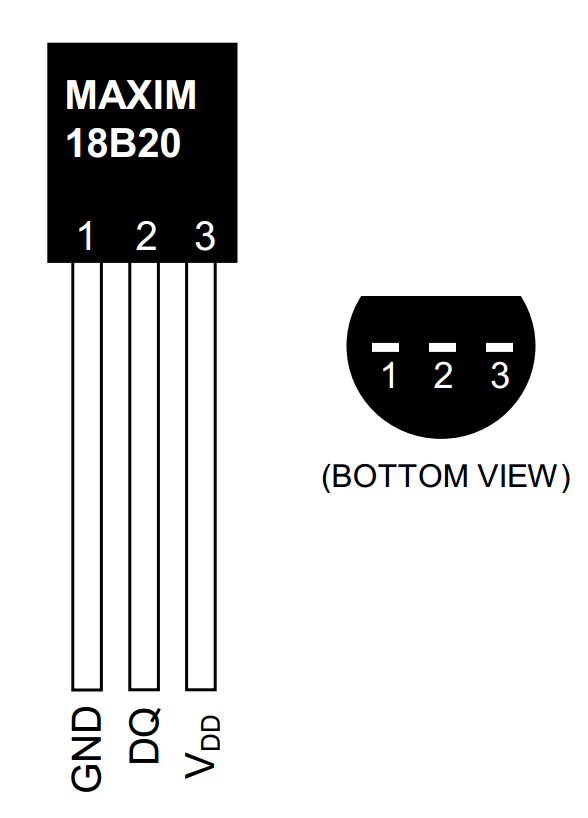
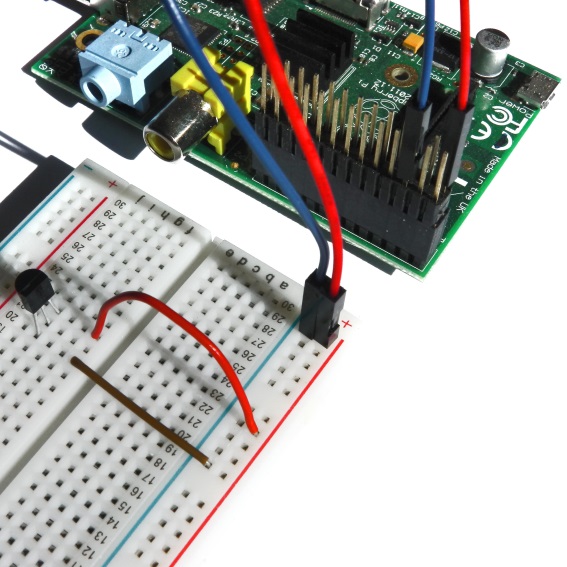
**The Raspberry Pi and 1-Wire Temperature Sensors**

In previous tutorials we’ve outlined the integration of simple sensors and switches with the Raspberry Pi. These components have had a simple on/off or high/low output, which is sensed by the Raspberry Pi. Our PIR movement sensor tutorial for example, simply says “yes, I’ve detected movement”.

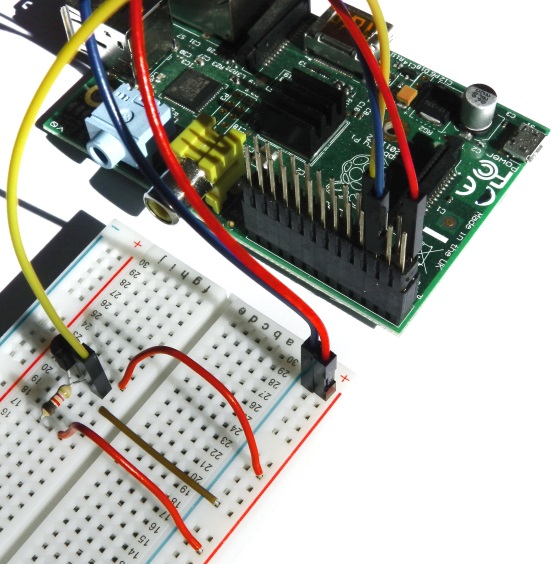
So, what happens when we connect a more advanced sensor and want to read data more complex data? In this tutorial we’ll be connecting a 1-Wire Digital Thermometer and programme our Raspberry Pi to read the output the temperature it senses!

In 1-Wire sensors, all data is sent down one wire, which makes it great for microcontrollers such as the Raspberry Pi, as it only requires one GPIO pin for sensing. In addition to this, most 1-Wire sensors will come with a unique serial code (more on this later) which means you can connect multiple units up to one microcontroller without them interfering with each other.

The sensor we’re going to use in this tutorial is the Maxim DS18B20+ Programmable Resolution 1-Wire Digital Thermometer. The DS18B20+ has a similar layout to transistors called the TO-92 package, with three pins: GND, Data (DQ), and 3.3V power line (VDD). You’ll also need some jumper wires, a breadboard and a 4.7kΩ (or 10kΩ) resistor.

The resistor in this setup is used as a 'pull-up' for the data-line, and should be connected between the DQ and VDD line. It ensures that the 1-Wire data line is at a defined logic level, and limits interference from electrical noise if our pin was left floating. We’re also going to be using ***GPIO 4 [Pin 7]*** as the driver pin for sensing the thermometer output. This is the dedicated pin for 1-Wire GPIO sensing.

**Hooking it up**

1. Connect ***GPIO GND [Pin 6]*** on the Pi to the negative rail on the breadboard and connect ***GPIO 3.3V [Pin 1]*** on the Pi to the Positive rail on the breadboard.
2. Plug the DS18B20+ into your breadboard, ensuring that all three pins are in different rows. Familiarise yourself with the pin layout, as it’s quite easy to hook it up backwards!
3. Connect ***DS18B20+ GND [Pin 1]*** to the negative rail of the breadboard.
4. Connect ***DS18B20+ VDD [Pin 3]*** to the positive rail of the breadboard.
5. Place your 4.7kΩ resistor between ***DS18B20+ DQ [Pin 2]*** and a free row on your breadboard.
6. Connect that free end of the 4.7kΩ resistor to the positive rail of the breadboard.
7. Finally, connect ***DS18B20+ DQ [Pin 2]*** to ***GPIO 4 [Pin 7]*** with a jumper wire.

That’s it; we’re now ready for some programming!

**Programming**

With a little set up, the DS18B20+ can be read directly from the command line without the need for any Python programs. However, this requires us to input a command every time we want to know the temperature reading. In order to introduce some concepts for 1-Wire interfacing, we’ll access it via terminal first, and we’ll then write a Python programme which will read the temperature automatically at set time intervals.

The Raspberry Pi comes equipped with a range of drivers for interfacing. However, it’s not feasible to load every driver when the system boots, as it will increase the boot time significantly and use a considerable amount of system resources for redundant processes. These drivers are therefore stored as loadable modules and the command ***modprobe*** is employed to boot them into the Linux kernel when they’re required. The following two commands load the 1-Wire and thermometer drivers on GPIO 4.

sudo modprobe w1-gpio

sudo modprobe w1-therm

We then need to change directory ***cd*** to our 1-Wire device folder and list ***ls*** the devices in order to ensure that our thermometer has loaded correctly.

cd /sys/bus/w1/devices/

ls

In the device drivers, your sensor should be listed as a series of numbers and letters. In this case, the device is registered as 28-000005e2fdc3. You then need to access the sensor with the ***cd*** command, replacing our serial number with your own.

cd 28-000005e2fdc3

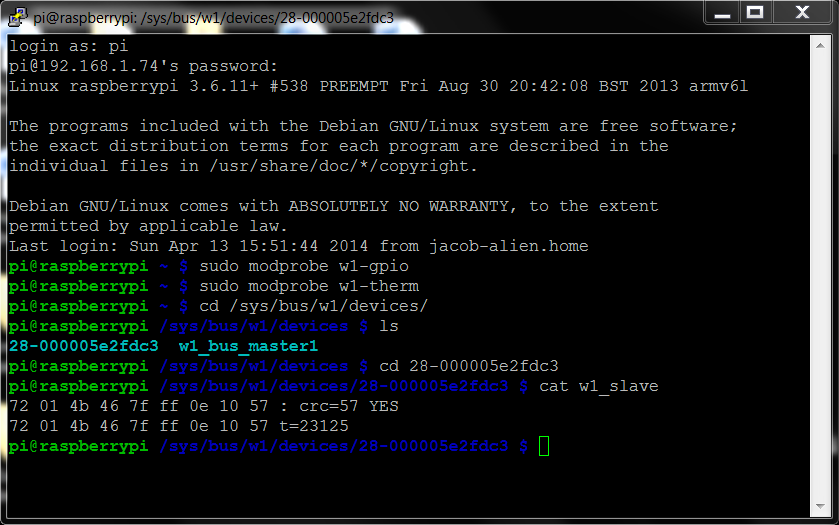
The sensor periodically writes to the w1\_slave file, so we simply use the ***cat*** command to read it.

cat w1\_slave

This yields the following two lines of text, with the output ***t=*** showing the temperature in degrees Celsius. A decimal point should be placed after the first two digits e.g. the temperature reading we’ve received is 23.125 degrees Celsius.

72 01 4b 46 7f ff 0e 10 57 : crc=57 YES

72 01 4b 46 7f ff 0e 10 57 t=23125



In terms of reading from the module, this is all that’s required from the terminal. Try holding onto the thermometer and taking another reading! With these commands in mind, we can write a Python program to output our temperature data automatically.

**Python Program**

Our first step is to import the required modules: ***os*** allows us to enable our 1-Wire drivers and interface with our sensor, and ***time*** allows our Raspberry Pi to define time, and enables the use of time periods in our code.

import os

import time

We then need to load our drivers:

os.system('modprobe w1-gpio')

os.system('modprobe w1-therm')

The next step is to define our sensor’s output file (the w1\_slave file) as defined above. Remember to utilise your own temperature sensor’s serial code!

temp\_sensor = ‘sys/bus/w1/devices/28-000005e2fdc3/w1\_slave’

We then need to define a variable for our raw temperature value (temp\_raw); the two lines outputted by the sensor demonstrated with our terminal example. We could simply print this statement now. However, we’re going to process it into something more useable. So, we open, read, record and then close our temperature file. We use the return function here, in order to recall this data at a later stage in our code.

def temp\_raw():

f = open(temp\_sensor, 'r')

lines = f.readlines()

f.close()

return lines

First, we check our variable from the previous function for any errors. If you study our original output as defined in the terminal example, we get two lines of code (Line 0 = 72 01 4b 46 7f ff 0e 10 57 : crc=57 YES); we ***strip*** this line except for the last three digits, and check for the “YES” signal, indicating a successful temperature reading from the sensor. In Python, not-equal is defined as “!=”, so here we’re saying whilst the reading does not equal YES, sleep for 0.2s and repeat.

def read\_temp():

lines = temp\_raw()

while lines[0].strip()[-3:] != 'YES':

time.sleep(0.2)

lines = temp\_raw()

Once the program is happy that the YES signal has been received, we proceed to our second line of output code (Line 1 = 72 01 4b 46 7f ff 0e 10 57 t=23125). We ***find*** our temperature output “t=”, check it for errors, ***strip*** the output of the “t=” phrase to leave just the temperature numbers, and run two calculations to give us the figures in Celsius and Fahrenheit.

temp\_output = lines[1].find('t=')

if temp\_output != -1:

temp\_string = lines[1].strip()[temp\_output+2:]

temp\_c = float(temp\_string) / 1000.0

temp\_f = temp\_c \* 9.0 / 5.0 + 32.0

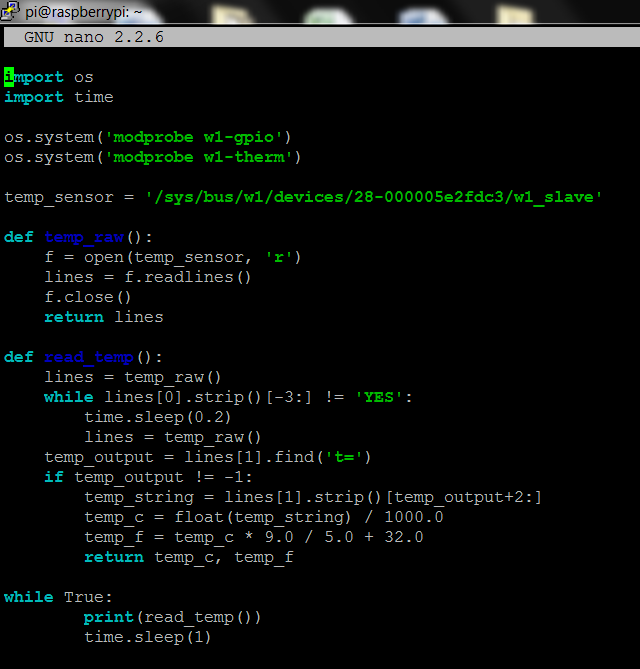
return temp\_c, temp\_f

Finally, we loop our process and tell it to output our temperature data every 1 second.

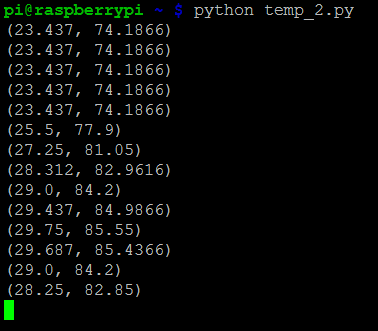
while True:

print(read\_temp())

time.sleep(1)

So that’s our code! 

Save your program, and run it to yield the temperature output:



DS18B20+ sensors can be run in parallel, and accessed using their unique serial directories. The Python example above can be edited to access and read from multiple sensors! As always; theDS18B20+sensor and all components are available separately or as part of our workshop kit from the ModMyPi website.