Instructions:

Project Box 1 for Raspberry Pi

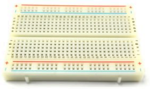
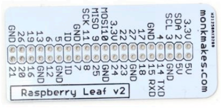
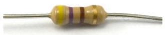
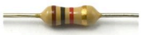
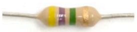
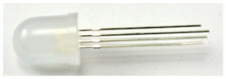
Compatible with Raspberry Pi 2, 3 and 4.

Download the Instructions by scanning the QR code or visit:
monkmakes.com/downloads/pi_box_1a.pdf

V1a-2

KIT CONTENTS

Before you start making your projects, please make sure that you have all the parts listed below:

Solderless breadboard	
Male to male jumper wires - Use to connect things together on the breadboard.	
Female to Male Jumper Wires - Use to connect the Raspberry Pi to the breadboard.	
Raspberry Leaf – to identify pins on the Raspberry Pi	
5 x 470Ω resistor (yellow, purple, brown stripes)	
2 x 1kΩ resistor (brown, black, red)	
4.7MΩ resistor (yellow, purple and green stripes)	
330nF capacitor	
2 x Red LED – the longer lead is the + (positive) lead	
RGB LED – the longest lead is the – (negative) lead	

2 x Tactile push switch	
Phototransistor	
Thermistor	
Buzzer	

You will of course also need a Raspberry Pi with a recent version of Raspberry Pi OS on it. The Raspberry Pi must also have a working internet connection in order to download and install the programs.

Table of Contents

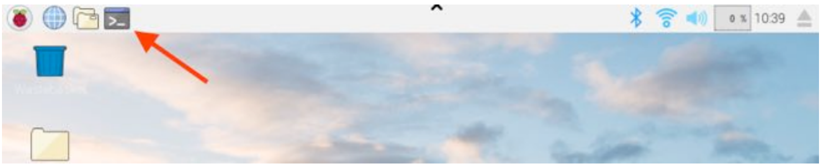
Setting up your Raspberry Pi

Before you can use the kit, you need to configure your Raspberry Pi and install some software. Your Raspberry Pi must be connected to the Internet for this.

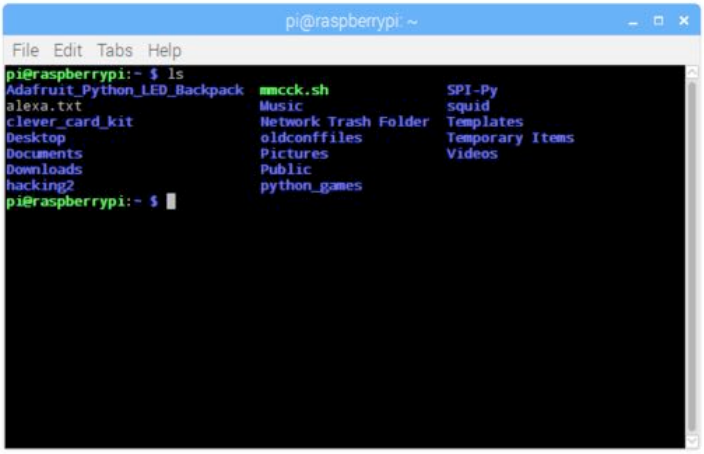
The Terminal

You will need to use the Terminal to type commands to set up your Raspberry Pi.

To start a Terminal window, click on the terminal icon indicated by the arrow below.



This will open a Terminal like the one shown below.



```
pi@raspberrypi:~ $ ls
Adafruit_Python_LED_Backpack  mmcck.sh             SPI-Py
alexa.txt                     Music                squid
clever_card_kit               Network Trash Folder Templates
Desktop                       oldconffiles         Temporary Items
Documents                     Pictures             Videos
Downloads                     Public
hacking2                      python_games
pi@raspberrypi:~ $
```

When the Terminal is ready for you to enter a command, it will prompt you with the $ character. For example, type the command 'ls'. This will show you a list of files in your 'current directory'.

In the instructions in this booklet, you will see the $ symbol with the commands that you are asked to type. This is just a reminder that you are using the Terminal, you do not actually need to type the $ again.

Step 1. Update your Raspberry Pi

Make sure that your Raspberry Pi is up to date by running the following commands in the Terminal:

```
$ sudo apt-get update
$ sudo apt-get upgrade
```

If you get error messages during the upgrade, **restart your Raspberry Pi and run both commands again.**

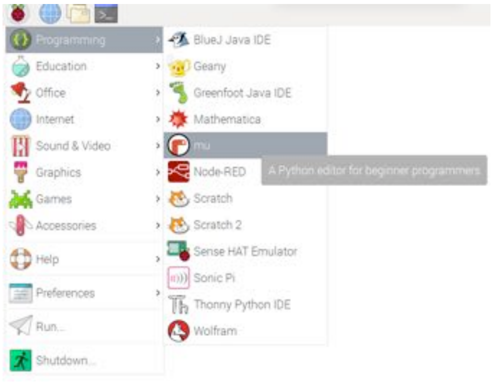
If you see a message in the Terminal that asks "Do you want to continue? [Y/n]" press "y" to continue.

Note that these commands could take a considerable time to complete if your Raspberry Pi has a lot to update.

Step 2. Setup the Mu Editor

This kit assumes that you are using the Mu editor to run the programs for the project and also for you to view the code and modify it if you are feeling adventurous. If you have a recent version of Raspbian, the good news is that Mu is ready-installed. If it is installed you should find it in the Programming section of the Main Menu. If its not there, you can install it by running the following command in the Terminal.

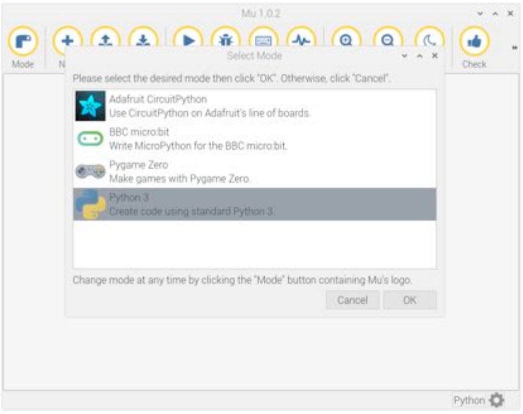
```
$ sudo apt-get install mu-editor
```



Once Mu is installed, find it in the Programs section of the Main menu and run it.

When you run Mu for the first time, it will open a *Set Mode* window that prompts you to select what you are going to be using Mu for. Select the option *Python 3*.

Once you have selected Python 3 the Mu editor will open. You can come back to this when you are ready to use the program for Project 1, but for now, lets install the programs needed for the projects.



Please select the desired mode then click "OK". Otherwise, click "Cancel".

Adafruit CircuitPython
Use CircuitPython on Adafruit's line of boards.

BBC micro:bit
Write MicroPython for the BBC micro:bit.

Pygame Zero
Make games with Pygame Zero.

Python 3
Create code using standard Python 3.

Change mode at any time by clicking the "Mode" button containing Mu's logo.

Step 3. Install the Project Software

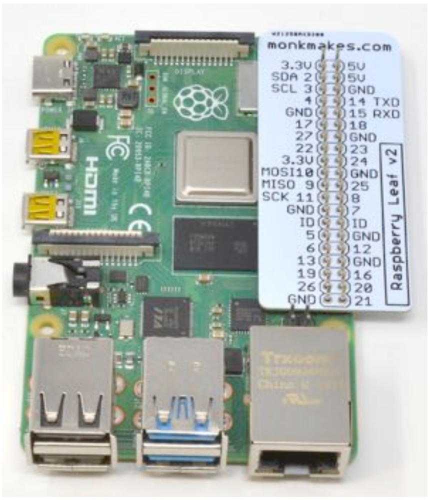
Its important that you carry out step 2 before step 3. Mu needs to have been run once in order to create the directory where the project programs will go.

Start by opening an Terminal on your Raspberry Pi and issue the following commands to fetch an installation script to install all the software that you will need for the projects in this book. Note its pb**1**.sh (the digit 1 not L) at the end of the wget command.

```
$ wget http://monkmakes.com/pb1.sh
$ sh pb1.sh
```

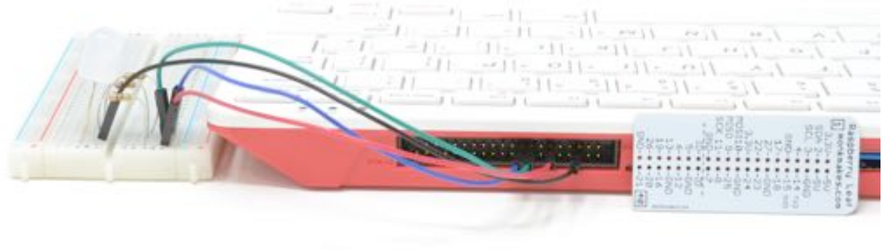
Step 4. Fit the Raspberry Leaf

Because we need to know which pin is which on the Raspberry Pi, fit the supplied Raspberry Leaf onto your Raspberry Pi as shown.



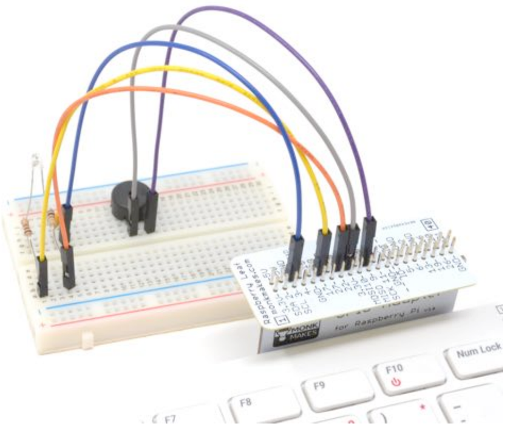
Raspberry Pi 400

This kit is designed for the Raspberry Pi 4 and earlier bare-circuit-board versions of the Raspberry Pi. However, you can still use it with the Raspberry Pi 400, but it will be more difficult to identify the GPIO pins.



Even though you cannot place the Raspberry Leaf over the GPIO pins, you can use it as a reference for when to connect the wires. You just need to count carefully along from one end of the connector to find the right pin.

You can also separately buy a MonkMakes GPIO Adapter for Pi 400 (https://monkmakes.com/pi_400_gpio), that makes it a bit easier to make the connections.



You are now ready to start making Project 1.

Project 1. Make an LED Blink

To make this first project (and all that follow), you are going to wire up the breadboard as shown below, then connect the breadboard to the pins on the Raspberry Pi, and then run the program for this project in Mu.

Step 1. Find the parts you need

For this project, you will need the following items:

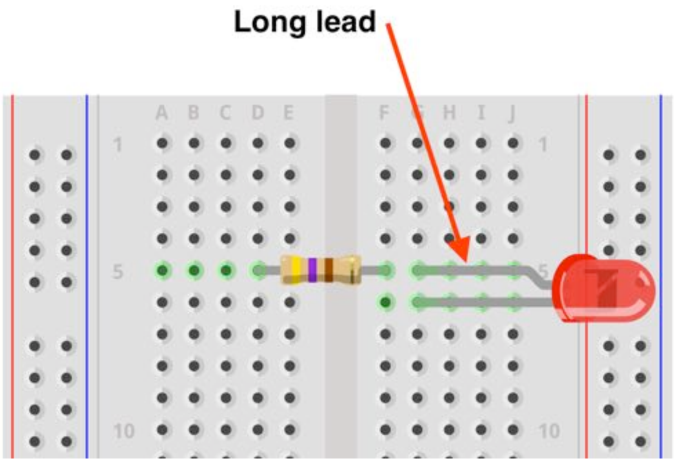
- 470Ω Resistor (yellow, purple and brown stripes)
- Red LED
- Two Female to male jumper wires

Use the table of parts on pages 2 and 3 to identify the parts you need. If you are interested in what the resistor color stripes mean see Appendix C.

Step 2. Wiring up the breadboard.

Lets start by placing the components onto the breadboard. Using the diagram below, push the component legs through the holes in the breadboard at the positions shown.

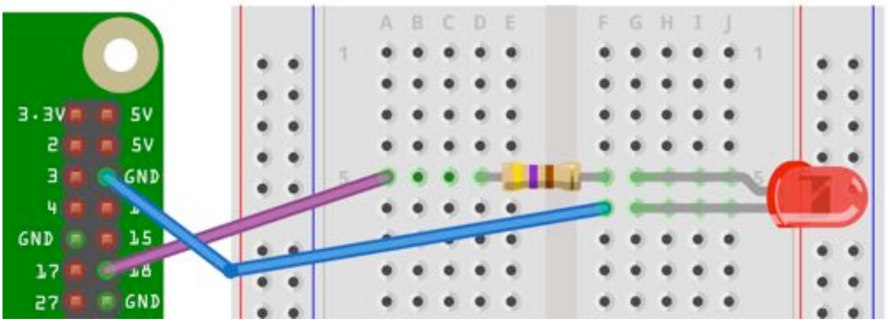
If you haven't come across breadboards in the past, Appendix B provides an explanation of how they work.



Be careful with the orientation of the LED - the long leg is the positive lead (row 5), and the shorter leg is the negative lead (row 6). It does not matter which way around the resistor goes.

Step 3. Connecting the Breadboard to the Raspberry Pi.

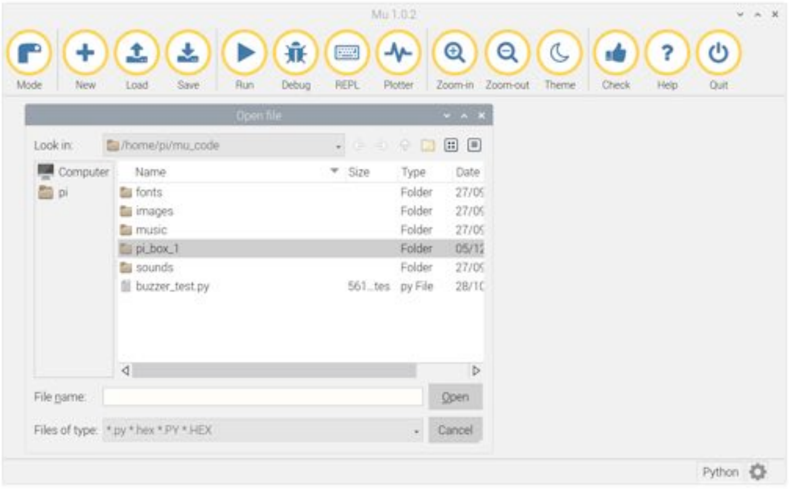
Now that the components are in the right place, take a purple and blue female to male jumper wire and connect the Raspberry Pi to the breadboard as shown below:



It doesn't really matter what color leads you use to make the connections. But when the projects get more complex, sticking to the colors used in the diagrams can make it easier to see what's going on.

Step 4. Running the Program

To run the program, we first need to load it into Mu. So, if Mu is not running, start it now from the Programs section of the Main menu and click on *Load*.



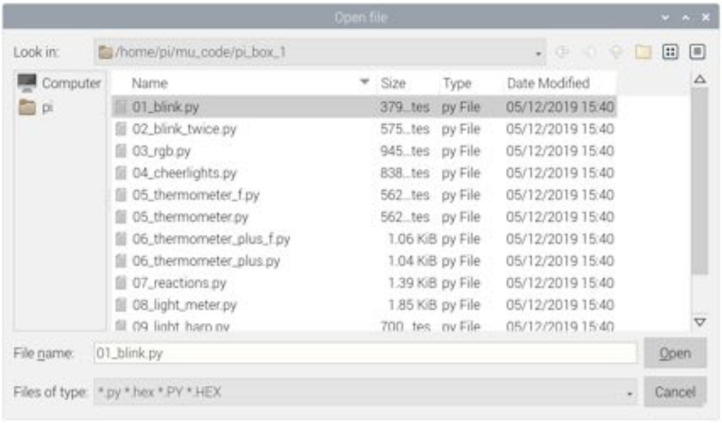
Look in: /home/pi/mu_code

Name	Size	Type	Date
fonts		Folder	27/0
images		Folder	27/0
music		Folder	27/0
pi_box_1		Folder	05/1
sounds		Folder	27/0
buzzer_test.py	561 ..tes	py File	28/1

File name:

Files of type: *.py *.hex *.PY *.HEX

All the project programs are held in the directory *pb1*. So double click on this to see the list of programs:

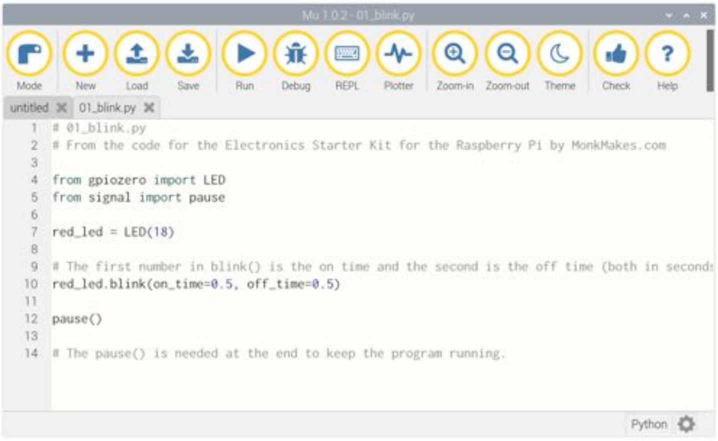


Name	Size	Type	Date Modified
01_blink.py	379...tes	py File	05/12/2019 15:40
02_blink_twice.py	575...tes	py File	05/12/2019 15:40
03_rgb.py	945...tes	py File	05/12/2019 15:40
04_cheerlights.py	838...tes	py File	05/12/2019 15:40
05_thermometer_f.py	562...tes	py File	05/12/2019 15:40
05_thermometer.py	562...tes	py File	05/12/2019 15:40
06_thermometer_plus_f.py	1.06 KiB	py File	05/12/2019 15:40
06_thermometer_plus.py	1.04 KiB	py File	05/12/2019 15:40
07_reactions.py	1.39 KiB	py File	05/12/2019 15:40
08_light_meter.py	1.85 KiB	py File	05/12/2019 15:40
09_light_harp.py	700...tes	py File	05/12/2019 15:40

File name: 01_blink.py

Files of type: *.py *.hex *.PY *.HEX

Select the first program (01_blink.py) and click *Open*.



```
# 01_blink.py
# From the code for the Electronics Starter Kit for the Raspberry Pi by MonkMakes.com

from gpiozero import LED
from signal import pause

red_led = LED(18)

# The first number in blink() is the on time and the second is the off time (both in second
red_led.blink(on_time=0.5, off_time=0.5)

pause()

# The pause() is needed at the end to keep the program running.
```

To start the program, click on the Run button. If everything is set up correctly, the LED should be blinking on and off.

When you want to exit the program, click *Stop*.

If the LED doesn't blink on and off, check that its the right way around and that everything is wired up as shown in the diagram.

The Code for Project 1

If you want to understand how the code works, take a look at the code in Mu.

Any lines beginning in # are called *comment* lines. These are not actually program code, but rather explain something that's going on in the code.

The *import* commands specify the library files that the code uses. In this case:

- The *gpiozero* library lets us turn the LED on and off
- From the signal library we just need the one function (*pause*) that is used right at the end of the program.

When setting up the LED by using *LED(18)*, the number 18 refers to the pin that the LED is connected to.

The *pause()* command at the end of the program is needed so that the program will continue running (and continue blinking) after the *blink* command is used. If the *pause* is removed, the LED will only blink once and then the program will end.

All the programs in this kit make use of *libraries* of Python code that are not actually part of the Python language, but rather provide useful toobags of Python code that you can make use of - to read sensors, make LEDs flash etc. Appendices D, E and F explain more about the libraries

Things to Try

To make the LED blink faster, try changing the values of *on_time* and *off_time* to 0.1.

Project 2. Make 2 LEDs Blink

Don't dismantle Project 1 just yet, for Project 2, you are going to add another LED and make the two LEDs blink alternately.

BEFORE you start changing what's on the breadboard, disconnect the breadboard from your Raspberry Pi. An accidental short-circuit could damage or break your Raspberry Pi.

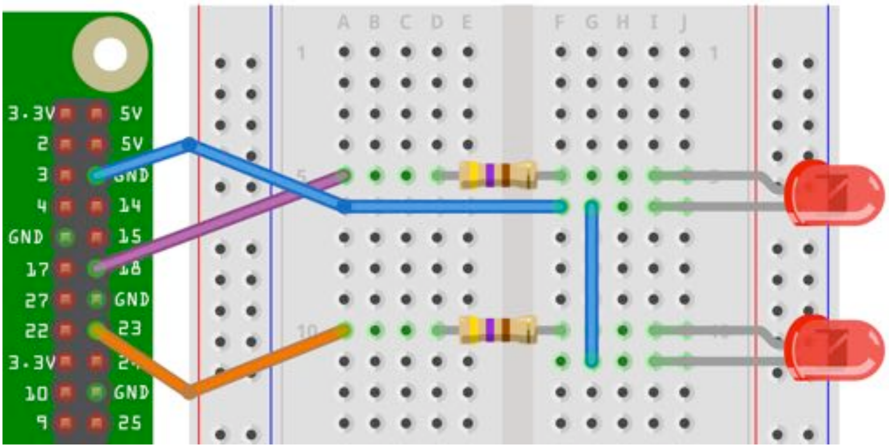
Step 1. Find the parts you need

For this project, you will need the following items:

- Two 470Ω Resistors
- Two Red LEDs
- Three Female to Male jumper wires
- Male to Male jumper wire

Step 2. Build and Connect the breadboard.

Wire up the breadboard and connect it to your Raspberry Pi using the diagram below as a guide.



Notice how the blue male to male jumper wire is used to connect the two negative connections of the LEDs.

Step 3. Running the Program

Load and run the program called 02_blink_twice.py in Mu. The LEDs should now blink in turn.

The Code for Project 2

The code is very similar to the 01_blink.py example, except for a small delay before starting to make the second LED blink.

```
from gpiozero import LED
import time
from signal import pause

red_led1 = LED(18)
red_led2 = LED(23)

red_led1.blink(0.5, 0.5)
time.sleep(0.5)
red_led2.blink(0.5, 0.5)

pause()
```

The *blink* function requires two arguments; the first represents the time the LED stays on and the second represents the time the LED stays off, both in seconds. In this example, the first LED will start blinking every half a second and then the second LED will join in after another 0.5 second delay so that the LED's will blink one after the other.

Project 3. RGB Color Display

This project will demonstrate how to control an RGB LED with your Raspberry Pi.

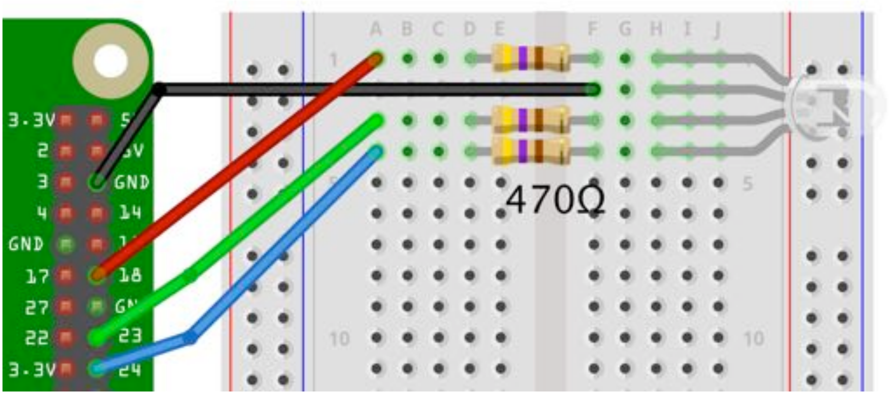
Step 1. Find the parts you need

For this project, you will need the following items:

- Three 470Ω Resistors
- An RGB LED
- Four Female to Male jumper wires

Step 2. Build and Connect the breadboard.

Wire up the breadboard and connect it to your Raspberry Pi using the diagram below as a guide.

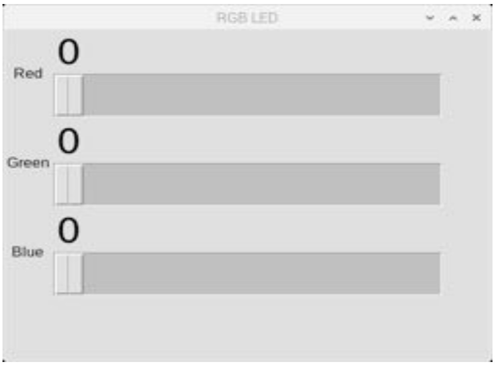


All three of the resistors used in this project are the same value of 470Ω. You should find that one of the RGB LED's leads is longer than the rest. Place the RGB LED such that the long lead is in row 2 of the breadboard and connected to the GND wire from the Pi. Make sure that the legs of the resistors are not touching each other.

Step 3. Running the Program

Load and run the program called 03_rgb.py in Mu. After a short delay, this should open a window with three sliders, one for each color: red, green and blue. Adjusting these sliders should change the color of the RGB LED.

Once you have finished, close the window (with the sliders in it).



The Code for Project 3

At first glance this program looks quite long, however this is mainly due to doing everything three times (one for each color/slider) and because there is a graphical user interface (GUI). The guizero library is used to make it easy to create a nice GUI. You can read more about guizero in Appendix E.

```
rgb_led = RGBLED(18, 23, 24)

red = 0
green = 0
blue = 0
```

This first section sets up the RGB LED by telling the Pi which pins we have used for each color (red, green and blue in that order). The next three lines initialize some variables that are used to keep track of the color values.

The next three methods are almost identical, the only difference is that each one controls a different color. Therefore, we can just look at how one works to understand the rest.

```
def red_changed(value):
    global red
    red = int(value)
    rgb_led.color = Color(red, green, blue)
```

Each of these functions will be called when its respective color slider is changed. This first function will therefore be called when the red slider is changed in order to update the red component of the color being displayed. The parameter *value* that is passed to the function will be the new value of the slider after it has been changed. The *global* keyword is used to define *red* as a global variable so that the variable *red* accessed inside the function is the same as the variable used outside of the function. The value of the red component is updated to the new value from the slider and the color of the RGB LED is then changed to reflect this.

The rest of the code is all for the GUI and again, there is some repetition for each of the three colors.

```
app = App(title='RGB LED', width=500, height=400,
layout="grid")
```

This line creates the window that the sliders will be in. It gives the window a title and sets its size to 500 pixels by 400 pixels. The layout is also set to a grid so that the text labels and sliders will be properly aligned.

```
Text(app, text="Red", grid=[0, 0])
Slider(app, command=red_changed, end=255, width=400,
height=50, grid=[1, 0]).text_size = 30
```

The first line creates a text label for the slider. This includes a grid reference to properly align everything in the window. The first number in the grid refers to the x or horizontal location and the second number corresponds to the y or vertical direction. The second line creates the slider itself. The argument *command=red_changed* defines what should happen when the slider is changed and in this case, will change the color of the LED using the functions that were defined before.

These two lines are then repeated for green and blue too.

Finally, the window needs to be displayed which is done using:

```
app.display()
```

Project 4. CheerLights

CheerLights is an Internet of Things (IoT) project that is designed to synchronise the colors of people's lights from all over the world. The idea is that anyone can tweet at the CheerLights Twitter account a color, which will then be set in a text document that can be accessed over the internet. This means that anyone using the CheerLights API (Application Programming Interface) can synchronise their lights to be the same color.

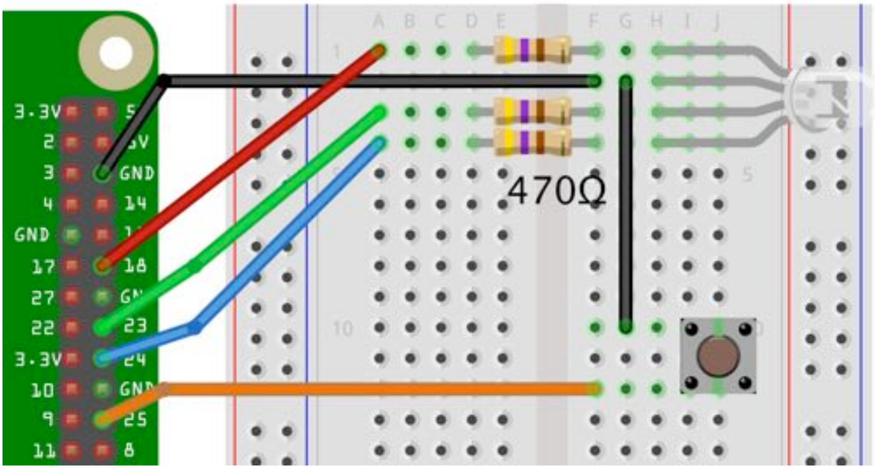
Step 1. Find the parts you need

For this project, you will need the following items:

- Three 470Ω Resistors
- An RGB LED
- Push Switch
- Five Female to Male jumper wires
- Male to male jumper wire

Step 2. Build and Connect the breadboard.

Wire up the breadboard and connect it to your Raspberry Pi using the diagram below as a guide. Note that this is the same basic layout as Project 3, but with an extra push button.



Step 3. Running the Program

To use this project your Raspberry Pi must be connected to the internet.

Load and run the program 04_cheerlights.py using Mu.

After a few seconds, the LED will automatically set itself to the current Cheerlights color, checking every 10 seconds. Pressing the button will turn the LED off until the Cheerlights color changes. You can force this to happen by tweeting a message such as *@cheerlights red.*

You should find that the LED color will change every few minutes, sometime much more often, as people around the world tweet to set a new Cheerlights color.

The Code for Project 4

In order to get the latest color from CheerLights, this program needs to be able to connect to the CheerLights API over the internet. This is handled by the Python *requests* library, which is used here to get the content of a web page containing the current CheerLights color in hex format.

```
from gpiozero import Button, RGBLED
from colorzero import Color
import time, requests

update_period = 10 # seconds
led = RGBLED(red=18, green=23, blue=24)
button = Button(25)

cheerlights_url = 
"http://api.thingspeak.com/channels/1417/field/2/last.txt"
old_color = None
```

This first section of code handles the import statements and does some initialization. The LED is set up on pins 18, 23 and 24 for red, green and blue respectively. The button is set up on pin 25.

The variable update_period sets the time in seconds between successive checks. 10 seconds is about right for this.

The variable *old_color* is used to determine whether the color retrieved from the cheerlights webservice has changed since the last request.

The *pressed* function is linked to a button press and sets the red, green and blue color values all to 0 turning the LED off.

The *while* loop contains the request to the Cheerlights web service and also checks to see if the Cheerlights color has changed. If it has, then it changes the LED color to the new color and then sets old_color to color.

Finally the program sleeps for the *update_period* (10 seconds).

```
def pressed():
    led.color = Color(0, 0, 0)
button.when_pressed = pressed

while True:
    try:
        cheerlights = requests.get(cheerlights_url)
        color = cheerlights.content
        if color != old_color:
            led.color = Color(color)
            old_color = color
    except Exception as e:
        print(e)
    time.sleep(update_period)
```

Project 5. Thermometer

This project uses a thermistor to measure the temperature. The temperature reading is not very accurate, but it should give you a rough idea.

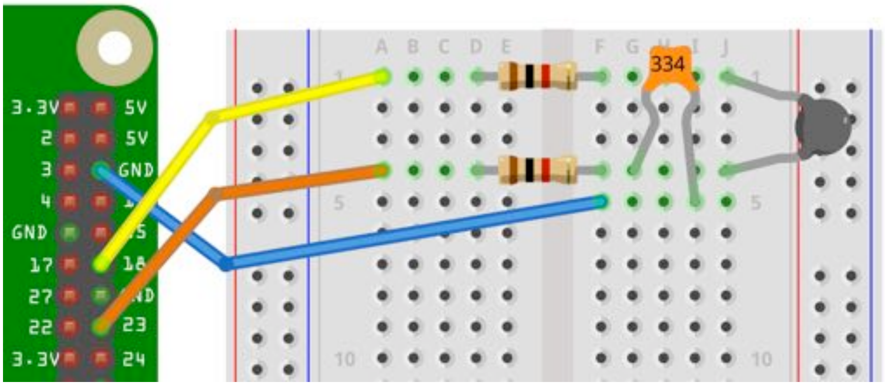
Step 1. Find the parts you need

For this project, you will need the following items:

- Two 1kΩ Resistors
- 330nF Capacitor
- Thermistor
- Three Female to Male jumper wires

Step 2. Build and Connect the breadboard.

Wire up the breadboard and connect it to your Raspberry Pi using the diagram below as a guide.



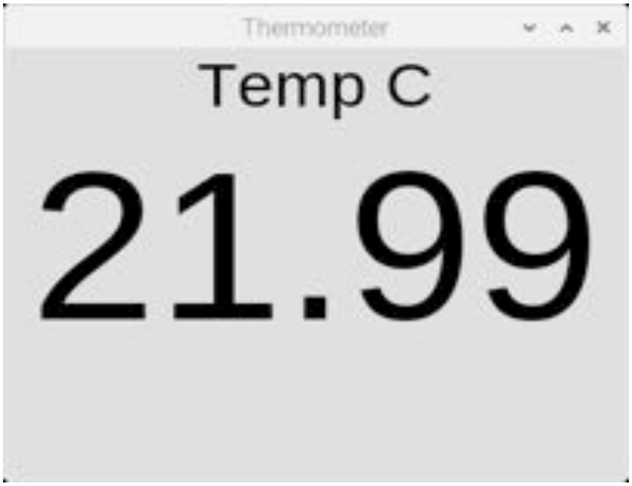
Note that the resistors in this project are different from the ones in the previous four projects. The capacitor and thermistor, like the resistors, do not have any preference for which way round they are placed.

Step 3. Running the Program

There are two versions of the program for this project. The program 05_thermomether.py displays the temperature in degrees Celsius, whereas the program 05_thermomether_f.py displays the temperature in Fahrenheit. Load the version you want to run into Mu and then click the Run button.

This should open a window that displays the approximate temperature of the thermistor, updating once per second. Try holding the thermistor (the black component) between your fingers to increase the temperature.

The program can be exited by simply closing the temperature reading window.



The Code for Project 5

The code for this project is almost exactly the same for each version, so here we will just look at the code for the Celsius version.

This project (and some of the subsequent projects) use a library called PiAnalog, written by Simon Monk. This library handles the analog side of using the Raspberry Pi's pins, which is not as straightforward as the digital pins we have been using so far. See Appendix F for more information on the Pi Analog library.

After initializing the PiAnalog library, a function must be declared to update the temperature reading. The function is reasonably self-explanatory:

```
def update_temp():
    temperature = p.read_temp_c()
    temperature = "%.2f" % temperature
    temp_text.value = temperature
    temp_text.after(1000, update_temp)
```

The temperature is read using the PiAnalog function *read_temp_c()* (more on what this function actually does in the Explanation section later). The temperature value is then rounded to two decimal places and updated in the text label in the GUI. The final line of the function calls the function again after 1000 milliseconds or 1 second.

This creates an infinite loop in the function to make sure that the temperature value is updated every second.

The rest of the program handles the GUI:

```
app = App(title = "Thermometer", width="400", height="300")
Text(app, text="Temp C", size=32)
temp_text = Text(app, text="0.00", size=110)
temp_text.after(1000, update_temp)
app.display()
```

The first line creates the window and the second line places a label to display "Temp C". The text label to display the current temperature reading is created next and assigned the name *temp_text*. The *after(1000, update_temp)* command that was used in the *update_temp* function is used again here to start the loop of updating the temperature and then the window can now be displayed.

The Explanation for Project 5

In the code for this project, the actual business of taking a temperature reading from the thermistor is hidden behind the *read_temp_c()* function from the PiAnalog library. If you wish, you can see exactly how this function works by typing the following commands in the Terminal:

```
$ cd /home/pi/pi_analog
$ nano PiAnalog.py
```

The operation of the circuit can be split into two parts: a thermistor (which is a device that has a resistance that varies with temperature) and the rest of the circuit which measures the resistance of the thermistor.

An equation called the Steinhart-Hart equation describes exactly how this temperature varies with the measured resistance of the thermistor. This means that by measuring the resistance of the thermistor, we can find the (approximate) temperature of the thermistor using the PiAnalog library.

The Fahrenheit version of the program works in exactly the same way except that the value for temperature in Celsius is converted into Fahrenheit at the end.

PROJECT 6. THERMOMETER PLUS

This project expands on the thermometer built in Project 5 to include a buzzer that will sound if the thermistor reaches a certain temperature (25 °C, 77 °F by default).

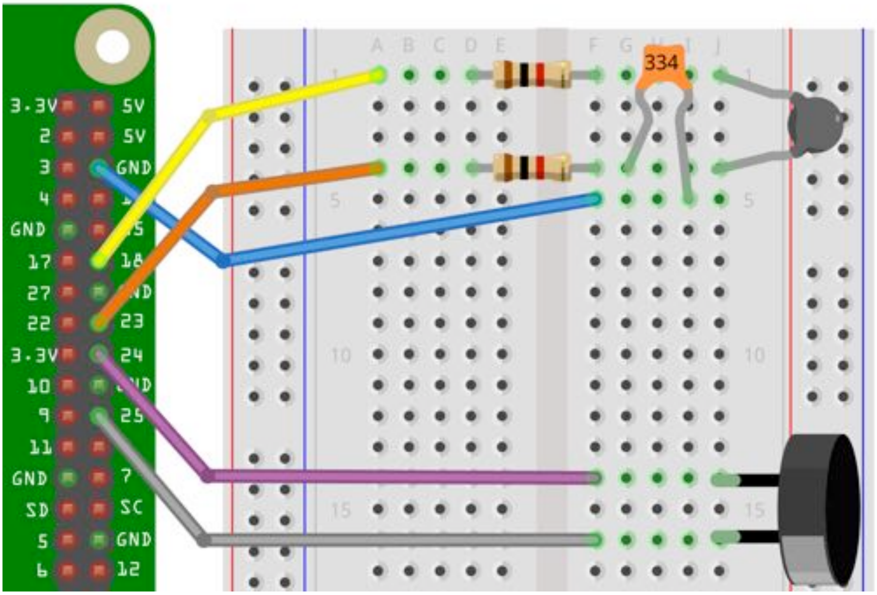
Step 1. Find the parts you need

For this project, you will need the following items:

- Two 1kΩ Resistors
- 330nF Capacitor
- Thermistor
- Buzzer
- Four Female to Male jumper wires
- Male to male jumper wire

Step 2. Build and Connect the breadboard.

If you have just built Project 5, you can leave everything where it is and simply add the extra wires and the buzzer. The orientation of the buzzer does not matter as it can be placed either way around.



Step 3. Running the Program

Similarly to Project 5, there are two programs for this project, one in Celsius and one in Fahrenheit (06_thermometer_plus.py and 06_thermometer_plus_f.py). Run the program for your preferred units of temperature and a window similar to that in Project 5, should open displaying the current temperature.

Warm the thermistor up by holding it between your fingers. Once the temperature gets above 25 °C (or 77°F), the buzzer should sound. If you can't get the temperature above the default value, you may need to edit the program and change the *set_temp* variable to something lower (see the Code section below if you are unsure how to do this).

The Code for Project 6

The code for this project is similar to Project 5 as the thermometer part of it is unchanged. For an explanation of how the thermometer works, see the Explanation for Project 5 section.

```
from PiAnalog import *
from guizero import App, Text
from gpiozero import DigitalOutputDevice
import time

set_temp = 25

pin1 = DigitalOutputDevice(24)
pin2 = DigitalOutputDevice(25)
p = PiAnalog()

def buzz(pitch, duration):
    period = 1.0 / pitch
    p2 = period / 2
    cycles = int(duration * pitch)
    for i in range(0, cycles):
        pin1.on()
        pin2.off()
        delay(p2)
        pin1.off()
        pin2.on()
        delay(p2)

def delay(p):
    t0 = time.time()
    while time.time() < t0 + p:
        pass

# Update the temperature reading
def update_temp():
    temperature = p.read_temp_c()
    if temperature > set_temp:
        buzz(2000, 0.5)
    temperature = "%.2f" % temperature # Round the
temperature to 2 d.p.
    temp_text.value = temperature
    temp_text.after(1000, update_temp)

# Create the GUI
app = App(title = "Thermometer", width="400", height="300")
Text(app, text="Temp C", size=32)
temp_text = Text(app, text="0.00", size=110)
temp_text.after(1000, update_temp) # Used to update the
temperature reading
app.display()
```

The variable at the beginning of the program, *set_temp* defines the threshold temperature above which the buzzer will sound. This is the variable to change if you can't get the thermistor up to 25 °C or you live somewhere warm.

The main difference between this program and the one from Project 5 is the code to control the buzzer.

First, a buzzer is set up on pin 24. Then, a new function is defined to control the actual buzzing of the buzzer. The note that the buzzer plays is determined by how quickly the buzzer turns on and off. The function *buzz()* is used so that the *pitch* can be given in Hertz (Hz) and is then converted to make the buzzer sound at this frequency.

The rest of the program is familiar from Project 5 except for two new lines in the *update_temp()* function. Quite simply, the two new lines will check if the current temperature value is above the *set_temp* value and, if it is, will sound the buzzer. This check occurs every time that the temperature is updated (once a second).

Project 7. Reaction Timer

This project will allow you to time your reactions using LEDs and buttons.

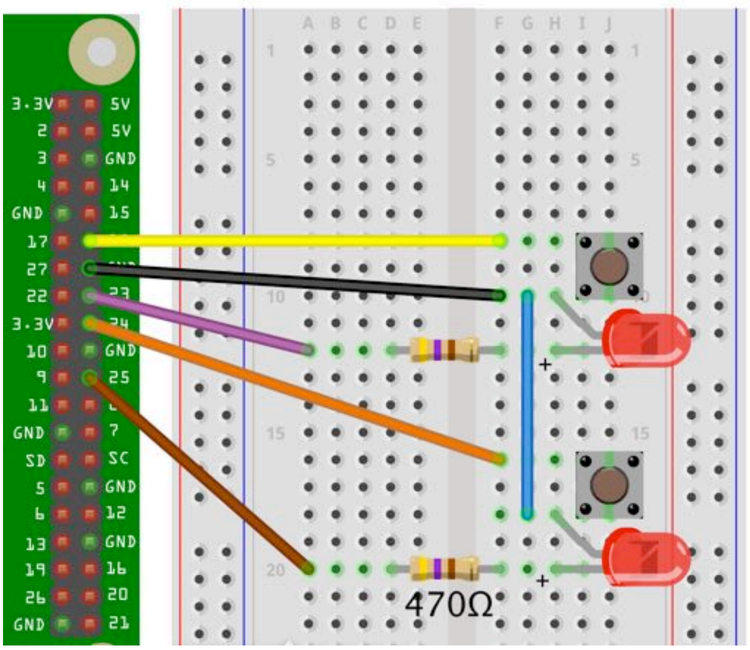
Step 1. Find the parts you need

For this project, you will need the following items:

- Two 470Ω Resistor
- Two Red LED
- Two Push switches
- Five Female to Male jumper wires
- Male to Male jumper wire

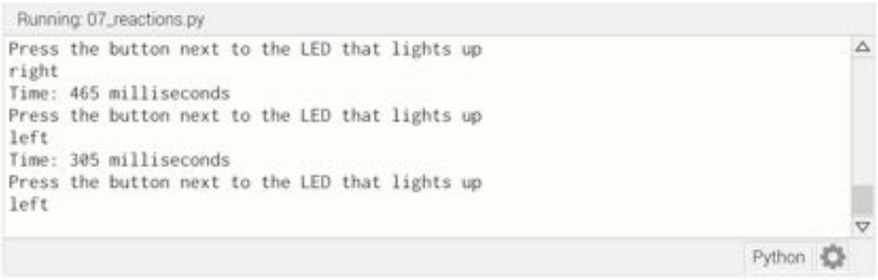
Step 2. Build and Connect the breadboard.

When assembling the circuit, make sure that the longest leg of the LEDs are in the positions marked with a + on the breadboard. Note also that the buttons are reversible and can be placed wither way around.



Step 3. Running the Program

To use the reaction timer, load and run the program in Mu. When the program starts, you will notice that the bottom part of the Mu window shows a message telling you to *Press the button next to the LED that lights up*.



```
Press the button next to the LED that lights up
right
Time: 465 milliseconds
Press the button next to the LED that lights up
left
Time: 305 milliseconds
Press the button next to the LED that lights up
left
```

After a while one of the LEDs will light, and you should press the button next to the LED that lights as fast as possible. You will then get a message telling you how many milliseconds you took to press the right button.

The Code for Project 7

This code may seem a bit complicated at first but this is mainly due to making sure the right button was pressed at the right time.

```
from gpiozero import LED, Button
import time, random

left_led = LED(25)
right_led = LED(23)
left_switch = Button(24)
right_switch = Button(18)

# find which buttons pressed 0 means neither, -1=both, 2=right, 1=left
def key_pressed():
    # if button is pressed is_pressed will report false for that input
    if left_switch.is_pressed and right_switch.is_pressed:
        return -1
    if not left_switch.is_pressed and not right_switch.is_pressed:
        return 0
    if not right_switch.is_pressed and left_switch.is_pressed:
        return 1
    if right_switch.is_pressed and not left_switch.is_pressed:
        return 2

while True:
    left_led.off()
    right_led.off()
    print("Press the button next to the LED that lights up")
    delay = random.randint(3, 7)    # random delay of 3 to 7 seconds
    led = random.randint(1, 2)      # random led left=1, right=2
    time.sleep(delay)
    if (led == 1):
        print("left")
        left_led.on()
    else:
        print("right")
        right_led.on()
    t1 = time.time()
    while not key_pressed():
        pass
    t2 = time.time()
    if key_pressed() != led:   # check the correct button was pressed
        print("WRONG BUTTON")
    else:
        # display the response time
        print("Time: " + str(int((t2 - t1) * 1000)) + " milliseconds")
```

The function *key_pressed()* is used to determine which combination of buttons have been pressed (or not). If both buttons are pressed, it will return -1, if neither are pressed, it will return 0, if just the left button is pressed it will return 1 and if the right button is pressed, it will return 2. The reason for picking these numbers will become clearer further on in the code.

To make sure that you cannot cheat by anticipating when the LED will light up, a random delay between 3 and 7 seconds is used. To choose which LED will be displayed, a random number between 1 and 2 is picked. If 2 is picked, we define that as being right, otherwise if 1 is picked, we set the *led* variable to 1 to represent left.

As soon as the LED to be lit is decided and lit, the current time is stored in *t1*. The while loop calls the *key_pressed()* function which returns the number corresponding to the combination of buttons pressed. As *key_pressed()* returns 0 when no buttons are pressed, the condition *not key_pressed()* is equivalent to *not 0* which is the same as *1* , or *True* and so the loop will keep on going until a button is pressed. The command *pass* is used in the body of the loop as nothing actually needs to happen in the loop itself, but it cannot be left blank.

Pressing a button will break the *while-pass* loop and so by recording the time in *t2*, the time taken to press the button can be found by doing *t2-t1*. If the wrong button is pressed, the user is informed of this. Otherwise, the correct button must have been pressed, so the reaction time is printed in milliseconds.

Project 8. Light Meter

In Projects 5 & 6, a special type of resistor called a thermistor was used to measure the temperature. In this project, a special type of transistor called a phototransistor is used to measure the light level.

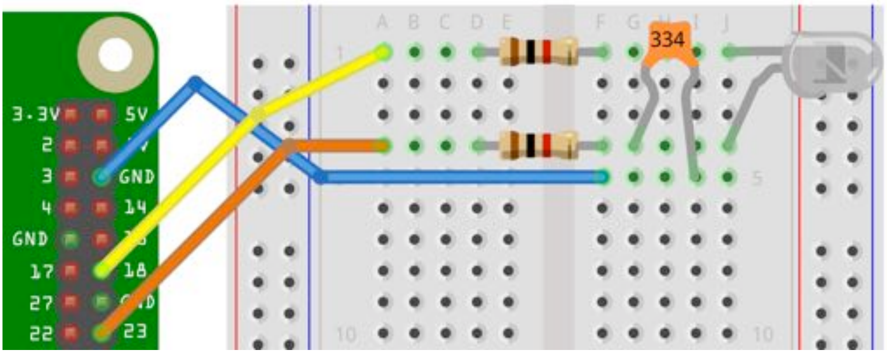
Step 1. Find the parts you need

For this project, you will need the following items:

- Two 1kΩ Resistors
- 330nF Capacitor
- Phototransistor
- Three Female to Male jumper wires

Step 2. Build and Connect the breadboard.

Note, the phototransistor must be placed with the longer lead to row 4 of the breadboard.



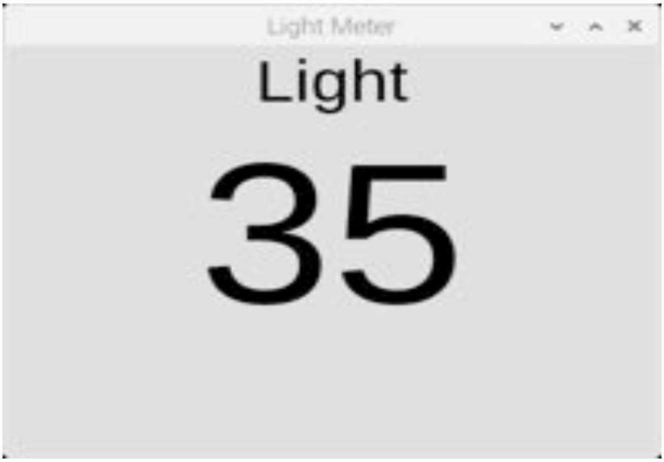
Step 3. Running the Program

Unlike the thermometer which has well defined units of temperature, the scale used for the light level here is somewhat arbitrary. Ultimately, the larger the number the more light is shining on the phototransister.

Load the program 08_light_meter.py into Mu and run it.

You should find that the light level changes by placing your hand over the phototransistor or by shining a torch on it.

When you want to exit the program, close the window that is displaying the light level.



The Code for Project 8

A lot of the code and general principles of this project are similar to those used in Projects 5 & 6. If you want a deeper explanation of some of the principles, have a look at the explanation section from Project 5.

```
from guizero import App, Text
from PiAnalog import *
import time, math

p = PiAnalog()

multiplier = 2000 # increase to make more sensitive

def light_from_r(R):
    light = 1/math.sqrt(R) * multiplier
    if light > 100:
        light = 100
    # Sqareroot the reading to compress the range
    return light
```

The values for the resistance of the phototransistor follow an inverse square root relationship with the perceived brightness, so the square root of the resistance is used in the function *light_from_r().*The variable *multiplier* can be used to adjust the sensitivity of the light meter.

The final parts of the program are related to the GUI and updating the reading.

```
def update_reading():
    light = light_from_r(r.read_resistance())
    reading_str = "{:.0f}".format(light)
    light_text.value = reading_str
    light_text.after(200, update_reading)

app = App(title="Light Meter", width="400", height="300")
Text(app, text="Light", size=32)
light_text = Text(app, text="0", size=110)
light_text.after(200, update_reading)
app.display()
```

First, the light level is calculated and then rounded to zero decimal places i.e. an integer. The value of the text label displaying the light level is then updated, and the final command in *update_reading()* is used to ensure that the reading will update every 0.2 seconds.

The final block of code is then just setting up the window, starting the *update_reading()* loop to run every 0.2 seconds and finally displaying the window.

Project 9. Light Harp

A more musical use of the phototransistor is to use it as a sort of light harp, where the musical note played by the buzzer depends on the level of light on the phototransistor.

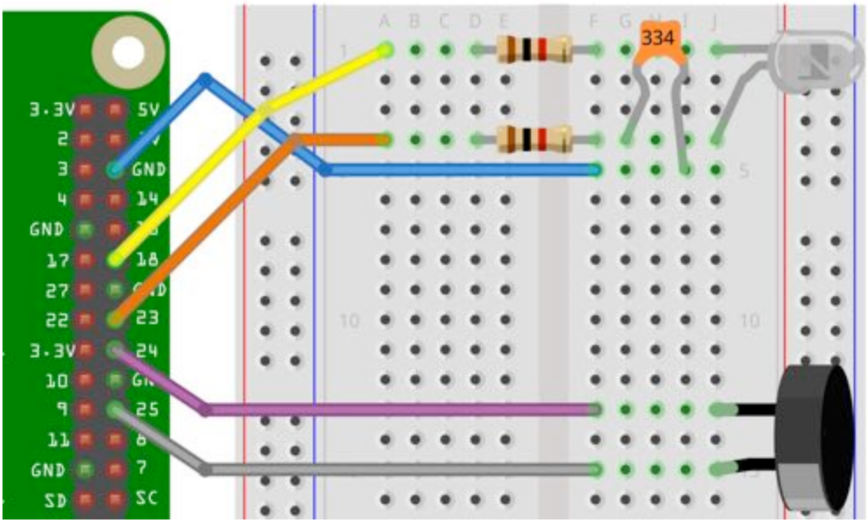
Step 1. Find the parts you need

For this project, you will need the following items:

- Two 1kΩ Resistors
- a 330nF Capacitor
- a Phototransistor
- a Buzzer
- Four Female to Male jumper wires
- a Male to Male jumper wire

Step 2. Build and Connect the breadboard

If you have just constructed Project 8, you do not need to take anything out of the breadboard and can simply add the extra wires and the buzzer as shown.



Step 3. Running the Program

Load and run the program 09_light_harp.py in Mu.

The piezo buzzer is not very musical! To change the tone, wave your hand over the phototransistor or try shining a torch on it.

The Code for Project 9

The code for this project seems much simpler than the code for the light meter in the previous project. Part of this is because this project does not need a GUI, and the other main reason is that there is no need to try and calculate the actual light level for this project (more on this below).

```
from gpiozero import DigitalOutputDevice
import time
from PiAnalog import *

pin1 = DigitalOutputDevice(24)
pin2 = DigitalOutputDevice(25)
p = PiAnalog()

def buzz(pitch, duration):
    period = 1.0 / pitch
    p2 = period / 2
    cycles = int(duration * pitch)
    for i in range(0, cycles):
        pin1.on()
        pin2.off()
        delay(p2)
        pin1.off()
        pin2.on()
        delay(p2)

def delay(p):
    t0 = time.time()
    while time.time() < t0 + p:
        pass

while True:
    reading = p.analog_read() # 1000 to 5000 ish for indoors
    f = reading * 2
    buzz(f, 0.1)
```

The way that the piezo buzzer works is by turning on and off very quickly. How quickly it turns on and off determines the tone emitted by the buzzer. The value of delay is taken from the *analog_read()* function from the PiAnalog library is proportional to the resistance of the phototransistor (and hence the light level), changing the light level of the phototransistor will change the tone of the buzzer.

If you want more information on how PiAnalog and the phototransistor works, see Projects 5 & 8.

Project 10. Proximity Detector

This project uses an open-ended wire to act as a proximity detector. If you get close enough to the wire, or actually touch it, the LED will light up. Moving away from the wire will then turn the LED off again.

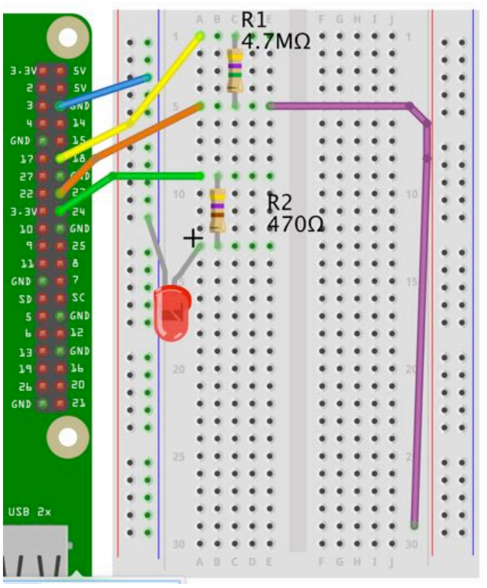
Step 1. Find the parts you need

For this project, you will need the following items:

- 470Ω Resistor
- 4.7MΩ Resistor
- Red LED
- Four Female to Male jumper wires
- Male to Male jumper wire

Step 2. Build and Connect the breadboard.

The color code of the two resistors used in this project are very similar, so be careful to get them the right way round.



Step 3. Running the Program

Load and run the program 10_proximity.py in Mu.

When you first run the program, it should tell you that it is calibrating. Try to keep away from the lead while this is happening. After it has finished calibrating, it should output a number and you can then try to get the LED to light by getting close to the lead.

You may well find that you can only get the LED to light by actually touching the floating/open-ended lead. If you want to improve the sensitivity, try attaching a square of aluminum foil to the end of the wire. This should allow you to turn on the LED by hovering your hand a few cm above the foil.

The Explanation for Project 10

If you look at the code before reading this explanation it will probably look a bit like magic, but it is actually using an idea we have seen before in the thermometer and light meter projects.

In the thermometer and light meter projects, the resistance of something needed to be measured, which was achieved by timing the charge time of a capacitor. The way to tell when the capacitor was charged above a certain level was to check when a pin connected to the capacitor became an input pin, which happens at about 1.65 V.

But this project doesn't use a capacitor, so why are we talking about this? The reason is that the lead and the air surrounding it have a very small capacitance. As was discussed in Project 5, the time taken for a capacitor to charge is proportional to the resistance times the capacitance. Because the lead and air form a small capacitor, a very large resistor (this project uses 4.7 MΩ) can be used to boost this time to something long enough to measure.

This means that by timing how long this “capacitor” takes to charge, the time constant (the product of resistance times capacitance) of the lead and air can be measured. If this time constant is repeatedly measured, then when someone places a finger near the lead, the capacitance (and time constant) will change. If the time taken changes above a certain threshold value, then the LED can be turned on, as someone has moved close enough to the lead to change the capacitance.

The Code for Project 10

With the above explanation in mind, the code for this project should (hopefully) make sense.

```
# 10_proximity.py

from gpiozero import DigitalOutputDevice, LED, Button
import time

# This project uses the Capsense technique modelled on this:
# http://playground.arduino.cc/Main/CapacitiveSensor


threshold = 0

# setup the pin modes
send_pin = DigitalOutputDevice(18)
sense_pin = Button(23, pull_up=None, active_state=True)
red_led = LED(24)
send_pin.on()

# return the time taken for the sense pin to flip state as
# a result of the capacitative effect of being near the sense pin
def step():
    send_pin.off()
    t1 = time.time()
    while sense_pin.value:
        pass
    t2 = time.time()
    time.sleep(0.1)
    send_pin.on()
    time.sleep(0.1)
    return (t2 - t1) * 1000000

# This function takes 10 readings and finds the largest
def calibrate():
    global threshold
    print("Wait! Calibrating")
    n = 10
    maximum = 0
    for i in range(1, n):
        reading = step()
        if reading > maximum:
            maximum = reading
    threshold = maximum * 1.2
    print(threshold)
    print("Calibration Complete")


calibrate()

while True:
    reading = step() # take a reading
    red_led.value = (reading > threshold)
```

The first step is to set up the pins. The *send* lead is used to charge the “capacitor” and the *sense* lead will indicate when the capacitor is charged above or below a certain level.

The *step()* function is used to take a reading of the current time constant (resistance times capacitance). This is done by turning off the *send* pin and timing how long the *sense* pin takes to drop below a certain voltage as the *sense* pin is connected to the *send* pin by the resistor and lead capacitance. The command *sense_pin.value* will return 1 if the pin is above this level or 0 if it is below it. Therefore the while loop will keep going until the *sense* pin drops below this level.

Once the timing is complete, the *send* pin is turned back on ready for the next reading.

To know whether or not the capacitance has changed (by moving your finger closer to the lead), the ambient capacitance must first be measured. This is done using the *calibrate()* function which takes 10 readings (using the *step()* function) and returning the largest reading taken multiplied by 1.1.

The reason the threshold is the maximum value times 1.1 is because the capacitance can fluctuate quite a bit, so if the threshold was simply set to the maximum value, the LED would keep being set off.

Once the calibration is complete, the program can then take continuous readings and compare them to the threshold value. Introducing a finger to the lead will increase the capacitance, so if the current reading is greater than the threshold, the condition *reading* > *threshold* will evaluate to 1 (or *True*) and the LED will be turned on.

Appendix A. Troubleshooting

Problem: Error messages when using update/upgrade commands

Solution: Reboot your Raspberry Pi and run the commands again. Make sure you are connected to the internet.

Problem: Error message when using wget command: unable to resolve host address 'monkmakes.com'

Solution: This most likely means that your Raspberry Pi is not connected to the internet. If this problem persists after connecting to the internet, wait about 10 minutes and then try again.

Problem: Error message when using wget command: “Error 404”

Solution: This usually indicates that the URL in the *wget* command has been mistyped. Try typing out the command again.

Problem: LED is not working

Solution: Check that the LED is the right way around. The long leg is the positive end and the short leg is the negative end. Make sure that the negative (shorter) leg is connected to GND.

Problem: Components are not working.

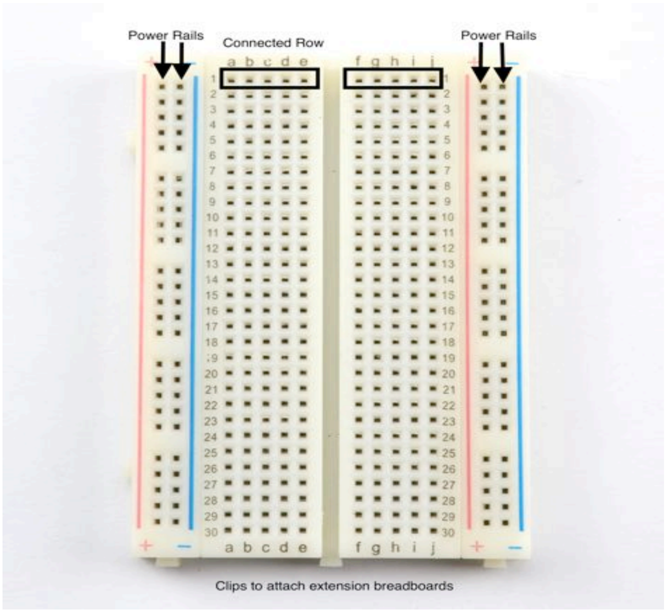
Solution: Carefully check that the legs of each components are not touching each other. Check if all of the component legs are properly pushed into the breadboard. Make sure that all of the leads are going to the correct pins on the Pi and to the correct holes on the board. Make sure that all LEDs are the correct way around. Double check that all resistors have the correct value (color stripes). Make sure that you are running the correct program for the project you have set up.

Support

For the most up-to-date help on this kit, see http://monkmakes.com/pb1 or contact MonkMakes support at support@monkmakes.com

Appendix B. How a Breadboard Works

All the projects in this kit are built on the provided breadboard which is connected to the Raspberry Pi GPIO pins using the Male to Female Jumper Wires.



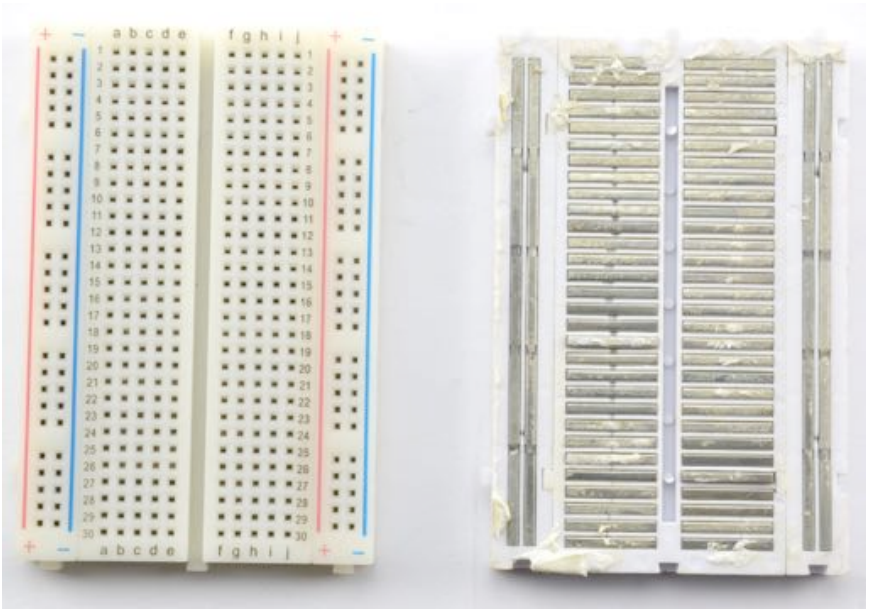
If you connect the projects in exactly the same way as the diagrams then they should work fine. However, it can be quite tedious to use exactly the same holes on the breadboard so it is worth understanding how the holes are all connected.

The rails on the board are labelled around the outside - two sets of outer rails labelled + and – and the center holes are labelled using numbers 1-30 for rows and a-j for columns.

The bulk of the board in the center is connected **horizontally**. Row 1 is not connected to row 2 however holes 1a, 1b, 1c, 1d and 1e are all connected. The break in the very center of the board also breaks the connections between rows. For example, hole 1e is not connected to 1f. Each row follows the same pattern.

Again, the rows are on the right hand side of the board are connected (1f-1j).

If you were to take your breadboard apart, here's what it would look like. You can see the metal clips connecting each section together .



To use the board, firmly push the metal leads of the components or wires into the holes of the board. Make sure that the leads are in the center of the holes as they can often get stuck on the sides.

The component leads often need to be bent to properly fit in the right holes, but be careful not to bend the leads too far (especially on the components with shorter leads such as the buzzer) as they can snap off.

Appendix C. The Resistor Color Code

Resistors have little stripes on them that tell you their value. Here's how to read them.

Each color has a value.

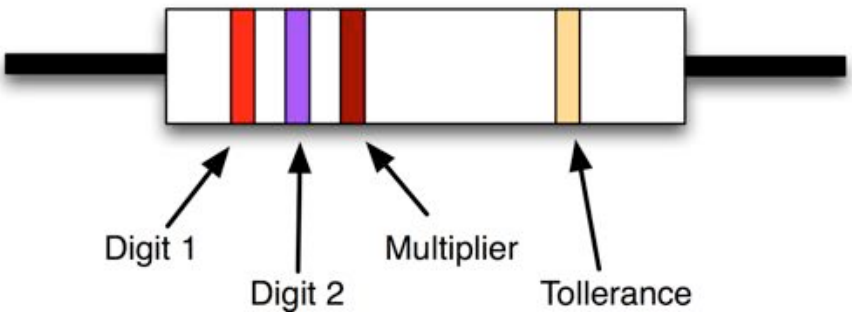
There will generally be three colored bands together starting at one end of the resistor, a gap, and then a single band at one end of the resistor. The single band at the far side indicates the accuracy of the resistor value.

The first band is the first digit, the second the second digit and the third 'multiplier' band is how many zeros to put after the first two digits.

The Gold and Silver stripes at the far end of the resistor are used to indicate how accurate the resistor is, so Gold is +-5% and Silver is +-10%. In other words a Gold (5%) 1000Ω (1kΩ) resistor could have an actual resistance between 950Ω and 1050Ω.

5% is plenty accurate enough for the projects in this kit.

Black	0
Brown	1
Red	2
Orange	3
Yellow	4
Green	5
Blue	6
Violet	7
Gray	8
White	9
Gold	5%
Silver	10%



Appendix D. The GPIO Zero Library

The GPIO Zero library is designed to make it easy to program electronics such as LEDs and switches that are attached to your Raspberry Pi's GPIO pins.

The library contains a number of classes that represent the electronics being attached.

For example, the class LED represents (you guessed it) an LED attached to a particular GPIO pin. To be able to use this class, we must first import it from the GPIO library like this:

```
from gpiozero import LED
```

gpiozero now needs to know which pin the LED is attached to, which you do as follows. In this case the LED is attached to pin 18, so we use:

```
red_led = LED(18)
```

Now, if we want to turn the LED on we can just write:

```
red_led.on()
```

and to turn it off again:

```
red_led.off()
```

We can also make it do fancy things like blink on and off, as we did in Project 1, using:

```
red_led.blink(on_time=0.5, off_time=0.5)
```

You can find an excellent tutorial on GPIO Zero here: https://www.raspberrypi.org/blog/gpio-zero-a-friendly-python-api-for-physical-computing/

And full documentation on the library here: https://gpiozero.readthedocs.io

Appendix E. The GUI Zero Library

Laura Sach and Martin O'Hanlon at The Raspberry Pi Foundation have created a Python library that makes it super easy to design GUIs. The installation script for this kit installs this on your Raspberry Pi.

For example, Project 5 of this kit uses guizero to display the temperature. Before you can use the library, you need to import the bits of it that you want to use in your program.

For example if we just wanted a window containing a message here's the import command:

```
from guizero import App, Text
```

The class *App* represents the application itself and every program you write that uses guizero needs to import this. The only other class needed here is *Text* that is used to display the message.

The following command creates the application window, specifying a title and the window's starting dimensions.

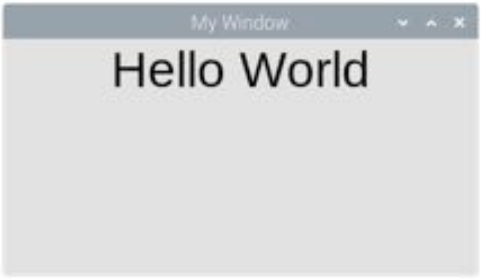
```
app = App(title = "My Window", width="400", height="300")
```

To add some text to the window, we can use the line:

```
Text(app, text="Hello World", size=32)
```

The window is now prepared for display but won't actually appear until the program runs the line:

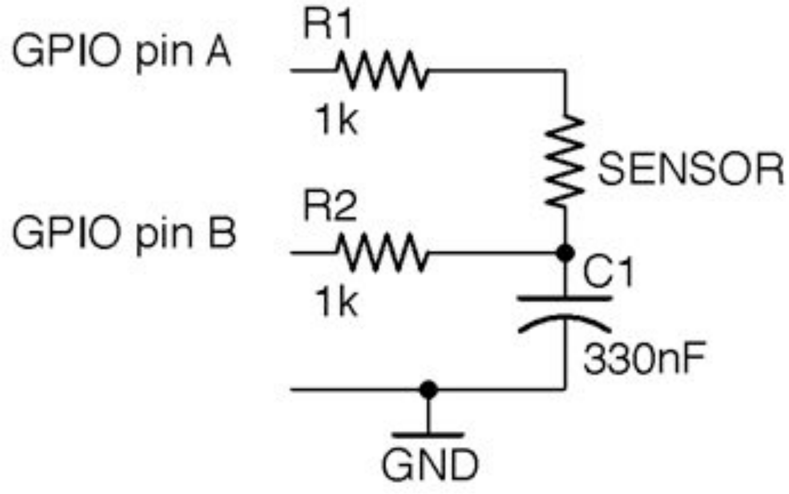
```
app.display()
```



You can find out more about guizero here: https://lawsie.github.io/guizero/start/

Appendix F. The Pi Analog Library

The PiAnalog library allows you to use analog sensors like the thermistor and phototransistor with the entirely digital IO pins of the Raspberry Pi.

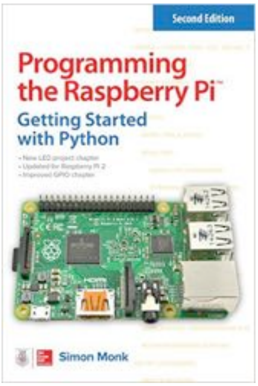


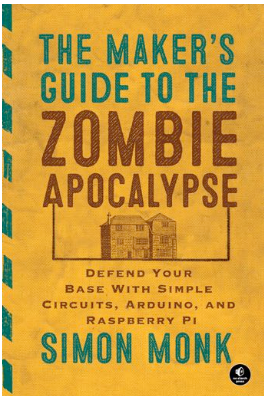
The resistance is measured using a capacitor, which acts as a temporary store of charge. The relationship between the time taken for a capacitor to charge (or discharge) is linked to the resistance and capacitance of the capacitor. The value of the capacitance is known (here we used a 330nF capacitor) and so by connecting the capacitor to the thermistor and timing how long the capacitor takes to discharge, the resistance of the thermistor can be found. This resistance can then be plugged into an equation (the Steinhart-Hart equation) to give an approximate value for the temperature of the thermistor.

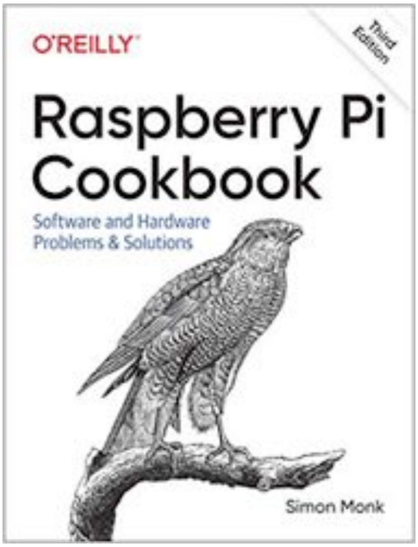
You can find full documentation on the Pi Analog library here, including an explanation of the electronics and theory of how it works, here:
https://github.com/simonmonk/pi_analog

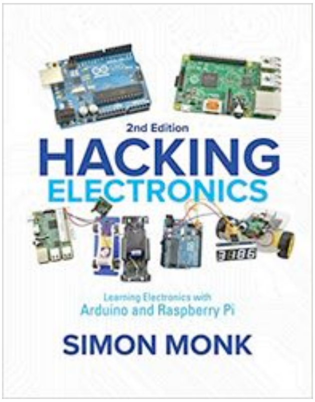
Books

This kit gives you a good set of parts to go off and start developing your own projects. You may find that you want to learn more about using and programming the Raspberry Pi. These books were written by the designer of this kit.




THE MAKER'S GUIDE TO THE ZOMBIE APOCALYPSE
Defend Your Base With Simple Circuits, Arduino, and Raspberry Pi
SIMON MONK



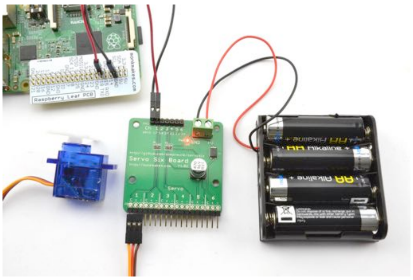


Other Kits

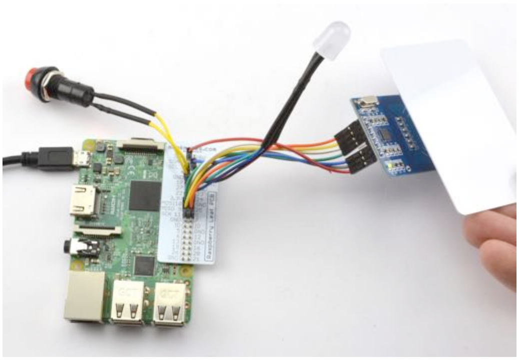
As well as this kit, MonkMakes makes all sorts of kits and gadgets to help with your projects.

Find out more, as well as where to buy here: https://monkmakes.com you can also follow MonkMakes on Twitter @monkmakes.

Amplified Speaker Kit for Raspberry Pi



ServoSix Kit for Raspberry Pi



Clever Card Kit for Raspberry Pi