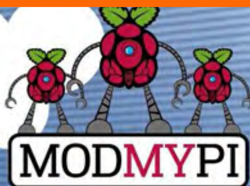


THE RASCLOCK
Get yours today from ModMyPi

Raspberry Pi timekeeping with a real time clock

Jacob Marsh

ModMyPi

DIFFICULTY : BEGINNER

In order to achieve its miniature size and low price tag, several non-essential items usually found on a desktop computer had to be omitted from the Raspberry Pi. Laptops and computers keep time when the power is off by using a pre-installed, battery powered 'Real Time Clock' (RTC). An RTC module is not included with the Raspberry Pi, which instead updates the date and time automatically over the internet via Ethernet or WiFi. Subsequently, your Pi will revert back to the standard date and time settings when the network connection is removed. For projects which have no internet connection, you may want to add a low cost battery powered RTC to help your Pi keep time!

The RasClock has been specifically designed for use with the Raspberry Pi and plugs directly in to the Raspberry Pi's GPIO Ports. This article will walk you through its installation!

Step 1 - plug it in!

To avoid any damage to the module, make sure your Raspberry Pi is switched off and the RTC battery is firmly seated before installation. Plug the coin battery into the RTC by matching the positive on the battery with the positive on the module and then plug the RTC into the Raspberry Pi's GPIO pins. It sits on the 6 GPIO pins at the SD card end of the Raspberry Pi.

Step 2 - set-up

This RTC module is designed to be used in Raspbian. So the first step is to make sure you have the latest Raspbian Operating System (OS) installed on your Raspberry Pi (http://www.raspberrypi.org/downloads).
Currently the module requires the installation of a driver that is not included in the standard Raspbian distribution; however a pre-compiled installation package is available which makes setup nice and easy.

Make sure your Pi has internet access and grab the installation package off the internet from an LXTerminal window:

```
wget
http://afterthoughtsoftware.com/files/linux-image-3.6.11-atsw-rtc_1.0_armhf.deb
```

(The wget command allows you to grab a file off the internet by providing a URL).

```
sudo dpkg -i linux-image-3.6.11-atsw-rtc_1.0_armhf.deb
```

(The dpkg command enables the management

of Debian packages. The -i installs the package, or upgrades it if it is already installed).

This may take a couple of minutes to complete.

```
sudo cp /boot/vmlinuz-3.6.11-atsw-rtc+
/boot/kernel.img
```

(The cp command stands for copy. Here, we need to copy the RTC module's boot file to the Raspberry Pi boot directory).

The next step involves editing the text in the Raspberry Pi boot files. I usually use nano text editor for these minor changes - it's basic, pre-installed and easy to master. System commands for nano are enabled by holding the CTRL key (denoted as ^ in nano) whilst pressing the relevant command e.g. CTRL+X to exit.

We need to configure Raspbian to load the RTC drivers at boot by adding the boot information to the /etc/modules configuration file:

```
sudo nano /etc/modules
```

(This will open the 'modules' file within nano text editor and allow you to make changes. To add text simply use the arrows keys to browse to the next line in the boot file and add the following text, one per line. Then exit nano (CTRL+X) and don't forget to save those changes!

```
i2c-bcm2708
rtc-pcf2127a
```

The final step in set-up is to register the RTC module when the Raspberry Pi boots and set the system clock from the RTC. When editing files always follow the instructions outlined at the top of the file denoted by #. For example, the file we are just about to edit requires any text to be put before the end of the file, denoted by 'exit 0'. Open the required file for editing:

```
sudo nano /etc/rc.local
```

For Rev 1. Raspberry Pi boards add the following text:

```
echo pcf2127a 0x51 > /sys/class/i2c-
adapter/i2c-0/new_device
( sleep 2; hwclock -s ) &
```

For Rev 2. Raspberry Pi boards add the following text:

```
echo pcf2127a 0x51 > /sys/class/i2c-
adapter/i2c-1/new_device
( sleep 2; hwclock -s ) &
```

Then reboot:

```
sudo reboot
```

Step 3 - using the RTC

After you reboot the Raspberry Pi you should be able to access the module using the hwclock command. The first time you use the clock you will need to set the time. To copy the system time into the clock module:

```
sudo hwclock –w
```

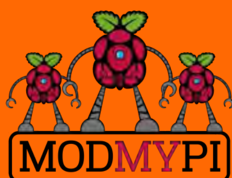
To read the time from the clock module:

```
sudo hwclock -r
```

To copy the time from the clock module to the system:

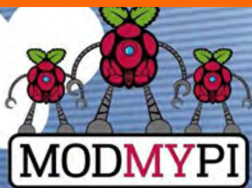
```
sudo hwclock –s
```

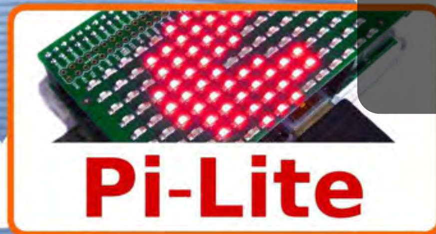
That's it... you can now keep time using your Raspberry Pi with no internet! Type hwclock into your resident search engine for a load more useful commands!



This article is sponsored by ModMyPi

All breakout boards and accessories used in this tutorial are available for worldwide shipping from the ModMyPi webshop at www.modmypi.com





THE PI-LITE
Get yours today from ModMyPi

A plug and play LED matrix board for the Raspberry Pi

Jacob Marsh

ModMyPi

DIFFICULTY : BEGINNER

The Pi-Lite is a versatile, plug and play, 126 LED (9x14 Grid) matrix display for the Raspberry Pi. Each pixel is individually addressable - so you can display scrolling text, graphics and bar graphs; basically anything that can fit in 126 pixels! It's a great starting place for doing something visual with your Raspberry Pi.

The Pi-Lite comes as a complete, fully assembled board that requires no soldering and it's designed to plug straight into the Raspberry Pi's GPIO ports. The matrix is controlled by an on-board ATMega 328 processor with pre-loaded software and works equally well with a Raspberry Pi using GPIO or with a PC, Mac or Linux machine via the on-board FTDI connector. You'll find a short beginner's guide to set the Pi-Lite up on the Raspberry Pi below.

Step 1 - setting up the Raspberry Pi for basic Pi-Lite functions!

The Pi-Lite is as Ciseco product, so requires a custom Wheezy Image to be loaded onto an SD card and used for this task. This image has reconfigured GPIO pins for serial access and the Minicom terminal emulator that's used to send and receive characters from the serial port is pre-installed. You can set all this up manually on your version of Raspbian; however for ease of this tutorial we'll use the custom image which can be downloaded at the following link:

```
http://openmicros.org/Download/2013-05-25-wheezy-raspbian-ciseco.img.zip
```

Simply unzip the image and load it onto an SD card like the standard Raspbian distribution.

Step 2 - the fun stuff!

Make sure your Raspberry Pi is switched off and then plug the Pi-Lite in. It sits on top of the GPIO ports within the footprint of the Raspberry Pi and fits neatly inside a ModMyPi case. Boot your Raspberry Pi up, log in and you'll be presented with the Raspberry Pi command line. The Pi-Lite will also auto-boot with a very cool sequence!

To access the Pi-Lite module via Minicom and send scrolling text messages, enter the command:

```
minicom -b 9600 -o -D /dev/ttyAMA0
```

Now, simply by typing, you can send any text to the Pi-Lite which will be scrolled across automatically. It's also possible to enter Minicom's command mode to change various settings, such as the scroll speed or pixel state.

To enter command mode type $$$ (three dollar signs) - which will stop all scrolling and Minicom will respond with "OK". All commands must be sent as one string in UPPER case and terminated with a carriage return (pressing enter). After receiving and carrying out a command the Pi-Lite leaves command mode and returns to scroll mode. If a command is not received within a few seconds or a command is inputted incorrectly, the command control will be terminated and the Pi-Lite will return to scroll mode.

As an example, we'll increase the scrolling speed using the SPEED command. By default the scroll speed is set to 80, but it can be set anywhere from 1 (very fast) to 1000 (very slow). Let's slow our scroll speed - simply type:

```
$$$SPEED200
```

Then hit enter. The Pi-Lite will automatically exit command mode and re-enter scroll mode. You can now check to see that your scroll speed has increased!

There's a full list of commands, as well as the example scripts utilised in Step 3 below, available at the following link. You'll need these to show bar graphs, turn on/off individual pixels, and generally make your Pi-Lite function:

```
https://www.modmypi.com/pi-lite-raspberry-pi-led-matrix
```

Step 3 - running scripts!

What's great about the Pi-Lite is that it enables you to run custom Python scripts and subsequently show graphics, repeated text strings, read the weather, run a real-time Twitter feed or display anything else you can imagine! I'll show you how to download and run some example Python scripts, but you can always edit them or write your own if you're feeling adventurous! Please note, use upper case in the commands where stated.

The Ciseco Wheezy image will already have a suitable version of Python installed. However, you'll also need to install the "Git Control System" and the "Python Serial Package":

```
sudo apt-get install git
sudo apt-get install python-serial
```

We now need to pull the library files from Github and put them in a directory. First ensure you are in your home directory by changing directory to the standard home location:

```
cd /home/pi
```

Then create a directory for the Github example files and browse to it:

```
mkdir git
cd git
```

Now obtain the Pi-Lite source code. This includes the Python examples:

```
git clone
git://github.com/CisecoPlc/PiLite.git
```

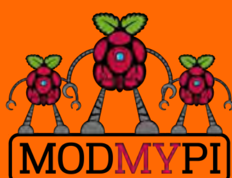
You can now browse to the example scripts:

```
cd PiLite/Python_Examples
```

Some of the scripts can be run straight from the command line via Python (CTRL+C will terminate). For this example we'll run the Pacman example script, which displays (you guessed it) Pacman on the Pi-Lite!:

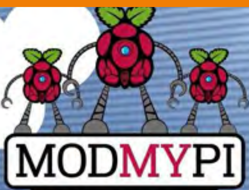
```
python Pacman.py
```

As with all Raspberry Pi projects - the best way to learn is to play and a great place to start is the Pi-Lite!



This article is sponsored by ModMyPi

All breakout boards and accessories used in this tutorial are available for worldwide shipping from the ModMyPi webshop at www.modmypi.com



Pi-Lite

Buttons and switches with the Raspberry Pi

Jacob Marsh

ModMyPi

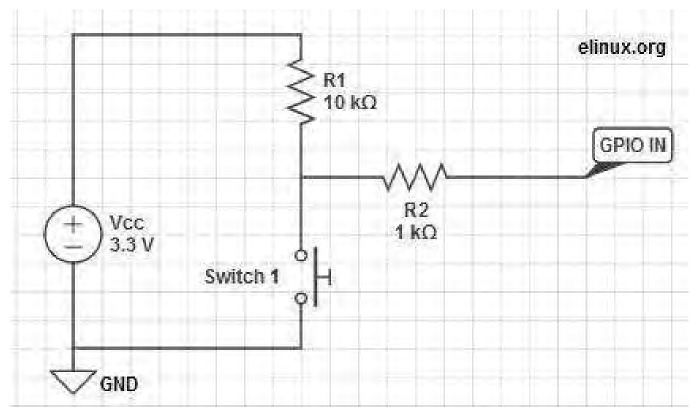
SKILL LEVEL : BEGINNER

Buttons and switches are a fundamental part of 'physical' computing. This beginner's tutorial is designed to teach the basics of physical operation and programming with the Raspberry Pi using a simple momentary switch setup.

The tutorial requires a few simple components that are available from ModMyPi (product codes in brackets):
- Medium Breadboard (BB2) – For laying out our components & circuit.
- Male to Female Jumper Wires (JW8) – For jumping between the RPi & breadboard.
- PCB Mount Switch (TAC001) – A four point basic momentary switch.
- ModMyPi's Ridiculous Resistor Kit (RK995) – to protect your Pi & calibrate the float voltage.
- 10KΩ Resistor - (Brown, Black, Black, Red, Brown)
- 1KΩ Resistor - (Brown, Black, Black, Brown, Brown)
- Breadboard Jumper Wire Kit (140KI) – For easy jumping on the breadboard.

The circuit

The purpose of this circuit is to enable the Raspberry Pi to detect a change in voltage and run a program when the button (Switch 1) is pressed. This requires three GPIO pins on our Raspberry Pi: the first will provide a signal voltage of 3.3V (Vcc), the next will ground the circuit (GND), and the third will be configured as an input (GPIO IN) to detect the voltage change.



When a GPIO pin is set to input, it doesn't provide any power and consequently has no distinct voltage level; defined as 'floating'. We need the pin to be capable of judging the difference between a high and low voltage, however in a floating state it's liable to incorrectly detect states due to electrical noise. To enable the pin to see the difference between a high or low signal we must 'tie' that pin, calibrating it to a defined value; 3.3V in this case!

To tie the input pin, we connect it to the Vcc 3.3V pin, hence when Switch 1 is open, the current flows through GPIO IN and reads high. When Switch 1 is closed, we short the circuit and the current is pulled to GND; the input has 0V, and reads low! The large R1 (10kΩ) resistor in this circuit ensures that only a

little current is drawn when the switch is pressed. If we don't use this resistor, we are essentially connecting Vcc directly to GND, which would allow a large current to flow, potentially damaging the Pi! To make the circuit even safer in case we get something wrong, we add the R2 (1kΩ) resistor to limit the current to and from GPIO IN.

The switch

Four point switches are wired in a very similar manner to two point switches. They're simply more versatile, as you can have multiple isolated inputs into the same switching point. Checking the diagrams, Pins 1 & 2 are always connected, as are Pins 3 & 4. However, both pin pairs are disconnected from each other when the button is not pressed e.g. Pins 1 & 2 are isolated from Pins 3 & 4. When the button is pressed the two sides of the switch are linked and Pins 1, 2, 3 & 4 are all connected!

In 'momentary' switches the circuit disconnects when pressure is removed from the button, as opposed to 'toggle' switches when one push connects and the next push disconnects the internal switch circuit.

Where does it all go?

WARNING. When hooking up to GPIO points on your Raspberry Pi care must be taken, as connecting the wrong points could permanently fry your Pi. Please use a GPIO cheat-sheet, and double check everything before switching it on. I will denote each GPIO point by its name, and physical location, for example GPIO P17 is actually located at Pin 11, denoted: GPIO P17 [Pin 11]. The irregularities are a result of the pin names being referenced by the on board chip rather than their physical location.

1. Connect Pi to Ground Rail. Use a black jumper wire to connect GPIO GND [Pin 6] on the Pi to the Negative rail on the breadboard – the rail on the edge of the board with the negative sign (-).

2. Connect Pi 3.3V to Positive Rail. Use a red jumper wire to connect GPIO 3.3V [Pin 1] on the Pi to the Positive rail on the breadboard – the edge rail with the positive sign (+).

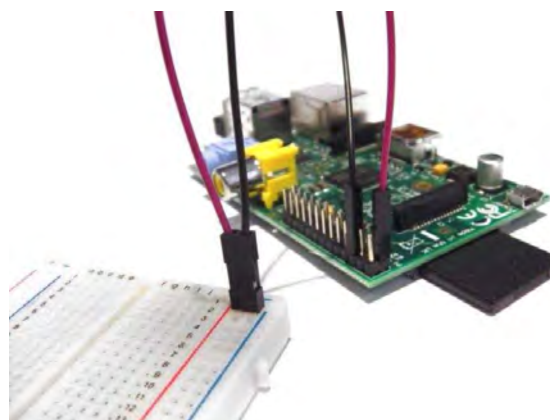
3. Plug your switch in. When breadboarding, make sure all of the legs are in separate rows. To achieve this straddle the central channel on the breadboard.

4. Add 10kΩ Resistor. Connect this from Switch Pin 1, to the positive (+) rail of the breadboard. Orientation of standard film resistors is unimportant.

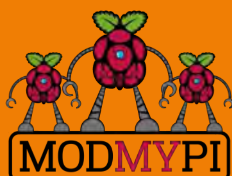
5. Connect Switch to Ground. Use a breadboard jumper wire to hook Switch Pin 3 to the ground (-) rail.

6. Connect Switch to 1kΩ Resistor. Add this resistor between Switch Pin 1 and the 10kΩ Resistor and take it to a clear rail.

7. Connect Switch to Signal Port. We'll be using GPIO P17 to detect the 3.3V signal when the switch is pressed. Simply hook up a jumper between GPIO P17 [Pin 11] on the Pi and the 1kΩ Resistor rail.

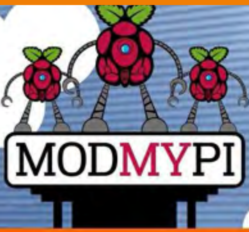


That's our circuit built! Next time, we'll write a simple program in Python to run when we press the switch!



This article is sponsored by ModMyPi

All breakout boards and accessories used in this tutorial are available for worldwide shipping from the ModMyPi webshop at www.modmypi.com



PHYSICAL COMPUTING

Buttons and switches with the Raspberry Pi - part 2

Jacob Marsh

ModMyPi

SKILL LEVEL : BEGINNER

Buttons and switches are a fundamental part of physical computing. This beginners tutorial is designed to teach the basics of physical operation with the Raspberry Pi using a simple momentary switch setup. In part one in Issue 17, we discussed and built our switch circuit. In this part, we will go through the steps of programming and interacting between the Pi and our components. The coding part of this tutorial is in Python, a widely used general purpose language. It is also very readable, so we can break down and explain the function of each line of code. The purpose of our code will be to read the I/O pin when the switch is pressed!

Using a switch

If you have not already done so, start X by typing startx and load the program IDLE3. Since we are starting a new project, open up a new window File>>New Window. Remember that Python is case sensitive and indentation is fundamental. Indentation, which is used to group statements, will occur automatically as you type commands, so make sure you stick to the suggested layout. The first line of our code imports the Python library for accessing the GPIO.

```
import RPi.GPIO as GPIO
```

Next, we need to set our GPIO pin numbering, as either the BOARD numbering or the BCM numbering. BOARD numbering refers to the physical pin numbering of the headers. BCM numbering refers to the channel numbers on the Broadcom chip. Either will do, but I personally prefer BCM numbering. If you're confused, use a GPIO cheat sheet to clarify which pin is which!

```
GPIO.setmode(GPIO.BCM)
```

Now you need to define the GPIO pins as either inputs or outputs. In Part 1, we set BCM Pin 17:BOARD Pin 11 (GPIO P17 [Pin 11]) as our input pin. So our next line of code tells the GPIO library to set this pin as an input.

```
GPIO.setup(17, GPIO.IN)
```

In part 1 of the tutorial, the input pin was tied high by connecting it to our 3.3V pin. The purpose of our Python program is to check to see if the input pin has been brought low e.g. when the button has been pressed. To check the high/low status of the pin we're going to use a True or False statement within an infinite loop. We need to tie

our True statement to the high value of our input pin. To do so we need to create a new variable called input_value and set it to the current value of GPIO P17 [Pin 11].

```
while True:
  input_value = GPIO.input(17)
```

The next step is to add a condition that will print something when the button is pressed. For this, we'll use a False condition. The input_value is False when the button is pressed and the associated signal is pulled low. This can be checked with a simple Python it statement.

```
  if input_value == False:
    print("Who pressed my button?")
```

When the button is pressed the program will now display the text: Who pressed my button?, feel free the change this to anything you want.

```
    while input_value == False:
      input_value = GPIO.input(17)
```

The last two lines in the above code are very important, they create a loop that tells Python to keep checking the status of GPIO P17 [Pin 11] until it's no longer low (button released). Without this, the program would loop while the button is still being pressed meaning the message will be printed multiple times on the screen before you release the button. The final program should look like this in python:

```
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)
GPIO.setup(17, GPIO.IN)
while True:
  input_value = GPIO.input(17)
  if input_value == False:
    print("Who pressed my button?")
    while input_value == False:
      input_value = GPIO.input(17)
```

Save the file as button.py. In order to run the program, open a new terminal window on the Pi and type the following command:

```
sudo python button.py
```

At first nothing will happen, but if you press the button the program will print the defined message. To exit a running Python script, simply hit CTRL+C on the keyboard to terminate. If it hasn't worked don't worry. First check the circuit is connected correctly on the breadboard as defined in part 1, then that the jumper wires are connected to the correct pins on the GPIO port. If it still fails to work, double check each line of the program is correct remembering that python is case-sensitive and checking if indentation is correct. I find that typing the code out by hand will give better results than a simple copy/paste. This is a deceptively simple program that can be used for many purposes. The same code could be used to read when the pins of separate devices, such as a sensor or external micro-controller, have been pulled high or low.

Adding an LED

We'll carry on using the same breadboard as before but will require a couple extra components:

- An LED (light emitting diode), any colour you like.
- ModMyPi's Ridiculous Resistor Kit (RK995)
  * 330Ω Resistor - (Orange, Orange, Black, Black, Brown)

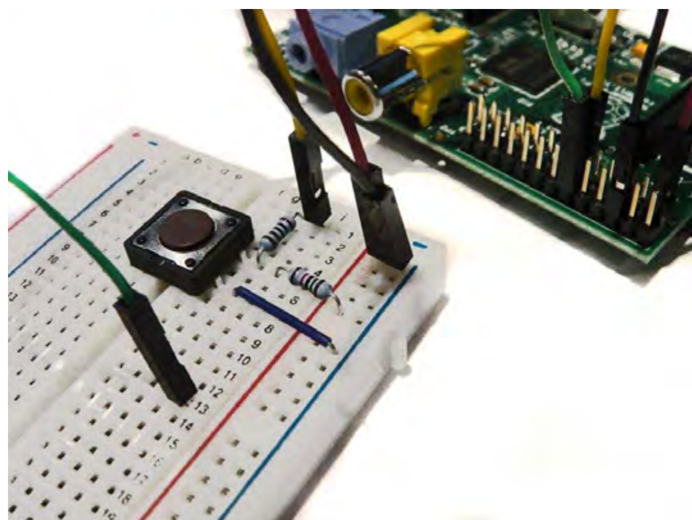
An output pin on the GPIO port can be set to either 0V (low) or 3.3V (high). Our expansion circuit will wire up an LED between an output pin and a ground pin on the Raspberry Pi's GPIO port. We'll be using Python to trigger our output pin high, causing a current to pass through the LED, lighting it up! We also add a 330Ω Resistor to limit the current that is passed through the LED.

For this exercise we will connect the LED to GPIO P18 [Pin 12] which will be defined later in

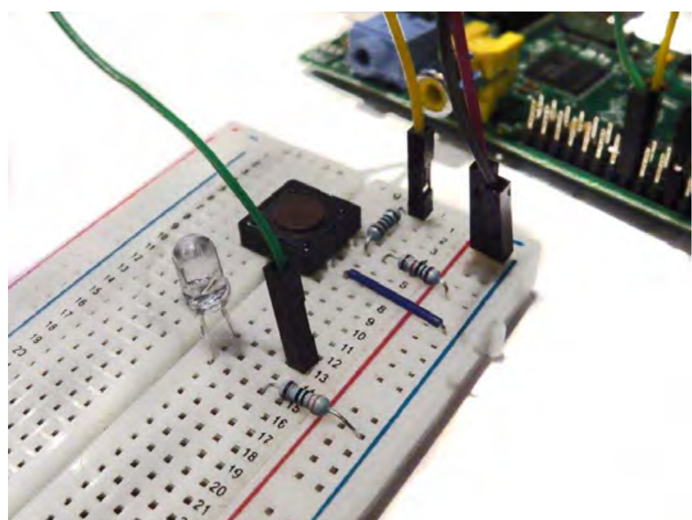
our program as an output. We will use the same ground pin GPIO GND [Pin 6] as before, which is already connected to the Negative rail (-) on our breadboard.

Building the Circuit

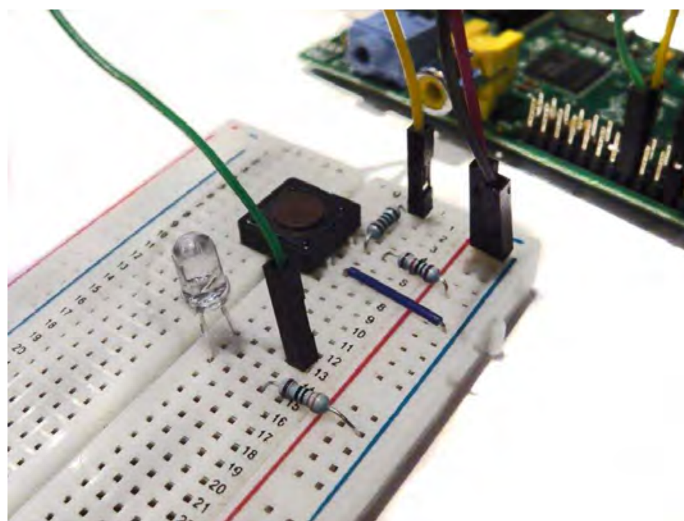
1. Connect GPIO output to breadboard. Use a green jumper wire to connect GPIO P18 [Pin12] on the Pi to an empty to row on the breadboard



2. Add an LED. Insert the long leg (anode) of the LED into the same row as the green jumper wire and insert the shorter leg (cathode) into another empty row on the breadboard. Note: Make sure the LED is inserted the correct way round as it will only allow current to pass through it in one direction. If it's wrong it won't light up!



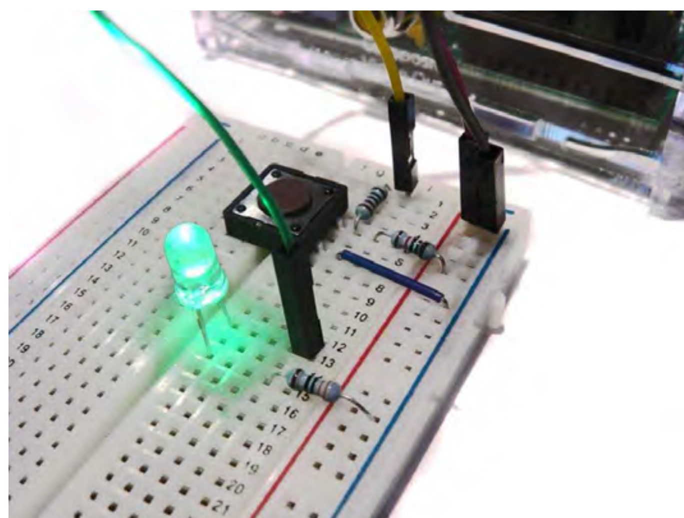
3. Add 330Ω Resistor. Connect the resistor between the cathode of the LED and the Negative rail (-) on the breadboard. This protects our LED from burning out by way of too much current.

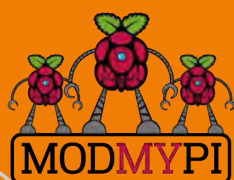


And that's our circuit built!

Next time

Next time, we'll add a timer function to our script and a trigger function for our LED. As the script will be more complicated, it requires a proper clean-up function, so we'll code that in too!



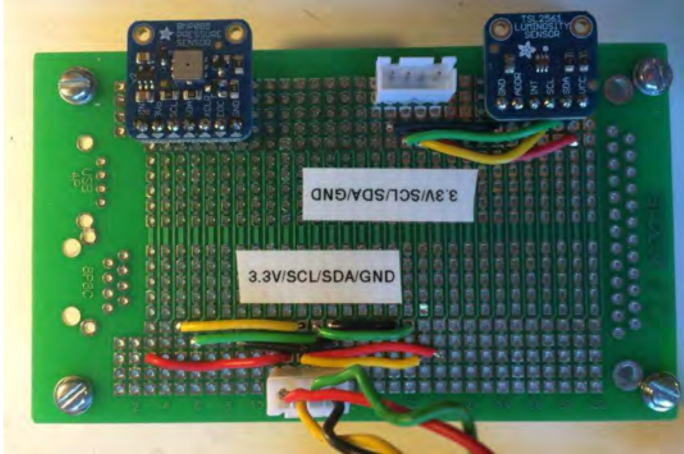


This article is sponsored by ModMyPi

All breakout boards and accessories used in this tutorial are available for worldwide shipping from the ModMyPi webshop at www.modmypi.com

What to measure?

We want to measure the temperature, humidity and local light levels both inside and outside of the containing box to see what is happening in the local environment. This information will be placed in a MySQL database for later analysis.

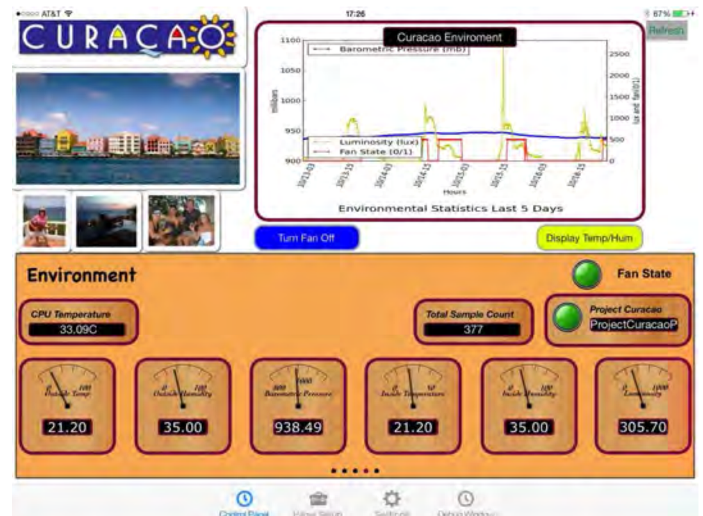


Putting in the sensors

The BMP085 and TSL2561 are soldered onto a prototype board with I^2C coming in from the Raspberry Pi. There is a plug to further extend the I^2C bus to an encased temperature and humidty sensor outside of the box (the AM2315). The AM2315 proved to be problematic (see below), but the other I^2C sensors worked like champions. The DHT22 inexpensive indoor temperature and humidity sensor worked well with additional software to filter out bad readings. This caused problems with the RasPiConnect control software because of the unreliable delay to good readings (see below). We control the fan with a relay that connects directly to the output of the solar cells (6.5V on a good day). We figured that the fan would be used on the sunniest days. The fan and relay coil each take 70mA. We are investigating replacing the relay with a lower energized coil current (see Sainsmart: http://goo.gl/aSTU0z) as 140mA really hurts our power budget.

Monitoring the sensors and fan remotely

Project Curacao is monitored remotely through the Internet by the use of SSH and RasPiConnect (www.milocreek.com). Each of the main subsystems has a different display. The environmental display has a graph for temperature/humidity and luminosity/barometric pressure/fan state (shown below). The software for generating this display is also on github.com/projectcuracao.



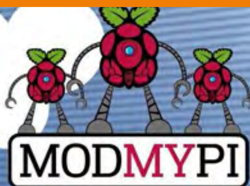
Problems with sensors

There were two major problems with sensors in this design. First of all the AM2315 is an odd duck. It has a power down sequence that requires it to be addressed twice (once to wake up and once to read - for example it takes two invocations of i2cdetect to see the sensor) and secondly, it just isn't reliable using 3.3V for I^2C. A solution might be to put a level converter in for the device, but since we had a 5V I^2C on the battery watchdog Arduino, we decided to add it to the watchdog. Secondly, the DHT22 has some very tricky timing in order to read it from the Raspberry Pi. Since the Raspberry Pi is a multitasking machine, you can't depend on the timing 100%. This means that readings from the DHT22 fail on a regular basis. The way to fix this problem is to keep reading it until the correct format is received. However, you can only read it every 3 seconds. This plays havoc with an HTTP based monitoring system such as RasPiConnect with 10-15 second timeouts. This problem was fixed by reading the DHT22 in the main software and having RasPiConnect read the last reading from the MySQL database.

What is coming up?

Part 3 goes through the Raspberry Pi Camera Subsystem and Part 4 describes the software system used for Project Curacao. All of the code used in this article is posted on GitHub at github.com/projectcuracao.

More discussion on Project Curacao at: http://switchdoc.blogspot.com



PHYSICAL COMPUTING

PHYSICAL COMPUTING
Brought to you by ModMyPi

Buttons and switches with the Raspberry Pi - part 3

Jacob Marsh

ModMyPi

SKILL LEVEL : BEGINNER

In our previous tutorial we built a simple button circuit connected to our Raspberry Pi via the GPIO ports and programmed a Python script to execute a command when the button was pressed. We then expanded our circuit with an LED. In this tutorial, we will be expanding our script to include timer and trigger functions for our LED. As the script will be more complicated, it requires a proper clean-up function. Start by reading the previous two tutorials featured in Issues 17 and 18, before trying this one!

Adding LED control code

Now that our LED expansion circuit has been built, we will add some code into our previous program. This additional code will make the LED flash on and off when the button is pressed. Start by booting your Raspberry Pi to the Raspbian GUI (startx). Then start IDLE3 and open the previous example program button.py. Save this file as button_led.py and open it.

Since we want the LED to flash on and off, we will need to add a time function to allow Python to understand the concept of time. We therefore need to import the time module, which allows various time related functionality to be used. Add another line of code underneath line 1:

```
import time
```

Next, we need to define GPIO P18 [Pin 12] as an output to power our LED. Add this to our GPIO.setup section (line 4), below the input pin setup line:

```
GPIO.setup(18, GPIO.OUT)
```

Once GPIO P18 [Pin 12] has been set as an output, we can turn the LED on with the command GPIO.output(18, True). This triggers the pin to high (3.3V). Since our LED is wired directly to this output pin, it sends a current through the LED tha turns it on. The pin can also be triggered low (0V) to turn the LED off, by the command GPIO.output(18, False).

Now we don't just want our LED to turn on and off, otherwise we would have simply wired it to the button and a power supply. We want the LED to do something interesting via our Raspberry Pi and Python code. For example, let us make it flash by turning it on and off multiple times with a single press of the button!

In order to turn the LED on and off multiple times we are going to use a for loop. We want the loop to be triggered when the button has been pressed. Therefore, it needs to be inserted within the if condition 'If input_value == False:', that we created in our original program. Add the following below the line 'print("Who pressed my button!")' (line 9), making sure the indentation is the same:

```
        for x in range(0, 3):
```

Any code below this function will be repeated three times. Here the loop will run from 0 to 2, therefore running 3 times. Now we will add some code in the loop, such that the LED flashes on and off:

```
            GPIO.output(18, True)
            time.sleep(1)
            GPIO.output(18,False)
            time.sleep(1)
```

The LED is triggered on with the command GPIO.output(18, True). However, since we do not want to immediately turn it back off, we use the function time.sleep(1) to sleep for one second. Then the LED is triggered off with the GPIO.output(18,False) command. We use the time.sleep(1) function again to wait before the LED is turned back on again.

The completed program should be of the form:

```
import RPi.GPIO as GPIO
import time
GPIO.setmode(GPIO.BCM)
GPIO.setup(17, GPIO.IN)
GPIO.setup(18, GPIO.OUT)
while True:
  input_value = GPIO.input(17)
  if input_value == False:
    print("Who pressed my button?")
    for x in range(0, 3):
      GPIO.output(18, True)
      time.sleep(1)
      GPIO.output(18, False)
      time.sleep(1)
    while input_value == False:
      input_value = GPIO.input(17)
```

Save the file and open a new terminal window. Then type the following command:

```
sudo python button_led.py
```

This time when we press the button a message will appear on the screen and the LED should also flash on and off three times!

To exit the program script, simply type CTRL+C on the keyboard to terminate it. If it hasn't worked do not worry. Do the same checks we did before. First, check the circuit is connected correctly on the breadboard. Then check that the jumper wires are connected to the correct pins on the GPIO port. Double check the LED is wired the right way round. If the program still fails, double check each line of the program, remembering that Python is case-sensitive and correct indentation is needed.

If is everything is working as expected, you can start playing around a bit with some of the variables. Try adjusting the speed the LED flashes by changing the value given to the time.sleep() function.

You can also change the number of times that the LED flashes by altering the number times that the for loop is repeated. For example if you wanted the LED to flash 30 times, change the loop to: for x in range(0, 30).

Have a go playing around with both these variables and see what happens!

Exiting a program cleanly

When a program is terminated (due to an error, a keyboard interrupt (CTRL+C) or simply because it's come to an end), any GPIO ports that were in use will carry on doing what they were doing at the time of termination. Therefore, if you try to run the program again, a warning message will appear when the program tries to 'set' a pin that’s already in use from the previous execution of the program. The program will probably run fine, but it is good practice to avoid these sorts of messages, especially as your programs become larger and more complex!

To help us exit a program cleanly we are going to use the command GPIO.cleanup(), which will reset all of the GPIO ports to their default values.

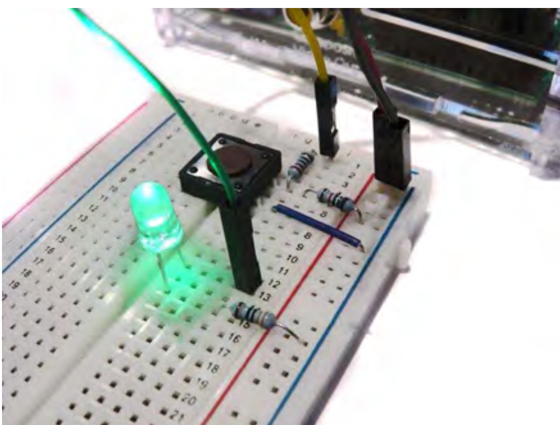
For some programs you could simply place the GPIO.cleanup() command at the end of your program. This will cause the GPIO ports to be reset when the program finishes. However, our program never ends by itself since it constantly loops to check if the button has been pressed. We will therefore use the try: and except syntax, such that when our program is terminated by a keyboard interruption, the GPIO ports are reset automatically.

The following Python is an example of how the try: and except command can be used together to exit a program cleanly.

```
# Place any variable definitions and
# GPIO  set-ups here

try:

# Place your main block of code or
# loop here

except KeyboardInterrupt:
  GPIO.cleanup()

# Program will end and GPIO ports
# cleaned when you hit CTRL+C

finally:
  GPIO.cleanup()
```

Note that Python will ignore any text placed after hash tags (#) within a script. You may come across this a lot within Python, since it is a good way of annotating programs with notes.

After we have imported Python modules, setup our GPIO pins, we need to place the main block of our code within the try: condition. This part will run as usual, except when a keyboard interruption occurs (CTRL+C). If an interruption occurs the GPIO ports will be reset when the program exits. The finally: condition is included such that if our program is terminated by accident, if there is an error without using our defined keyboard function, then the GPIO ports will be cleaned before exit.



Open button_led.py in IDLE3 and save it as button_cleanup.py. Now we can add the code previously described into our script. The finished program should have the form:

```
import RPi.GPIO as GPIO
import time
GPIO.setmode(GPIO.BCM)
GPIO.setup(17, GPIO.IN)
GPIO.setup(18, GPIO.OUT)
try:
  while True:
    input_value = GPIO.input(17)
    if input_value == False:
      print("Who pressed my button?")
      for x in range(0, 3):
        GPIO.output(18, True)
        time.sleep(1)
        GPIO.output(18, False)
        time.sleep(1)
      while input_value == False:
        input_value = GPIO.input(17)
except KeyboardInterrupt:
  GPIO.cleanup()
# Program will end and GPIO ports cleaned
# when you hit CTRL+C
finally:
  GPIO.cleanup()
```

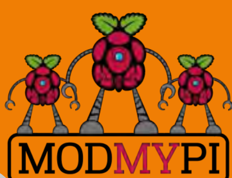
Notice that only the loop part of the program is within the try: condition. All our imports and GPIO set-ups are left at the top of the script. It is also important to make sure that all of your indentations are correct!

Save the file. Then run the program as before in a terminal window terminal:

```
sudo python button_cleanup.py
```

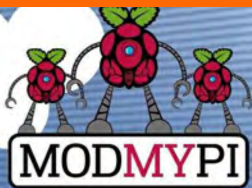
The first time you run the file, you may see the warning message appear since the GPIO ports have not been reset yet. Exit the program with a keyboard interruption (CTRL+X). Then run the program again and hopefully this time no warning messages will appear!

This extra code may seem like a waste of time because the program still runs fine without it! However, when we are programming, we always want to try and be in control of everything that is going on. It is good practice to add this code, to reset the GPIO when the progam is terminated.



This article is sponsored by ModMyPi

All breakout boards and accessories used in this tutorial are available for worldwide shipping from the ModMyPi webshop at www.modmypi.com



PHYSICAL COMPUTING

PHYSICAL COMPUTING
Brought to you by ModMyPi

GPIO Sensing: Motion Detection - Part 1

Jacob Marsh

ModMyPi

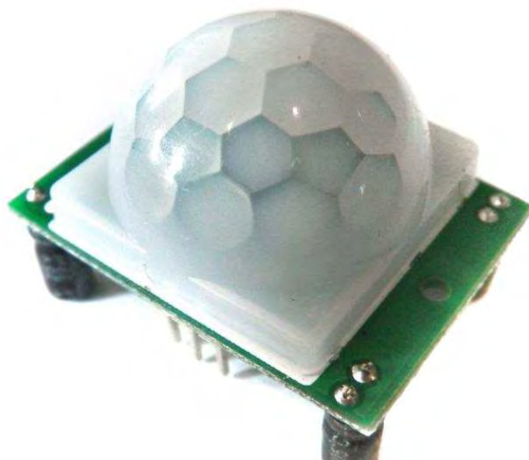
SKILL LEVEL : BEGINNER

In previous tutorials in Issues 15 to 19 of The MagPi, the basics behind physical computing with the Raspberry Pi computer were introduced. The previous articles covered basic Python programming and Board/BCM GPIO numbering, which are both used in this tutorial.

In this tutorial, a passive infrared sensor (PIR) is used instead of a switch. The PIR is used to activate a print statement within a Python program when motion is detected.

PIR sensors

PIR sensors provide a simple way of detecting motion. Everything on the Earth emits a small amount of infrared radiation, where hotter objects emit more radiation. PIR sensors are able to detect a change in infrared levels within their detection zone. For example, when someone comes into a room the PIR detects the infrared radiation change.



Assembling the circuit

For this tutorial, the following parts are required:

* a breadboard
* 6 x male to female jumper wires
* a PIR Sensor

All of the components can be purchased from the ModMyPi online shop.

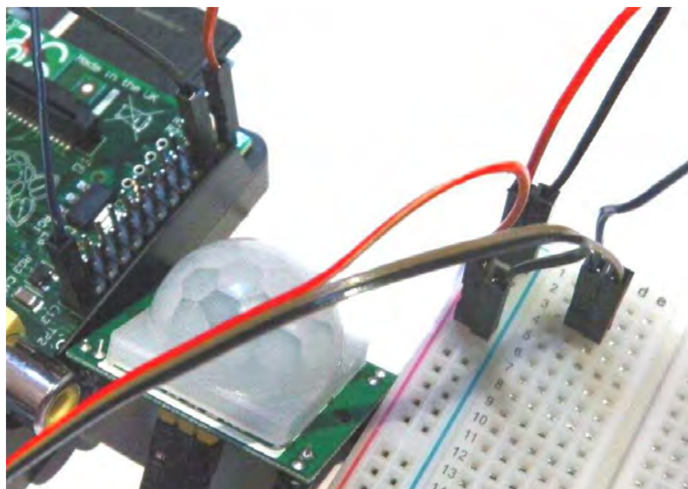
1. Use three male to female jumper wires and connect the female ends to the PIR sensor terminals. The three pins on the PIR are labelled as: Red; PIR-VCC (3-5VDC in), Brown; PIR-OUT (digital out) and Black; PIR-GND (ground).

2. Plug the male jumper wire connected to PIR-VCC into the positive rail of the breadboard, plug the PIR-GND wire into your negative rail and plug the PIR-OUT wire into any other blank rail.

3. Use a black jumper wire to connect GPIO GND [Pin 6] on the Raspberry Pi to the negative rail of the breadboard, to which the PIR-GND wire is already connected.

4. Use a red jumper wire to connect GPIO 5V [Pin 2] on the Raspberry Pi to the positive rail of the breadboard, to which the PIR-VCC wire has already been connected.

5. Connect GPIO 7 [Pin 26] to the same rail as the PIR-OUT.



Sensing with Python

The status of the PIR can be checked in a similar way as the switch mentioned in previous tutorials. However, the PIR does not need a pull-up resistor since it outputs 0V or 3V3. The GPIO 7 [Pin 26] connected to the PIR-OUT just needs to be set as an input pin.

Type in the following using the nano text editor:

```
import RPi.GPIO as GPIO
import time
GPIO.setmode(GPIO.BCM)
PIR = 7
GPIO.setup(PIR, GPIO.IN)
try:
  print("PIR Module Test")
  print("(CTRL+C to exit)")
  time.sleep(2)
  print "Ready"
  while True:
    if GPIO.input(PIR):
      print("Motion detected!")
    time.sleep(1)

except KeyboardInterrupt:
  print("Quitting")
  GPIO.cleanup()
```

The program includes several steps that were introduced in the previous articles. At the start there are two import statements to allow the GPIO and time functions to be used.

```
import RPi.GPIO as GPIO
import time
```

Then there is the setmode function,

```
GPIO.setmode(GPIO.BCM)
```

to configure the GPIO in BCM numbering mode. More information on the BCM numbering scheme can be found at:
http://elinux.org/RPi_Low-level_peripherals

After the GPIO setmode, a variable PIR is used to store the connection associated with the PIR digital out. This connection number is used to setup the associated pin as an input with:

```
GPIO.setup(PIR, GPIO.IN)
```

The example program includes a try-except statement, to clean up the GPIO connections when the program exits. Inside the try statement there are some print statements and a while loop. The while loop polls the GPIO input every second. If the GPIO pin is high then,

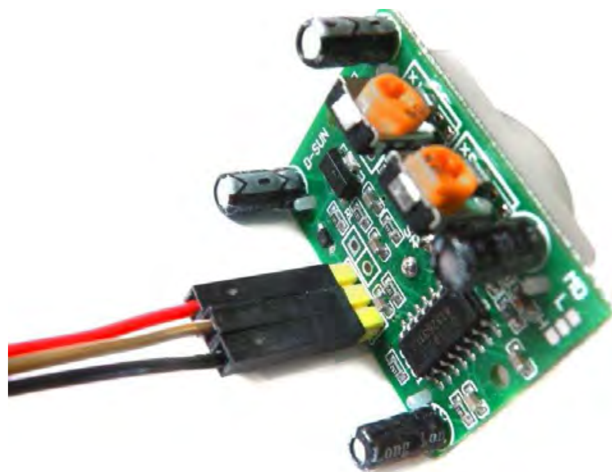
```
Motion detected!
```

will be printed on the screen.

PIR settings

On the PIR, there are two potentiometers that adjust the amount of time the PIR is high and also its sensitivity. Follow the labelling on the

board, which indicates the function of each potentiometer. If the program continues to report that motion is detected then try turning down the sensitivity of the PIR. If the potentiometer is turned clockwise then sensitivity will be increased.



GPIO callbacks

In the previous examples, GPIO polling has been used to check the state of a GPIO input pin. However, the GPIO also allows callbacks. This means that a pin can be set to call a Python function if the state of the input pin changes.

This is much better than polling since the function is called immediately, without the need for a high polling frequency.

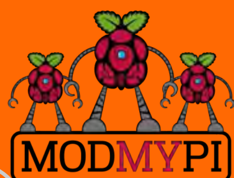
This time when the program runs, the CPU of the Raspberry Pi waits for the signal on the PIR to rise. Once the signal rises, the function callback_up is called. The callback function is passed the channel number of the GPIO pin that went high. Possible GPIO tests are `GPIO.RISING`, `GPIO.FALLING` and `GPIO.BOTH`.

Next time

That's it, a very simple method of connecting a low cost movement sensor to the Raspberry Pi.

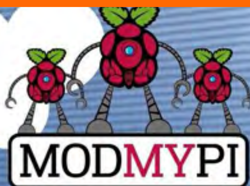
The next article will include a buzzer, some LEDs and a magnetic door lock. These parts will be used to build a low cost Raspberry Pi based security system!

```
import RPi.GPIO as GPIO
import time

def callback_up(channel):
  print("Up detected on channel %s" % channel)

PIR = 7
GPIO.setmode(GPIO.BCM)
GPIO.setup(PIR, GPIO.IN)

try:
  GPIO.add_event_detect(PIR, GPIO.RISING, callback=callback_up)
  while 1:
    time.sleep(100)

except KeyboardInterrupt:
  print("Cleaning up the GPIO")
  GPIO.cleanup()
```



This article is sponsored by ModMyPi

All breakout boards and accessories used in this tutorial are available for worldwide shipping from the ModMyPi webshop at www.modmypi.com



GPIO Sensing: Using 1-Wire temperature sensors - Part 2

Jacob Marsh

ModMyPi

SKILL LEVEL : BEGINNER

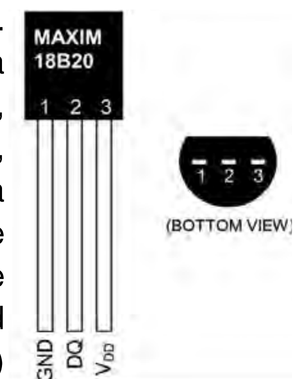
1-Wire sensors

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 in Issue 21, for example, simply says "Yes, I've detected movement".

So, what happens when we connect a more advanced sensor and want to read more complex data? In this tutorial we will connect a 1-Wire digital thermometer sensor and programme our Raspberry Pi to read the output of the temperature it senses!

In 1-Wire sensors all data is sent down one wire, which makes it great for microcontrollers and computers, 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 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 (V_{DD}). You 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 V_{DD} 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 are also going to use GPIO 4 [Pin 7] as the driver pin for sensing the thermometer output. This is the dedicated pin for 1-Wire GPIO sensing.

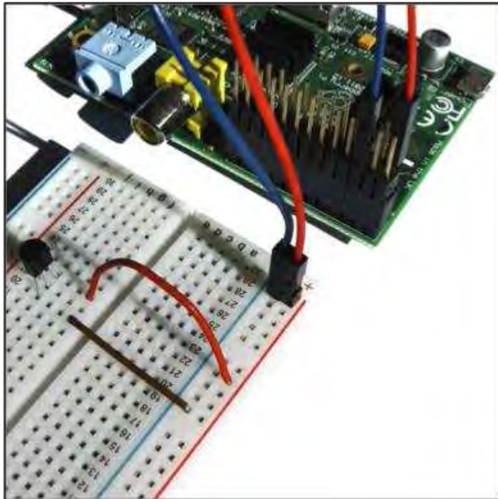
Hooking it up

1. Connect GPIO GND [Pin 6] on the Raspberry Pi to the negative rail on the breadboard.
2. Connect GPIO 3.3V [Pin 1] on the Raspberry Pi to the positive rail on the breadboard.
3. Plug the DS18B20+ into your breadboard,

ensuring that all three pins are in different rows. Familiarise yourself with the pin layout, as it is quite easy to hook it up backwards!

4. Connect DS18B20+ GND [Pin 1] to the negative rail of the breadboard.

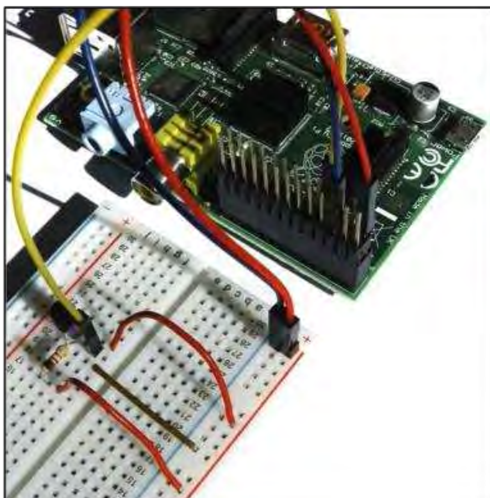
5. Connect DS18B20+ V_{DD} [Pin 3] to the positive rail of the breadboard.



6. Place your 4.7kΩ resistor between DS18B20+ DQ [Pin 2] and a free row on your breadboard.

7. Connect that free end of the 4.7kΩ resistor to the positive rail of the breadboard.

8. Finally, connect DS18B20+ DQ [Pin 2] to GPIO 4 [Pin 7] with a jumper wire.



That's it! We are 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 programming. 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 will access it via the command line first and then we will write a Python program 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 increases the boot time significantly and uses 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. At the command line enter:

```
sudo modprobe w1-gpio
sudo modprobe w1-therm
```

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

```
cd /sys/bus/w1/devices
ls
```

In the device list, your sensor should be listed as a series of numbers and letters. In my case, the device is registered as 28-000005e2fdc3. You then need to access the sensor with the `cd` command, replacing the serial number with that from your own sensor. Enter:

```
cd 28-000005e2fdc3
```

The sensor periodically writes to the `w1_slave` file. We can 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 milli-degrees Celsius. Divide this number by 1000 to get the temperature in degrees, 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 sensor, this is all that's required from the command line. Try holding onto the thermometer for a few seconds and then take another reading. Spot the increase? With these commands in mind, we can now write a Python program to output our temperature data automatically.

Python program

Our first step is to import the required modules. The os module allows us to enable our 1-Wire drivers and interface with the sensor. The time module allows the 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 the 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/28-000005e2
fdc3/w1_slave'
```

We then need to define a variable for our raw temperature value, `temp_raw`; the two lines output by the sensor, as demonstrated by the command line example. We could simply print this statement now, however we are going to process it into something more usable. To do this we open, read, record and then close the `temp_sensor` 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 shown in the command line example, we get two lines of output code. The first line was "72 01 4b 46 7f ff 0e 10 57 : crc=57 YES". We strip this line, except for the last three characters, and check for the "YES" signal, which indicates a successful temperature reading from the sensor. In Python, not-equal is defined as "!=", so here we are saying that while 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 a YES signal has been received, we proceed to our second line of output code. In our example this was "72 01 4b 46 7f ff 0e 10 57 t=23125". We find our temperature output "t=", check it for errors and strip the output of the "t=" phrase to leave just the temperature data. Finally we run two calculations to give us the temperature 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)
```

That's our code! A screenshot of the complete program is shown below. Save your program and run it to display the temperature output, as shown on the right.

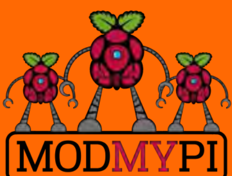
Multiple sensors

DS18B20+ sensors can be connected in parallel and accessed using their unique serial number. Our Python example can be edited to access and read from multiple sensors!

```
pi@raspberrypi ~ $ sudo python temp_2.py
(23.437, 74.1866)
(23.437, 74.1866)
(23.437, 74.1866)
(23.437, 74.1866)
(23.437, 74.1866)
(25.5, 77.9)
(27.25, 81.05)
(28.312, 82.9616)
(29.0, 84.2)
(29.437, 84.9866)
(29.75, 85.55)
(29.687, 85.4366)
(29.0, 84.2)
(28.25, 82.85)
```

As always, the DS18B20+ sensor and all components are available separately or as part of our workshop kit from the ModMyPi website http://www.modmypi.com.

```
import os
import time

os.system('modprobe w1-gpio')
os.system('modprobe w1-therm')

temp_sensor = '/sys/bus/w1/devices/28-000005e2fdc3/w1_slave'

def temp_raw():
    f = open(temp_sensor, 'r')
    lines = f.readlines()
    f.close()
    return lines

def read_temp():
    lines = temp_raw()
    while lines[0].strip()[-3:] != 'YES':
        time.sleep(0.2)
        lines = temp_raw()
    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

while True:
        print(read_temp())
        time.sleep(1)
```



This article is sponsored by ModMyPi

All breakout boards and accessories used in this tutorial are available for worldwide shipping from the ModMyPi webshop at www.modmypi.com