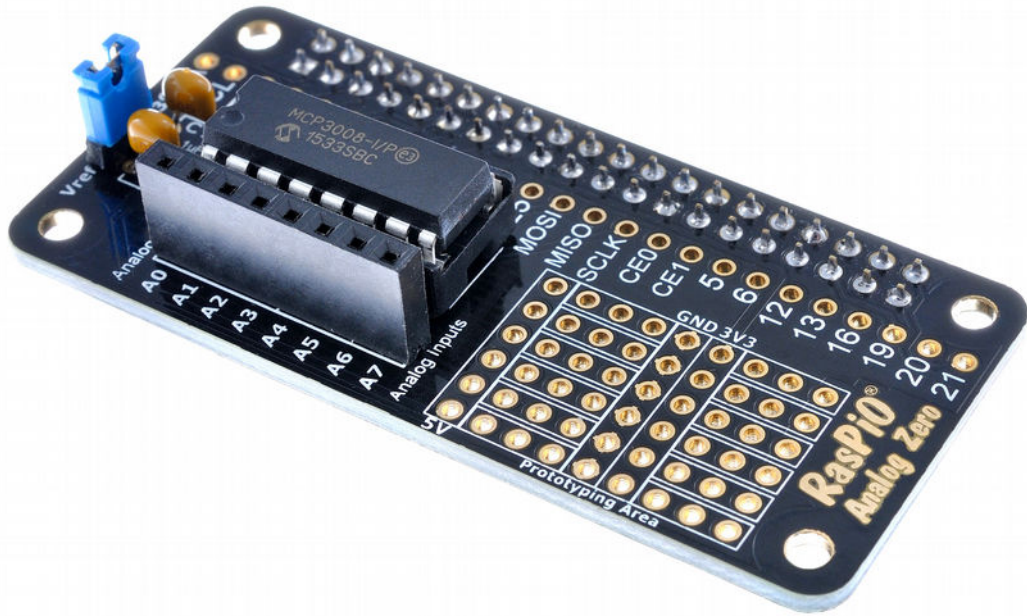


Using Your *RasPi0 Analog Zero*



To Have Fun With GPIO Zero

by Alex Eames

Introduction

Adding 8 Channels of Analog Input

The RasPiO^{®1} Analog Zero offers a compact, inexpensive, easy way to add eight analog² channels to your Raspberry Pi.

This enables simultaneous reading of 8 input Voltages up to 3.3V (or more with some tricks I'll show you later). This can be used to...

- read sensors
- use potentiometer dials for control or display
- make a weather station
- make a digital thermometer
- make a multi-channel Voltmeter

GPIO Zero

Ben Nuttall and Dave Jones have created GPIO Zero as the ideal way into Python GPIO programming. Using it with the RasPiO Analog Zero means there is *nothing to install* before you can start playing.

Also, by keeping the board inexpensive, I hope it's realistic for individuals, schools and jams to be able to get hold of some and discover the joys of measuring and controlling the world with the RasPiO Analog Zero, Raspberry Pi and GPIO Zero.

All code in this guide and in the [analogzero Github repository](#) is in Python 3.

1 RasPiO is a trademark of Alex Eames. Raspberry Pi is a trademark of the Raspberry Pi Foundation

2 I use the US spelling for the word analog. In British English it's spelt analogue, but I got in the habit of using analog when learning Arduino programming. Perhaps it's a silly reason, but everyone has their quirks.

Analog Zero Instructions

The RasPi0 Analog Zero uses the BCM GPIO port numbering scheme. This is a perfect match for GPIO Zero.

Hardware Technical Overview

This page is mainly for the technically minded. If you just want to get on with experimenting, you can skip to the next section.

MCP3008

RasPi0 Analog Zero uses an MCP3008 analog to digital converter. It's an SPI driven, 10-bit, 8-channel ADC. The [MCP3008 datasheet is here](#).

Ports Used by the Board

The MCP3008 analog to digital converter chip is connected to the SPI ports MOSI, MISO, SCLK and CE0. All the Pi's GPIO ports³ are broken out to through-holes.

V_{ref} is Tweakable

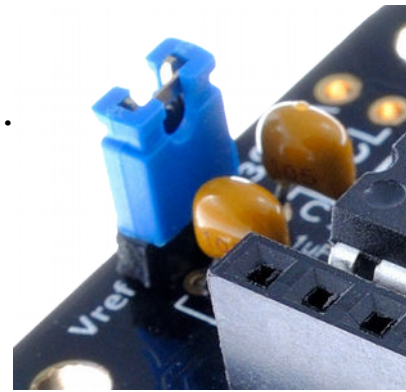
The MCP3008 is powered by the Pi's 3.3V (3V3) rail. This means that **the highest voltage the chip can measure directly is 3.3V**.

If you are reading a sensor that outputs a lower voltage, you can tweak V_{ref} to a lower value in order to set the full-scale range of the 10-bit (1023 steps) resolution. *e.g.* a TMP36 temperature sensor outputs 1V at 50°C. If you're going to be measuring temperatures below that, tweaking V_{ref} to 1.0V would get you 0.1°C resolution. If you used 3V3, you'd have 0.33°C resolution.

³ Except GPIO26

By default V_{ref} is set to 3V3 by placing the jumper to connect V_{ref} to 3V3 (as in the photo).

To set V_{ref} to your own value (not greater than 3.3V) connect the V_{ref} pin to your chosen voltage source (it must have common GND with the Pi).



V_{ref} pin showing jumper

SPI Usage

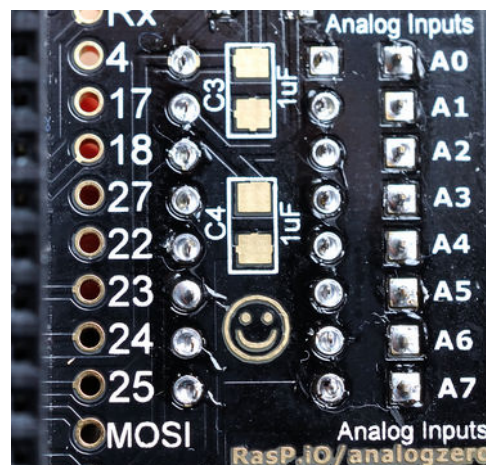
The Pi's SPI can handle two devices natively. If you wish to add another SPI device, ensure its chip-select pin is connected to CE1 or it will interfere with the MCP3008 chip, which uses CE0.

SMT Pads On Rear

There are two surface mount pads on the rear of the board.

These give an alternative location for the bypass capacitors (for V_{DD} and V_{ref}).

If you prefer to use these you can fit your own 1 μ F 1206 capacitors (not supplied).



SMT pads on reverse of board

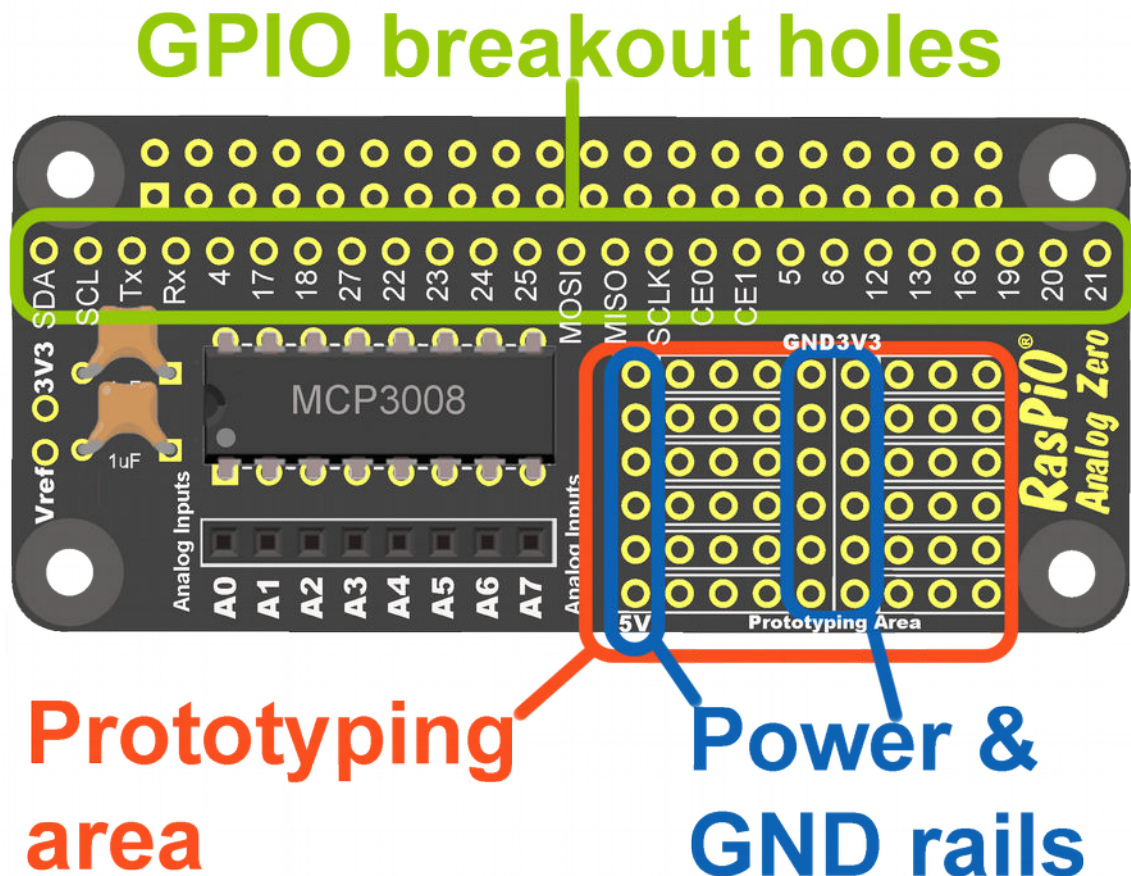
If You Want More Than 10-bit Resolution...

The MCP3008 is a 10-bit ADC. 10-bit gives you $2^{10} - 1 = 1023$ steps of resolution (values 0 to 1023).

If you have an application that requires greater resolution, the 12-bit MCP3208 chip (not supplied) is a pin-compatible drop-in replacement. It's also supported in GPIO Zero. 12-bit provides 4095 steps of resolution.

Know Your RasPiO Analog Zero

The RasPiO Analog Zero has been designed to fit directly⁴ on any 40-pin consumer model of Raspberry Pi and make it as easy as possible for people to get into analog sensing and control using GPIO Zero on the Pi.



Anatomy of RasPiO Analog Zero

The RasPiO Analog Zero connects an MCP3008 ADC chip to your Pi, breaks out the GPIO ports and provides a prototyping area with power and ground rails, where you can add components of your choice.

There is also an 8-way female header for the analog inputs if you choose to fit it.

⁴ It can fit on older 26-pin Pis with the use of a 26-pin stacking header. The MCP3008 will still work, but some of the GPIO breakouts (the bottom 8) will not

Analog to Digital Converters

We live in an analog world, but computers can only “speak” digital information. For a computer to process information, it has to be converted into 0s and 1s, (On/Off, HIGH/LOW, 0V/3V3 etc).

So we need a mechanism of converting the analog information around us into digital form so that we can do something with it. To do this, we use a device called an analog to digital converter (also known as ADC or AD).

Your eyes, with their vast numbers of rods and cones are an incredible example of an analog to digital converter. Even the most modern camera sensor with all its millions of pixels cannot simultaneously determine as many colours, shades and intensities as your eyes.

Analog information is continuous, whereas digital information comes in steps. If you have enough steps, you can get a usefully high degree of measurement precision. The MCP3008 is a 10-bit ADC. This means that it has $2^{10} - 1 = 1023$ steps. This is referred to as “resolution”. In practice, the ADC output values are 0 to 1023. (In GPIO Zero, this is converted into a float variable where 0 is 0 and 1023 = 1.0)

ADCs are used to measure the voltage of a signal. Most sensors are designed to output a voltage proportional to the property they are measuring. So this gives us an elegant way of getting information from the analog world into the Raspberry Pi.

The MCP3008 ADC is powered via 3V3 from the Raspberry Pi. This is also its default reference voltage⁵ V_{ref} . So if the measured analog input signal is 3.3V, the ADC will output 1023. If the input signal is 0V, the ADC output will be 0.

⁵ See Vref is Tweakable to find out how to adjust the reference voltage

If we divide 3.3V by 1023, we get the resolution of the device. $3.3V / 1023 = 0.00322 \text{ V/step}$. That's 3 mV.

If we want to read an analog sensor's voltage we do the following calculation...

$\text{ADC reading} / 1023 * 3.3 \text{ V} = \text{Sensor Voltage}$

From the sensor voltage, we can usually calculate temperature, pressure or whatever our sensor is measuring.

But we can just measure and report the voltage(s) as well.

GPIO Zero Converts ADC Output For Us

When using GPIO Zero, the ADC output (0 to 1023) is converted into a float variable where $0 = 0$ and $1023 = 1.0$

This can simplify our calculations, as we'll see a little later on in the code section.

Soldering Instructions

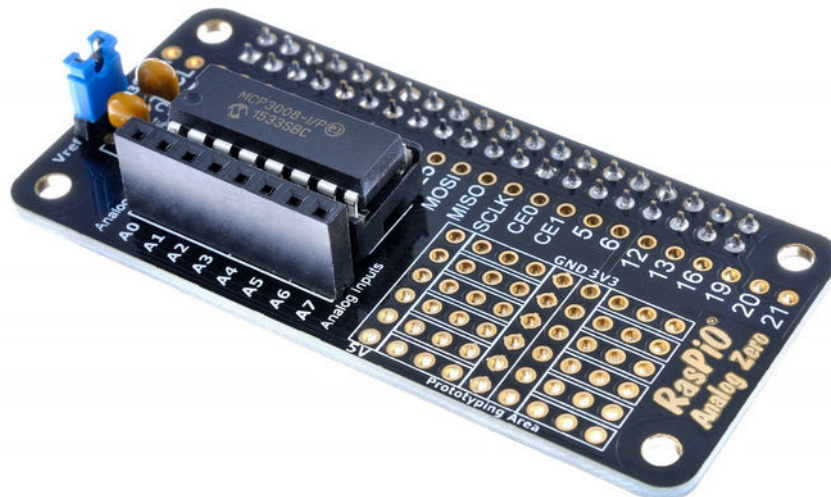
If you prefer an assembly video, you can find one here, with lots of background information. It's 17 minutes, but covers every single joint...

<https://youtu.be/HjzZm9Rgaks>

It's usually best to start with "low" components and get progressively higher. Suggested assembly order is...

1. chip socket
2. capacitors (doesn't matter which way round they are)
3. 2-way male header
4. 8-way female header
5. 40-way GPIO header
6. gently roll the legs of the chip on a flat surface to push them inwards slightly, then press into the chip socket with the dimple at the top end (next to capacitors)
7. fit the jumper connecting V_{ref} to 3V3

Now it should look something like this...



If you want to use the Pi's hardware SPI capability, you can ensure that SPI is enabled [in the same way as enabling i2c here](#). GPIO Zero will fall back to 'bit-banging' if SPI is not enabled, but should still work.

Basic Usage of GPIO Zero with MCP3008

To read and display the value of a single analog channel...

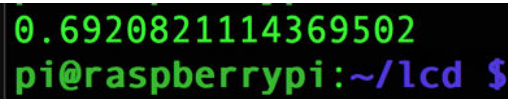
```
from gpiozero import MCP3008
adc = MCP3008(channel=0, device=0)
print(adc.value)
```

`channel=0` specifies that we want to read channel 0 (A0).

`device=0` specifies SPI device 0.

The default is 0, which is correct for the RasPiO Analog Zero, so it may be omitted.

`adc.value` returns a float variable from 0.0 to 1.0. The output will look something like this...



A terminal window with a black background. The first line shows the output '0.6920821114369502' in green text. The second line shows the prompt 'pi@raspberrypi:~/lcd \$' in blue text.

Read & display a single channel

This is a scaled representation of the 0 to 1023 that the MCP3008 returns via SPI (in this case 0.69208 represents 708).

To read and display the voltage of a single analog channel...

```
from gpiozero import MCP3008
adc = MCP3008(channel=0)
voltage = 3.3 * adc.value
print("channel 0 voltage is: ", voltage)
```

The only thing we're doing differently here is multiplying the `adc.value` by 3.3 to calculate the value of the voltage at A0 (ADC channel 0). `voltage` will be a float variable from 0.0 to 3.3. The script

output will look something like this...

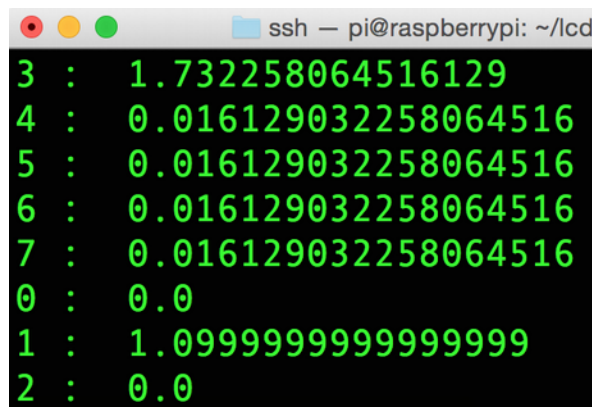
```
channel 0 voltage is: 2.2903225806451615
pi@raspberrypi:~/lcd $
```

As you can see, this is just the raw output.

Continually read/display voltage of all 8 analog channels...

```
from gpiozero import MCP3008
from time import sleep
voltage = [0,0,0,0,0,0,0,0]
vref = 3.3

while True:
    for x in range(0, 8):
        with MCP3008(channel=x) as reading:
            voltage[x] = reading.value * vref
        print(x,": ", voltage[x])
    sleep(0.1)
```



```
3 : 1.732258064516129
4 : 0.016129032258064516
5 : 0.016129032258064516
6 : 0.016129032258064516
7 : 0.016129032258064516
0 : 0.0
1 : 1.0999999999999999
2 : 0.0
```

Output from the above script

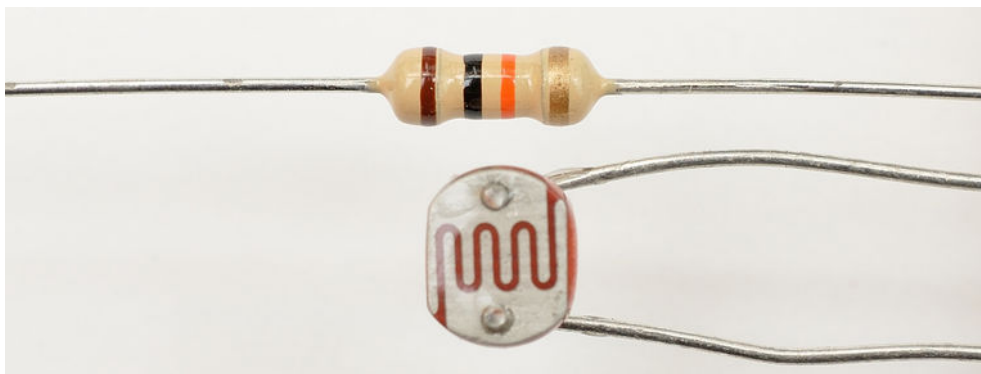
This was a quick overview of the basics to get you up and running. We'll make the raw output look more presentable a bit later on. Now let's look at some specific experiments and circuits...

Using the MCP3008 with GPIO Zero

LDR Circuit

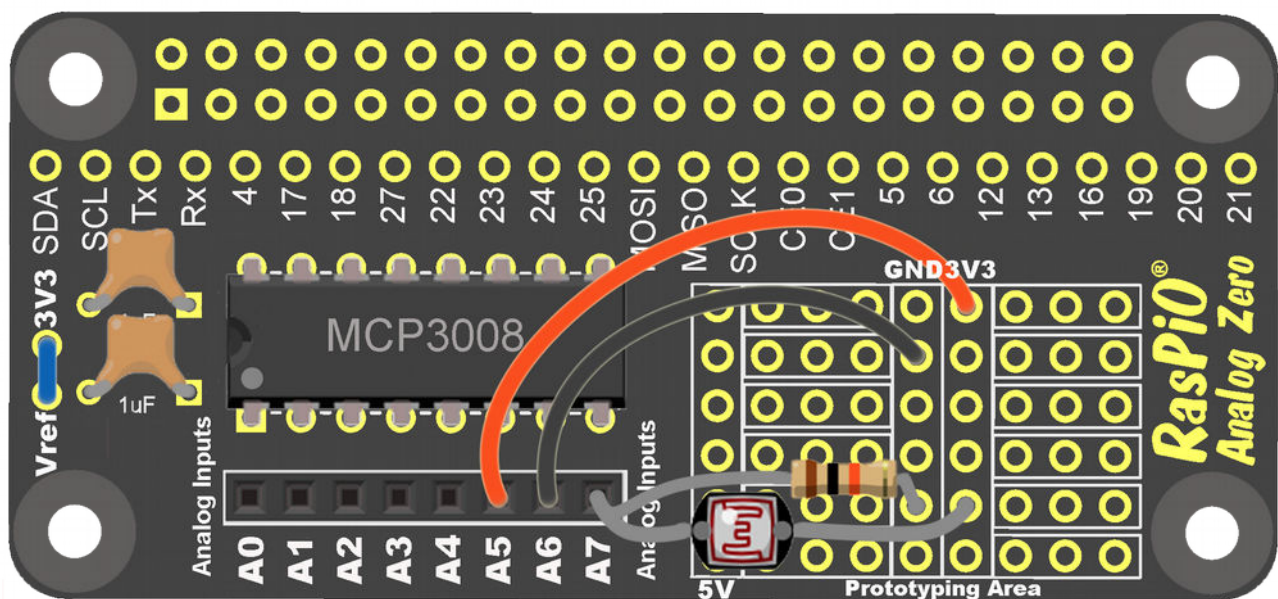
If we tried to read the channels on the MCP3008 now, without connecting anything to them, the reading values would likely be jumping all over the place due to random RF and electrostatic influences.

Let's do something a bit more controlled than that. Let's use the Light Dependent Resistor (LDR) and 10 k Ω resistor from the kit...



10 k Ω resistor (top), LDR (bottom)

The circuit will look like this...



10 k Ω resistor and LDR circuit

In the circuit, the LDR is connected to A7 and 3V3. The 10 k Ω resistor is connected to A7 and GND.

A wire connects A6 to GND and another wire connects A5 to 3V3.

So the values of A5 and A6 should stay constant and the value of A7 will vary with light level. Let's write a script to read the values and show them on the screen...

LDR Code

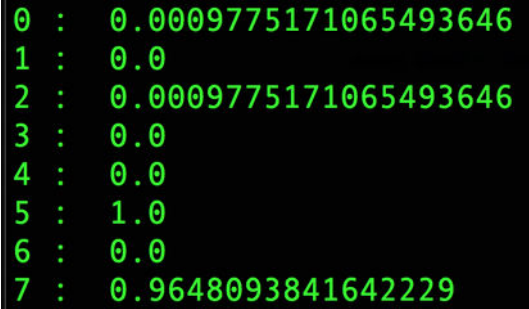
The following code [ldr.py](#) reads all eight channels (0 to 7) of the MCP3008 and displays their value on the screen, repeatedly, forever...

```
#!/usr/bin/python3
from gpiozero import MCP3008
from time import sleep

while True:
    for x in range(0, 8):
        with MCP3008(channel=x) as reading:
            print(x, ": ", reading.value)
        sleep(0.1)
```

GPIO Zero outputs an MCP3008 reading as a float variable between 0.0 and 1.0, where 0.0 is 0V and 1.0 = 3.3V (unless you change V_{ref} - it's 3.3V by default).

Your output should look something like this...



```
0 : 0.0009775171065493646
1 : 0.0
2 : 0.0009775171065493646
3 : 0.0
4 : 0.0
5 : 1.0
6 : 0.0
7 : 0.9648093841642229
```

Simple LDR script output

This script will keep reading the MCP3008 until you hit <CTRL> + C

The output from channels 0-4 is a bit random. It's floating about.

5 should stay fixed firmly at 1.0 as it's connected to 3V3 (3.3V).

6 should stay fixed firmly at 0.0 as it's connected to GND (0V).

7 should vary with changing light conditions. You can verify this with a torch to increase light and a pen lid to decrease it.

In total darkness, channel 7 should read very close to 0. In bright light it will be very close to 1.0.

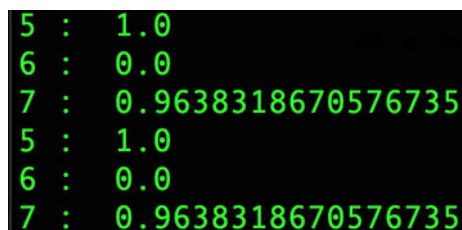
If you change the range, you can ignore the channels we're not interested in...

```
for x in range(0, 8):
```

...becomes...

```
for x in range(5, 8):
```

Then our program only reads and displays 5, 6 and 7 - the channels we're interested in...

A terminal window with a black background and green text. It shows the output of a script that reads channels 5, 6, and 7 of the MCP3008. The output is repeated twice. Channel 5 is always 1.0, channel 6 is always 0.0, and channel 7 is approximately 0.9638318670576735.

```
5 : 1.0
6 : 0.0
7 : 0.9638318670576735
5 : 1.0
6 : 0.0
7 : 0.9638318670576735
```

Modified LDR script output

Measuring Actual Voltages?

We can tweak our script by adding three lines and changing one to show the measured voltage instead of a float between 0.0 and 1.0.

Our code (with changes highlighted) becomes...

```
#!/usr/bin/python3
```

```

from gpiozero import MCP3008
from time import sleep
voltage = [0,0,0,0,0,0,0,0]
vref = 3.3

while True:
    for x in range(5, 8):
        with MCP3008(channel=x) as reading:
            voltage[x] = reading.value * vref
            print(x,": ", voltage[x], "V")
        sleep(0.1)

```

And the output now looks like this...

```

5 : 3.3 V
6 : 0.0 V
7 : 3.170967741935484 V

```

LDR tweak 2 output

But that long value on channel 7 looks a bit silly. There's no way we can claim an accuracy or resolution of that many decimal places. With a 10-bit (1023 steps) ADC we have 0.003V of resolution (at $V_{\text{ref}} = 3.3\text{V}$), so we should only quote an absolute maximum of 3 decimal places, but 2 will be less “flickery” and more “trustworthy”.

Format Output to 2 Decimal Places

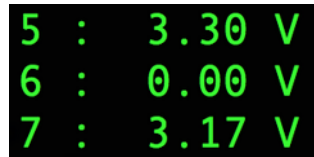
To format the output we can use string formatting...

```
print(x,": ", voltage[x], "V")
```

...becomes...

```
print(x,": ", '{:.2f}'.format(voltage[x]), "V")
```

The `.2f` makes the output two decimal places. Your output should now look nice and neat, like this...



```
5 : 3.30 V
6 : 0.00 V
7 : 3.17 V
```

LDR tweak 3 output

And this is what the final script should now look like...

```
#!/usr/bin/python3
from gpiozero import MCP3008
from time import sleep
voltage = [0,0,0,0,0,0,0,0]
vref = 3.3

while True:
    for x in range(5, 8):
        with MCP3008(channel=x) as reading:
            voltage[x] = reading.value * vref
            print(x,": ", '{:.2f}'.format(voltage[x]), "V")
        sleep(0.1)
```

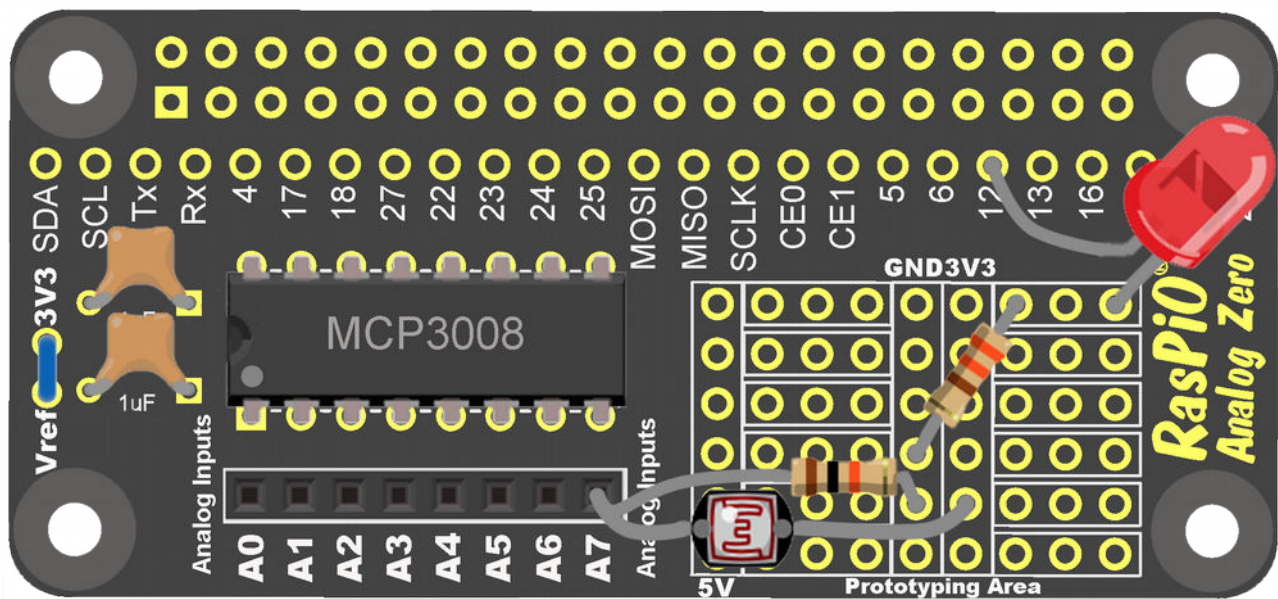
Using ADC Output to Make a Decision

So we've learned how to display ADC output on the screen and how to convert the reading into a voltage and format it. Now let's use the measured LDR value to make a decision. We'll add an LED and 330 Ω resistor to our circuit and switch the LED on when the LDR shows less than half brightness.

Connect LED positive end (long leg) to GPIO12.

LED negative end (flat side) to a 330 Ω resistor.

Connect the other end of the 330 Ω resistor to GND.



Decision-making circuit

And now we just write a few simple lines of code [ldr-led.py](#). The yellow highlights show the parts we use to control the LED.

```
#!/usr/bin/python3
from gpiozero import MCP3008, LED
from time import sleep
red = LED(12)
ldr = MCP3008(channel=7)
while True:
    print("LDR: ", ldr.value)
    if ldr.value < 0.5:
        red.on()
        print ("LED on")
    else:
        red.off()
        print ("LED off")
    sleep(0.1)
```

We only read one channel (7) this time, so the code is a bit simpler. We create `ldr = MCP3008(channel=7)` on channel 7 and read it with `ldr.value`. Then, if `ldr.value` is lower than 0.5, we switch on the LED, otherwise we switch it off.

The on-screen output looks something like this. I caught the moment of covering the LDR, and when the LED switched on...

```
LED off
LDR:  0.512218963831867
LED off
LDR:  0.2961876832844575
LED on
LDR:  0.2727272727272727
LED on
LDR:  0.2570869990224829
```

Switching an LED using LDR reading

You now know how to read one or more analog channels of the MCP3008 and manipulate, display and format the results or use them to make a decision.

Next we're going to look at a couple of RasPiO Analog Zero projects involving i²c character LCDs...

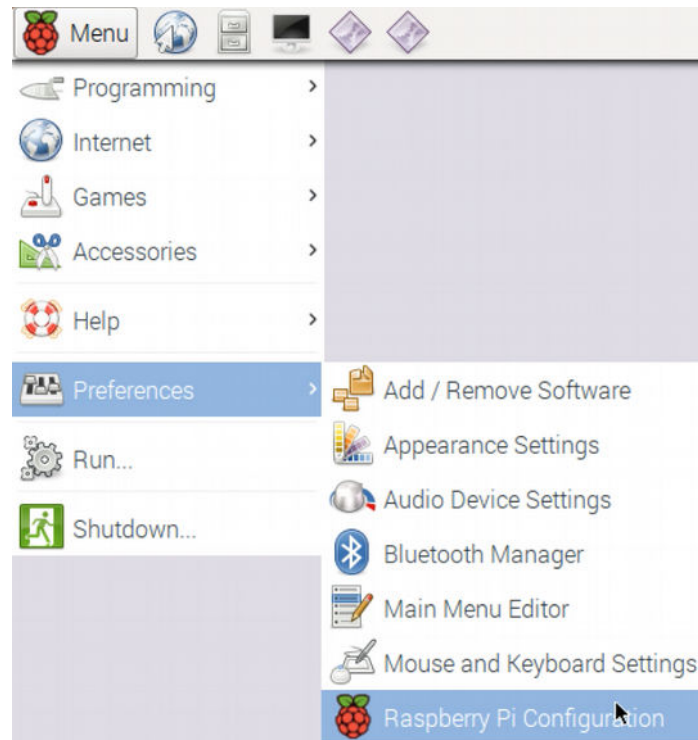
- Weather Station/Digital Thermometer
- Voltmeter

But before we can do that properly, we'll need to install some simple scripts and make sure our Pi is set up to use an i²c character LCD.

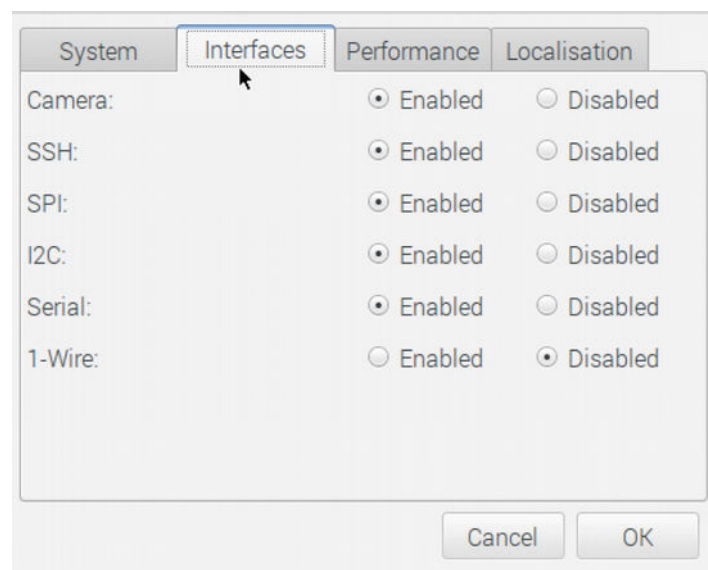
Setting Up for Weather Station & Voltmeter

Ensure i²c is Enabled

Menu > Preferences > Raspberry Pi Configuration



Click the *Interfaces* tab and ensure i2c is enabled



Then click OK. If it asks you to reboot, do that now.

Install Python i2c Drivers

These projects use i²c LCDs so we need to be able to run these from Python.

Open a terminal window...



and type...

```
sudo apt-get update
```

then

```
sudo apt-get install -y python-smbus i2c-tools python3-smbus
```

```
pi@raspberrypi:~ $ sudo apt-get install -y python-smbus i2c-tools python3-smbus
```

Once this has installed, type...

```
i2cdetect -y 1
```

(for original rev 1 model B Pi, change 1 to 0)

```
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:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
40:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
50:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
60:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
70:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
```

No i2c devices attached

This tool allows us to check the i²c address of any i²c devices we have connected to the Pi. For example, our character LCDs are usually on 0x27 (or 0x3f). So, if connected, the output would look like this...

```

pi@raspberrypi:~ $ i2cdetect -y 1
    0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
10:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
20:  --  --  --  --  --  --  --  27  --  --  --  --  --  --  --
30:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
40:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
50:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
60:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
70:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --

```

i2c device at 0x27

If `i2cdetect -y 1` shows a number other than 27, we'll need to edit one of our [LCD driver files \(lcd_driver.py\)](#) to change the address. But we haven't installed them yet. It's only a couple of files. We'll get them from github with the following command...

```

cd
git clone https://github.com/raspitv/analogzero.git

```

This will create a directory called `analogzero` and place all the required files in it. To go there...

```

cd analogzero
ls

```

The python drivers for the i²c LCDs are...

```

i2c_lib.py
lcd_driver.py

```

```

pi@raspberrypi:~/analogzero $ ls
adc_voltmeter_16x2.py
adc_voltmeter_20x4.py
i2c_lib.py
lcd_demo16x2.py
lcd_demo20x4.py
lcd_driver.py
ldr-led.py
ldr.py
README.md
weather_16x2.py
weather_20x4.py

```

These files must be in the same directory as the scripts which use them. **If your LCD's i²c address is not 27, you'll need to edit `lcd_driver.py` to change line 7 `ADDRESS = 0x27` to show the correct i²c address (e.g. `0x3f`).** If your LCD is `0x3f`, it should be labelled.

Now let's have a play with the LCD and make it display something...

In the `analogzero` directory there are two LCD demo scripts (one for each screen size)...

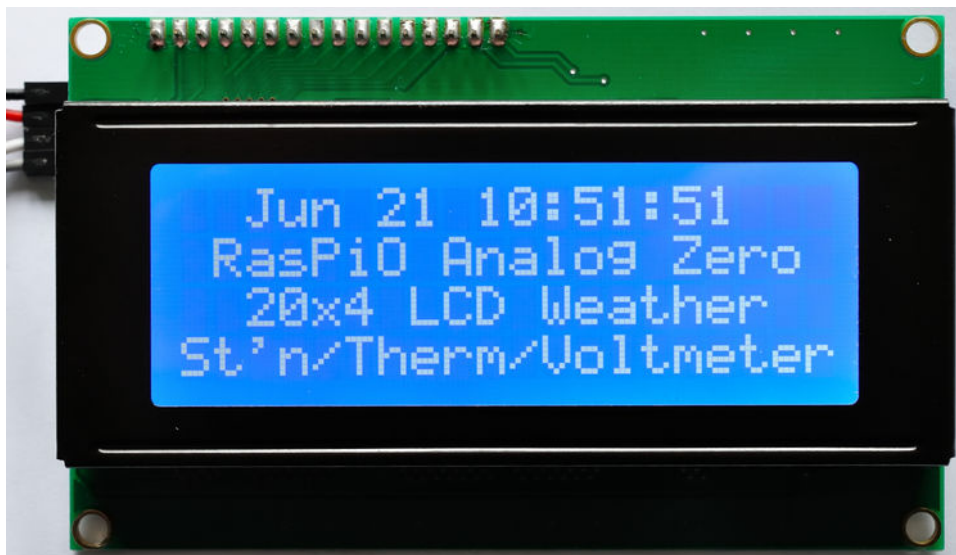
[`lcd\_demo16x2.py`](#)

[`lcd\_demo20x4.py`](#)

Choose the right one for your LCD, and run it with...

```
python3 lcd_demo16x2.py
```

This should demonstrate the capabilities of the display and give you a nice clock (only accurate if connected to internet).



Demo script showing 20x4 LCD clock

Let's go through some of the key parts of the code to highlight the LCD controls...

```
import lcddriver
```

imports the driver files.

```
# LCD custom character variables
degree = chr(0)
squared = chr(1)
cust_chars = [[0x1c,0x14,0x1c,0x0,0x0,0x0,0x0,0x0],      # degree
               [0x8,0x14,0x8,0x10,0x1c,0x0,0x0,0x0]]      # squared
```

Allows us to create and use our own custom characters. In this case I've made characters for ° and ². If you want to create your own custom characters (you can have up to 8), there is a [useful web page here](#) to help generate the hex codes.

```
lcd = lcddriver.lcd()
lcd.lcd_load_custom_chars(cust_chars)
lcd.lcd_clear()
```

...creates and object lcd to control the LCD with. Then we load the custom characters and clear the LCD.

```
# display an intro message
lcd.lcd_display_string('{:^16}'.format("RasPi0 Analog Zero"), 1)
lcd.lcd_display_string('{:^16}'.format("16x2 Weather Kit"), 2)
```

...displays 2 lines of text center justified '{:^16}'.format() to 16 character length.

```
def update():
    lcd.lcd_display_string('{:^16}'.format(row_one), 1)
    lcd.lcd_display_string('{:^16}'.format(row_two), 2)
```

...defines a function to write the value of row_one and row_two to their respective rows. So all we have to do to write to the LCD is change the value of these variables and call update()

```
# activate every single pixel to test the display
pixel_test = chr(255) * 16
for x in range(1,3):
    lcd.lcd_display_string(pixel_test, x)
    sleep(1)
```

Character 255 is a solid bar with every pixel illuminated. You can use the above technique to display any of the built-in characters, but you can also use normal Latin-based keyboard characters (ASCII) as well.

You can find the [character table for the LCDs here](#). (ROM code A00)

HD44780U

Table 4 Correspondence between Character Codes and Character Patterns (ROM Code: A00)

Lower 4 Bits	Upper 4 Bits	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111	
xxxx0000	CG RAM (1)			0	@	P	`	P					-	9	≡	α	p	
xxxx0001	(2)			!	1	A	Q	a	q				。	ア	チ	△	ä	q
xxxx0010	(3)			"	2	B	R	b	r				「	イ	ツ	×	β	θ
xxxx0011	(4)			#	3	C	S	c	s				」	ウ	フ	ε	ε	ω
xxxx0100	(5)			\$	4	D	T	d	t				、	エ	ト	ト	μ	Ω
xxxx0101	(6)			%	5	E	U	e	u				・	オ	ナ	1	σ	Ü
xxxx0110	(7)			&	6	F	V	f	v				ヲ	カ	ニ	ヨ	ρ	Σ
xxxx0111	(8)			'	7	G	W	g	w				ア	キ	ヌ	ラ	g	π
xxxx1000	(1)			(	8	H	X	h	x				イ	ク	ネ	リ	フ	×
xxxx1001	(2)			)	9	I	Y	i	y				う	ケ	ル	ル	'	y
xxxx1010	(3)			*	:	J	Z	j	z				エ	コ	ン	レ	j	≠
xxxx1011	(4)			+	;	K	L	k	l				オ	サ	ヒ	ロ	*	π
xxxx1100	(5)			,	<	L	¥	1	l				カ	シ	フ	ワ	φ	π
xxxx1101	(6)			-	=	M	J	m	}				ユ	ズ	ン	ン	÷	
xxxx1110	(7)			.	>	N	^	n	÷				ヨ	セ	ホ	°	ñ	
xxxx1111	(8)			/	?	O	_	o	+				ッ	ッ	マ	°	ö	■

Note: The user can specify any pattern for character-generator RAM.

Note: The user can specify any pattern for character-generator RAM.

```
lcd lcd_display_string('{:<16}'.format("align left"), 1)
lcd lcd_display_string('{:>16}'.format("align right"), 2)
lcd lcd_display_string('{:^16}'.format("center"), x)
```

These lines show you how to align text using < left, > right, ^ center

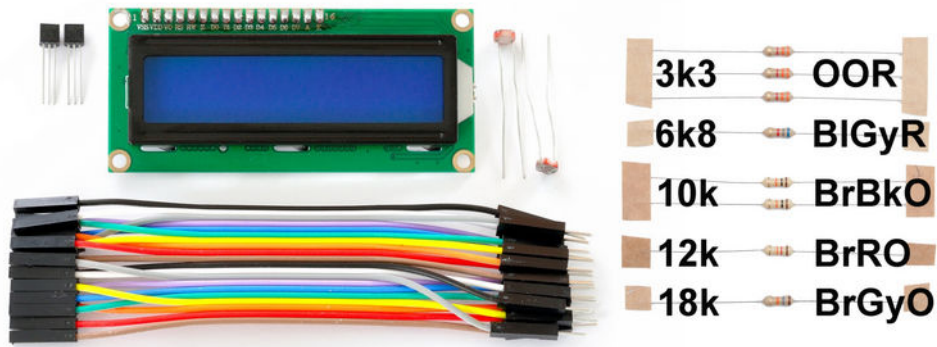
```
lcd.backlight(0) # swap 0 for 1 turns backlight on
```

Switches off the lcd backlight.

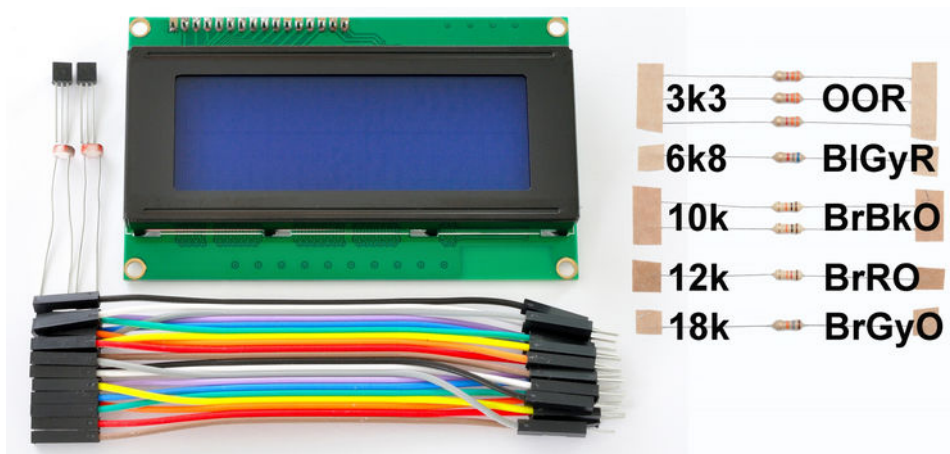
Now we've learnt a bit about the LCD, let's see how to build a weather station/thermometer.

Weather Station / Thermometer Project

This project is designed around the optional RasPiO 16x2 and 20x4 weather station kits.



RasPiO 16x2 weather/thermometer/voltmeter kit



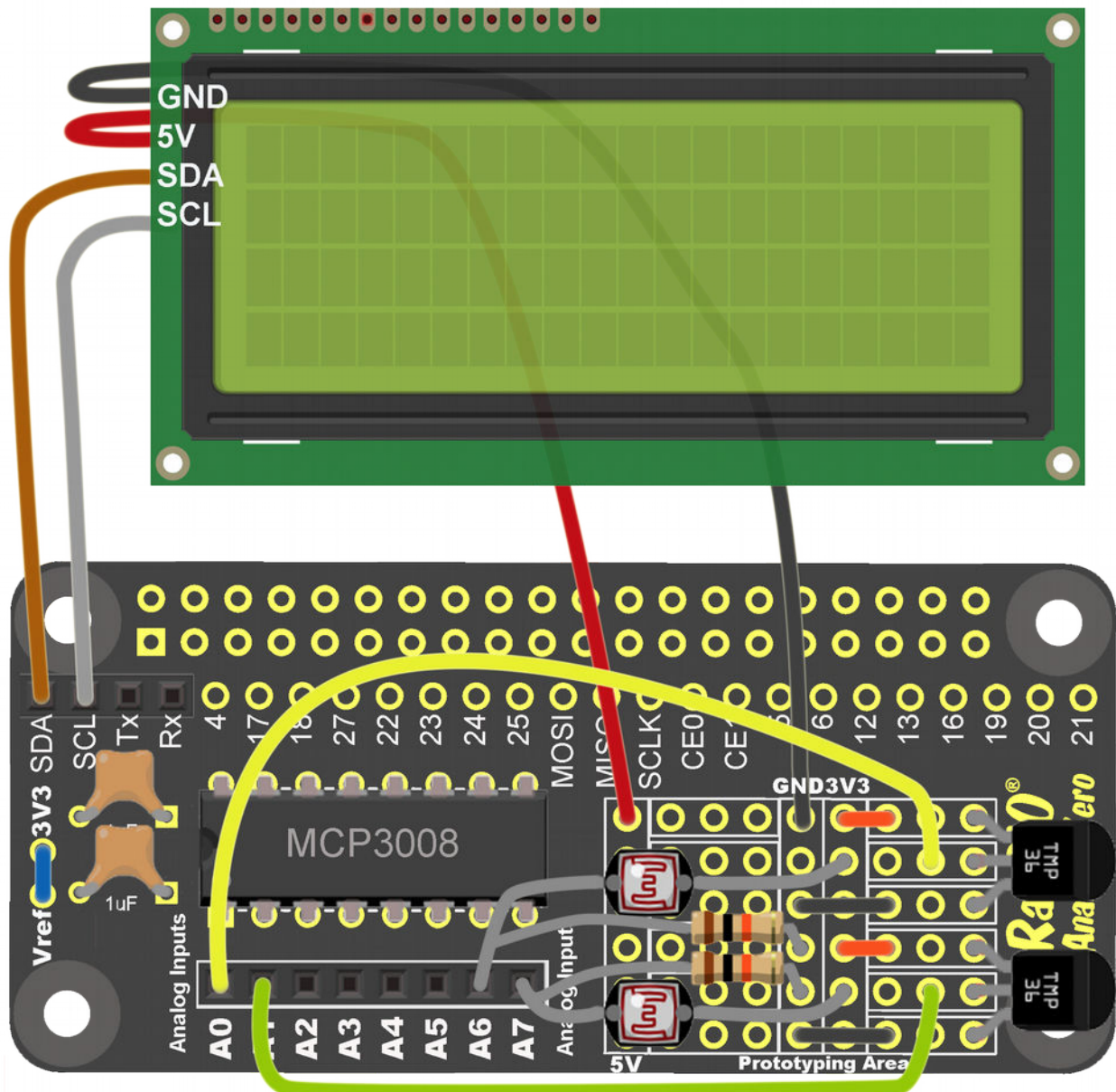
RasPiO 20x4 weather/thermometer/voltmeter kit

Each kit contains...

- an i²c LCD
- two TMP36 temperature sensors
- two LDRs
- 20 jumper wires
- some resistors

The weather station and thermometer circuits and code are essentially

the same, so we'll treat those together. Wire up the following circuit...



Weather Station/Thermometer circuit

Each LDR has one end in 3V3 and one end in A6 or A7.

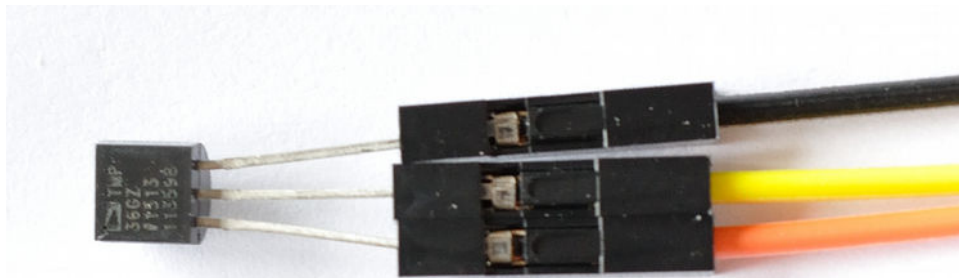
A 10k Ω resistor connects A6 and A7 to GND. Be careful not to short 3V3 to GND with the LDR and resistor wires.

The TMP36 temperature sensors have their middle pins connected to A0 and A1. Their other pins are connected to 3V3 and GND.

The i²c LCDs (same for both) connect to SDA, SCL, 5V and GND.

So we have temperature sensors on channels 0 and 1 and light sensors on channels 6 and 7.

You can use jumper wires for the temperature sensors. They don't have to be mounted on the PCB proto area. I've just shown it that way for clarity. You can even attach the temperature sensor to really long wires and put them in a remote location if you want to.



TMP36 sensor connected to jumper wires

So now we have a circuit, let's write some code to make it work.

As before, there are two Python 3 scripts – one for each LCD size...

[weather_16x2.py](#)

[weather_20x4.py](#)

The 16x2 display has a lot less 'real-estate', so we alternate temperature and light readings to be able to fit it all in clearly.



16x2 Weather Station output

With the 20x4 we don't need to do this, so can display it all at once. You've even got space to display something else as well...



20x4 Weather Station output

The terminal output is the same for both. It just prints the voltage reading for each of the 8 ADC channels on the terminal screen...

```
channel 0: 0.707 Volts
channel 1: 0.713 Volts
channel 2: 0.000 Volts
channel 3: 0.000 Volts
channel 4: 0.000 Volts
channel 5: 0.000 Volts
channel 6: 2.997 Volts
channel 7: 3.111 Volts
```

Terminal output for both

Weather Station / Thermometer Code

Below is a listing of [weather_16x2.py](#)

The script reads each of the analog channels. If everything is wired according to the diagram, the output on the LCDs and terminal should be as described above.

```
#!/usr/bin/python3
from time import sleep
from gpiozero import MCP3008
import lcd_driver
vref = 3.3
channels = [0,0,0,0,0,0,0,0]
temperatures = [0,0]
light_levels = [0,0]
count = 0

lcd = lcd_driver.lcd()    # create object for lcd control
lcd.lcd_clear()          # clear LCD ready for start
```

```

def update():
    lcd lcd_display_string('{:^16}'.format(row_one), 1)
    lcd lcd_display_string('{:^16}'.format(row_two), 2)

# display a centered intro message
row_one = '{:^16}'.format("RasPi0 Analog Zero")
row_two = '{:^16}'.format("16x2 Weather Kit")
update()
sleep(3)

while True:
    for x in range(8):
        adc = MCP3008(channel=x)
        volts = 0.0
        for y in range(20):
            volts = volts + (vref * adc.value)
        volts = volts / 20.0
        if x < 2:
            temperatures[x] = '{:4.1f}'.format((volts - 0.5) * 100)
        if x > 5:
            light_levels[x-6] = '{:4.1f}'.format(volts / vref * 100)
        volts = '{:.3f}'.format(volts)
        channels[x] = volts

# on-screen output useful for debug when tweaking
# shows the actual voltage at each analog input
print("channel " + str(x) + ":", volts, "Volts")

# update the character LCD once every cycle then a short delay
# because we have limited characters, we alternate display
if x == 7:
    if count % 2 == 0:
        row_one = "Temp 0: " + temperatures[0] + "C"
        row_two = "Temp 1: " + temperatures[1] + "C"
    else:
        row_one = "Light 0: " + light_levels[0] + "%"
        row_two = "Light 1: " + light_levels[1] + "%"
    update()
    count += 1
    sleep(0.2) # you can adjust refresh rate here

```

This script uses techniques we've already covered, such as...

- **reading the analog channels**
- **converting the reading to a voltage or percentage**
- **displaying results in the terminal**
- **formatting the output to the required number of characters**
- **displaying output on an i²c character LCD**

But we also use a couple of new techniques...

- converting a voltage reading to a temperature
- selectively treating different channels differently

```
if x < 2:
    temperatures[x] = '{:4.1f}'.format((volts - 0.5) * 100)
if x > 5:
    light_levels[x-6] = '{:4.1f}'.format(volts / vref * 100)
```

We've connected our TMP36 temperature sensors to channels 0 and 1, so we only want to convert the voltages from those two channels into temperatures. We use `if x < 2:` to select these two only. Then

```
(volts - 0.5) * 100
```

...gives us the temperature measured by the TMP36 in °C. So a voltage of 0.712V would give 21.2 °C.

In the same way, we've used channels 6 and 7 for the LDRs so we use `if x > 5:` to select just those channels for our light level percentage calculations. LDRs are not linear, so they only give an approximate indication of light level. Hence expressing the output as a percentage, rather than trying to pretend it's measuring something really accurately.

Suggested Ideas to Extend Your Weather Station

This is just a simple introduction to the weather station concept. There is a lot more you could do with it to make it your own...

- add custom characters for ° ([see demo script](#))
- change the LCD output to suit your own needs
- add internet weather feed information
- indoor/outdoor temperatures
- add more sensors to use the 4 remaining channels (could place them further away on longer wires)
- monitor your refrigerator or freezer temperature
- add a news or stock ticker or twitter feed
- sms, tweet, or push notifications alerts to your phone

You could end up making a sophisticated device just by adding more and more to it.

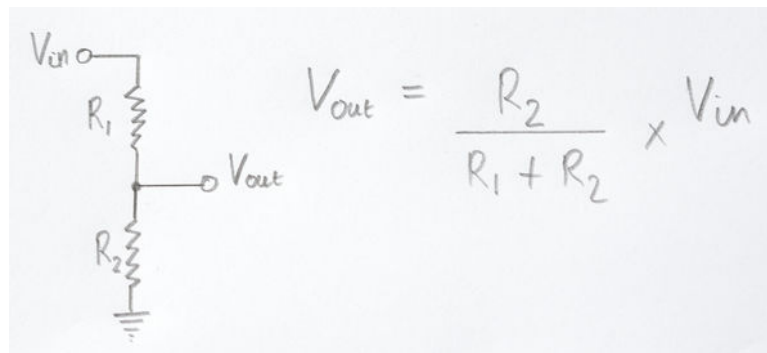
But what about the Voltmeter Project? That's up next...

Voltmeter Project

We've already seen that the MCP3008 can measure up to 3.3V when connected to the Pi. But what if we wanted to measure a higher voltage? Is there a way we could achieve that?

Voltage Dividers

Yes there is. We can use a voltage divider (also known as a resistor divider) to reduce the input voltage below 3V3 so we can measure it...



Voltage divider circuit and formula

If you choose your values for R_1 and R_2 appropriately, you can divide your input voltage (V_{in}) by any number you choose. This is ideal for us. In this case R_2 is fixed at 3.3k Ω and we've chosen three different R_1 values for to make three voltage dividers. Each one gives us a different range. This is exactly why you have three 3k Ω , one 6.8k Ω , one 12k Ω and one 18k Ω resistors in your kit.

R2 /				
R1	R2	(R1+R2)	Vin	Vout
6800	3300	0.3267	10	3.2673
12000	3300	0.2157	15	3.2353
18000	3300	0.1549	20	3.0986

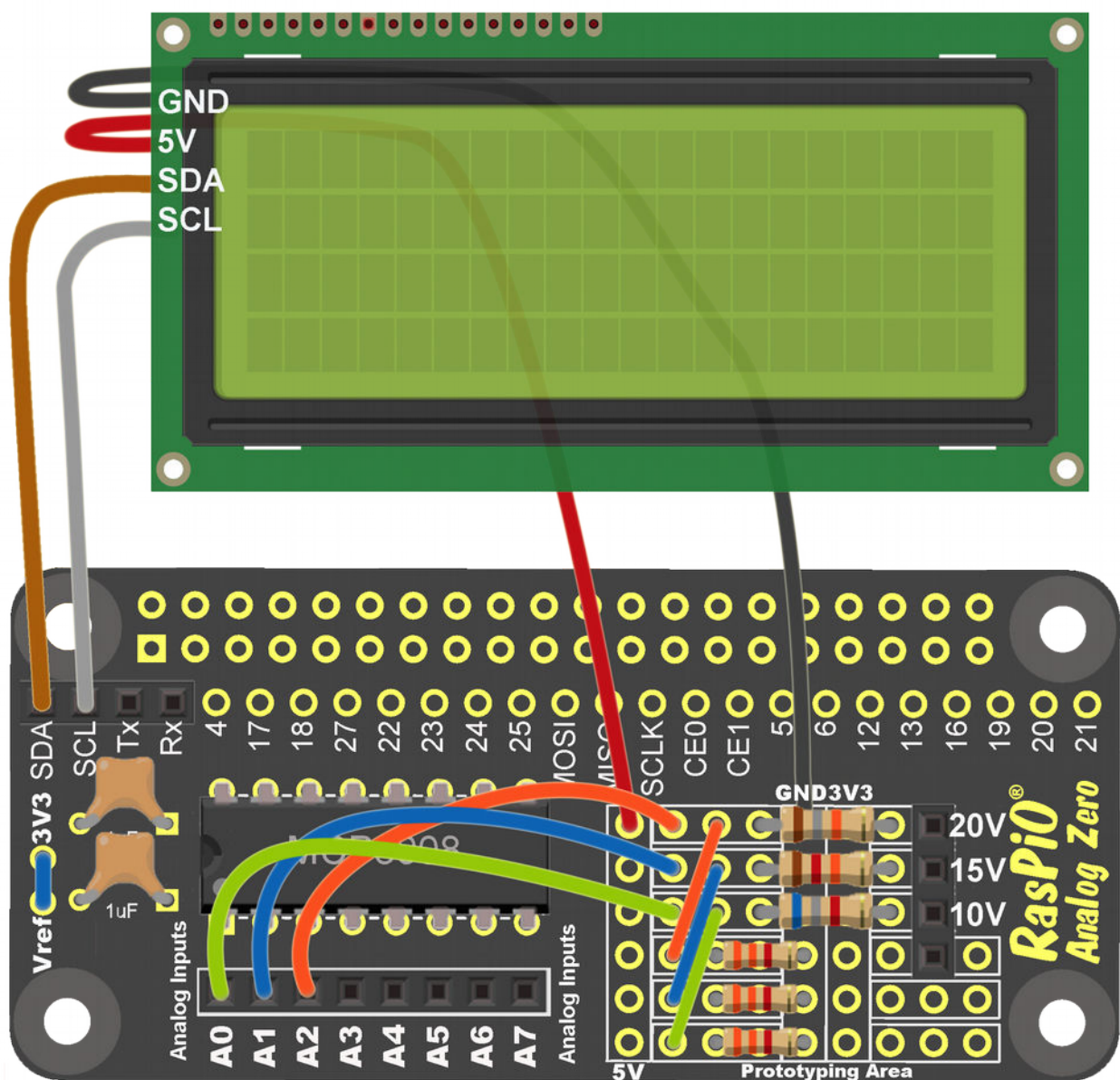
Using voltage dividers allows us to make inputs which can measure up

to 10V, 15V and 20V respectively by reducing the voltage that the ADC 'sees' (V_{out}) to below 3.3V so it can be measured.

If that makes your head hurt, don't worry. Just take care to make sure the resistors are in the correct places.

18k Ω	Brown, Grey, Orange	20V _{in}
12k Ω	Brown, Red, Orange	15V _{in}
6.8k Ω	Blue, Grey, Red	10V _{in}
3.3k Ω	Orange, Orange, Red	

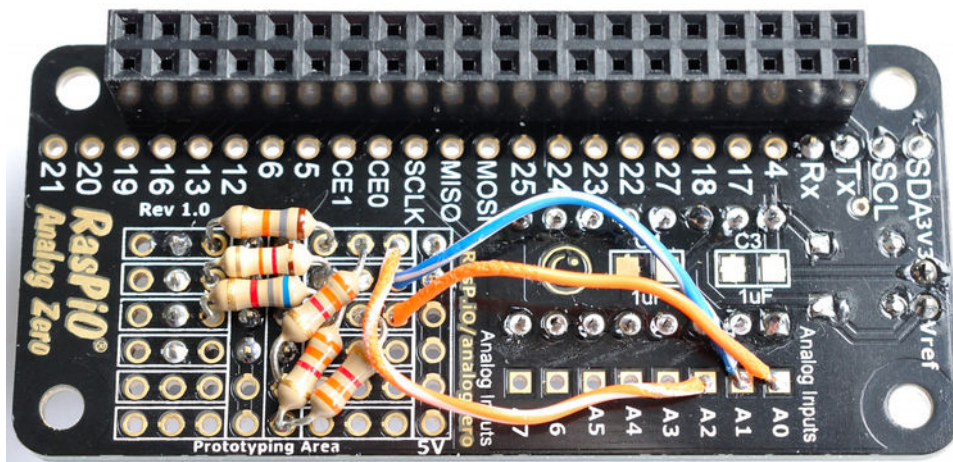
Voltmeter Circuit



The divided 20V input goes to A2.
The divided 15V input goes to A1.
The divided 10V input goes to A0.

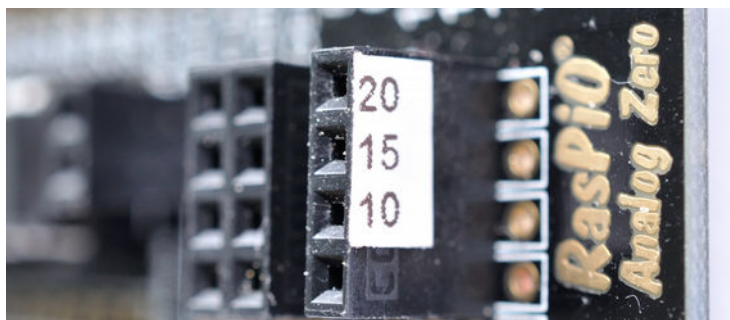
You don't strictly need the short orange, blue and green wires. They are included just for clarity. You can use the resistor leads to make those connections directly if you are careful to avoid shorting against other resistors or connections.

When I made up this circuit, I put the resistors on the underside of the board so they were out of the way.



Voltmeter circuit on underside of board

I also labelled my V_{in} headers 20, 15, 10 so I would not connect too high a voltage to the wrong input.



Labelled headers to avoid confusion

If you connected, say, a 20V voltage to the 10V input, you'd be sending 6.5V into the ADC and would quite possibly damage it. So do be careful and have your wits about you to avoid damage.

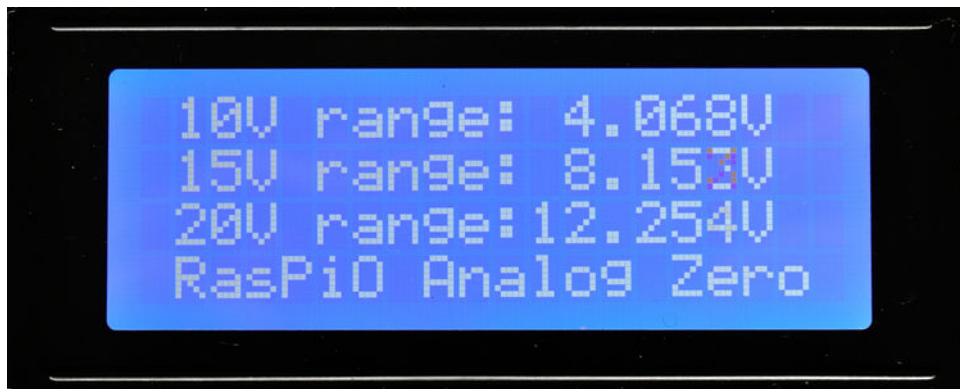
Voltmeter Code

As usual, there are two scripts for the voltmeter, one for each size of LCD...

[adc_voltmeter_16x2.py](#)

[adc_voltmeter_20x4.py](#)

The 20x4 version displays all three channels on the screen at once. You can see in the photo below I just caught it updating the 15V range value.



The 16x2 version alternates the displayed channels every few seconds.



Both versions output the same data to the console...

```
channel 0: 4.068 Volts
channel 1: 8.152 Volts
channel 2: 12.254 Volts
```

A code listing of [adc_voltmeter_20x4.py](#) follows, with comments

```

#!/usr/bin/python3
from time import sleep
from gpiozero import MCP3008
import lcddriver
vref = 3.296
adc_list = [0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0]
conversion_factors = [3.080,4.668,6.43,1,1,1,1,1]
lcd = lcddriver.lcd()      # create object for lcd control
lcd.lcd_clear()           # clear LCD ready for start

def update():
    lcd.lcd_display_string('{:^20}'.format(row_one), 1)
    lcd.lcd_display_string('{:^20}'.format(row_two), 2)
    lcd.lcd_display_string('{:^20}'.format(row_three), 3)
    lcd.lcd_display_string('{:^20}'.format(row_four), 4)

# display an intro message
row_one = "Hi 20x4 RasPi0"
row_two = "Analog Zero"
row_three = "Multi-range"
row_four = "Voltmeter"
update()
sleep(2)

while True:
    for x in range(3):
        adc = MCP3008(channel=x)
        readings = 0.0
        repetitions = 200          # how many times we sample
        for y in range(repetitions):
            readings += adc.value
        average = readings / repetitions
        volts = '{:6.3f}'.format(vref * average *
conversion_factors[x])
        print("channel " + str(x) + ":", volts,"Volts")
        adc_list[x] = volts
        if x == 2:
            row_one = str("10V range:"+adc_list[0])+"V"
            row_two = str("15V range:"+adc_list[1])+"V"
            row_three = str("20V range:"+adc_list[2])+"V"
            row_four = str("RasPi0 Analog Zero")
            update()
        sleep(0.05)

```

Picking out some key parts of the code...

```
vref = 3.296
```

This is an accurate measurement on the Pi's 3V3 rail output.

```
conversion_factors = [3.080,4.668,6.43,1,1,1,1,1]
```

This also relates to calibration of our 20V, 15V and 10V input channels.

It's fully explained in [the calibration section](#)

```
repetitions = 200                # how many times we sample
for y in range(repetitions):
    readings += adc.value
average = readings / repetitions
```

This little block of code reads the inputs multiple times and averages them. This helps to give more stable readings. You can change the value of repetitions from 200 to something else. If you change it to 1, you'll see that the display is likely to be more 'jittery'. If you change it to a really high number, the program will slow down.

The rest of the program uses techniques and code we've already covered.

Calibrating the Voltmeter – Why?

- The accuracy of any instrument can only ever be as good as its calibration. The voltmeter scripts contain the calibration factors for my setup. This section will show you how to tweak your setup so that it is calibrated to your own voltmeter or voltage source.
- Remember that the 3V3 regulator on the Pi may not be exactly 3.3000V. Mine, measured with my best multimeter, was 3.296V. That's only 0.13% low *if we believe my multimeter*. But in any case it's close enough for nearly anything you might need.
- Resistor tolerances. We're using 5% resistors. They're usually pretty close to their nominal value, but they are specified to be within 5%. In theory, if your voltage divider was made up of a resistor that was 5% high and another that was 5% low, your division could be “a little off”.

By calibrating our meter we can ensure that it is as accurate as it can be. It's never going to be perfect, but it's surprisingly good.

Step 1: Set your vref in line 5.

Using your best measuring device, measure the voltage between GND and 3V3 of your Pi. The most convenient way to do this is to use the GND and 3V3 outputs on the RasPiO Analog Zero board. Then edit line 5 of the voltmeter script to show your measured value. It should be pretty close to 3.3 Volts. (e.g. 3.296V)

Step 2: Reset conversion factors.

In line 7 of the voltmeter script, change all the `conversion_factors = [3.080,4.668,6.43,1,1,1,1,1]` values to 1, like this...

```
conversion_factors = [1,1,1,1,1,1,1,1]
```

Step 3: Connect Inputs to 3V3. Connect all three inputs (20V, 15V, 10V) to the Pi's 3V3 and run the voltmeter script.

Step 4: Note Values

The script will show some wrong-looking values on the screen and LCD. Don't worry! These will enable us to calculate our conversion factors and calibrate our voltage dividers to our specific setup.

```
channel 0: 1.070 Volts
channel 1: 0.706 Volts
channel 2: 0.512 Volts
```

Step 5: Calculate and amend conversion_factors

Taking the results from above, conversion factors are calculated thus...

$vref / 1.070 = \text{conversion factor}$

Channel 0 (10V input): $3.296 / 1.070 = 3.080$

Channel 1 (15V input): $3.296 / 0.706 = 4.668$

Channel 2 (20V input): $3.296 / 0.512 = 6.438$

Now we change line 7 of the voltmeter script...

```
conversion_factors = [1,1,1,1,1,1,1,1]
```

...to use our new numbers. It should look something like this...

```
conversion_factors = [3.080,4.668,6.438,1,1,1,1,1]
```

And now if you re-run the voltmeter script (with all three inputs connected to 3V3), your values should be very close to your vref value (3.296V in my case).

```
channel 0:  3.295 Volts
channel 1:  3.294 Volts
channel 2:  3.297 Volts
```

Remember the resolution of the ADC is 0.0032V. So we should really treat the 3rd decimal place with suspicion. But also note that all three channels are now reading within 0.0032V of 'the true value'.

So now you've made and calibrated your multi-channel, multi-range Voltmeter! I hope you enjoyed making it. Now enjoy trying it out and measuring some voltages with it. Stay away from the mains and AC though!

Suggested Ideas to Extend the Voltmeter

- modify the code to use more channel(s)
- add more dividers for more ranges e.g. 5V (but be warned – voltages above 20V can bite – best stay in that region and NEVER try to use this on mains or AC)
- add a button to switch on/off the LCD backlight
- add other button/menu features
- add calibrate or auto-calibrate functionality to the hardware/software
- add auto-ranging to the hardware/software

Final Word

You should now have a thorough overview of how to use the RasPiO Analog Zero with GPIO Zero.

I hope you have a lot of fun with it. There is always more to learn and further to go. As time goes by, I hope to add more to this guide to cover more aspects of GPIOZero and more components.

You can check for the latest version at <http://rasp.io/analogzero>

And if you haven't yet got yourself a RasPiO Analog Zero, or need another, you can get that from here as well...

<http://rasp.io/analogzero>

RasPiO Analog Zero

