00 (the \x syntax in a byte string lets you specify a raw hex value instead of a printable character).

pyboard note: The pyboard uses a slightly different function for writing to memory, **mem_write**. See the pyboard documentation (http://adafru.it/roC) for more details.

Now try reading the temperature again and notice there are less digits after the decimal point, i.e. the reading is less accurate:

```
temp_c(i2c.readfrom_mem(address, temp_reg, 2))
```

```
>>> i2c.writeto_mem(address, res_reg, b'\x00')
>>> temp_c(i2c.readfrom_mem(address, temp_reg, 2))
23.5
>>>
```

You can put the device back into its most accurate resolution by setting the register to the value 3:

```
i2c.writeto_mem(address, resolution, b'\x03')
temp_c(i2c.readfrom_mem(address, temperature, 2))
```

```
>>> i2c.writeto_mem(address, res_reg, b'\x03')
>>> temp_c(i2c.readfrom_mem(address, temp_reg, 2))
24.3125
>>>
```

That's all there is to reading and writing data with an I2C device that exposes memory & registers! You'll find almost all I2C sensor use this memory/register mode of operation. Check the device datasheet to see which register addresses and values to use when interacting with the device!

Device Operations

Some I2C devices don't expose data with memory or registers. For these devices you simply send and receive bytes of data with no associated memory/register address. To work with these devices you'll use the following functions that are similar to the register/memory operations above:

- **readfrom** function: Read a number of bytes from the device and return them as a new byte string, just like **readfrom_mem** but without the memory/register address parameter.
- **readfrom_into** function: Read a number of bytes from the device and save them in a buffer, just like **readfrom_mem_into** but without the memory/register address parameter.
- **writeto** function: Write a byte string to the device, just like**writeto_mem** but without the memory/register address parameter.

pyboard note: The pyboard device operations are slightly different and use the name**recv** and **send**. See the pyboard documentation (http://adafru.it/roC) for more details.

Also for the ESP8266 MicroPython port you can even perform 'raw' or primitive I2C operations that act at an even lower level then the device and register operations. Check out the ESP8266 I2C primitive operation documentation (http://adafru.it/roB) for more details. In most cases you'll want to stick to the higher-level device & register operations

above!

I2C Slave

Some MicroPython boards support the ability to act as an I2C 'slave' or peripheral. This means they can be assigned an I2C address and listen for requests from other I2C devices. You could use this for example if you're creating a sensor in MicroPython that exposes its data to other boards with I2C.

Currently the pyboard is the only board that supports I2C slave mode. In particular the ESP8266 MicroPython port does not currently support I2C slave mode. Check out the pyboard documentation (http://adafru.it/roC) for more details on I2C slave mode usage.