Raspberry Pi temperature sensor

Written by Matthew Kirk.

In this tutorial, we will be building a circuit to connect a temperature sensor to our Raspberry Pi, and writing a program to read the sensor data. The circuit we will build is going to connect to the Raspberry Pi using the GPIO pins.

GPIO Explained

GPIO stands for General Purpose Input/Output. General purpose because all they are is simple connections that can be either high or low, a binary choice. This means we can easily do things that involves binary choices, and it will still be nice and simple to understand what is going on. In this tutorial, we are going to be turning LEDs off and on, and checking whether buttons are being pressed—all very binary actions, which makes them ideally suited for GPIO pins.

However, this is not all that GPIO pins can be used for. A large range of hardware devices can communicate by transmitting binary data along a connection, and this can also go straight onto the GPIO. If we have the right program to work out what the data means, we can do even more complicated things. We are going to be making use of a temperature sensor that sends the temperature information into a GPIO pin, and some software to convert this binary data into a useful format.

Once we have built our circuit, the next step is to write a program to read the temperature, and give it to us in a nice format. Once that is done, we will add some LEDs and a button to our circuit, and find out how to measure button presses and turn the LEDs on and off from our program.

Finally, we will put all these bits together to make a button controlled temperature logger—a program that will measure the temperature every second and put this in a file, which can be started and stopped by the press of a button.

Step One: Updating the Kernel

- The first step is to change where our Pi updates from, by editing a text file. We need to open up the file `/etc/apt/sources.list.d/raspi.list` as root, so type:
  `sudo leafpad /etc/apt/sources.list.d/raspi.list`
- Now change the line in this file so that it reads "`deb http://archive.raspberrypi.org/debian/ wheezy main untested`", then save and close the file.
- Next, do the following commands:
  `apt-get update`
  `apt-get upgrade`

Kernel Updates

What we are doing here is making the Pi updte itself using the very latest kernel. The kernel is the lowest level bit of software, that makes the connection between the actual hardware, and all the other software that can run. Once the Pi knows it can update itself from this brand new place, we upgrade the kernel. This new kernel has all the extra code it needs to access the temperature sensor we will be using.

Step Two: Connecting the Temperature Sensor

- Turn off the Pi—it is generally a bad idea to connect wires while there is electricity running through the pins.
- Obtain:
  - An AdaFruit ribbon cable connector.
  - A DS18B20 temperature sensor (or this version if you want to be able to move your sensor around).
  - A 4.7kΩ resistor, a breadboard, and a selection of wires.
  - Solder, and a soldering iron.
- Solder the ribbon connector parts together, and attach the ribbon cable connector to the Pi and breadboard.
- Connect pin 3 on the sensor to the 3.3V GPIO pin (labelled 3V3 on the AdaFruit connector).
- Connect pin 1 to the ground GPIO pin (labelled GND on the AdaFruit connector).
- Connect pin 2 to the GPIO pin 4 (labelled #4 on the AdaFruit connector).
- Put the 4.7kΩ resistor between pin 2 and pin 3 of the temperature sensor.
- Turn the Pi on, then put your finger against the sensor. If it is connected the wrong way round, then it will get very hot within a second or two, in which case, turn the Pi off once it has booted, and wait a while (it was still hot after 10 minutes when I did it) for it to cool down, and then take out the sensor and put it back in again the right way round.
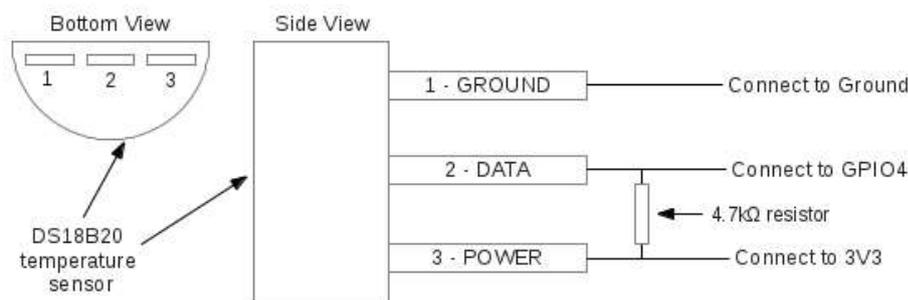
Ribbon Cable

The AdaFruit ribbon cable connector saves us from having to attach wires directly to the Raspberry Pi board, as this would just be really fiddly. It also has better labelling on the GPIO pins, which makes it easier to work with.
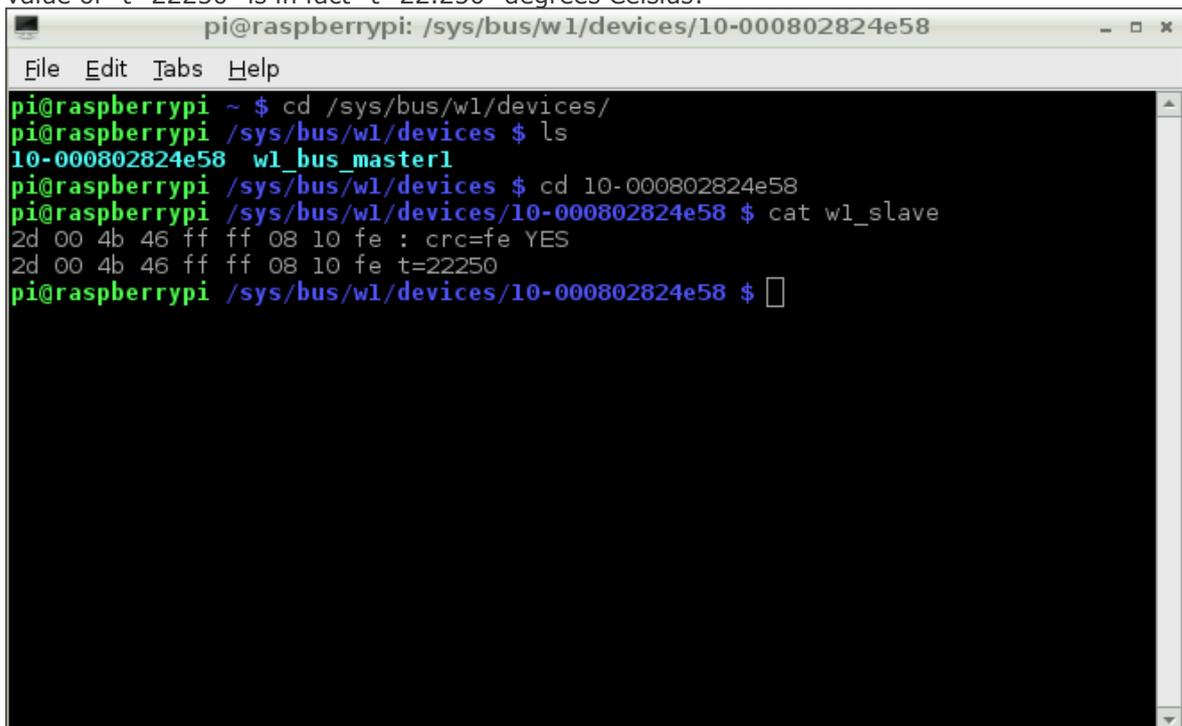
The temperature sensor we are using here, the DS18S20, has three legs. Two of these pins put the sensor in a circuit between a high voltage pin and ground. This connection provides the power for the sensor to measure the temperature. The third pin connects to a GPIO pin, and it is with this connection that we can communicate with the sensor. The resistor that we have put between this data pin and the high voltage pin is what is known as a pull-up resistor—unless the data leg grounds itself to send a 0 along the wire, its voltage on the wire will be "pulled-up" to 3.3V. This makes it simpler for the sensor to communicate with our Pi, as to send a 1 signal, it doesn't have to do anything at all.

Step Three (A): Reading the Sensor from the Terminal

- Turn on the Pi.
- Once a user is logged on to the Pi, type these commands into a terminal, or just at the prompt given upon login and before typing "startx":
  ```
  sudo modprobe w1-gpio
  sudo modprobe w1-therm
  cd /sys/bus/w1/devices/
  ls
  ```
- The entry on the screen that is mostly numbers is the serial number of the sensor. The sensor used for this tutorial has the serial number "10-000802824e58". Replace the serial number in the next command with the one for the sensor used.
  ```
  cd 10-000802824e58
  cat w1_slave
  ```
- Two lines of text will be printed. On the second line, the section starting "t=" is the temperature in degrees Celsius. A decimal point goes after the first two digits, so the example value of "t=22250" is in fact "t=22.250" degrees Celsius:

Modprobe Explained

The first step here was to run the command `modprobe`. This command can load individual modules that do particular things into the kernel, so we can use them. If you want to avoid typing the `modprobe` commands every time you start up your Pi, you will need to tell Linux that you want these modules to be loaded automatically when it starts. This can be done by opening up the file `/etc/modules`—you need to be root to do this, so type `sudo leafpad /etc/modules` into the terminal. Now add the lines "w1-gpio" and "w1-therm" to the end of this file, save it and close.

The business with finding the right folder and then reading the file to get the temperature might seem reasonably simple, but as you might guess, there is a bit of work being hidden here. What actually happens when you try and read what is in the `w1_slave` file is that the Pi sends a message to the temperature sensor asking it what the temperature is. The sensor spends half a second or so working it out and sending it back (which is why there is a slight delay after you enter the command), at which point the Pi puts this information in the file and gives it back to you.

Try opening the file like any other, in the graphical file manager interface. Open up the file browser, go to the directory the file is in and open up the file in Leafpad. The same text as before will be displayed.

Step Three (B): Reading the Temperature from Python

- Open a terminal and type `python`.
- At the python IDLE command prompt, type the commands below, ignoring the lines starting with #—they explain what the commands do:

```
# Open the file that we viewed earlier so that python can see what is in it.
Replace the serial number as before.
tfile = open("/sys/bus/w1/devices/10-000802824e58/w1_slave")
# Read all of the text in the file.
text = tfile.read()
# Close the file now that the text has been read.
tfile.close()
# Split the text with new lines (\n) and select the second line.
secondline = text.split("\n")[1]
# Split the line into words, referring to the spaces, and select the 10th word
(counting from 0).
temperaturedata = secondline.split(" ")[9]
# The first two characters are "t=", so get rid of those and convert the
temperature from a string to a number.
temperature = float(temperaturedata[2:])
# Put the decimal point in the right place and display it.
temperature = temperature / 1000
print temperature
```

New Python Commands

There are quite a few Python commands in this section, which might be unfamiliar. In the Python IDLE terminal, type `help(something)` and a description of whatever was put between the brackets will display and some general information about what can be done with the command. For example, with `tfile.read()`, there exists also `tfile.readline()` and `tfile.readlines()`, which as the name suggests do different things. If the variable assignment (the `temperature =` bit) is not included in the lines written above, the result of the `.split` method will be output.

In the commands just now, the text is slowly broken down until just the temperature values remain, with several lines of code. This many lines of code is not necessary, `temperaturedata = text.split("\n")[1].split(" ")[9]` would also work, or splitting by spaces from the start, not newlines. The code may be harder to read if it is squashed onto one line, however. This operation can be made faster—play around with it, and if a quicker way becomes apparent, maybe a classmate can work out what it does?

See here my example code: **How to read the temperature in Python.**

```
#Copyright (c) 2012 Matthew Kirk
# Licensed under MIT License, see
# http://www.cl.cam.ac.uk/freshers/raspberrypi/tutorials/temperature/LICENSE

tfile = open("/sys/bus/w1/devices/10-000802824e58/w1_slave")
text = tfile.read()
tfile.close()
temperature_data = text.split()[-1]
temperature = float(temperature_data[2:])
temperature = temperature / 1000
print temperature
```

Step Four: Write a Script to Print Out the Temperature.

- Install a text editor such as Geany or SciTE or use the preinstalled Leafpad or equivalent.
- Open the editor and copy the code written above (or the version with the different splitting that was suggested).
- Save the file as `temperature.py` in `/home/pi`.
- Open a terminal and run **python temperature.py**. The program should output the temperature as it did before.

Writing Scripts

What we've done here is put the commands we typed out earlier into a script—so that we can easily run the program without having to type in all the commands again.

Here, it is advised to install a different text editor for writing code. The reason for this is that the default text editor—Leafpad—that the Pi comes with isn't really designed for writing code in. If you open up your Python script in IDLE and in Leafpad, you'll see that IDLE has added colour to the code—this is called syntax highlighting. This is something that almost all text editors designed for writing code in do, because it makes it easier to see what is happening in the program. SciTE or Geany are recommended in this tutorial. SciTE is a bit quicker to open files, and looks simpler to start with, whereas Geany has lots more features that start to be useful when your program get a bit longer, such as listing all the names of variables in your program down the side, which makes it easier to remember what names have been used.

Step Five: Connecting an LED

- Turn off the Pi.
- Obtain an LED, a 220Ω resistor, and probably some more wires.
- Pick one of the numbered pins on the ribbon cable that connects your Pi to the breadboard and connect the long leg (on the opposite side to the flat edge of the LED base) of the LED to this pin.
- Connect the 220Ω resistor between the ground pin of the Pi and the other (longer) leg of the LED.

Circuit Building

What we have done here is set up a circuit from one of the GPIO pins, through the LED and resistor, down to ground. With the Pi, we can switch on a specific GPIO pin, so it has a high voltage, and current will flow through the LED and resistor. The resistor we've put in stops too much current flowing, which could heat up the LED so much that it would burn out.

Step Six: Turning On the LED

- Turn on the Pi.
- Open a root terminal from the menu, or by running `sudo su` in a normal terminal.
- Now, type these commands, where "18" in this example is replaced by whichever pin the LED is connected to:

```
cd /sys/class/gpio/
echo "18" > export
cd gpio18
echo "out" > direction
echo "1" > value
```

- The LED should now be turned on.
- To turn the LED off, type:

```
echo "0" > value
cd ..
echo "18" > unexport
```

Powering our Circuit

We have opened up a root terminal here to control the LED. Why did we have to do this here, but not when we were working with our temperature sensor? With the temperature sensor, all we could do was ask for the temperature, which is pretty harmless. With the LED, we wanted to change something about the hardware (make the GPIO pin we had chosen into an output), which could lead to bad things happening if we changed something we shouldn't have, or changed the wrong pin number. With the Raspberry Pi, it is designed in a way so that you can't break anything just by changing the wrong GPIO pin, but the restriction is left over from more expensive computers that are more easily broken.

Once we've open a root terminal, first we have to tell the Pi that we want to do something with a particular GPIO pin, which we do by using `echo` to write the pin number into the export file. The Pi notices we've done this, and sets up a folder for that pin to allow us to control it. Then we tell the Pi that we want this pin to be an output pin, again using `echo`, so that we can turn the LED on and off. Writing the number "1" to the value file makes the pin turn on, so that current flows out of that pin. Once we've got our LED working, we turn it off by `echo`ing "0", and then telling the Pi that we are done working with that pin by writing the pin number to the unexport file.

We've just seen how to turn on and off the LED using the terminal, so now it's time to make it work with Python. The program needs to write to a file. Here's an example of writing "Hello" to a file named "example". Build upon that.

```python
# Open the file and write to it, hence "w"
f = open("example", "w")
f.write("hello")
f.close()
```

Run this program in a terminal using `sudo python` for the reasons explained above. It should turn the LED on.

See here my example code: **How to turn on and off an LED from Python.**

```
# Copyright (c) 2012 Matthew Kirk
# Licensed under MIT License, see
# http://www.cl.cam.ac.uk/freshers/raspberrypi/tutorials/temperature/LICENSE

export_file = open("/sys/class/gpio/export", "w")
export_file.write("18")
export_file.close()

direction_file = open("/sys/class/gpio/gpio18/direction", "w")
direction_file.write("out")
direction_file.close()

value_file = open("/sys/class/gpio/gpio18/value", "w")
value_file.write("1")
value_file.close()

value_file = open("/sys/class/gpio/gpio18/value", "w")
value_file.write("0")
value_file.close()

unexport_file = open("/sys/class/gpio/unexport", "w")
unexport_file.write("18")
unexport_file.close()
```

Step Seven: There is an Easier Way!

We are going to install a library for Python that will let us turn on GPIO pins, without having to go through the hassle of opening and closing lots of files like in the last exercise.

- Install the package with `sudo apt-get install python-rpi.gpio` in a terminal.
- Launch Python as root with `sudo python` and type:

```
import RPi.GPIO as GPIO
# Use the pin numbers from the ribbon cable board.
GPIO.setmode(GPIO.BCM)
# Set up the pin you are using ("18" is an example) as output.
GPIO.setup(18, GPIO.OUT)
# Turn on the pin and see the LED light up.
GPIO.output(18, GPIO.HIGH)
Turn off the pin to turn off the LED.
GPIO.output(18, GPIO.LOW)
```

Python Libraries

This library that we have installed is just a piece of code, this time written in the C programming language, that hides away all the detail of what needs to be done to get the GPIO pins turned on and off. When the Python command GPIO.setup(18, GPIO.OUT) library goes away and does all the things that we did before—writing the pin number to the export file, and writing "out" to the file called direction but they are just hidden away.

The setmode command tells the library that we are labelling our pins like they are on the ribbon cable board. The board refers to the pins in terms of what they do, which is why the pins have

labels like "3V3", "GND", "5V0". The alternative, which would be done with GPIO.setmode(GPIO.BOARD) is to number the pins in the same way the house numbering on a street is done—the pins on one side have the numbers 1,3,5 all the way up to 25, and the pins on the other side have the numbers 2,4,6 all the way up to 26.

Step Eight: Connecting a Button

- Here, a push-to-make button and a 1kΩ will be needed.
- Pick another free input pin and connect this pin to one side of the button, and the other side of the button to the 3.3V pin.
- Put the resistor beteen the input pin and the ground pin on the Pi.

Button Connection

The type of button we are using here is called push-to-make, which means that the two sides of the switch are not connected unless you are pushing it. Push-to-break buttons exist too, which are always connected unless the button on them is pushed. We are also using a resistor to connect the input GPIO pin to ground, which makes sure that the input is fixed to off when the button isn't pressed, rather than the voltage floating around somewhere between high and low. We need a larger resistance than we might normally use for other things, to stop current flowing straight through to ground when the button is pressed.

Step Nine (A): Checking for the Button Press from the Terminal

- Turn the Pi on.
- Open a root terminal.
- Type in the following, replacing "17" with the pin your button is connected to:

```
cd /sys/class/gpio/
echo "17" > export
cd gpio17
echo "in" > direction
cat value
```

- If the button is not pressed, this code should print "0".
- Hold the button down and run the program again: the number "1" should have printed out.
- Now the testing is complete, type:
-

```
cd ..
echo "17" > unexport
```

Button Bash-ing

As we did with the LEDs, we have to tell the Pi that we want to do something with a particular GPIO pin, which we do by using echo, just like earlier. Then we tell the Pi that we want this pin to be an input pin, by us echoing "in" rather than "out", so that we can measure whether or not the button is pressed. When we read the file "value", we are checking whether the input is high or low (1 or 0 respectively). Finally, the "unexport" the pin as before.

Step Nine (B): Checking for the Button Press Using Python

- Launch Python as root, and type:

```
import RPi.GPIO as GPIO
# Use the pin numbers from the ribbon cable board
GPIO.setmode(GPIO.BCM)
# Set up this pin as input.
GPIO.setup(17, GPIO.IN)
# Is the button pressed or not?
GPIO.input(17)
# Hold down the button, run the command again. The output should be "true".
GPIO.input(17)
```

The instructions above have shown how to check for a button press. How about putting this into a loop, so that every time the button is pressed, something happens, like a message printing to the screen. However, the Pi will be able to print out a message and move on to checking the button again much faster than a finger can be taken off the button. So, your program will have to sit and wait for some seconds, maybe with a "while" loop and the "pass" command. Once this works, try to make the program the temperature whenever the button is pressed.

See here my example code: **How to check a button press from Python.**

```
# Copyright (c) 2012 Matthew Kirk
# Licensed under MIT License, see
# http://www.cl.cam.ac.uk/freshers/raspberrypi/tutorials/temperature/LICENSE

import RPi.GPIO as GPIO

GPIO.setmode(GPIO.BCM)
GPIO.setup(17, GPIO.IN)

while True:
        if GPIO.input(17):
                tfile = open("/sys/bus/w1/devices/10-000802824e58/w1_slave")
                text = tfile.read()
                tfile.close()
                temperature_data = text.split()[-1]
                temperature = float(temperature_data[2:])
                temperature = temperature / 1000
                print temperature
        while GPIO.input(17):
                pass
```

Step Ten: Writing a Temperature Logging Program

The final step in this tutorial is putting together all the different things we can do to make a temperature logging program—something that will record the temperature to a file for as long as we want.

- We need another LED, and another 220Ω resistor. The two LEDs will be used to show when our program is ready to start measuring temperature, and when it is actually logging it to a file. To make this more obvious, a different colour LED would be useful here.
- With these extra parts, wire the LED up to another free pin, just like the first LED is.
- Open up text editor, and follow these steps:
  - Import the RPi.GPIO library.
  - Set up the three GPIO pins (the button and two LEDs) as input and outputs.
  - Turn on one of the LEDs to indicate we are ready to start logging.
  - Use the code from the previous exercise to check for a button press, and wait for the button to be un-pressed.
  - When the button is pressed, we want our first LED to turn off and the other to come on, to indicate that our Pi is now taking data.
  - Next, open up a file which we can write, to give us somewhere to put the data.
  - Add another while loop to wait until the button is pressed again, and in the loop read the temperature in the same way we've done before.
  - Instead of printing the temperature, we will write the temperature to the file which we opened earlier. One change we will need to make from how we learnt to write to files earlier is that now, we want each temperature value to be on a newline. This can be done by adding "\n" to the end of the string we are writing, like `str(temperature) + "\n"`.
  - When the last while loop exists, we want to close the file, as we are finished with it, and we do this with `datafile.close()`, and turn off the LED.

See here my example code: **A complete temperature logger.**

```
# Copyright (c) 2012 Matthew Kirk
# Licensed under MIT License, see
```

```
# http://www.cl.cam.ac.uk/freshers/raspberrypi/tutorials/temperature/LICENSE

import RPi.GPIO as GPIO

GPIO.setmode(GPIO.BCM)
GPIO.setup(17, GPIO.IN)
GPIO.setup(18, GPIO.OUT)
GPIO.setup(25, GPIO.OUT)

GPIO.output(18, GPIO.HIGH)

while True:
        if GPIO.input(17):
                break

while GPIO.input(17):
        pass

GPIO.output(18, GPIO.LOW)
GPIO.output(25, GPIO.HIGH)

datafile = open("temperaturedata.log", "w")

while True:
        tfile = open("/sys/bus/w1/devices/w1_bus_master1/10-000802824e58/w1_slave")
        text = tfile.read()
        tfile.close()
        temperature_data = text.split()[-1]
        temperature = float(temperature_data[2:])
        temperature = temperature / 1000
        datafile.write(str(temperature) + "\n")
        if GPIO.input(17):
                break

datafile.close()
GPIO.output(25, GPIO.LOW)
```

Tutorial over! Enjoy your temperature monitoring, and see below for Appendices.

Appendices

Programs can almost always be improved, perhaps by adding some more code so that it can do something new, or just by changing what is there to make it better, or more readable. This is what this section is all about—pointing out things that don't work as well as they could, or things which would be better done in a different way.

Appendix One: Name of the Data File

The same filename is always used for the script, so old data disappears everytime there is some more. Filenames can be made unique with "temperaturedata1.log", "temperaturedata2.log", but a more helpful way is to add what is known as a timestamp to the name. To do this in Python:

- Import the "time" module into Python, find out the current time for the logging file and put it in an easy to read form, then append this to the file name, like so:

```
import time
# This displays the time in the format year-month-day, hour-minute-second.
timestamp = time.strftime("%Y-%m-%d-%H-%M-%S")
# Put together the different bits of the filename into one, and open the file.
filename = "".join(["temperaturedata", timestamp, ".log"])
datafile = open(filename, "w", 1)
```

Fancy Functions

There are quite a few more complicated bits in this new bit of code. The `strftime()` function returns the time formatted in a particular way, a list of which can be found in the Python documentation. The format in the example means that if the files are sorted alphabetically, the newest files come last and the oldest first.

The join function is a function that all strings have, that allows us to take any list of strings and quickly join them together, putting whatever string we are using to call join on between every item in the list. Here we use an empty string so that the bits just get put straight together. This is generally the quickest way to connect a bunch of text together in Python.

You will also see that the command to open the file has gained an extra option, the "1" at the end. If you run the old code we had, and look in your file browser while doing so, you'll notice that the file appears as soon as we start logging, but is listed as having a size of zero bytes until we press the button to stop logging. This happens because Python stores all the data in memory somewhere, and only actually puts it down on the SD card when you close the file. The issue with this behaviour is that, if we were to suddenly pull the plug, or stop the program running, we would lose all the data we had collected, because none of it was down on the card. By adding the option "1" to our open command, we tell Python that we would like it to only keep a single line of text in memory at a time, and that it should put write each bit of text to the SD card when the line ends.

See here my example code: **Making the data file timestamped.**

```
# Copyright (c) 2012 Matthew Kirk
# Licensed under MIT License, see
# http://www.cl.cam.ac.uk/freshers/raspberrypi/tutorials/temperature/LICENSE

import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BCM)
GPIO.setup(17, GPIO.IN)
GPIO.setup(18, GPIO.OUT)
GPIO.setup(25, GPIO.OUT)

GPIO.output(18, GPIO.HIGH)

while True:
        if GPIO.input(17):
                break
```

```
while GPIO.input(17):
        pass

GPIO.output(18, GPIO.LOW)
GPIO.output(25, GPIO.HIGH)

timestamp = time.strftime("%Y-%m-%d-%H-%M-%S")
filename = "".join(["temperaturedata", timestamp, ".log"])
datafile = open(filename, "w", 1)

while True:
        tfile = open("/sys/bus/w1/devices/10-000802824e58/w1_slave")
        text = tfile.read()
        tfile.close()
        temperature_data = text.split()[-1]
        temperature = float(temperature_data[2:])
        temperature = temperature / 1000
        datafile.write(str(temperature) + "\n")
        if GPIO.input(17):
                break

datafile.close()
GPIO.output(25, GPIO.LOW)
```

Appendix Two: Changing the GPIO pins

Another idea is that up to here, the pin numbers for our two LEDs, button, and temperature sensor have been put straight into the program as needed. Which works fine, but what happens when we move stuff about on the board? Suddenly, everything is connected to a different GPIO pin, and all our clever instructions are being sent to the wrong things—we'll be checking for a push of an LED, and trying to light up a button. Fixing this means going though the whole of our program, checking for each time we used a pin number, and changing each of them to the new one. So instead, how about we fix this properly?

▪   At the beginning of the script, put in a few variables called, say, LED1_GPIO_PIN, LED2_GPIO_PIN, and BUTTON_GPIO_PIN. Give them the value of the appropriate pin number. Then, replace all the occurrences of actual pin numbers in the code with these variables.

So now if we change the position of the pins on the Pi, all we have to do is change the value of these new variables, and everything will still work.

Why Capitals?
In this new bit of code, capital letters are used for the variable names. This is recognised by most programmers as meaning that these variables are constants that aren't supposed to change during the course of running the code.

See here my example code: Cleaning up our code.

```
# Copyright (c) 2012 Matthew Kirk
# Licensed under MIT License, see
# http://www.cl.cam.ac.uk/freshers/raspberrypi/tutorials/temperature/LICENSE

import RPi.GPIO as GPIO
import time

LED1_GPIO_PIN = 18
LED2_GPIO_PIN = 25
BUTTON_GPIO_PIN = 17
```

```
GPIO.setmode(GPIO.BCM)
GPIO.setup(BUTTON_GPIO_PIN, GPIO.IN)
GPIO.setup(LED1_GPIO_PIN, GPIO.OUT)
GPIO.setup(LED2_GPIO_PIN, GPIO.OUT)

GPIO.output(LED1_GPIO_PIN, GPIO.HIGH)

while True:
        if GPIO.input(BUTTON_GPIO_PIN):
                break

while GPIO.input(BUTTON_GPIO_PIN):
        pass

GPIO.output(LED1_GPIO_PIN, GPIO.LOW)
GPIO.output(LED2_GPIO_PIN, GPIO.HIGH)

timestamp = time.strftime("%Y-%m-%d-%H-%M-%S")
filename = "".join(["temperaturedata", timestamp, ".log"])
datafile = open(filename, "w", 1)

while True:
        tfile = open("/sys/bus/w1/devices/10-000802824e58/w1_slave")
        text = tfile.read()
        tfile.close()
        temperature_data = text.split()[-1]
        temperature = float(temperature_data[2:])
        temperature = temperature / 1000
        datafile.write(str(temperature) + "\n")
        if GPIO.input(BUTTON_GPIO_PIN):
                break

datafile.close()
GPIO.output(LED2_GPIO_PIN, GPIO.LOW)
```

Appendix Three: Logging at a variable rate

With our program as it currently is, the temperature is always measured as fast as possible, which is about once every three-quarters of a second (this is basically how long it takes the sensor itself to measure the temperature, rather than any slow working by the Pi). If we are measuring temperature over 30 seconds, or a few minutes perhaps, this sampling rate is fine. But if we wanted to look at how the temperature varied over an hour, or even a day, then we would end up with thousands, maybe tens of thousands of readings, which would change quite slowly. In this case, it would be nice to be able to make our program only make measurements at much larger intervals.

If we want to do this, we need to make our program wait for the right amount of time between measurements, so that they are done at the right intervals. Using the function `time` in the Python `time` module, we can work out how long has passed since we last asked for the temperature, and then compare this to how long we want to wait between measurements.

▪   So in your code, add in a variable that stores the time just before you read the temperature, and at the end of the while loop. You can compare these two times, and if the difference is less than the wait we want, make the program wait for however long it needs to make up the difference. In Python, making your program wait for a while is as simple as using the `sleep` command in the `time` module, and giving it the length of time you want to sleep for.

But there is a problem with this. When the program is sleeping, it isn't paying attention to anything else—in particular, not the button which tells our program to stop logging. This means to stop our program, as it currently is, requires you to press the button during the small amount of time when the Pi is not sleeping or reading the temperature. To solve this, we will make the Pi do lots of small sleeps, and check the button between each of them.

- Replace the single `sleep` with a `for` loop, that will sleep for 0.1 seconds however many times is required to make up the difference, and check for the button press after each little sleep. If the button is pressed, you will want to use the `break` command to exit the `for` loop, but first, set some variable to `True`, so that we can test this variable in the outer `while` loop, and break out of that is it is `True`. Don't forget to define this variable as `False` before the loop starts, to stop Python becoming confused.
- Your code for this section will probably look a bit like:

```
if (time_2 - time_1) < measurement_wait:

    no_of_sleeps = int(round((measurement_wait - (time_2 - time_1)) / 0.1))
    for i in range(no_of_sleeps):

        time.sleep(0.1)
        if GPIO.input(BUTTON_GPIO_PIN):

            button_pressed = True
            break

        if button_pressed:

            break
```

See here my example code: **Allowing for a variable measurement rate**

```
# Copyright (c) 2012 Matthew Kirk
# Licensed under MIT License, see
# http://www.cl.cam.ac.uk/freshers/raspberrypi/tutorials/temperature/LICENSE

import RPi.GPIO as GPIO
import time

LED1_GPIO_PIN = 18
LED2_GPIO_PIN = 25
BUTTON_GPIO_PIN = 17
GPIO.setmode(GPIO.BCM)
GPIO.setup(BUTTON_GPIO_PIN, GPIO.IN)
GPIO.setup(LED1_GPIO_PIN, GPIO.OUT)
GPIO.setup(LED2_GPIO_PIN, GPIO.OUT)

GPIO.output(LED1_GPIO_PIN, GPIO.HIGH)

while True:
        if GPIO.input(BUTTON_GPIO_PIN):
                break

while GPIO.input(BUTTON_GPIO_PIN):
        pass

GPIO.output(LED1_GPIO_PIN, GPIO.LOW)
GPIO.output(LED2_GPIO_PIN, GPIO.HIGH)

timestamp = time.strftime("%Y-%m-%d-%H-%M-%S")
```

```
filename = "".join(["temperaturedata", timestamp, ".log"])
datafile = open(filename, "w", 1)

measurement_wait = 5
button_pressed = False
while True:
        time_1 = time.time()
        tfile = open("/sys/bus/w1/devices/10-000802824e58/w1_slave")
        text = tfile.read()
        tfile.close()
        temperature_data = text.split()[-1]
        temperature = float(temperature_data[2:])
        temperature = temperature / 1000
        datafile.write(str(temperature) + "\n")
        time_2 = time.time()
        if (time_2 - time_1) < measurement_wait:
                no_of_sleeps = int(round((measurement_wait - (time_2 - time_1)) /
0.1))
                for i in range(no_of_sleeps):
                        time.sleep(0.1)
                        if GPIO.input(BUTTON_GPIO_PIN):
                                button_pressed = True
                                break
        if button_pressed:
                break


datafile.close()
GPIO.output(LED2_GPIO_PIN, GPIO.LOW)
```

Where Next?

Now we've got a complete temperature sensor program, what can we do next? Well, having written all this code just to measure temperature, wouldn't it be good if we could just write some small piece of code to use a different type of sensor, and then have this just work?

Fortunately for you, this is exactly what the next page is all about. So let's move on and explore how we can start using plugins.

My code as python files is found zipped in a separate folder M_Kirk_temp_sensor_code.zip

All code licensed under the MIT license.