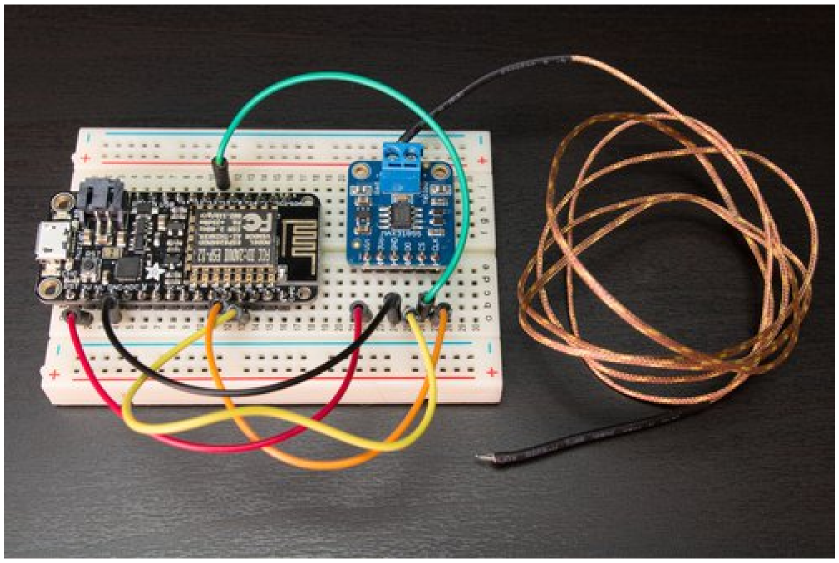
MicroPython Hardware: SPI Devices

Created by Tony DiCola

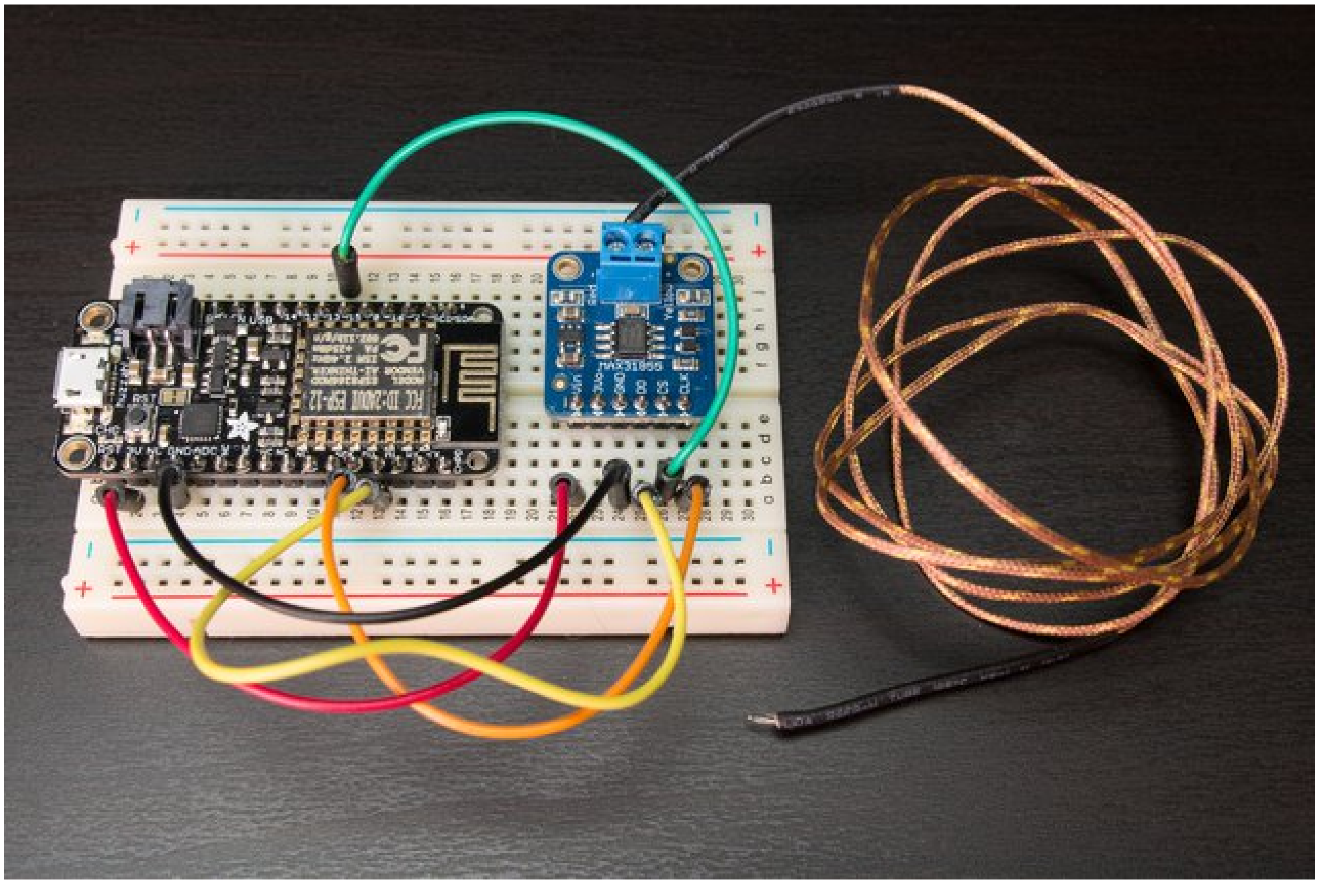


Last updated on 2017-01-27 10:52:34 AM UTC

Guide Contents

Overview

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



Serial peripheral interface (http://adafru.it/qhB), or SPI, is a protocol for two devices to send and receive data. For example a LCD display might use a SPI interface to receive pixel data, or a temperature sensor might use SPI to send readings to a microcontroller. With MicroPython you can use the SPI protocol to talk to devices and write scripts that interface with interesting hardware!

This guide will explore how to use SPI with MicroPython. In particular MicroPython on the ESP8266 will be used to talk to a simple SPI device, the MAX31855 thermocouple temperature sensor (http://adafru.it/qhC). You'll learn how to create a SPI interface, and send and receive data over that interface. With just the basics of sending and receiving SPI data you can start to write your own MicroPython code that interacts with other interesting SPI devices!

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)

SPI Master

With the SPI protocol there's an important distinction between the 'master' device which controls the communication and 'slave' devices that send and receive data with the master. In almost all cases the microcontroller is the master device that controls the SPI communication. In this example we'll look at SPI master mode where a MicroPython board acts as the master to control the SPI communication with other devices.

In most cases a SPI connection is made using at least four wires:

- **Clock**, the SPI master device toggles this line high and low to tell connected devices when they should send and receive bits of data.
- **MOSI (master output, slave input)**, this line sends bits of data from the master device to other connected devices. Think of this as the data output from the master device.
- **MISO (master input, slave output)**, this line sends bits of data from connected devices to the master device. Think of this as the data output from the connected devices.
- **Chip select**, although not required most connected devices have a chip select, or CS, line. This line is driven high or low by the SPI master to tell the connected device that it should listen for SPI commands. As long as each device has a separate chip select line you can usually share the clock, MOSI, and MISO lines of a SPI connection between multiple devices.

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

- MicroPython ESP8266 SPI documentation (http://adafru.it/r9b)
- MicroPython pyboard SPI documentation (http://adafru.it/pXb)
  - The pyboard currently uses an older pyb module instead of the machine module for the SPI interface. You also need to explicitly tell the SPI API that you're creating a SPI master device connection. The functions for sending and receiving data have slightly different names like recv instead of read and send instead of write.
- MicroPython WiPy SPI documentation (http://adafru.it/reG)
  - The WiPy SPI API also requires you tell it you're creating a SPI master device connection.
- MicroPython micro:bit SPI documentation (http://adafru.it/qhD)
  - The micro:bit uses a SPI mode parameter to set polarity and clock phase instead of explicit parameters for each value. Use the table shown in the

documentation (http://adafru.it/qhD) to understand which mode is necessary for the appropriate polarity and clock phase of your connection.

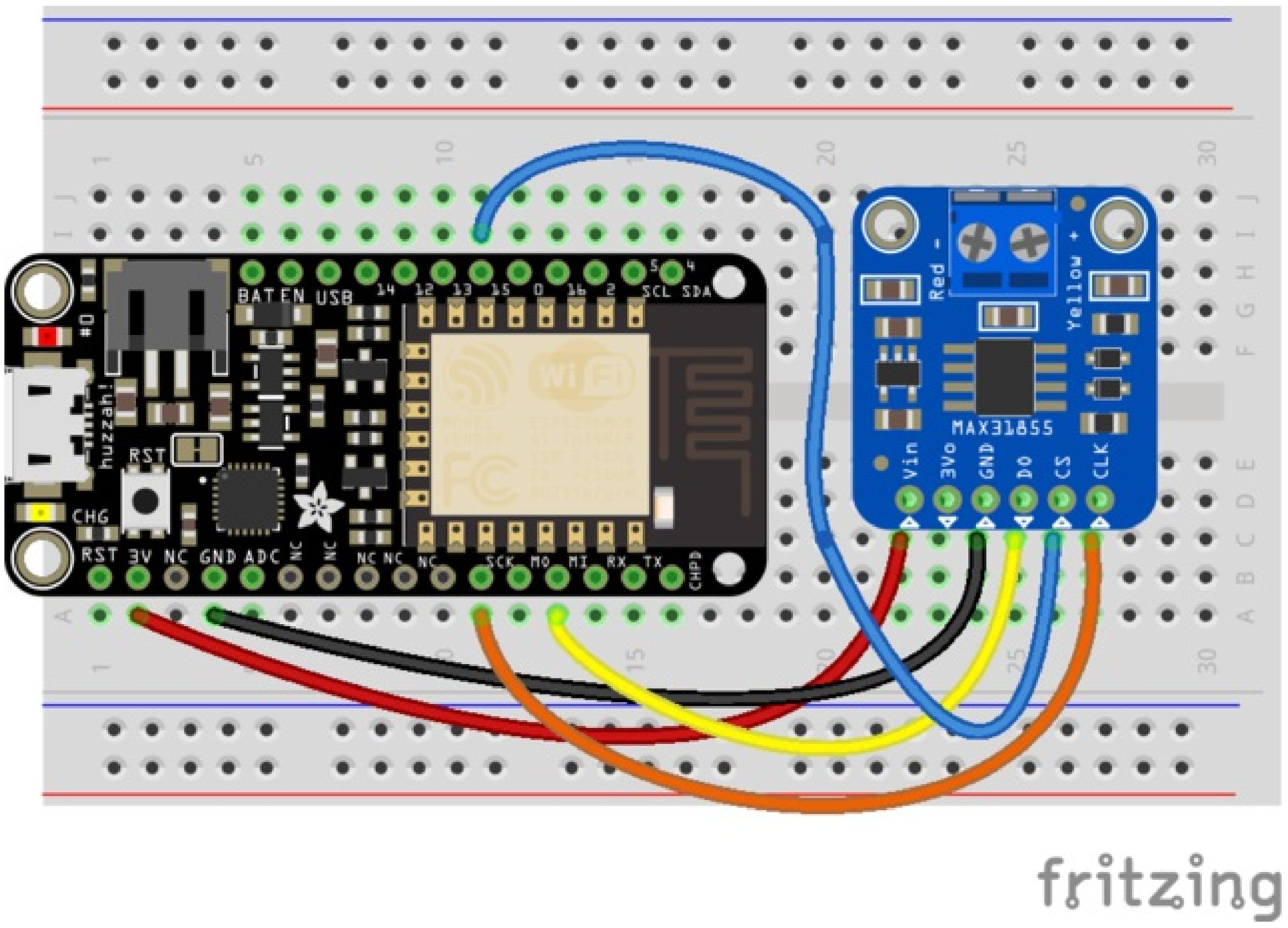
Note SPI protocol support across MicroPython boards is under development and might change over time. Be sure to check your board's documentation to find the latest information on SPI usage!

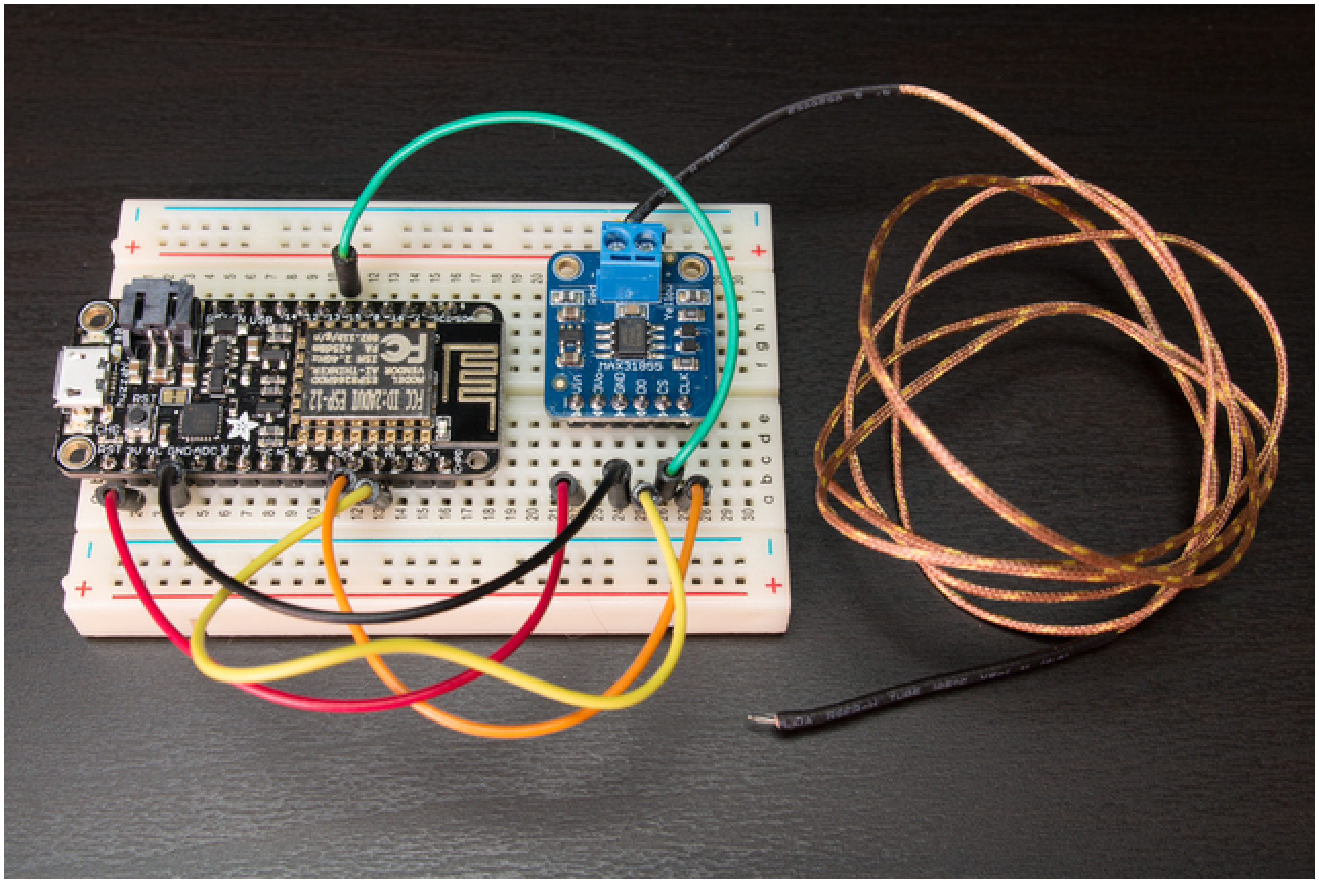
For this example we'll look at how to read data from a MAX31855 thermocouple sensor connected to an ESP8266 running MicroPython. The MAX31855 sensor has a very simple SPI interface that's easy to read temperature data with a few commands.

To follow this example you'll need the following parts:

- **ESP8266 board running MicroPython**, for example the Feather HUZZAH ESP8266 (http://adafru.it/n6A).
- **MAX31855 thermocouple sensor breakout (http://adafru.it/qhC) and a compatible thermocouple (http://adafru.it/270).**
- **Breadboard (http://adafru.it/64) & jumper wires (http://adafru.it/153).**

Connect the components as follows:


BAT EN USB
14 12 13 15 0 16 2 SCL SDA
RST 3V NC GND ADC NC NC NC NC NC SCK MO MI RX TX CHPD
huzzah!
RST
CHG
Red -
Yellow +
MAX31855
Vin 3Vo GND DO CS CLK
fritzing



- **Board ground** to **MAX31855 ground**.
- **Board 3.3V output** to **MAX31855 Vin (voltage input)**.
- **Board SCK (serial clock, ESP8266 GPIO14)** to **MAX31855 CLK (clock)**.
- **Board MI (master input, ESP8266 GPIO12)** to **MAX31855 DO (data output)**.
- **Board pin 15** to **MAX31855 CS (chip select)**.

You'll also want to make sure a thermocouple is connected to the MAX31855 breakout's terminal block. Make sure to use an appropriate thermocouple like a simple K-type thermocouple. Check out this handy thermocouple guide (http://adafru.it/qhE) if you're unfamliar with their usage.

Although it's not necessary with the MAX31855, if you're using a SPI device that should receive data from the ESP8266 you should connect the board MOSI/MO (master output, ESP8266 GPIO13) to the MOSI or data input line of your device.

Note the above connections will use the hardware SPI interface on the ESP8266 to talk to the MAX31855 sensor. Hardware SPI uses hardware built in to the microcontroller to talk the SPI protocol. This is useful because the microcontroller can handle all of the SPI communication even at very high speeds (multiple megahertz). However when using hardware SPI you have to use specific clock, MOSI, and MISO pins on the board as shown.

An alternative to hardware SPI is 'software SPI' which uses any GPIO pins for the clock, MOSI, and MISO lines. This is more flexible than hardware SPI, but it's slower because the SPI protocol is implemented in code instead of special hardware. Most MicroPython boards don't yet support software SPI so it won't be shown in this guide, however be aware the MicroPython ESP8266 port does support a software SPI interface (http://adafru.it/r9b) if you need it.

Setup Hardware SPI Connection

To setup the SPI connection connect to the board's serial or other REPL and run the following commands to import the machine module:

```
import machine
```

pyboard note: The pyboard SPI interface is currently using the older pyb module.Check the pyboard documentation (http://adafru.it/pXb) for more details on its usage.

Next create a SPI interface by running the following command:

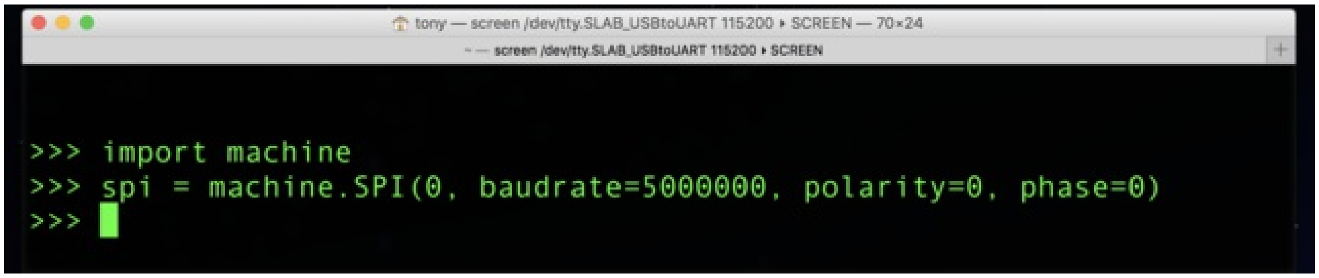
```
spi = machine.SPI(1, baudrate=5000000, polarity=0, phase=0)
```

Note this command might fail if you're running a slightly older version of MicroPython ESP8266 firmware! In particular if you see the following error:

ValueError: no such SPI peripheral

Then instead run the command as:

```
spi = machine.SPI(0, baudrate=5000000, polarity=0, phase=0)
```


tony — screen /dev/tty.SLAB_USBtoUART 115200 ▸ SCREEN — 70×24
~ — screen /dev/tty.SLAB_USBtoUART 115200 ▸ SCREEN

```
>>> import machine
>>> spi = machine.SPI(0, baudrate=5000000, polarity=0, phase=0)
>>>
```

The reason for this difference is that the first parameter of the SPI class initializer specifies which SPI interface on the board to use. Older versions of MicroPython ESP8266 used interface 0 for the hardware SPI interface, however later versions (after ~September 2016) used interface 1 for the hardware SPI interface.

pyboard & WiPy note: For the pyboard & WiPy their SPI class currently requires an explicit SPI.MASTER parameter in addition to the other parameters above. Check their documentation for more details on the usage.

micro:bit note: The micro:bit doesn't use polarity and phase parameters, instead it uses a single mode parameter that implies both polarity and phase. Check the micro:bit documentation (http://adafru.it/qhD) for more details.

The other parameters to the initializer control how the SPI interface is configured:

- **baudrate** controls the speed of the clock line in hertz.
- **polarity** controls the polarity of the clock line, i.e. if it's idle at a low or high level.
- **phase** controls the phase of the clock line, i.e. when data is read and written during a clock cycle.

You'll need to look at your device's datasheet to see what clock speed/baudrate it supports, and what polarity & phase values to specify too. Check out this description of SPI polarity and phase (http://adafru.it/qhB) for more details on what they mean, and how to convert from SPI modes (a common way to describe polarity & phase with one number) to explicit polarity & phase values.

For the MAX31855 thermocouple sensor a polarity & phase of 0 (i.e. SPI mode 0) are necessary to talk to the device, and a baudrate of 5mhz is supported for quickly reading data.

After initializing the SPI interface you'll need to also setup the chip select line as a standard digital output (http://adafru.it/qhF). Run the following command to make a digital output for the chip select line and set it to a high value:

```
cs = machine.Pin(15, machine.Pin.OUT)
cs.high()
```

```
>>> cs = machine.Pin(15, machine.Pin.OUT)
>>> cs.high()
```

For the MAX31855 it will send data only when the chip select line is low. Check your device's datasheet to see if it has a chip select line and how it expects the line to be driven.

Read SPI Data

You can read SPI data using the **read** or **readinto** functions on the SPI interface object.

For the MAX31855 sensor it has a very simple interface where you read 4 bytes of data (32 bits total) to get the current temperature reading and other sensor state. Remember before the chip will respond it needs to have the chip select line driven low. Run the following commands to toggle chip select low, read 4 bytes, and then set chip select back to a high level:

```
cs.low()
data = spi.read(4)
cs.high()
```

```
>>> cs.low()
>>> data = spi.read(4)
>>> cs.high()
```

pyboard note: On the pyboard the SPI interface currently uses the**recv** function name instead of **read**.

The **read** function takes one parameter which is the total number of bytes to read from the SPI interface. The board will drive the clock line appropriately and listen for incoming data on the MISO line, then return a byte array with the received data.

An alternate way to read data is with the**readinto** function. This function takes in a buffer of bytes you create ahead of time and will fill the buffer with received data. Using a buffer you create can help save memory and increase performance because MicroPython doesn't have to dynamically create a result buffer like with the read function above.

For example to use readinto to read the sensor you would run:

```
data = bytearray(4)
cs.low()
spi.readinto(data)
cs.high()
```

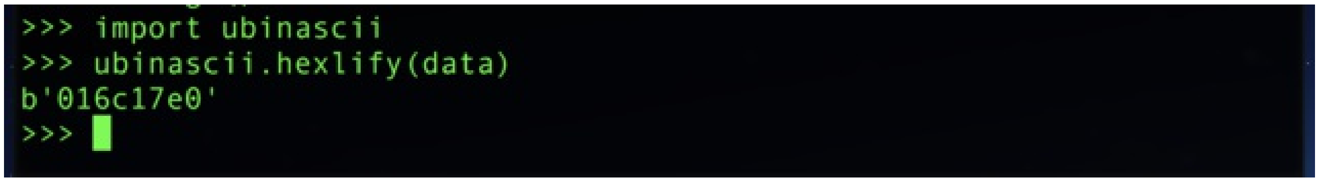
```
>>> data = bytearray(4)
>>> cs.low()
>>> spi.readinto(data)
>>> cs.high()
>>>
```

micro:bit note: The micro:bit doesn't currently support the readinto function, instead you'll have to use read.

If you're reading SPI data continually in a loop consider using readinto instead of read. By creating and managing the buffer of received data yourself it can speed up the program and reduce memory usage.

Now that the data is received you can run the following commands to print out its value in hexadecimal:

```
import ubinascii
ubinascii.hexlify(data)
```



```
>>> import ubinascii
>>> ubinascii.hexlify(data)
b'016c17e0'
>>> 
```

This will print the 4 received bytes as a hex string, like:

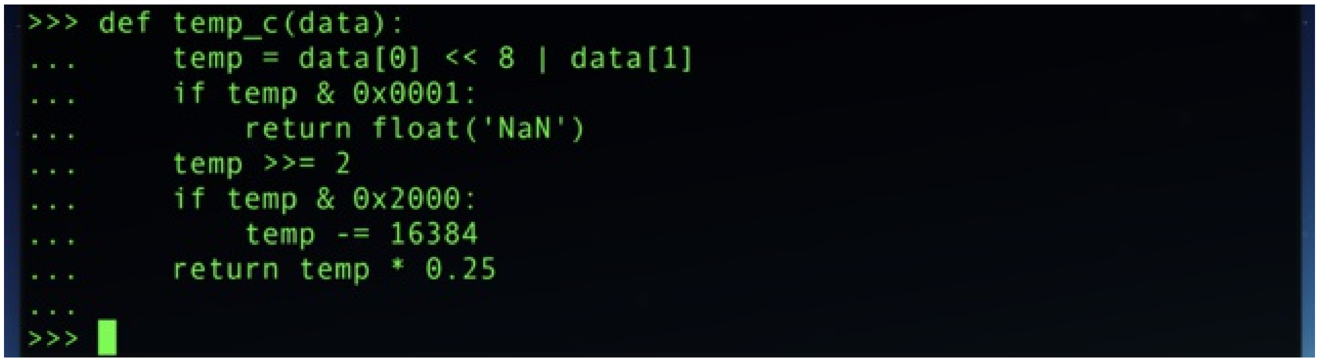
b'016c17e0'

If you only see zeros make sure the MAX31855 is connected exactly as shown above. Most likely the MISO line is not connected, or the MAX31855 isn't powered up.

The read and readinto functions are all you really need to read SPI data with MicroPython. After reading the raw data it's up to you to interpret it appropriately.

For the MAX31855 its datasheet (http://adafru.it/rfU) specifies the format of the received 32 bits and how to get the thermocouple temperature out of them. You can actually make a helper function to do this conversion from the raw data:

```
def temp_c(data):
    temp = data[0] << 8 | data[1]
    if temp & 0x0001:
        return float('NaN')  # Fault reading data.
    temp >>= 2
    if temp & 0x2000:
        temp -= 16384  # Sign bit set, take 2's compliment.
    return temp * 0.25
```

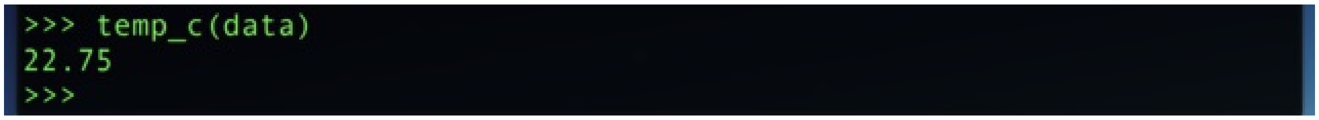


```
>>> def temp_c(data):
...     temp = data[0] << 8 | data[1]
...     if temp & 0x0001:
...         return float('NaN')
...     temp >>= 2
...     if temp & 0x2000:
...         temp -= 16384
...     return temp * 0.25
...
>>> 
```

This function just pulls out the 14 bits of signed temperature data from the received bytes. It also checks if a fault bit is set and returns a 'not a number' result to indicate a problem.

Now try calling the temp_c function on the previously received data:

```
temp_c(data)
```



```
>>> temp_c(data)
22.75
>>>
```

You should see a temperature in degrees Celsius printed!

Try holding the thermocouple tip with your hand and taking another reading (remember toggle the CS line low, then read 4 bytes, and toggle CS back high), then print the result with the temp_c function again. You should see a higher temperature from your hand heating the thermocouple!

Write & Transfer SPI Data

You can write SPI data using the**write** and **write_readinto** functions on the SPI interface object. The MAX31855 doesn't actually read any data written over SPI (it has no data input line!), but for completeness you can see how to write SPI data below.

To simply write out an array of bytes send them to the**write** function on the SPI interface:

```
spi.write(b'1234')
```

pyboard note: On the pyboard the SPI interface currently uses the**send** function name instead of **write**.

This will write the 4 byte string '1234' (actually the ASCII byte values of each character) out the MOSI line of the board (GPIO13 on the ESP8266).

You can also write data while at the same time reading data from the MISO line. This operation is sometimes called a 'transfer' in the SPI protocol and MicroPython implements it with the **write_readinto** function. For example to write a 4 byte string and read 4 bytes at the same time you can call:

```
data = bytearray(4)
spi.write_readinto(b'1234', data)
```

pyboard note: On the pyboard the SPI interface uses the**send_recv** function which returns data like the **read**/**recv** function instead of populating a buffer.

This will write the same 4 byte string '1234' out the MOSI line, while at the same time reading data on the MISO line and saving it into the provided buffer.

That's all there is to writing SPI data with MicroPython!

SPI Slave

SPI slave device support allows a board to act as a device that's controlled by a SPI master. This is handy if you need a device to look like a SPI peripheral, or if you want to control multiple devices from one master device. Each device can act as a SPI slave and the master can send and receive data with them as necessary.

In MicroPython SPI slave support is still in development for some boards. The ESP8266 port in particular does not currently support SPI slave mode. This guide won't go into detail about SPI slave support, instead check your board's documentation (http://adafru.it/pXc) to see if it has SPI slave support and learn about its usage.