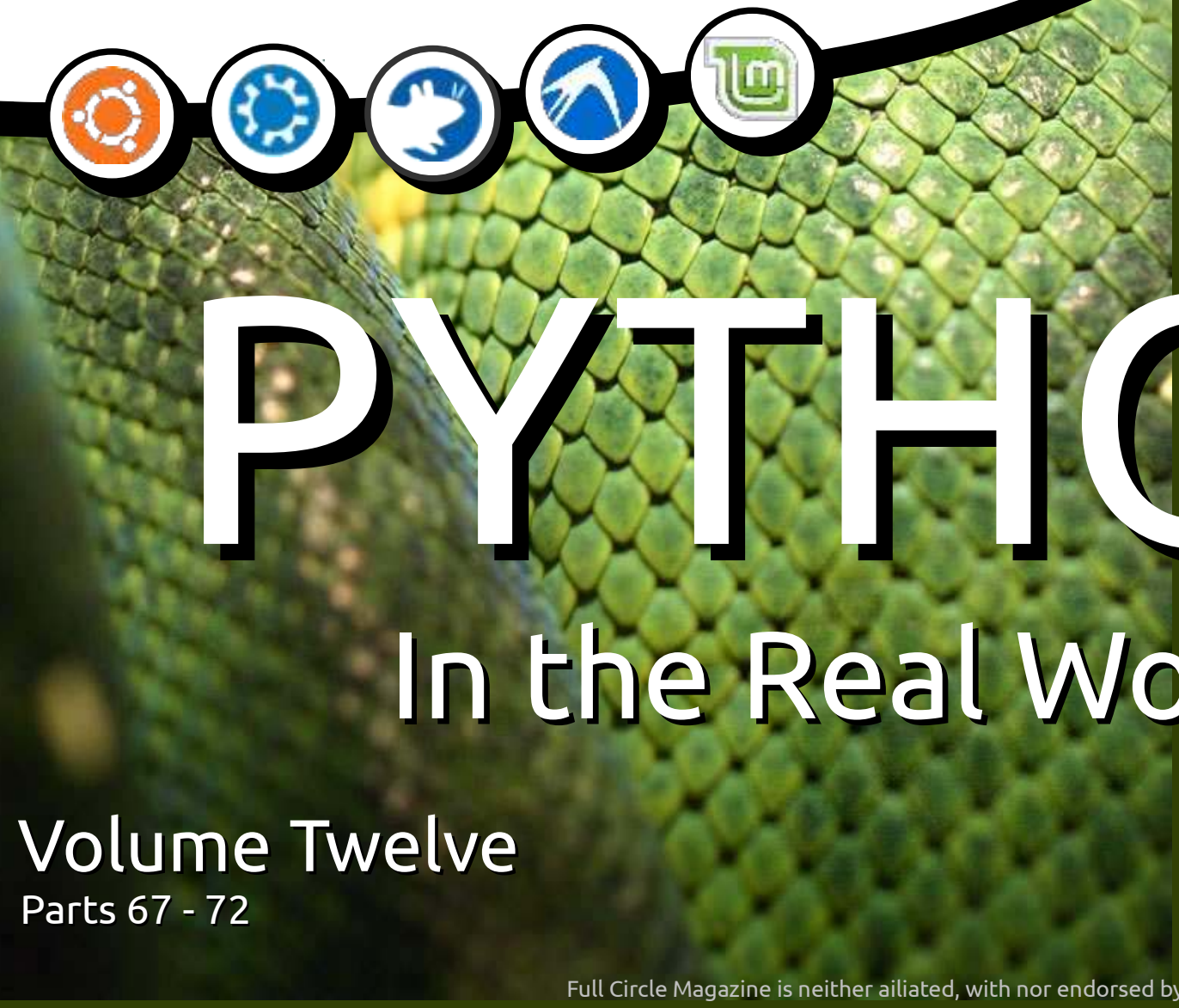
Full Circle

THE INDEPENDENT MAGAZINE FOR THE UBUNTU LINUX COMMUNITY

PROGRAMMING SERIES SPECIAL EDITION

PROGRAMMING SERIES
SPECIAL EDITION



PYTHON

In the Real World

Volume Twelve

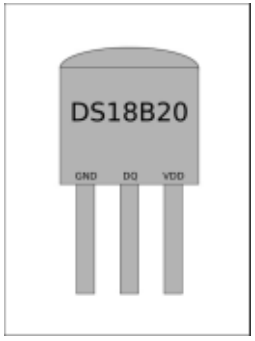
Parts 67 - 72

Python In The Real World - Pt 67

Written by Greg D. Walters

This month, we will be using my current favorite temperature sensor; the Dallas Semiconductor DS18B20 One Wire sensor. It looks like a 'normal' transistor, but is a very accurate sensor, much more so than the DHT11 that we used last month. It doesn't do humidity, but for temperature readings, it's a very good and inexpensive device. All data requests and output are sent on a single pin. It has an operating range from -55°C to 125°C (-67°F to 257°F) and should be able to run about 3 metres (9.8 feet). It also has a parasitic mode that allows power to be derived from the data line.

The data sheet can be found at https://datasheets.maximintegrated.com/en/ds/DS18B20.pdf. Here is what one sort of looks like…

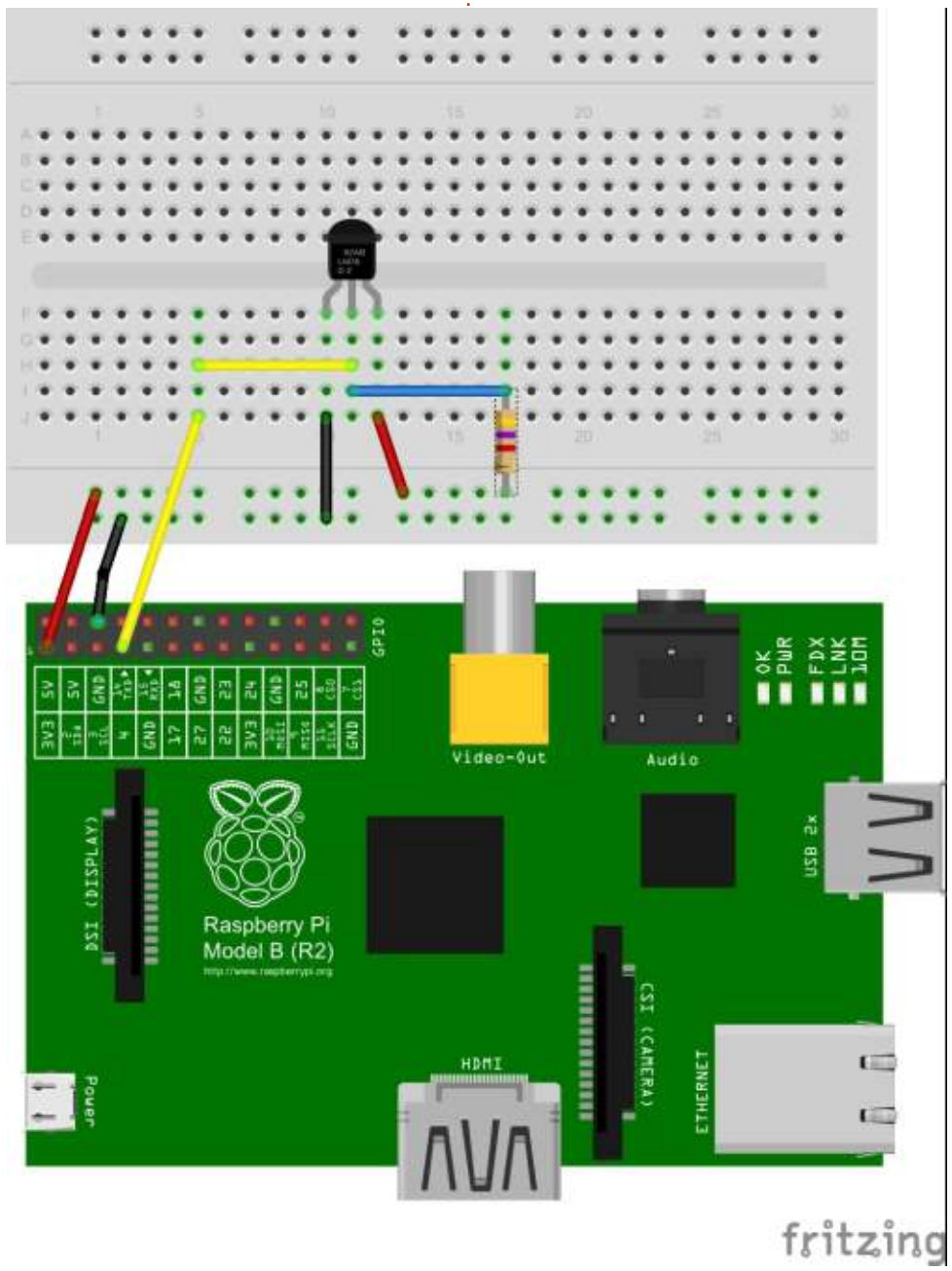


Wiring a single sensor is very easy. Shown right is the diagram.

There are only three connections to the RPi. Ground (sensor pin 1) to RPi pin 6, 3.3v (sensor pin 3) to RPi pin 1, and data (sensor pin 2) to RPI pin 7 (GPIO 4). You need to put a 4.7k resistor between sensor pins 2 and 3 (data and +Voltage). That's it. If you wish to add more sensors to the project, simply connect them ground to ground, +voltage to +voltage and pin 2 to pin 2 of the "main" sensor. No additional resistors should be needed for a reasonable line length. Next page, right-hand side, is an example of a three sensor project.

The Code

One thing you have to do is tell the operating system you want to use kernel support for one-wire devices. If you are using Raspbian Jessie, this is done in raspi-config. If you are using another OS, then you must add the following line to the /boot/config.txt file.



contents

```
dtoverlay=w1-gpio
```

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 before the ending three digits, e.g. the temperature reading we've received is 23.125 degrees Celsius.

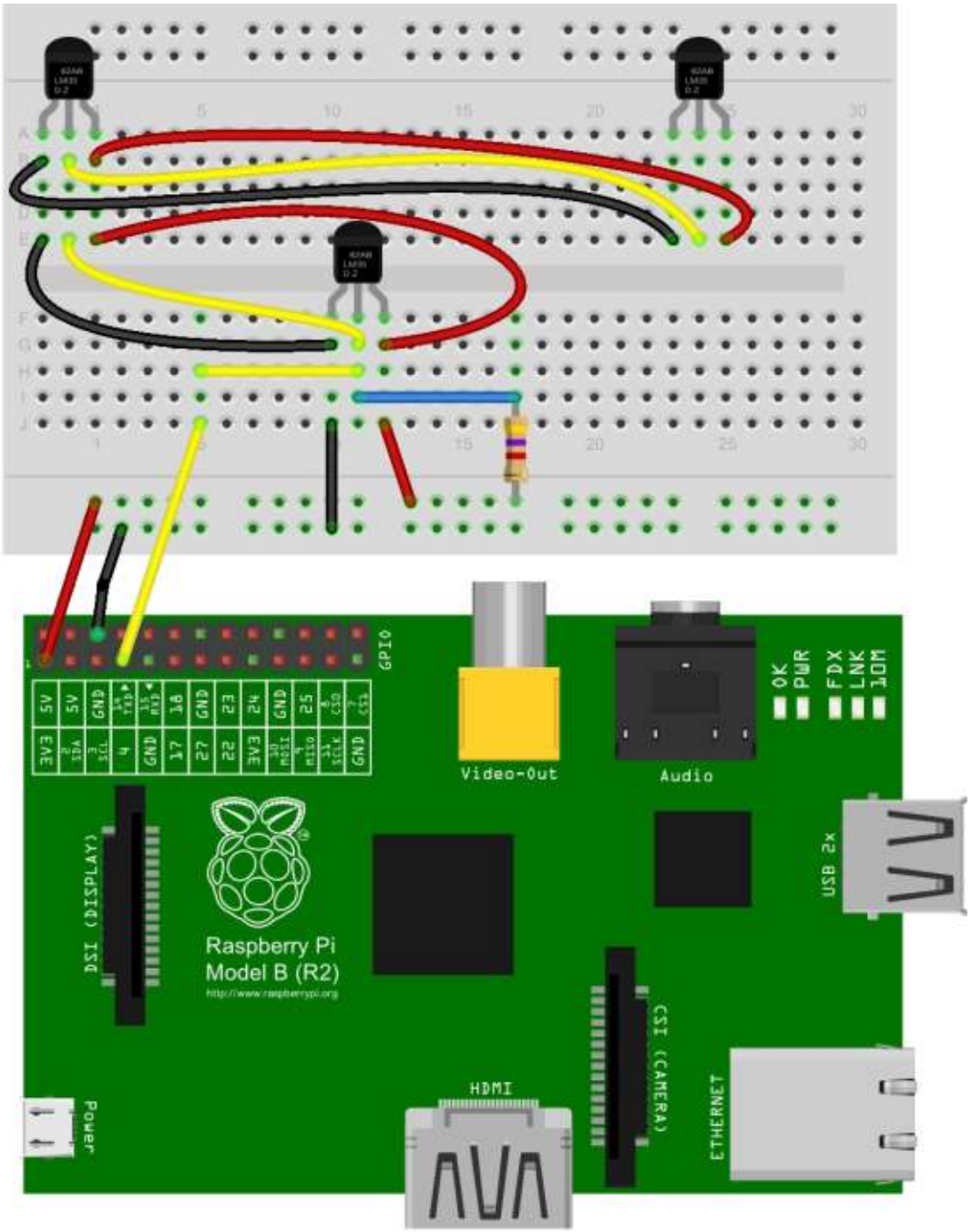
```
72 01 4b 46 7f ff 0e 10 57 :
crc=57 YES

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

Next page, top right, is how we had to do it in the "old days", the library that we will actually use.

Timo Furrer has provided a wonderful library for us to use on the RPi written in pure Python. You can get it at https://github.com/timofurrer/w1thermsensor. The current version is 0.3.1 and is also available through pypi.

The beauty of this library is that it takes almost all the work out of dealing with the sensors and allows you to just concentrate on your code.

Next page, bottom right, is the "current day" code using Timo's library...



There are actually only 7 lines of code needed here. Those lines that are commented out are so you can see the other ways to get and print the data in various temperature units (celsius and kelvin).

As I mentioned above, you can have more than one sensor on the same data line. So here is the code to make a single call to get the temp readings from all sensors in the system…

```
from w1thermsensor import
W1ThermSensor

for sensor in
W1ThermSensor.get_available_s
ensors():

    print("Sensor %s has
temperature %.2f" %
(sensor.id,
sensor.get_temperature()))
```

Of course, you'll want to do more than one data pull, so use the code above to modify it the way you want.

If you would like to make a call to a particular sensor, you can use this code as a baseline.

```
from w1thermsensor import
W1ThermSensor

sensor =
W1ThermSensor(W1ThermSensor.T
HERM_SENSOR_DS18B20, "28-
000007444532")

temperature_in_celsius =
sensor.get_temperature()
```

So, you can see, by using Timo's library, we can basically go from 27 lines of code to 3 (for a single call). That's wonderful.

I wanted to show you how to use a 16x2 LCD display with this, but I think I'll leave room for the other authors and we'll push that part out to next month. Don't lose your project hardware, we'll use it next month.

Until then, enjoy checking out the temperatures around your office/abode.

```
import os
import glob
import time
os.system('modprobe w1-gpio')
os.system('modprobe w1-therm')
base_dir = '/sys/bus/w1/devices/'
device_folder = glob.glob(base_dir + '28*')[0]
device_file = device_folder + '/w1_slave'
def read_temp_raw():
    f = open(device_file, 'r')
    lines = f.readlines()
    f.close()
    return lines
def read_temp():
    lines = read_temp_raw()
    while lines[0].strip()[-3:] != 'YES':
        time.sleep(0.2)
        lines = read_temp_raw()
    equals_pos = lines[1].find('t=')
    if equals_pos != -1:
        temp_string = lines[1][equals_pos+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)
```

```
from w1thermsensor import W1ThermSensor
from time import sleep
sensor = W1ThermSensor()
while 1:
    # temp_in_celsius = sensor.get_temperature()
    temp_in_fahrenheit = sensor.get_temperature(W1ThermSensor.DEGREES_F)
    # temp_in_all_units = sensor.get_temperatures([W1ThermSensor.DEGREES_C, _
          W1ThermSensor.DEGREES_F, W1ThermSensor.KELVIN])
    print temp_in_fahrenheit
    # print temp_in_celsius
    # print temp_in_all_units
    sleep(3)
```

Python In The Real World - Pt 68

Written by Greg D. Walters

Last month, we worked with the DS18B20 Temperature Sensor. This month we will start to interface a 16x2 LCD display to show our temperatures. Don't tear down your setup, but make sure you have enough room to mount the display on your breadboard. You'll need about 32 pinholes for the length of the device and 16 for the pins to connect to. You will have only three pinholes left if you mount the display at the bottom of the vertical holes, so you will need to use some jumpers to connect the bottom verticals to the top verticals.

Of course, the 16x2 display has 16 characters on two rows. The backlight comes in many colours. I chose a blue one. We can address each of the 32 character positions individually, or print pretty much like we do to the regular monitor.

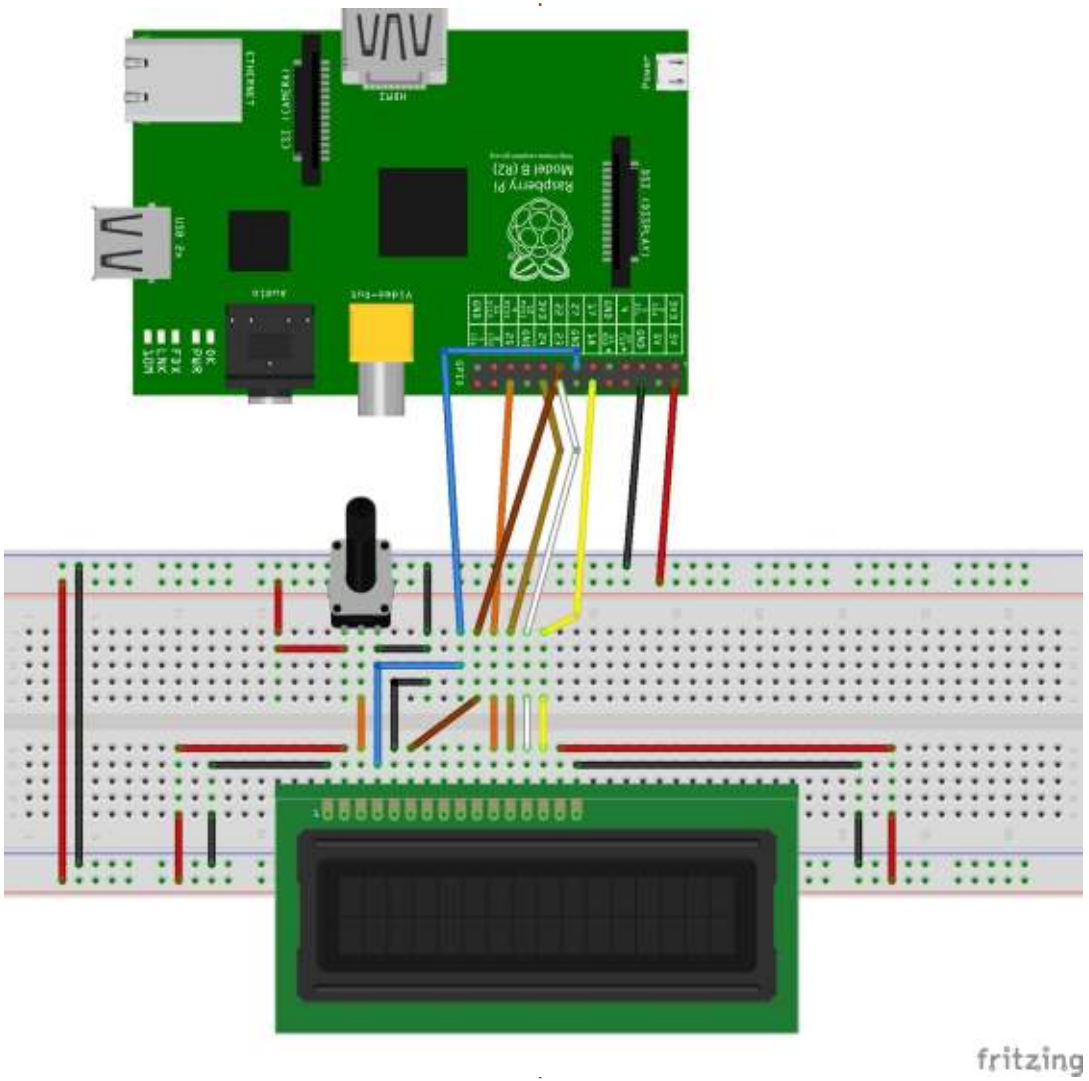
We will be making 8 connections to the RPi as well as the three that we used for the temperature sensor last month. You will need the following additional items for this month:

- 10K Potentiometer
- 16x2 LCD Display
- Many breadboard jumpers, Male to Male and 8 Male to Female

By the time you are done, the wiring diagram (and the resulting board) will look like a bit of a rat's nest, but go slowly – make sure you have the wiring correct.

As you can see in the graphic above, it's pretty crazy, so I'll lay out all the wiring for you in text.

First, you will need to put a jumper between the two horizontal busses on both the top and bottom. That way, you'll have power and ground on both busses. I chose to do it on the left side, but you can put it anywhere that is convenient for you. The next thing to do is to wire in the potentiometer. One side (it doesn't matter which) needs to go to ground and the other side to our 5 volt supply. The center contact (the wiper) will wire to pin 3 of the LCD display. This controls the contrast, so you can control how bright the characters appear. You



should already have 5 volts to the board, as well as ground, from last month.

On the display, connect pin 1 to ground and pin 2 to the +5 volt buss. That makes three connections out of the twelve we need. Pin 6 of the display goes to pin 22 of the RPi. This is the Enable

contents

pin. Pin 5 on the display goes to ground, and pin 4 to pin 27 on the RPi. We are up to 6 connections so far. That makes us halfway there. Because we have to use pin 4 for our sensor, we can't control the backlight.

Now we will work backwards from pin 16. Pin 16 goes to ground, and pin 15 to +5v. Pin 15 is actually the backlight voltage on mine. If you find the display too bright, you could put the wiper of another potentiometer connected between +5v and ground and control the display backlight.

Now for the data lines. There are actually 8 data lines, but thankfully, we will be using only 4. Pins 11 to 14 are D4, D5, D6 and D7 (counting from 0). Here is the connection list.

Display Pin	Raspberry Pi Pin
11	25
12	24
13	23
14	18

Now everything is hooked up, so we will continue with some sample code to test the display. But we need to get the Adafruit python library for LCDs. In a

```
#!/usr/bin/python
# Example using a character LCD connected to a Raspberry Pi or BeagleBone Black.
import time
import Adafruit_CharLCD as LCD
# Raspberry Pi pin configuration:
lcd_rs      = 27  # Note this might need to be changed to 21 for older revision Pi's.
lcd_en      = 22
lcd_d4      = 25
lcd_d5      = 24
lcd_d6      = 23
lcd_d7      = 18
lcd_backlight = 4
# Define LCD column and row size for 16x2 LCD.
lcd_columns = 16
lcd_rows    = 2
# Alternatively specify a 20x4 LCD.
# lcd_columns = 20
# lcd_rows    = 4
# Initialize the LCD using the pins above.
lcd = LCD.Adafruit_CharLCD(lcd_rs, lcd_en, lcd_d4, lcd_d5, lcd_d6, lcd_d7,
                           lcd_columns, lcd_rows, lcd_backlight)
# Print a two line message
lcd.message('Hello\nworld!')
# Wait 5 seconds
time.sleep(5.0)
# Demo showing the cursor.
lcd.clear()
lcd.show_cursor(True)
lcd.message('Show cursor')
time.sleep(5.0)
# Demo showing the blinking cursor.
lcd.clear()
lcd.blink(True)
lcd.message('Blink cursor')
time.sleep(5.0)
# Stop blinking and showing cursor.
lcd.show_cursor(False)
lcd.blink(False)
# Demo scrolling message right/left.
lcd.clear()
message = 'Scroll'
lcd.message(message)
for i in range(lcd_columns-len(message)):
    time.sleep(0.5)
    lcd.move_right()
for i in range(lcd_columns-len(message)):
    time.sleep(0.5)
    lcd.move_left()
# Demo turning backlight off and on.
lcd.clear()
lcd.message('Flash backlight\nin 5 seconds...')
time.sleep(5.0)
# Turn backlight off.
lcd.set_backlight(0)
time.sleep(2.0)
# Change message.
lcd.clear()
lcd.message('Goodbye!')
# Turn backlight on.
lcd.set_backlight(1)
```

terminal window, type the following...

```
git clone
https://github.com/adafruit/A
dafruit_Python_CharLCD

cd Adafruit_Python_CharLCD

sudo python setup.py install

cd examples
```

Now load char_lcd.py into your favorite editor. Or, you could type it in from the previous page.

Ignore the backlight messages, but you should see...

```
Hello World!
Show Cursor_
Blink Cursor_
Scroll (right and left)
Flash backlight in 5
seconds...
Goodbye!
```

If everything worked, we are ready to proceed. If not, go back and check your wiring.

Here is the modified program from last month that includes snippets from this example (top right) from Adafruit. (New code is in bold.)

That's about it for this month. Next month we will look at changing out our regular 16x2 display for a 16x2 I2C display (which uses only 2 lines for data and all control, and 2 lines for power.) We will also discuss the different ways of using serial communication for interfacing displays and other devices. Until then, have fun!

```
from w1thermsensor import W1ThermSensor
from time import sleep
import Adafruit_CharLCD as LCD
# Raspberry Pi pin configuration:
lcd_rs        = 27
lcd_en        = 22
lcd_d4        = 25
lcd_d5        = 24
lcd_d6        = 23
lcd_d7        = 18
lcd_backlight = 4
lcd_columns = 16
lcd_rows    = 2
# Initialize the LCD using the pins above.
lcd = LCD.Adafruit_CharLCD(lcd_rs, lcd_en, lcd_d4, lcd_d5, lcd_d6, lcd_d7,
                           lcd_columns, lcd_rows, lcd_backlight)
sensor = W1ThermSensor()
while 1:
    # temp_in_celsius = sensor.get_temperature()
    temp_in_fahrenheit = sensor.get_temperature(W1ThermSensor.DEGREES_F)
    print temp_in_fahrenheit
    lcd.clear()
    lcd.message(str(temp_in_fahrenheit))
    # print temp_in_celsius
    sleep(3)
```

Greg Walters is owner of RainyDay Solutions, LLC, a consulting company in Aurora, Colorado, and has been programming since 1972. He enjoys cooking, hiking, music, and spending time with his family.

Python In The Real World - Pt 69

This month, we will be using a 16x2 LCD display with an i2c interface to do the same thing that we did last month (FCM 111) with the Dallas Semiconductor DS18B20 Temperature sensor. If you remember, we had to use many I/O pins on the RPi. This time, thanks to the i2c display, we will need only 5 pins (+5v, Gnd, Sensor Data, SDA (data) and SCL (clock)) to do the same thing.

Before we get started with our project, a discussion about i2c is in order. I have distilled the following discussion from a wonderful tutorial by 'SFUPTOWNMAKER' at Sparkfun.com which can be found at https://learn.sparkfun.com/tutorials/i2c?_ga=1.242063243.863781319.1463423290

i2c

Inter-Integrated Circuit (i2c) protocol is intended to allow multiple "slave" digital ICs to communicate with one or more master chips. Like Serial Peripheral Interface (SPi), it is intended for only short distance communications within a single device. Like Asynchronous Serial Interfaces (RS-232 or UARTs), it requires only two signal wires to exchange information.

RS232 (Asynchronous Communications)

No clock data is required on the lines; however, both sides must agree on the communications data rate. Requires hardware overhead (UART) at each end.

In addition to the 8 data bits, at least one start and one stop bit are required for each data frame. While it is possible to connect multiple devices on a single serial port, there are issues with multiple devices trying to use the two lines at the same time.

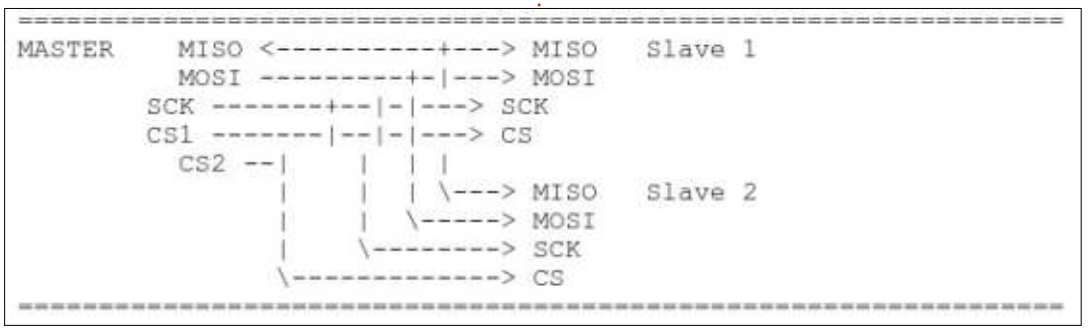
Most UART devices can support only certain set baud rates, the usual maximum is 230400 bits per second.

```
RX <--------- TX
TX ---------> RX
```

SPI

The biggest drawback with SPI is the number of pins required. Connecting a single master to a single slave requires four lines, each additional slave requires one addition chip select (CS) IO pin to the master. If you want to use many sensors/devices connected to the master, the required number of pins can quickly overwhelm the number of available inputs/outputs.

SPI (see diagram below) is good for high data rate full-duplex (simultaneous send/receive of data). Data rates can be upwards of 10 MHz.



i2c

i2c requires only 2 lines like async serial, but those two lines can support up to 1008 slave devices. Unlike SPI, i2c can support a multi-master system – allowing multiple slave devices to communicate to multiple master devices. There is a limitation that the masters can not communicate with each other on the i2c bus, and they must take turns using the bus lines. i2c has a similar overhead to Async in that, for each 8 bits of data, one extra bit is required as an "Ack/Nack" bit. The hardware requirements are more complex than SPI, but less than Async.

Data rates are somewhere between Async and SPI. Most i2c devices can communicate at between 100 KHz to 400 KHz.

```
=============================================================
Master 1     SDA <------+------------------------> SDA    Slave 1
             SCL -------|-------------+----------> SCL
                        |             |
Master 2     SDA <------+-------------|----------> SDA    Slave 2
             SCL --------------------------------> SCL
=============================================================
```

In the diagram above, SDA is the data line and SCL is the clock line.

Hopefully I didn't completely confuse you and you are ready to go ahead with our project.

A very good resource for what we are about to do is at http://www.circuitbasics.com/raspberry-pi-i2c-lcd-set-up-and-programming/

Make sure that i2c is enabled on the RPi. This is set in raspi-config.

Now, in a terminal, use apt-get to install two support libraries. (I wasn't able to get it to work as a single liner for some reason.):

```
sudo apt-get install i2c-tools

sudo apt-get install python-smbus
```

I was able to get Fritzing to come up with the i2c backpack (shown top right).

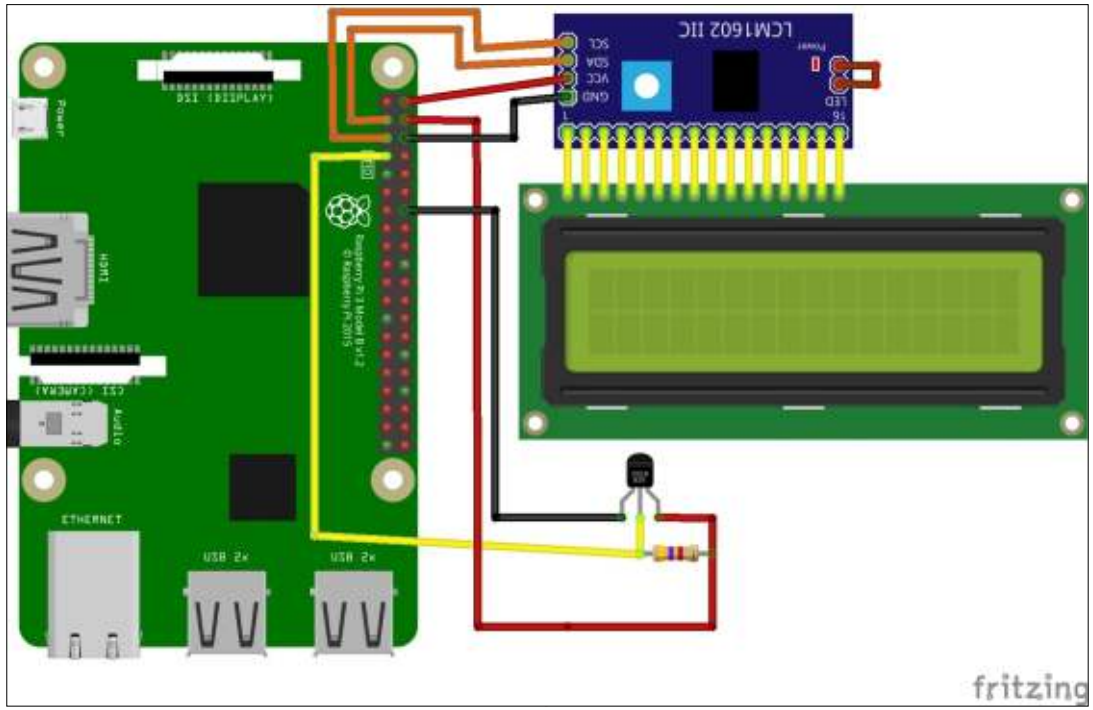
Hook up the SDA pin of the i2c backpack to PHYSICAL pin 3 on the RPi (this is GPIO2) and the SCL on the backpack to PHYSICAL pin 5 (GPIO3). Pick a free 5v pin on the RPi (either pin 2 or 4) and a free ground (pins 6 or 9) and connect them to the backpack VCC and Gnd. Don't forget we need the temp sensor connected to GPIO4 as last month (along with the resistor to +5VDC).

Now reboot and once the RPi is up, in a terminal type:

```
i2cdetect -y 1
```

This will verify that the i2c interface is working on your Pi, and also tell you what address is used by the LCD display device. Look at the screen dump shown right to see what it should look like.

As you can see, my device is at 3f, but yours might be at a different address. When you create the driver below (either typing directly from the article or from the pastebin page), you will need to enter your device address in line 22.




```
pi@raspberrypi:~ $ i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- 3f
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
pi@raspberrypi:~ $
```

The first set of code is a library that will work as a driver for the i2c LCD. This should be saved as i2c_lcd_driver.py. The code is on http://pastebin.com/ueu18fNL to save you typing.

Now, we are going to do a short test to make sure everything works. Type in the following code and save it as i2c_test1.py, into the same folder as the driver that we just wrote...

```
import i2c_lcd_driver
from time import *

mylcd = i2c_lcd_driver.lcd()

mylcd.lcd_display_string("Thi
s is a test",1)
```

If everything here is correct, you should see “This is a test” at the first character position on the first line of the LCD display. If it’s good, we can move forward. The following code is basically the same code from last month modified to use the i2c display instead of the parallel version.

That pretty much wraps up our discussion of LCDs and i2c. We will be using i2c LCDs and other i2c devices in the future, so you should keep them safe for later on. Next month, we will start working with motors, servos and stepper motors. So, run out, and get a hobby motor to be ready. In a few months, we’ll start working with the Arduino microcontroller and then learn to interface the RPi to control the Arduino.

Until then have fun.

```
import i2c_lcd_driver
from w1thermsensor import W1ThermSensor
from time import *

mylcd = i2c_lcd_driver.lcd()

#mylcd.lcd_display_string("This is a test",1)

sensor = W1ThermSensor()
#setup_lcd()
while 1:
      # This is basically the same code as last month, so use
      # whichever temp type you want.
    temp_in_fahrenheit = sensor.get_temperature(W1ThermSensor.DEGREES_F)
    # Print the temp to the terminal...
    print temp_in_fahrenheit
    # Now print it to the i2c LCD module...
    mylcd.lcd_clear()
    mylcd.lcd_display_string(str(temp_in_fahrenheit),1)
    sleep(3)
```

Greg Walters is owner of RainyDay Solutions, LLC, a consulting company in Aurora, Colorado, and has been programming since 1972. He enjoys cooking, hiking, music, and spending time with his family.

Python In The Real World - Pt 70

This month, we will be using the RPi to control a simple DC Hobby motor. This can be obtained from most hobby stores, electronics suppliers, and even some big box hardware stores. Here is a "shopping list" of what we will be needing.

- DC Hobby Motor
- L293D Dual H-Bridge Motor Driver Chip
- 4 AA (or AAA) Battery Holder and batteries
- Breadboard
- Male to Male jumpers
- RPi (of course)

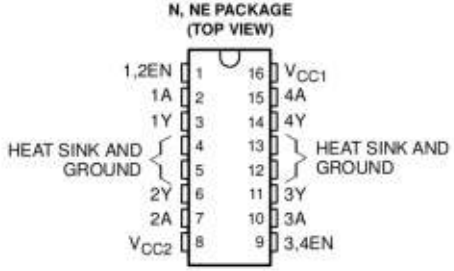
Before we start wiring and coding, we need to talk about a couple of things.

First, **NEVER EVER** connect a motor of **any kind** directly to the RPi. You are asking for disaster. The current requirements can cause the RPi to "melt down". The driver chip is less than $5.00 US and is a lot cheaper than a $39.00 RPi.

Second, we will discuss the L293D H-bridge motor driver for a few moments so you understand how this device works.

According to wikipedia, *"An H bridge is an electronic circuit that enables a voltage to be applied across a load in either direction. These circuits are often used in robotics and other applications to allow DC motors to run forwards and backwards."*

Here is a pinout of the driver chip ("borrowed" from hardwarefun.com)...



Pins 1 and 9 are enable pins. Think of these pins as an On/Off switch. A low state on the enable pin means the motor is off. A high state means that the motor CAN BE on. Let's look at it as a logic table or truth table. Pins 1A and 2A are one side of the chip and are control lines like the enable pins. The same logic applies to 3A and 4A (the other half of the chip) as well. Pins 1Y and 2Y are the outputs to the motor.

Enable	1A	2A	Result
LOW	HIGH	LOW	Not spinning - Enable is "off"
LOW	LOW	HIGH	Not spinning - Enable is "off"
HIGH	LOW	LOW	Not spinning - Both control inputs are "Off"
HIGH	HIGH	LOW	Turning Clockwise*
HIGH	LOW	HIGH	Turning Anti-Clockwise*
HIGH	HIGH	HIGH	Not Spinning - Both control inputs are "On"
			* Direction is based on motor wiring.

The bottom line of the crazy table above is this.
If you want the motor to turn on you MUST…
- Have the Enable pin HIGH (pin 1 and/or pin 9)
- AND EITHER 1A OR 2A HIGH BUT NOT BOTH (chip pin 2 and pin 7 respectively)

Now that we have decoded the logic of the magic chip, we can start to wire our breadboard and RPi.

Wiring

The Fritzing drawing (next page, top right) shows our wiring diagram for this month. Notice that we are only using one half of the chip, so we could actually control two small DC motors instead of just one. That, however, will be up to you to experiment with.

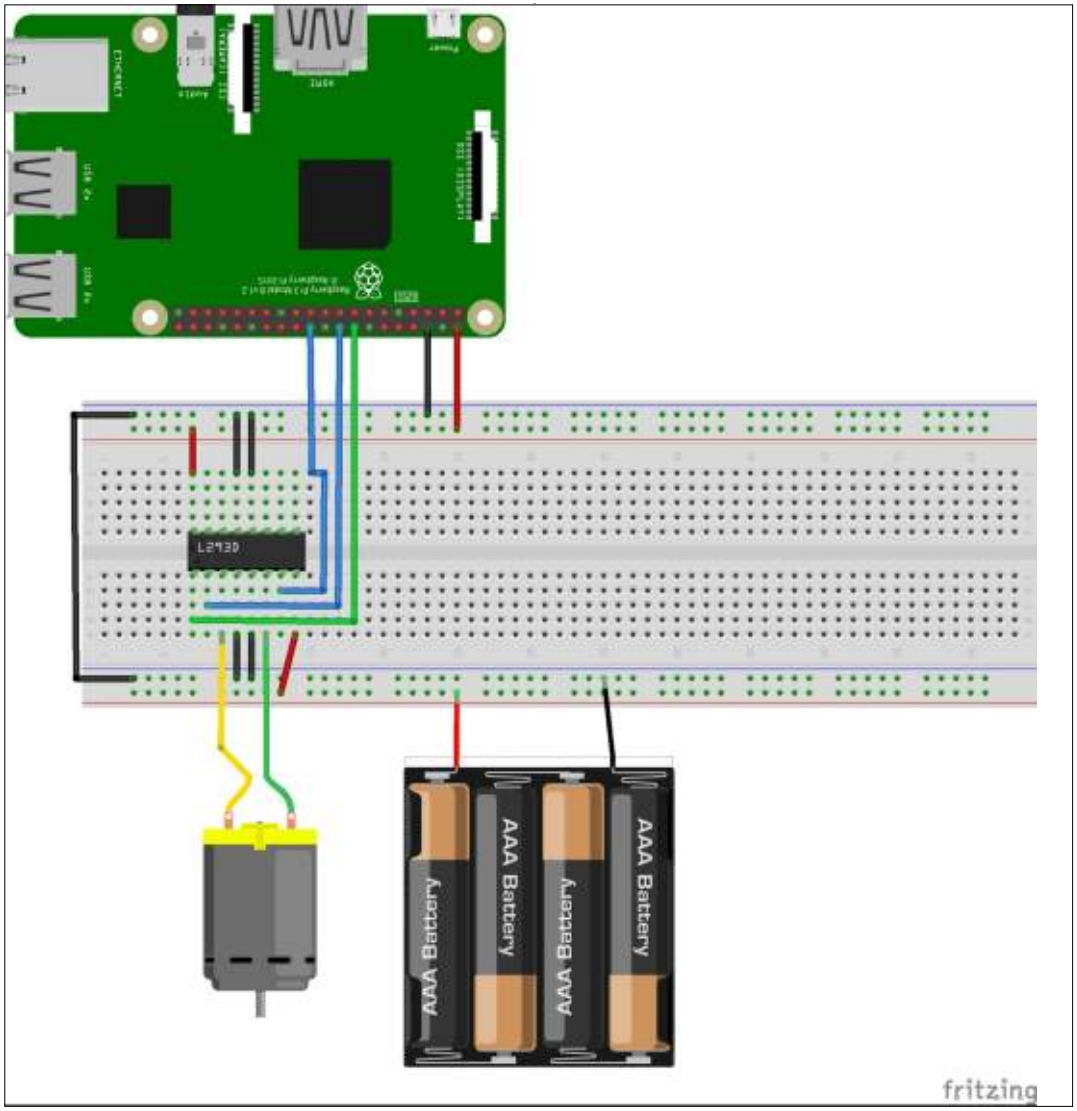
As always, make the wiring connections to the RPi BEFORE you power the RPi on. Also double check your wiring, especially since we have an external power source. You might not be happy if something is on the wrong pin.

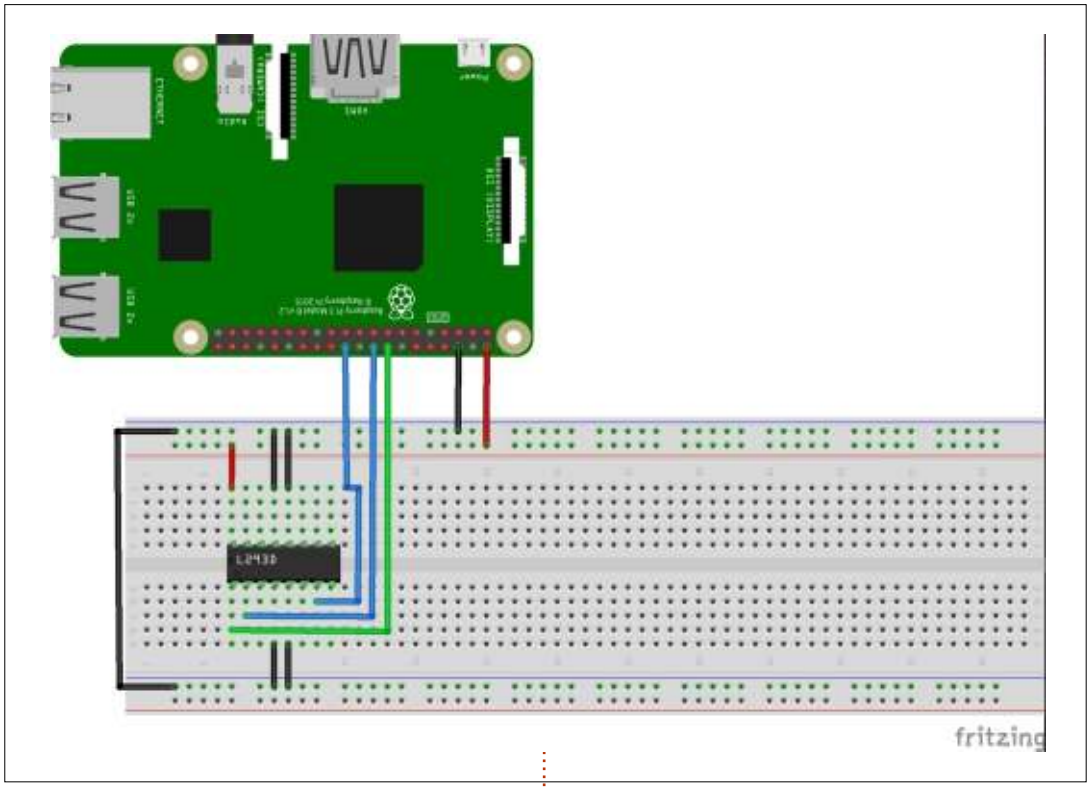
This first Fritzing image shows

the connections to the RPi and to the breadboard/chip. Basically it breaks down like that shown in the table bottom right

The next Fritzing diagram (below) shows the battery and motor hook-ups.

We are using the +5 VDC power from the RPi to power the motor driver chip (RPi pin 2 to L293D pin 16). While the above diagram shows AAA batteries, you can use a battery pack that uses AA batteries as well. We are also providing Ground from the RPi (pin





FROM		TO	
RPi	Pin 2 (+5VDC)	Breadboard	VCC Rail
RPi	Pin 6 (Gnd)	Breadboard	Gnd Rail
RPi	Pin 16 (GPIO 23)	Chip	Pin 1 (Enable)
RPi	Pin 18 (GPIO 24)	Chip	Pin 2 (1A)
RPi	Pin 22 (GPIO 25)	Chip	Pin 7 (2A)
Chip	Pins 4,5,12,13	Breadboard	Gnd Rail
Chip	Pin 16	Breadboard	VCC Rail
Chip	Pin 8	Breadboard	BOTTOM VCC Rail
Chip	Pins 12,13	Breadboard	BOTTOM Gnd Rail
Breadboard	Gnd Rail Top	Breadboard	Gnd Rail Bottom

6) to the chip (pins 4,5,12,13). The motor is driven on chip pin 3 (1A) and pin 5 (2A). The battery connects to chip pin 8 to provide the voltage for the motor.

Code

We will deal with code in two programs. The first simply turns on the motor, runs for a few seconds then stops it. The second is a modified version of the first that shows how to reverse the motor.

Dcmotor1.py

This program (below) will simply turn on the motor in forward (clockwise) mode and let it run, then stop it. Basically it just proves that everything is working correctly.

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

from time import sleep

```
GPIO.setmode(GPIO.BCM)
GPIO.setup(23,GPIO.OUT)   # 1A
GPIO.setup(24,GPIO.OUT)   # 2A
GPIO.setup(25,GPIO.OUT)   # Enable
GPIO.output(24,GPIO.LOW)
```

Set everything up and set 2A to low.

```
print "Starting motor"
GPIO.output(23,GPIO.HIGH)
GPIO.output(25,GPIO.HIGH)

sleep(5)
```

Set 1A to HIGH and Enable to HIGH to start the motor and let it run for 5 seconds.

```
print "Stopping motor"
GPIO.output(25,GPIO.LOW)
sleep(2)
GPIO.cleanup()
```

Stop the motor by setting the Enable to LOW, sleep for 2 seconds, then run GPIO.cleanup().

The first part of the program will be used in the next one.

Dcmotor2.py

In this program (next page), we set up the GPIO pins just as we did before, but we are now using PWM to modulate the speed of the motor. If you don't remember PWM, please refer to Part 64 back in FCM 107.

In the forward mode, the longer the duty cycle (closer to 100) means the motor will go faster.

In the reverse mode, the SHORTER the duty cycle (closer to 0) means the motor will go faster.

We speed up the motor by setting the duty cycle to a LOWER percentage, let it run for 5 seconds, then stop it, do a GPIO.cleanup(), then end the program.

Well, that's it for this month. Next month, we will be working with servos. All you need is a small inexpensive one with three wires. We will not be using parts from this month's project, but keep them for future projects.

Until then, have fun.

Greg Walters is owner of RainyDay Solutions, LLC, a consulting company in Aurora, Colorado, and has been programming since 1972. He enjoys cooking, hiking, music, and spending time with his family.

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

from time import sleep

```
GPIO.setmode(GPIO.BCM)
GPIO.setup(23,GPIO.OUT)   # 1A
GPIO.setup(24,GPIO.OUT)   # 2A
GPIO.setup(25,GPIO.OUT)   # Enable
GPIO.output(24,GPIO.LOW)
```

As I stated earlier, the above code is pretty much the same thing as we started with in dcmotor1.py.

```
fwd = GPIO.PWM(23,40)
```

We are setting pin 23 to be a PWM Output line with 40% duty cycle (on 40% of the time and off 60% of the time.

```
print "Starting motor"
GPIO.output(25,GPIO.HIGH)
fwd.start(70)
sleep(5)
```

We start the motor by setting the enable to High and setting the Duty Cycle to 70. The motor will run for 5 seconds.

```
print "Stopping motor"
GPIO.output(25,GPIO.LOW)
sleep(2)
```

Now, we stop the motor by setting enable to low.

```
print "Starting motor in reverse"
rev = GPIO.PWM(24,50)
GPIO.output(23,GPIO.LOW)
GPIO.output(25,GPIO.HIGH)
rev.start(50)
sleep(5)
```

We now set the motor to reverse (pin 23 to low and starting the PWM duty cycle to 50% and run for 5 seconds…

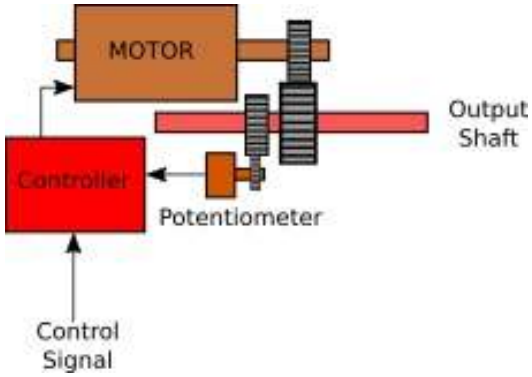
```
print "Speeding up the motor..."
rev.ChangeDutyCycle(10) # When reversing the motor, a smaller duty
                        # Cycle means faster.
sleep(5)
print "Stopping motor"
GPIO.output(25,GPIO.LOW)
GPIO.cleanup()
```

Python In The Real World - Pt 71

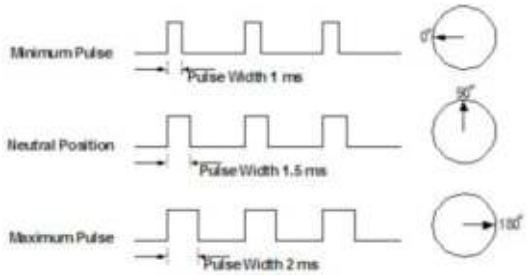
Written by Greg D. Walters

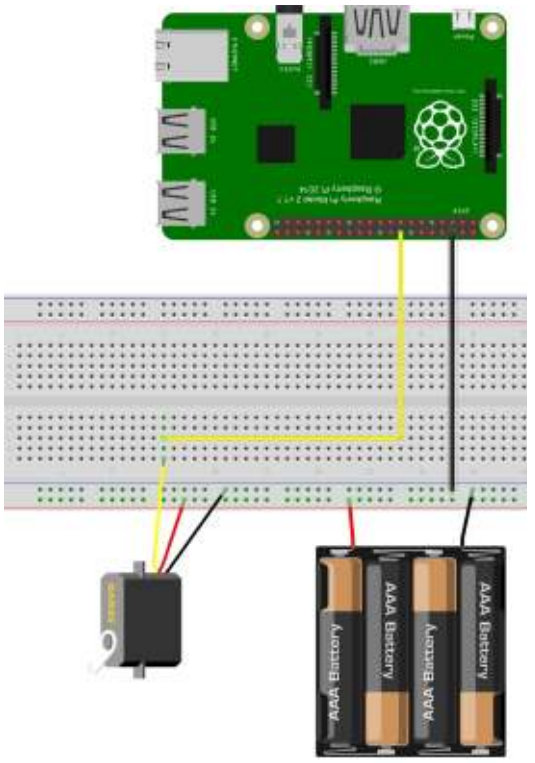
This month, we are going to interface a servo motor to our RPi. This requires only a servo motor, the Raspberry Pi, the breadboard, some jumpers, and the battery pack we used last month.

A servo motor is simply a motor that has a control circuit and a potentiometer to give the control circuit the position of the output shaft. MOST servos will rotate the shaft between 0° and 180°. There are some that go 360° and more, but they cost a lot. Gears do all the connections between the motor, the potentiometer, and the output shaft. We provide the power through batteries or another external power source, and the RPi will send out the control signals. Most servos have only three wires, Positive voltage, Negative voltage (ground) and Control Signal. Colors of the wires vary from manufacture to manufacture, but the voltage wires should be close in colour to red and black, and the control wire is the only one left. When in doubt, check for a data sheet from the manufacture.



The control signals are expected in a very specific "format", and we will use PWM (Pulse Width Modulation) for that. First, the pulses must come every 20 milliseconds. The width of the pulse determines where the output shaft turns to. If the pulse is 1 ms in width, the motor will move toward 0°. If the pulse is 1.5 ms, then the shaft moves toward 90°. If the pulse is 2 ms, the shaft moves toward 180°





The Wiring

The connections are very simple this month. The battery pack powers the motor, so +voltage on the servo goes to the + voltage rail, and the negative servo wire goes to the negative rail. We connect the negative voltage from the battery pack (negative rail) to pin 6 of the RPi. GPIO pin 23 (pin 16) goes to the control wire of the servo.

Now for some math. As we discussed earlier, the servo expects a signal every 20 ms in order to work, and we need to keep sending those pulses to keep the output shaft in the position that we want it in. The GPIO command to set the Pulse Width Modulation is

```
Pwm = GPIO.pwm({RPi Pin},{Frequency})
```

We know the pin number (23), but we need to convert the 20 ms to Hertz in order to set the pwm setup command. How do we do that? It's simple.

```
Frequency = 1/time
Frequency = 1/.02 (20ms)
Frequency = 50 Hertz
```

So now, when we set up our code, we can set the GPIO.pwm command to the control pin and use 50 for our frequency.

Our first program will start somewhere close to 0° and then move to close to 90° then move to

close to 180°. I keep saying close, because every servo is a bit different. We don't want to set the DutyCycle to 0 and have it slam to the limit and potentially damage the servo, so we will start off with a low number close to 0 and end with a number close to 12 in order to start to "dial in" a set of values that work. For my servo, the numbers 3 for 0° and 12 for 180° worked well.

SERVO1.PY

```
import RPi.GPIO as GPIO
from time import sleep

GPIO.setmode(GPIO.BCM)
GPIO.setup(23,GPIO.OUT)

pwm = GPIO.PWM(23,50)
pwm.start(2)
pwm.ChangeDutyCycle(3)
sleep(5)
```

3 should give you the first angle. You might need to try changing it to 2,1,0 or 4 to get it there. Write down this number.

```
pwm.ChangeDutyCycle(6)

sleep(5)
```

This should put the rotor into the centre position (90°). If you have changed the first number or the next number, this will have to change as well.

```
pwm.ChangeDutyCycle(12)

sleep(5)
```

The final number should take you to the 180° position. Again, try a few numbers one side or the other to tweak this, and once you have it, write down the number.

```
GPIO.cleanup()
```

Finally we call GPIO.cleanup() to set everything to normal.

Now comes the heavy duty math part. We will use the values 3 and 12 as y1 and y2 respectively for the formula. Substitute your numbers.

```
Offset = (y2-y1)/(x2-x1)
Offset = (12-3)/(180-0)
Offset = 9/180
Offset = .05
```

Now when we set the DutyCycle based on an angle between 0° and 180° we use the following formula

```
DutyCycle = (Offset * angle)
+ 2.0

DutyCycle = (.05 * angle) +
2.0
```

Servo2.py

```
import RPi.GPIO as GPIO
from time import sleep
GPIO.setmode(GPIO.BCM)
GPIO.setup(23,GPIO.OUT)
pwm = GPIO.PWM(23,50)
pwm.start(2)
def SetAngle(angle):
        DutyCycle=(.061 * angle) + 2.0
        print(DutyCycle, angle)
        pwm.ChangeDutyCycle(DutyCycle)

try:
        while True:
                for a in range(0,180):
                        SetAngle(a)
                        sleep(.05)

                for a in range(180,0,-1):
                        SetAngle(a)
                        sleep(.05)

except KeyboardInterrupt:
        GPIO.cleanup()
```

When I did this, it worked, but I found the value of .061 worked a little bit better.

Well, that's about it for this month. Next time, we will be working with a stepper motor, somewhat a cross of both a servo and a regular motor.

Until then, enjoy!

Greg Walters is owner of RainyDay Solutions, LLC, a consulting company in Aurora, Colorado, and has been programming since 1972. He enjoys cooking, hiking, music, and spending time with his family.

Python In The Real World - Pt 72

Written by Greg D. Walters

Welcome back for another entry into what I lovingly call 'Greg's Python Folly'. As promised, we will be working on interfacing a stepper motor to the Raspberry Pi. You will need your Raspberry Pi, a hobby stepper motor, a 4 x AA size battery pack, the L293D driver chip we used previously, a breadboard, and some jumpers.

While I was doing research for this particular project, I stumbled across a tutorial at tutorials-raspberrypi.de. I was so impressed by the information at this website, I am using the majority of their information and code in this article. The website is: http://tutorials-raspberrypi.com/how-to-control-a-stepper-motor-with-raspberry-pi-and-l293d-uln2003a/. If you get confused by my explanations, you can always drop by and maybe get some clarifications.

The motor I chose is a Radio Shack mini stepper motor. Basically it is a 28BJY-48 low-voltage stepper. Before you try to interface any stepper motor, please research the data sheet for as much information as you can get. In this case, the data sheet is located at: http://www.tutorials-raspberrypi.de/wp-content/uploads/2014/08/Stepper-Motor-28BJY-48-Datasheet.pdf

Now, let's examine stepper motors in general, then we'll expand that information to the 28BJY specifically and work on interfacing it to the Pi through our L293D driver chip.

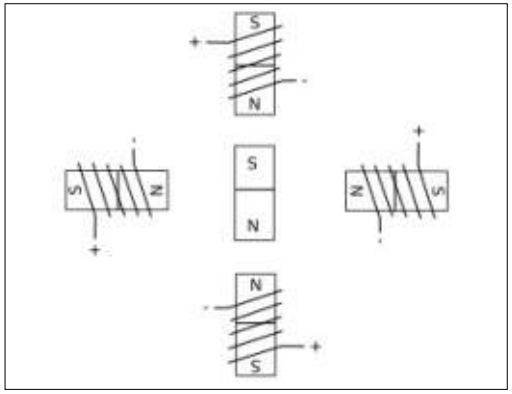
Stepper Motors

Stepper motors are used in robotics and in CNC type machines where you want the ability to move an item to a specific location easily. There are two basic types of stepper motors, one called Unipolar and one called Bipolar. The difference will become obvious as we go through this tutorial. The 28BJY is a Bipolar motor and also has a gearing system.

In both models, there are multiple electromagnetic coils that are turned on and off in a sequence to make the motor turn. Each time we apply power to one of the coils, the motor rotates a small amount (if powered in the correct sequence for the motor), called a step, hence the name stepper motor.

Unipolar Motors

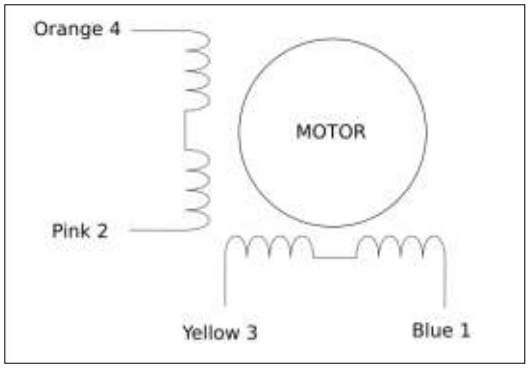
Unipolar motors have coils that are powered in only one direction, hence the UNI in Unipolar. The rotor of the motor is controlled by powering the various electromagnetic coils on and off in a specific sequence for a certain amount of time. In a simplified version of this model, let's look at the following diagram…



Turning on each coil one at a time will cause the magnet in the rotor to turn toward the proper coil. Using a clock face as a guide, turning on the coils in the sequence of 12 o'clock, 3 o'clock, 6 o'clock, 9 o'clock and then again at 12 o'clock will cause the rotor to turn clockwise one full rotation. This requires 4 "steps" to make one rotation. This is called the Unipolar wave. If we go a bit further, we could make a more granular movement by alternating the coils from a single coil turned on and then turning on the next coil as well, which makes the rotor turn in an eighth turn when both coils are turned on. The sequence would then be: 12, 12 and 3, 3, 3 and 6, 6, 6 and 9, 9, 9 and 12, and then finally 12 alone again. This then is 8 steps per rotation which is called half stepping. To make the motor reverse (counter-clockwise), we simply reverse the sequence. This is a VERY simple representation, and many stepper motors have a resolution that can be as high as 200 steps per revolution.

Bipolar Motors

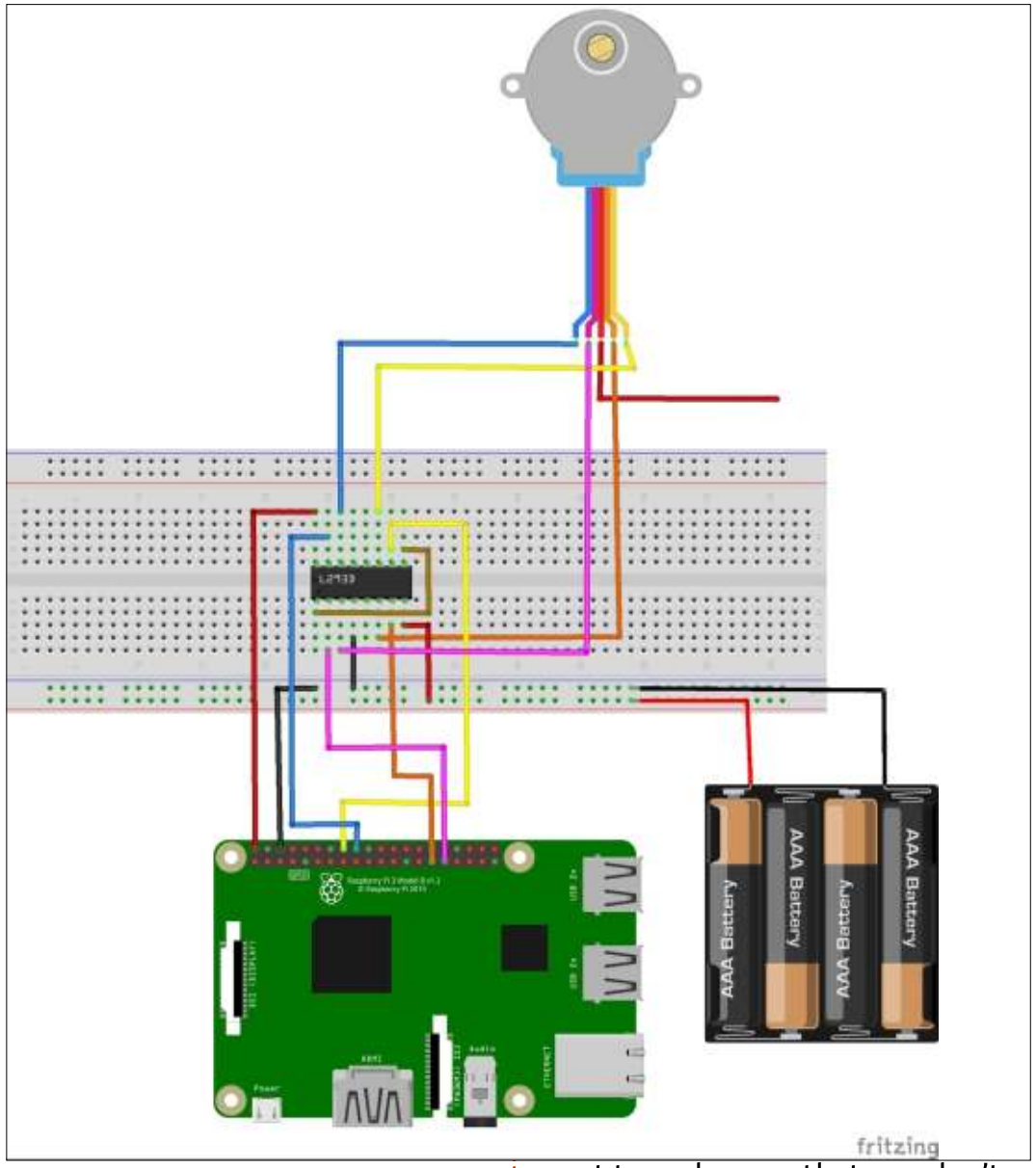
The 28BJY, as I stated earlier, is a Bipolar motor. In this case, the coils can have their current reversed and two coils are powered at any time. This creates a situation where the switching is more complex, but the amount of turn force (power) of the rotor is increased. A simple block diagram of the 28BJY is shown below.



The numbers shown with the colors of the wires are for the 28BJY and yours may be different. The wire connector (if there is one) might differ from unit to unit. You can use an ohmmeter to verify the coils.

The Wiring

A couple of words of warning on this before we start.



First, do all of your wiring BEFORE you power on the Raspberry Pi. We are working with an external power source, so you want to make sure that you don't short any wires or apply the battery power to the wrong pin.

Second, BE SURE of your wiring before you power on your RPi. If you get the wiring confused, at best your project will not work and the motor will just sit there and buzz.

When you look at the fritzing drawing, it looks fairly simple (and it is). I made sure that the wiring from the RPi to the driver chip were the same color as the intended segment of the motor. We will be using only 4 of the 5 motor wires. The red one (if yours has a red one) is not connected for this project.

Since the central component in this project is the L293D driver chip, here is a quick breakdown to try to make things easier for you...

```
L293D
Pin  1 -> Pin 9
Pin  2 -> Pi GPIO 6
Pin  3 -> Motor Pink
Pin  4 -> Breadboard Negative
Rail
Pin  5 -> No Connect
Pin  6 -> Motor Orange
Pin  7 -> Pi GPIO 5
Pin  8 -> Breadboard Positive
Rail
Pin  9 -> Pin 1
Pin 10 -> PI GPIO 23
Pin 11 -> Motor Yellow
Pin 12 -> No Connect
Pin 13 -> No Connect
Pin 14 -> Motor Blue
Pin 15 -> Pi GPio 24
Pin 16 -> Pi +5VDC
```

If you follow this, you should have no problems with the wiring.

The Code

As always, I will discuss the code in blocks. So let's get started.

```
import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)
coil_A_1_pin = 6 # pink
coil_A_2_pin = 5 # orange
coil_B_1_pin = 23 # blue
coil_B_2_pin = 24 # yellow
```

Here we are simply defining the imports, setting the GPIO mode, and setting the warnings to False so we don't get any nagging notices about already initialized pins. We also define which GPIO pins control the motor coils through the driver chip.

```
# adjust if different
StepCount = 8
Seq = range(0, StepCount)
Seq[0] = [0,1,0,0]
Seq[1] = [0,1,0,1]
Seq[2] = [0,0,0,1]
Seq[3] = [1,0,0,1]
Seq[4] = [1,0,0,0]
Seq[5] = [1,0,1,0]
Seq[6] = [0,0,1,0]
Seq[7] = [0,1,1,0]
```

Now this is the key to making our project work. This motor wants to have 8 steps (internal) per revolution of the motor (per the data sheet). We also define the sequence of which coil(s) are energized per step as a series of lists. Each sequence array explains which coil(s) is energized at any given time.

```
GPIO.setup(coil_A_1_pin,
GPIO.OUT)
GPIO.setup(coil_A_2_pin,
GPIO.OUT)
GPIO.setup(coil_B_1_pin,
GPIO.OUT)
GPIO.setup(coil_B_2_pin,
GPIO.OUT)
```

Here we are going through the setup steps, defining each of our pins used as outputs.

```
def setStep(w1, w2, w3, w4):

GPIO.output(coil_A_1_pin, w1)

GPIO.output(coil_A_2_pin, w2)

GPIO.output(coil_B_1_pin, w3)

GPIO.output(coil_B_2_pin, w4)
```

This subroutine is called each time we want to step the motor and we pass a 0 or 1 to each coil wire port on the driver chip to energize or deenergize the various coils to turn the rotor.

```
def forward(delay, steps):
        for i in range(steps):
                for j in range(StepCount):
                        setStep(Seq[j][0], Seq[j][1], Seq[j][2], Seq[j][3])
                        time.sleep(delay)

def backwards(delay, steps):
        for i in range(steps):
                for j in reversed(range(StepCount)):
                        setStep(Seq[j][0], Seq[j][1], Seq[j][2], Seq[j][3])
                        time.sleep(delay)
```

These two routines allow for easily commanding the motor forwards or backwards a specific number of steps in the proper direction.

```
if __name__ == '__main__':
        while True:
                delay = raw_input("Time Delay (ms)?")
                steps = raw_input("How many steps forward? ")
                forward(int(delay) / 1000.0, int(steps))
                steps = raw_input("How many steps backwards? ")
                backwards(int(delay) / 1000.0, int(steps))
```

And finally our "main" routine which loops over and over again asking the amount of the time delay and the number of steps in that given direction. For my motor, it takes 512 steps to make close to a full rotation.

On my system, with my motor, a time delay of 1ms works well. However, you might have to add a few milliseconds to yours for it to work.

Notice I stated that it takes 512 steps to make CLOSE to a full

rotation. This motor has a 64:1 gearing ratio, which leaves a rather ugly fractional step angle. But for the purposes of this tutorial, it works pretty well.

If you want to learn more about stepper motors, adafruit.com has a very nice article on the subject.

Hopefully you have enjoyed the series so far. Next up will be learning to use the Arduino microcontroller board. We'll use this information in the third section of the series where we control the Arduino with a Raspberry Pi (or other computer). So, that having been said, you should be ready and have an Arduino (Uno or Mega) ready and dust off the components we used in the early part of this series for next time.

Until then, keep learning and above all, HAVE FUN!

Greg Walters is owner of RainyDay Solutions, LLC, a consulting company in Aurora, Colorado, and has been programming since 1972. He enjoys cooking, hiking, music, and spending time with his family.

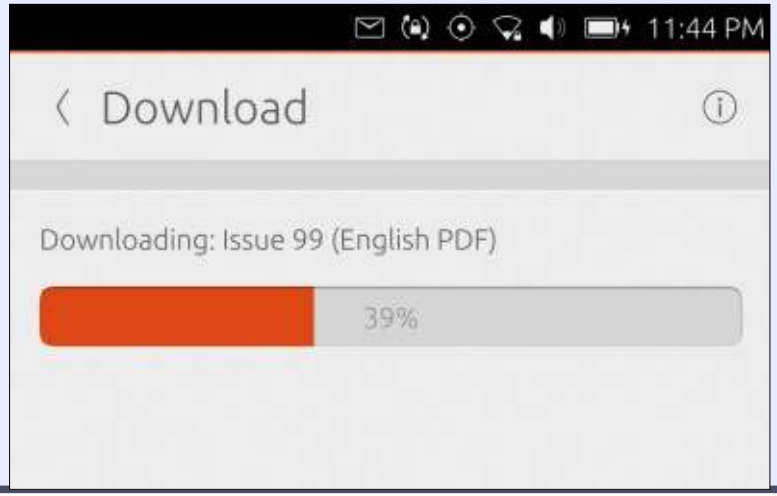
The Official Full Circle App for Ubuntu Touch

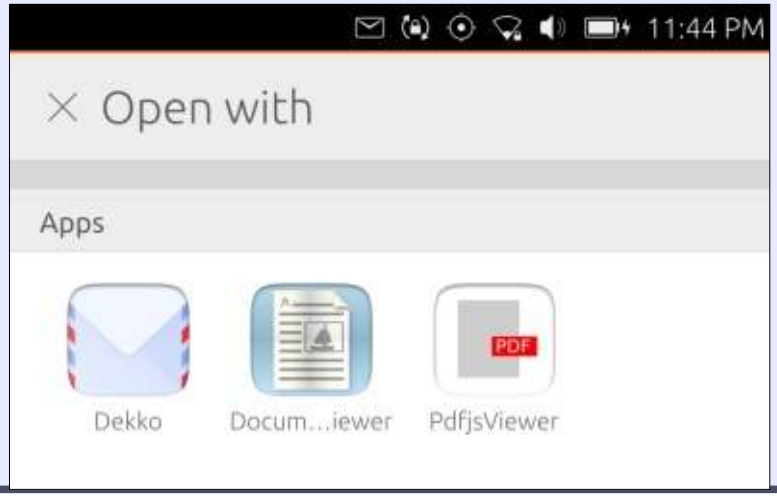
Brian Douglass has created a fantastic app for Ubuntu Touch devices that will allow you to view current issues, and back issues, and to download and view them on your Ubuntu Touch phone/tablet.

Install

Either search for 'full circle' in the Ubuntu Touch store and click install, or view the URL below on your device and click install to be taken to the store page.

https://uappexplorer.com/app/fullcircle.bhdouglass





HOW TO CONTRIBUTE

FULL CIRCLE NEEDS YOU!

A magazine isn't a magazine without articles and Full Circle is no exception. We need your opinions, desktops, stories, how-to's, reviews, and anything else you want to tell your fellow *buntu users.
Send your articles to: articles@fullcirclemagazine.org

We are always looking for new articles to include in Full Circle. For help and advice please see the Official Full Circle Style Guide: http://url.fullcirclemagazine.org/75d471

Send your comments or Linux experiences to: letters@fullcirclemagazine.org
Hardware/software reviews should be sent to: reviews@fullcirclemagazine.org
Questions for Q&A should go to: questions@fullcirclemagazine.org
Desktop screens should be emailed to: misc@fullcirclemagazine.org
... or you can visit our site via: fullcirclemagazine.org

Please note:

Special editions are compiled from originals and may not work with current versions.

Full Circle Team

Editor - Ronnie Tucker
ronnie@fullcirclemagazine.org

Webmaster - Lucas Westermann
admin@fullcirclemagazine.org

Special Editions - Jonathan Hoskin

Editing & Proofreading
Mike Kennedy, Gord Campbell, Robert Orsino, Josh Hertel, Bert Jerred, Jim Dyer and Emily Gonyer

Our thanks go to Canonical, the many translation teams around the world and Thorsten Wilms for the FCM logo.

For the Full Circle Weekly News:

You can keep up to date with the Weekly News using the RSS feed: http://fullcirclemagazine.org/feed/podcast

Or, if your out and about, you can get the Weekly News via Stitcher Radio (Android/iOS/web):
http://www.stitcher.com/s?fid=85347&refid=stpr

and via TuneIn at: http://tunein.com/radio/Full-Circle-Weekly-News-p855064/

Getting Full Circle Magazine:

EPUB Format - Most editions have a link to the epub file on that issues download page. If you have any problems with the epub file, email: mobile@fullcirclemagazine.org

Issuu - You can read Full Circle online via Issuu: http://issuu.com/fullcirclemagazine. Please share and rate FCM as it helps to spread the word about FCM and Ubuntu.

Magzster - You can also read Full Circle online via Magzster: http://www.magzter.com/publishers/Full-Circle. Please share and rate FCM as it helps to spread the word about FCM and Ubuntu Linux.