Project Kit Project Guide

Initial Setup

Hardware Setup

Amongst all the items in your Raspberry Pi project kit, you should find a Raspberry Pi 2 model B board, a breadboard (a plastic board with lots of holes in it, to allow you to connect electronic components), a Perspex board with a number of holes in it, plus a number of screws, nuts, plastic spacers and rubber pads. The first thing you'll need to do is to get all of these bits together in order to assemble your prototyping platform.

First of all, take the Perspex board and pass the 4 screws through the holes from the underside. From above, you should be able to see the lettering on the Perspex board such that it is the right way (not reversed). Place the Perspex board on a flat surface so that the 4 screws are held in place and then slide the 4 plastic spacers over the top of each of the 4 screws.

Having done this, you should be able to place the Raspberry Pi such that it sits on top of the 4 spacers, the various ports and components are visible from above, and the 4 screws pass through the holes in the Raspberry Pi board. When you do this, ensure that the GPIO pins (the 40 pin header) is facing towards the centre of the Perspex board, and all other ports (USB, Ethernet, A/V, HDMI, micro USB and SD card slot) are facing outwards. Having done this, carefully screw the 4 nuts onto the 4 screws in order to hold the Raspberry Pi in place. Use a small screw driver and a pair of mini long nose pliers to tighten them, but be careful not to over tighten these nuts, as this may damage the Raspberry Pi board.

Next, remove the backing from the adhesive pad on the underside of the breadboard and stick the breadboard in place on the Perspex board, within the rectangular area marked on the top of the Perspex board next to the Raspberry Pi. It doesn't matter too much which way round you mount the breadboard onto the Perspex board, although if you mount the breadboard such that the 2 small plastic lugs on the longer side of the breadboard face away from the Raspberry Pi it will be possible to connect additional breadboards by slotting them in place.

Finally, one at a time remove the 4 rubber pads from their protective backing and stick them in place on the underside of the Perspex board; one in each corner of the board. These rubber pads can be pressed into place by putting the Perspex board on a firm surface (such as a table) and pressing down onto the 4 corners of the board.

Software setup

You've now completed the build of your hardware, so the next step is to get your hardware up and running and ready to run some of the great projects that we've designed for you. There are loads of excellent resources available on the internet that will tell you everything you could possibly need to know to get your Raspberry Pi up and running. The quick start guide at the Raspberry Pi foundation's web site is an excellent starting point.

https://www.raspberrypi.org/help/quick-start-guide/

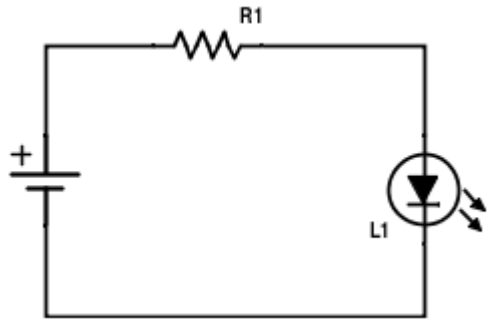
If you follow the instructions at this link, you may be a little unsure about what kind of SD card we have provided you with in your Project kit. The micro SD card provided with the Raspberry Pi project kit is pre-installed with NOOBS, and is set up to boot automatically into the Raspbian operating system. This means that when you insert the micro SD card and switch on the power supply to your Raspberry Pi, you should notice on your monitor that the Pi runs through its boot sequence, and will eventually display a desktop which will allow you to start working on your projects.

Project One – Light an LED

Introduction

So you've got your Pi and breadboard all set up. Everything is up and running and you've upgraded to the latest version of Raspbian etc. Now you've got that interesting looking bag of components burning a hole in your pocket, right? Let's use some of them and make the Pi control something in the real world. This first project is simple; it gets you used to using the breadboard and understanding how an LED works in a circuit.

This is the circuit we are going to build; it is a simple circuit designed to illuminate one of the LEDs in your component pack.

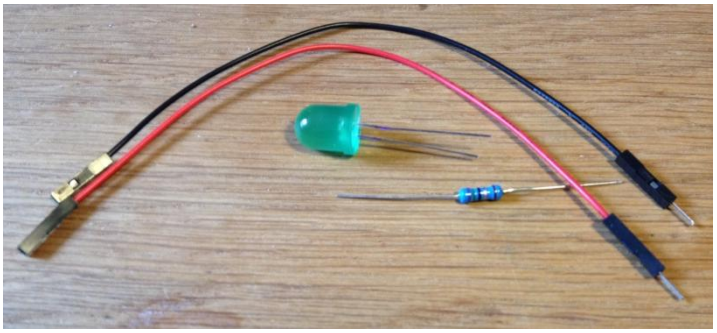


http://www.instructables.com/id/Choosing-The-Resistor-To-Use-With-LEDs/?ALLSTEPS

You will need:

- 1 LED (choose your favourite colour)
- 1 75R Resistor (these are the ones with the first three bands being Purple (7) Green (5) and Black (x1). 75 x 1 = 75ohms, for a full explanation of resistor colour codes see the links below)
  http://www.allaboutcircuits.com/textbook/reference/chpt-2/resistor-color-codes/

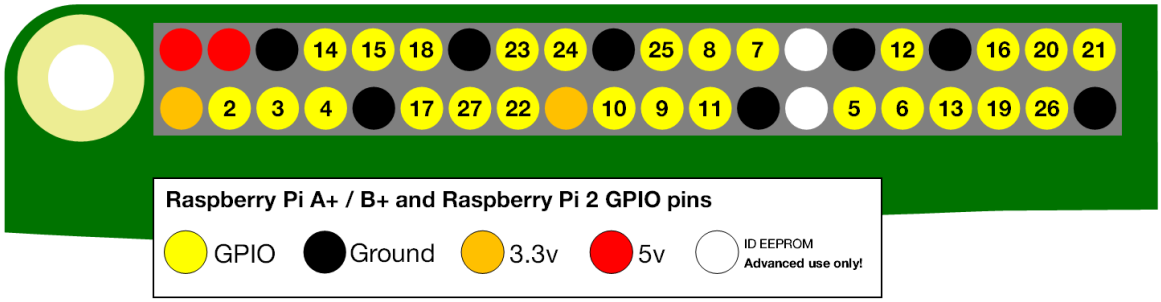
  https://physics.ucsd.edu/neurophysics/courses/physics_120/resistorcharts.pdf
- 2 Jumper wires (black and red)



We are going to build this circuit on our breadboard. Breadboards are useful electronics prototyping tools and are wired like this -

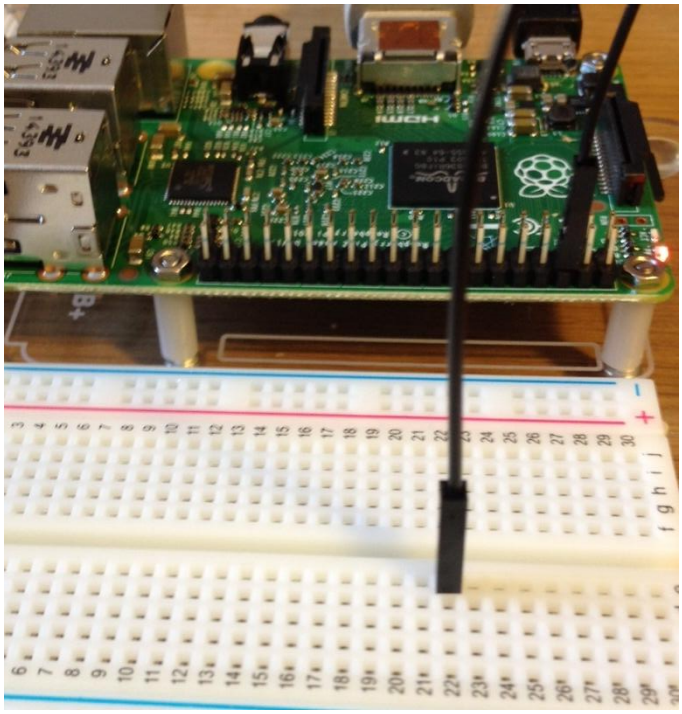
https://learn.sparkfun.com/tutorials/how-to-use-a-breadboard

We are going to connect our circuit to the Pi using the 40-pin GPIO connector. The pinout for the GPIO connector looks like this: For more info on the GPIO pinouts take a look at the excellent www.pinout.xyz web page from @Gadgetoid

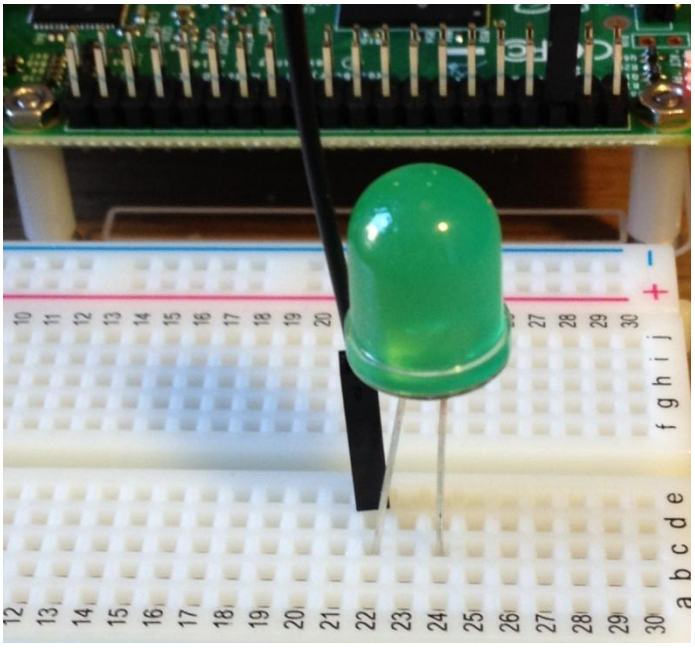


So let's get building our first circuit:

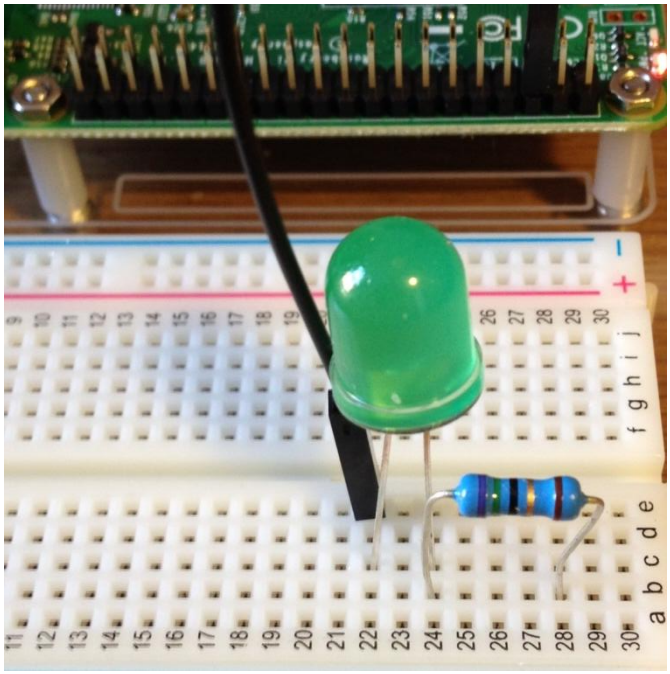
1. The first part of the circuit we will build is the link to the –ve side of our power source, in this instance we are using the Ground signal on the Pi GPIO. Take a black jumper wire and insert the pin into the breadboard next to the central row. Plug in the socket end to one of the Ground connectors on the GPIO. I used pin 6 as shown below:



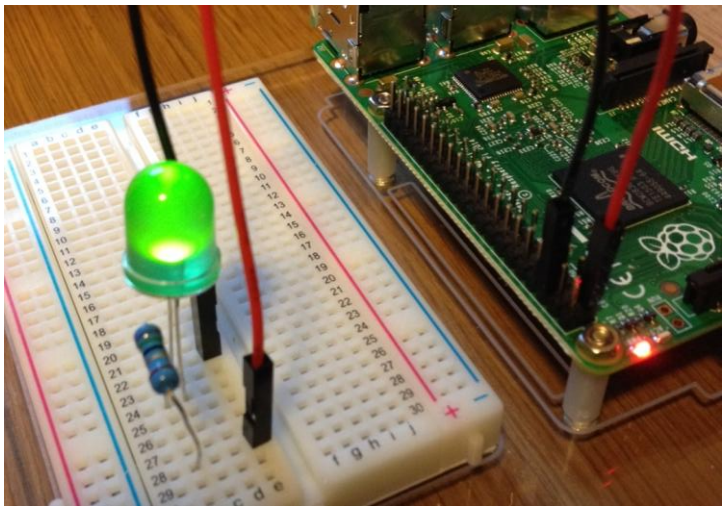
2. Now let's add the LED into the circuit. Insert the shorter leg of the LED, remember this is the –ve side called the cathode, into the same row as the black jumper wire. Insert the other, longer, leg into a different row.

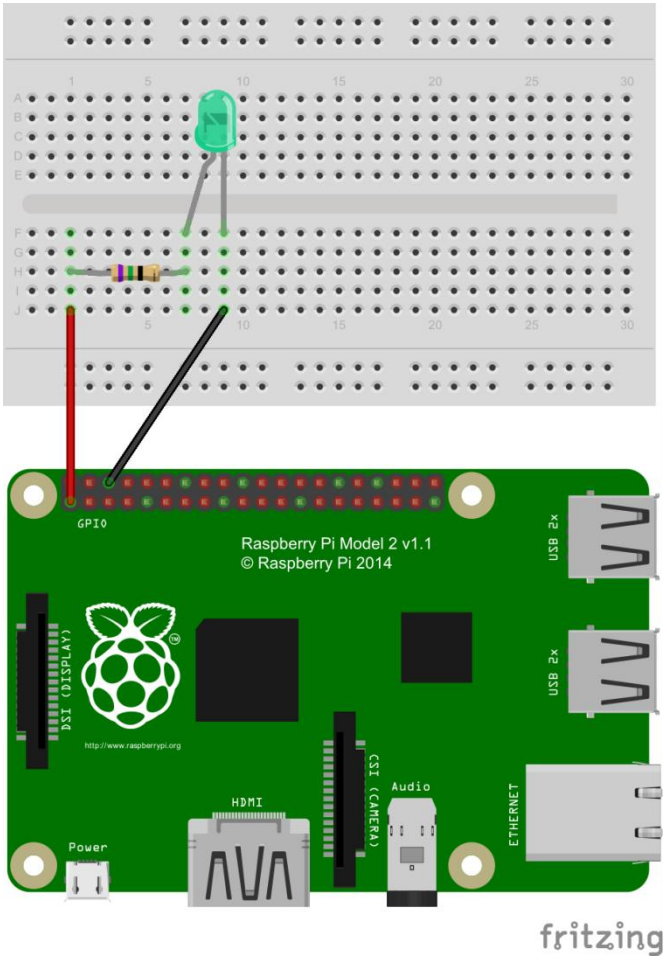


3. Next we need to add the resistor into the circuit. Take the 75R resistor and insert one leg into the breadboard on the same row as the +ve anode of the LED (remember the anode of the LED is the one with the longer leg). Insert the other leg into a different row of the breadboard as below. The good news here is that, unlike LEDs, resistors are not polarised so you cannot plug it in the wrong way around!



4. Now to complete our first circuit. We need to connect the resistor to the +ve side of our power source. Take a red jumper lead and insert the pin into the same row as the second leg of the resistor. Plug the socket end to one of the +3.3V connectors on the Pi GPIO. I've plugged it into pin 1 below. And let there be light!

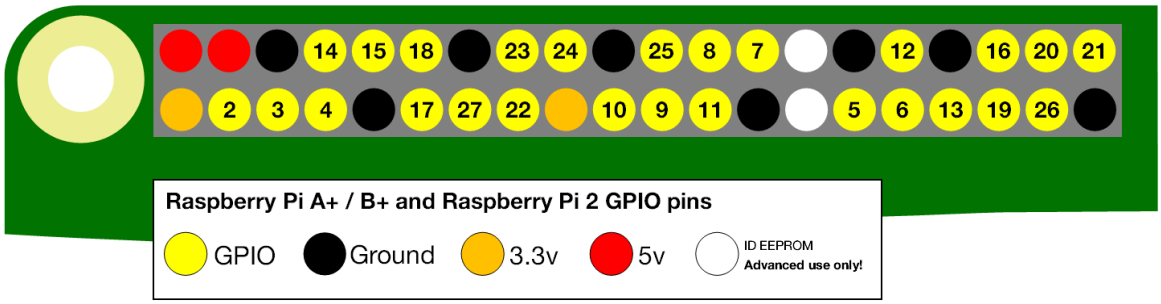




Project Two – Controlling the LED with the Pi

Introduction

This project builds directly on from what we learnt in Project One. We are going to use the same LED circuit but this time we will control the LED from the Raspberry Pi. That means that it is time to do some coding!



So that we can control the LED from the Pi we need to make one small change to our circuit. The Ground, 3.3V and 5V pins of the Pi are always at those voltages. The GPIO pins, however, we can control by writing a Python programme to run on the Pi. So let's move the red jumper wire from 3.3V on Pin 1 and connect it to GPIO2 on Pin 3.

In order to make it easier to control the GPIO pins and connect them to real world electronic components we are going to use a library of programming commands called GPIO Zero.

https://pythonhosted.org/gpiozero/

To install GPIO Zero type the following commands at the command prompt

```
sudo apt-get install python-pip python3-pip
sudo pip install gpiozero
sudo pip-3.2 install gpiozero

sudo pip install gpiozero --upgrade
sudo pip-3.2 install gpiozero --upgrade
```

Once installed type:

```
sudo idle3
```

This will open the Python programming shell. Use File->New Window to open a new editor window and then type in the following code:

```
from gpiozero import LED

green = LED(2)
green.blink()
```

The first line imports the LED class into your code from the GPIO Zero library. The second line creates an LED object called "green" which is connected to GPIO2. The final line calls the blink

method which as the name suggests makes our LED blink. By default the LED turns on for 1 second and then off for one second and then repeats forever. The blink method is quite powerful and actually allows you to pass arguments which control how long the LED is on, how long it is off and how often it blinks. That's all a bit easy and we're supposed to be learning some Python so let's write our own code to control the blinking. Enter the following code:

```
from gpiozero import LED
from time import sleep

green = LED(2)

while True:
    green.on()
    sleep(1)
    green.off()
    sleep(1)
```

This programme will do exactly the same as our three line piece of code before. This time however we have imported the sleep class from the time library. We've then created a loop and used the .on and .off methods from the LED class to turn our LED on and off. Try adjusting the arguments of the two sleep commands and see how the blinking of the LED changes.

But what if you don't want the LED to blink forever? Let's try the same code but now replace the While loop with a For loop.

```
from gpiozero import LED
from time import sleep

green = LED(2)

for i in range (0, 3):
    green.on()
    sleep(1)
    green.off()
    sleep(1)
    print ("%d blink" % (i))
```

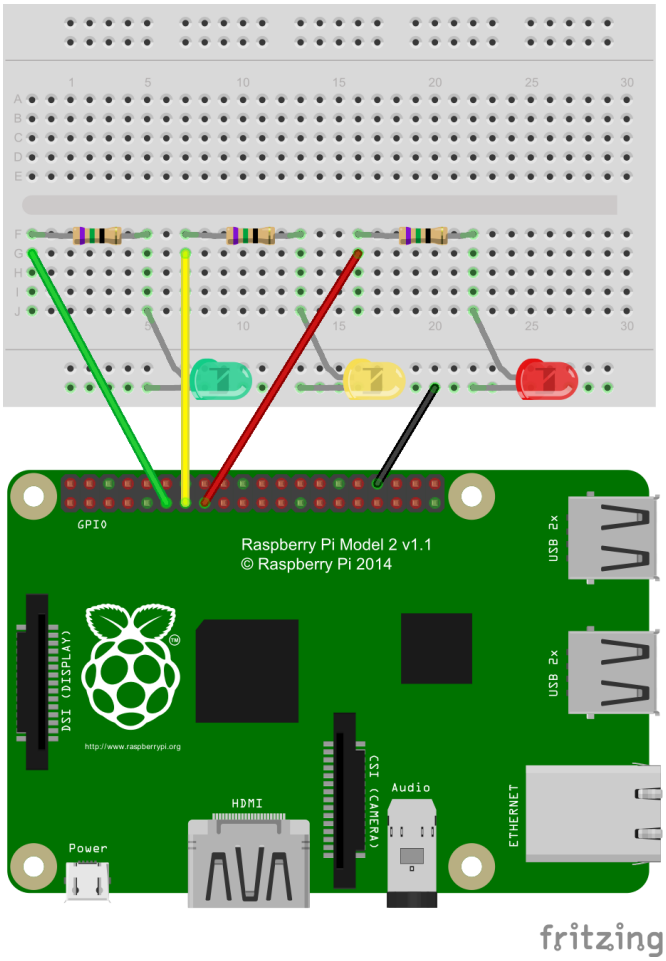
Project 3 – Basic Traffic Lights

Introduction

So, having learned how to control a single LED, let's move on to something a little more complex. This time we're now going to have a go at switching 3 LEDs on and off, following a defined sequence; your very own set of traffic lights.

Wiring up a single LED is fairly simple. However, if we were to have a single jumper wire for every LED lead, we would very quickly run out of jumper wires. For this reason we're going to start making use of the breadboard strips that run along the length of the breadboard. These strips are labelled '+' and '-', and are specifically designed to be used as a common ground or positive voltage strip. For the purposes of this particular project, we will use the strip marked as '-', and will connect the negative terminals (or cathodes) of each of our LEDs directly to this '-' strip. We call this a 'common cathode' configuration, because all the cathodes are joined together.

For this project, we will require 3 LEDs, 3 75R resistors (1 for each LED), and 4 jumper wires. 1 jumper wire is required to connect the negative strip on breadboard to one of the ground pins on the Raspberry Pi. We also require 1 jumper wire for each of the LEDs; to connect from the GPIO pins on the Pi to a strip on the breadboard which will connect to the resistor for each of the LEDs. The following diagram shows how we need to connect our components.



The jumper leads controlling the LEDs are connected to physical pins 11, 13 and 15 of the Pi. You will notice that these are actually labelled as pins 17, 27 and 22; and this is how we will need to refer to them in our Python code. It doesn't matter which ground pin the black jumper lead is connected to on the Raspberry Pi, but for this project we are going to use physical pin 34, as this will make the wiring of our later projects a little bit easier to follow.

Having completed and checked the wiring, it's time to take a look at the code we will need to control our traffic lights. As with our previous projects, we're going to make use of the GPIO zero library, since this simplifies things for us. You'll notice that the first few lines of code for this project are almost identical to those from our previous project. The only difference is the pin number that is used for our LED.

```
from gpiozero import LED
from time import sleep

green = LED(17)
yellow = LED(27)
red = LED(22)

while True:
    green.on()
    yellow.off()
    red.off()
    sleep(10)
    green.off()
    yellow.on()
    red.off()
    sleep(1)
    green.off()
    yellow.off()
    red.on()
    sleep(10)
    green.off()
    yellow.on()
    red.on()
    sleep(1)
```

Something else you might have noticed about this piece of code is that it is quite repetitive. When we write code, it is always a good idea to look at possible ways of simplifying the code, to make it easier to understand. One way that we can do this, is by making use of a programming construct known as a function, or procedure. Functions are used to define a sequence of instructions that would otherwise be repeated many times in our code. Functions and procedures also allow us to pass parameters to them in order to control the way that they work

So, let's look at an alternative way of writing this code. We're going to write a function, called switchLights, that switches the 3 LEDs on or off, depending on 3 parameters which are passed to the function, and then sleeps for a specified amount of time. The code below shows how we define this function in our code.

```
def switchLights (greenLight, yellowLight, redLight, sleepTime):
    if greenLight:
        green.on()
    else:
        green.off()

    if yellowLight:
        yellow.on()
    else:
        yellow.off()

    if redLight:
        red.on()
    else:
        red.off()

    sleep(sleepTime)
```

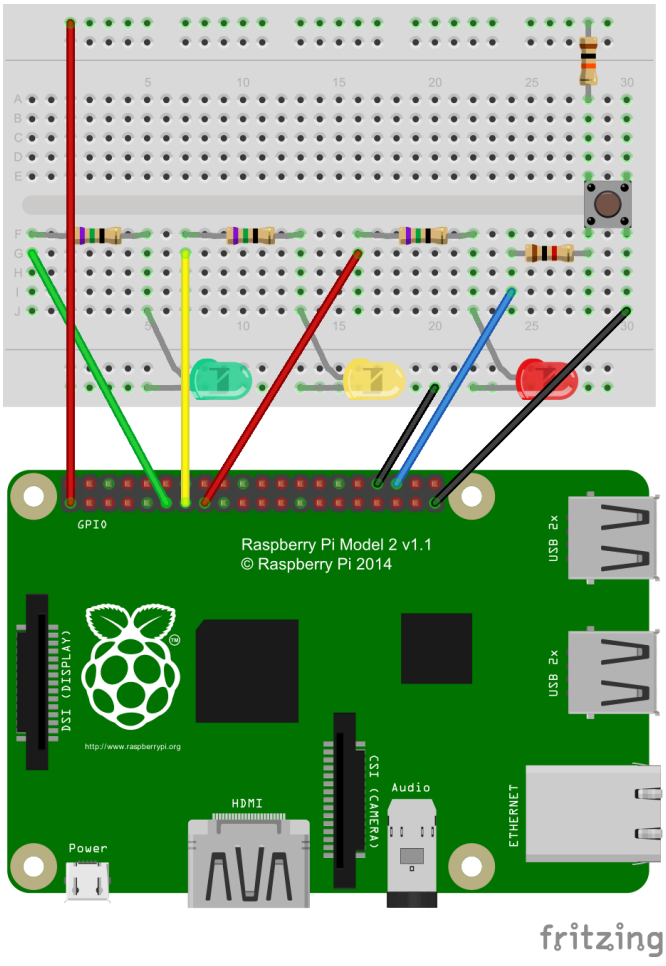
The full code to make use of this function is as follows:-

```
from gpiozero import LED
from time import sleep

green = LED(17)
yellow = LED(27)
red = LED(22)

def switchLights (greenLight, yellowLight, redLight, sleepTime):
    if greenLight:
        green.on()
    else:
        green.off()

    if yellowLight:
        yellow.on()
    else:
        yellow.off()

    if redLight:
        red.on()
    else:
        red.off()

    sleep(sleepTime)

while True:
    switchLights (True, False, False, 10)
    switchLights (False, True, False, 1)
    switchLights (False, False, True, 10)
    switchLights (False, True, True, 1)
```

Project 4 – Controlling our LEDs with a Switch (add a switch into our traffic light circuit and use it to trigger the light sequence)

Introduction

So, we have our traffic lights working, so let's move on again. Again, we're going to build upon our previous project, so all the LEDs, resistors, and jumper wires that you've already got in place can stay where they are for now. For this project all we're going to do is to add a simple push button so that we can add a little more control over how our traffic lights work. However, when we add a push button to a circuit for the Raspberry Pi, we also need to add a couple of other components. As well as the push button, you will also need 2 more jumper wires, the 1k resistor and the 10k resistor. The 1k resistor has a brown band, a black band and a red band. The 10k resistor has a brown band, a black band and an orange band. Connect these new components as shown in the diagram below.



Before looking at the code for this project, let's try to understand the circuit connections a little first; for example, why do we need the resistors? You'll notice that one of the leads of the 10k resistor is connected to one of the +3.3v GPIO pins on the Raspberry Pi through the +ve strip running along the length of the breadboard and the red jumper wire. The other lead of the resistor is connected to one of the pins of the push button. Because of the way the push button is constructed, this pin is actually connected internally to the pin directly opposite it (the pin connected to one of the leads of the 1k resistor), and thus to the GPIO pin 16 (physical pin 36) by the 1k resistor and the blue jumper wire.

We are going to use GPIO pin 16 as an input to determine whether or not the push button has been pressed. Because the 10k resistor is connected to the +3.3v supply, it has the effect of 'pulling up' the voltage on this pin of the push button to +3.3v, so that when the push button is not pressed, this pin will always register a voltage of +3.3v, so the input value at GPIO pin 16 will also be +3.3v. If we didn't have this 'pull up' resistor, the value of the voltage at this GPIO pin 16 would have a floating value of anything between 0 and +3.3v; so we couldn't actually tell whether or not the button is pressed. When the button is pressed, the voltage at GPIO pin 16 will be pulled down to 0v, because the other pole of the push button is connected directly to one of the GPIO ground pins (physical pin 39) through the black jumper wire. That (hopefully) explains the 10k resistor; but what about the 1k resistor?

The 1k resistor is there to protect us from damaging our Raspberry Pi if we were to accidentally make a mistake in our code. For example, if we were to configure GPIO pin 16 as an output, and to set pin 16 to +3.3v in our code, and then pressed the push button, GPIO pin 16 (+3.3v) would become shorted straight through to the ground pin. Without the 1k resistor, this would cause damage to the Raspberry Pi because of the current flowing directly between these 2 pins. The 1k resistor would limit this current to a tolerable level were these 2 pins to become shorted together in this way.

The code needed to detect when the button is pressed is really simple, thanks to the GPIO zero library. The following lines of code show how to import the necessary class (Button) from the GPIO zero library, and how to define a button object.

```
from gpiozero import LED, Button

button = Button(16, pull_up = True)
```

The 'pull_up = True' parameter tells GPIO zero that when the button is not pressed pin16 will have a voltage of +3.3v, and when pressed it will have a voltage of 0v.

The full code for this project is as follows:-

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

green = LED(17)
yellow = LED(27)
red = LED(22)
button = Button(16, pull_up = True)

def switchLights (greenLight, yellowLight, redLight, sleepTime):
    if greenLight:
        green.on()
    else:
        green.off()

    if yellowLight:
        yellow.on()
    else:
        yellow.off()

    if redLight:
        red.on()
    else:
        red.off()
    sleep(sleepTime)

while True:
    switchLights (True, False, False, 10)
    button.wait_for_press()
    switchLights (False, True, False, 1)
    switchLights (False, False, True, 10)
    switchLights (False, True, True, 1)
```

You'll see from this code that the green LED will light for 10 seconds, following which the code will wait for the button to be pressed. When the button is pressed, the lights will immediately start to switch until they turn back to green again and then wait for a further button press.

As it stands, this is not very realistic for a set of button controlled traffic lights, so we'll look at modifying our code so that it waits for a random amount of time after the button is pressed before starting to switch the lights. In order to do this, we'll need to import another Python library and generate a random wait time; as follows:-

```
import random

wait = random.randint(10,15)
```

This code imports the Python library that allows us to generate a random number, and then generates a random wait time of between 10 and 15 seconds.

The full code for our more realistic set of traffic lights is shown below:-

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

green = LED(17)
yellow = LED(27)
red = LED(22)
button = Button(16, pull_up = True)

def switchLights (greenLight, yellowLight, redLight, sleepTime):
    if greenLight:
        green.on()
    else:
        green.off()

    if yellowLight:
        yellow.on()
    else:
        yellow.off()

    if redLight:
        red.on()
    else:
        red.off()
    sleep(sleepTime)

while True:
    switchLights (True, False, False, 1)
    button.wait_for_press()
    wait = random.randint(10, 15)
    print ("Wait for %d seconds ..." % wait)
    switchLights (True, False, False, wait)
    switchLights (False, True, False, 1)
    switchLights (False, False, True, 10)
    switchLights (False, True, True, 1)
```

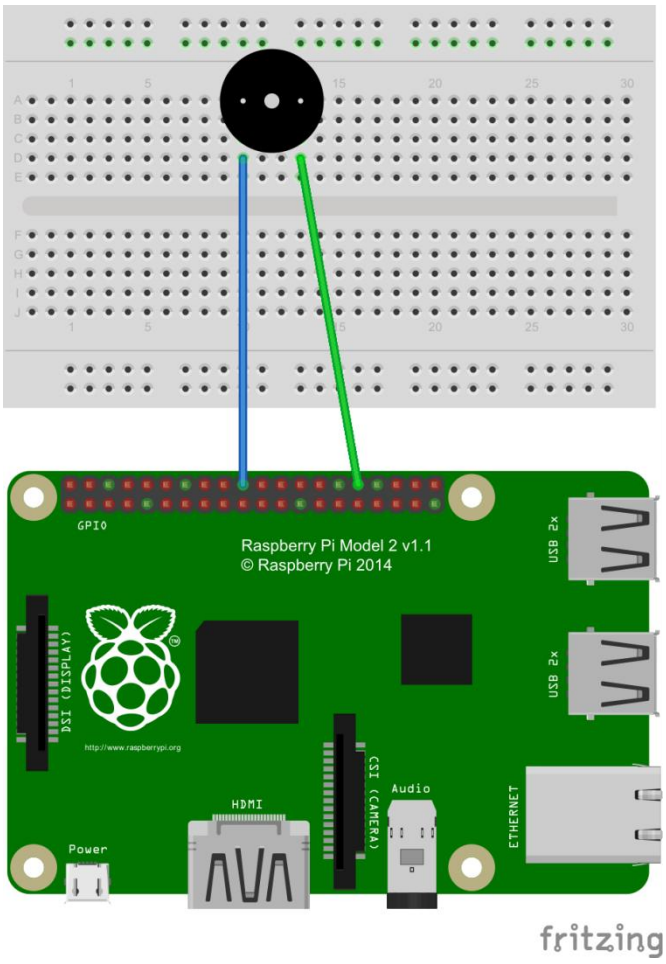
Project 5 – Basic buzzer circuit

Introduction

For our next project we're going to have a go at making some noise. If you look in your little bag of components, you should find a small black round cylinder, approximately 15mm in diameter and 10mm in height, which has two short pins sticking out of the bottom of it. On top it may have a small label stuck to it, which you can remove to reveal a small hole. This item is a piezo buzzer.

Note that, although we're only using the buzzer for this particular project, you can leave all the components from the previous project where they are on the board, as we'll be using them for our next project anyway.

The circuit for this project requires the buzzer and 2 jumper wires only. One of the terminals of the buzzer (it doesn't matter which one, as the buzzer does not have polarity) is connected to ground, and the other terminal is connected to GPIO pin 12. The full circuit diagram for this project is shown in the following diagram. Be careful, when placing the buzzer on the breadboard, to ensure that the terminal pins line up correctly with the correct breadboard strip and jumper wires.

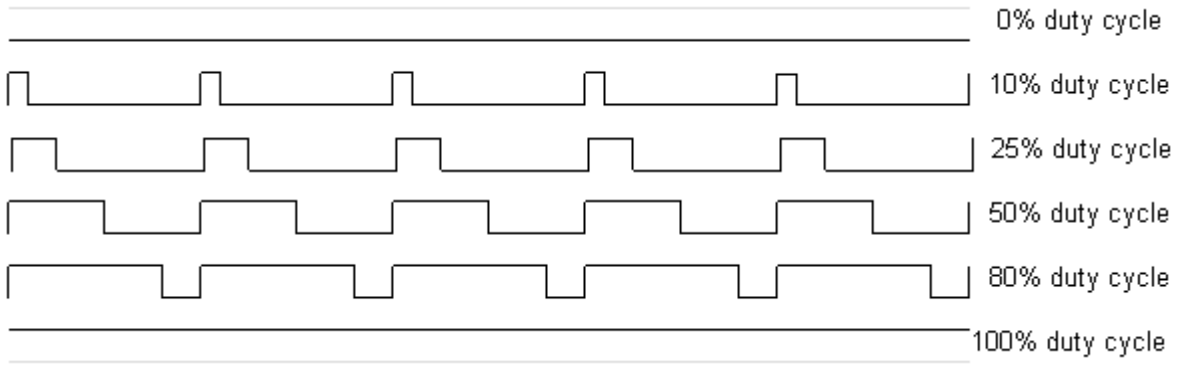


The code for our basic buzzer circuit is really simple, because again we can make use of the GPIO zero Python library. Our code, shown below, will create a buzzer object as a PWMOutputDevice. It then sets the frequency of the output signal to 5000 Hertz (or cycles per second), and gives the buzzer a value of 0.0. This value seems somewhat arbitrary just now, but we'll explain this soon. The code also defines a function called 'buzz' which sets the buzzer to a specified frequency and a value of 0.5 (again, we'll explain this in a moment) for a period of time before setting the value back to 0.0. The main piece of code calls the 'buzz' function with a frequency of 5,000 Hertz for a period of 1 second, and then sleeps for 1 second. Try running it and see what happens. You should hear the buzzer buzz for 1 second, and then stop for 1 second; and it should do this continuously until you halt the program.

```
from gpiozero import PWMOutputDevice
from time import sleep

BUZZERPIN = 12

buzzer = PWMOutputDevice(BUZZERPIN)
buzzer.frequency = 5000
buzzer.value = 0.0

def buzz (frequency, period):
    buzzer.frequency = frequency
    buzzer.value = 0.5
    sleep(period)
    buzzer.value = 0.0

while True:
    buzz(5000, 1)
    sleep(1)
```

The buzzer is set up as a PWMOutputDevice, where PWM stands for Pulse Width Modulation. This is a good way of controlling the buzzer, since the piezo buzzer we are using is not like a normal mechanical buzzer which works by having a positive DC (Direct Current) voltage applied to it. A piezo buzzer needs to have an AC (Alternating Current) signal applied to it in order to make it work. The ideal signal for the piezo buzzer would be something like a sine wave, but because the Raspberry Pi can only output digital signals, it can't generate a sine wave signal. The closest we can get to this is a square wave signal; but this is fine for our purposes.

A good way of generating a square wave signal from the Pi is to use a PWM signal. This is a square wave signal where we control the frequency of the signal, and also something called the duty cycle. The duty cycle is expressed as a percentage, and put simply it defines the ratio of time that the output is on to the time that the output is off. So, for example, a 50% duty cycle means that the output is on for half of the time and off for the other half of the time. The diagram below shows this concept.



So, as you can see, a duty cycle of 100% would mean that the output is always on, and a duty cycle of 0% would mean that the output is always off. For the piezo buzzer, a duty cycle of 0% or 100% is of no use, since this means a fixed voltage of either 0v or +3.3v, so we need to use a duty cycle value of something between 0% and 100%, and we'll have a look at that in a moment after looking at some other aspects of PWM signals.

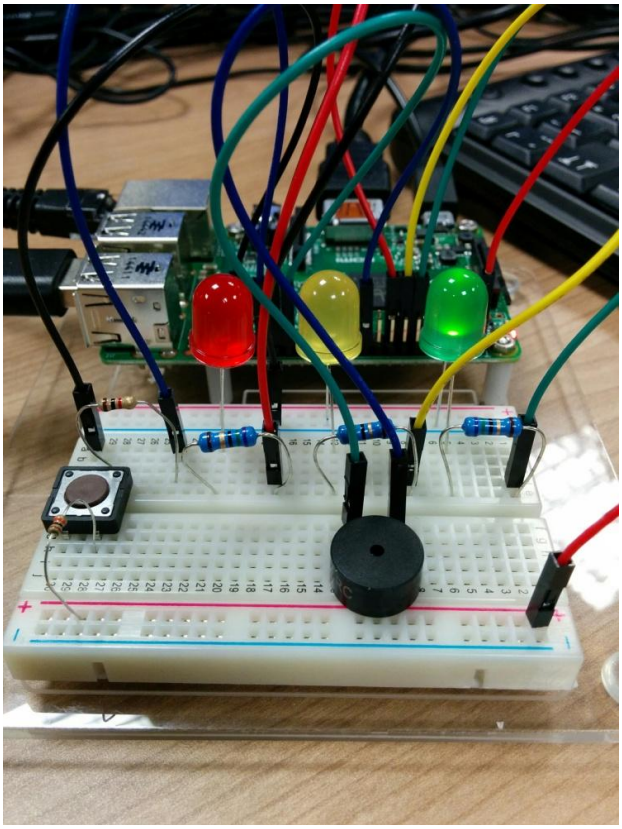
PWM signals are also useful when driving devices such as LEDs or DC (Direct Current) electric motors, since by increasing the duty cycle of the signal we are effectively increasing the mean voltage being supplied to the device. So, by increasing the duty cycle, we can thus make an electric motor run faster or an LED shine brighter. You might think therefore, that by modifying the duty cycle of the signal supplied to the piezo buzzer, you could alter the loudness of the buzzer; however, this is unfortunately not the case. You will see that, in our code, we use a fixed duty cycle of 50% (or 0.5), since this is the closest that we can get to an ideal square wave.

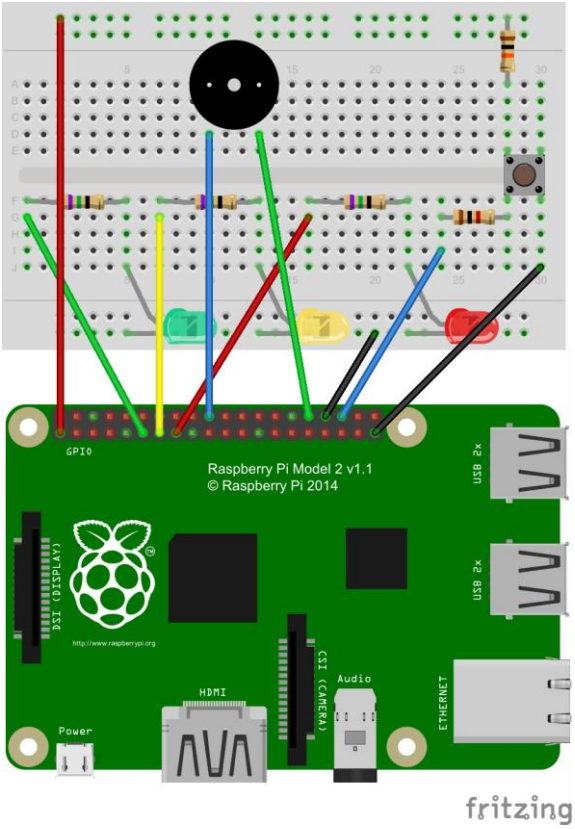
Although we aren't able to alter the loudness of the buzzer, you might want to try modifying your buzzer circuit to use an LED and resistor in place of the buzzer to see how altering the duty cycle of the signal supplied to the LED allows you to adjust the brightness of the LED. Be careful to ensure that the cathode (negative terminal) of the LED is connected to the ground GPIO pin.

Project 6 – Traffic Lights with Switch and Buzzer

Introduction

For out next project we're going to put everything together that we've learned so far to combine our LEDs, push button and buzzer to make a set of traffic lights that gives an audible indication of when the lights are at red. The circuit diagram below shows how the various components need to be wired up, and hopefully you'll still have this set up from our previous projects.





The code for this project is as shown below:

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

green = LED(17)
yellow = LED(27)
red = LED(22)
button = Button(16, pull_up = True)
buzzer = PWMOutputDevice(12)
buzzer.frequency = 500
buzzer.value = 0.0

def buzz (frequency, period):
    buzzer.frequency = frequency
    buzzer.value = 0.5
    sleep(period/2)
    buzzer.value = 0.0
    sleep(period/2)

def switchLights (greenLight, yellowLight, redLight, sleepTime):
    if greenLight:
        green.on()
    else:
        green.off()

    if yellowLight:
        yellow.on()
    else:
        yellow.off()

    if redLight:
        red.on()
    else:
        red.off()
    sleep(sleepTime)

while True:
    switchLights (True, False, False, 1)
    button.wait_for_press()
    wait = random.randint(10, 15)
    print ("Wait for %d seconds ..." % wait)
    switchLights (True, False, False, wait)
    switchLights (False, True, False, 1)
    switchLights (False, False, True, 0)
    for i in range(0, 10):
        buzz(500, 1)
    switchLights (False, True, True, 1)
```

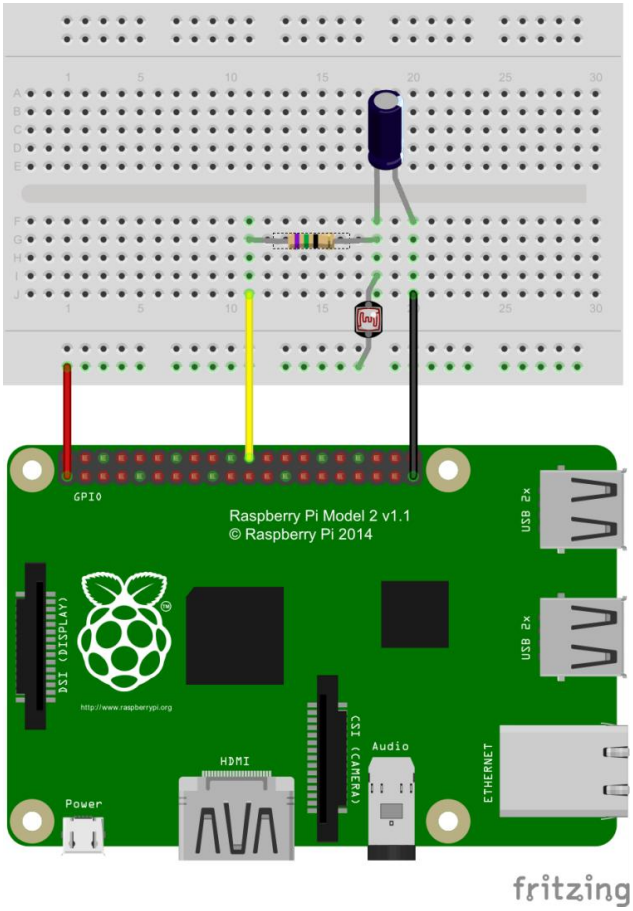
Project 7 – Measuring Light Levels with the Light Dependent Resistor (LDR)

Introduction

A light dependent resistor (or LDR for short) is effectively a variable resistor that reacts to the level of light that it sees. So, for example, in a dark room the LDR would have quite a high resistance; possibly in the order of a MegaOhm or more. When light is shone upon the LDR, however, the resistance decreases, and, depending on how bright the light is, the LDR resistance could be as low as a few hundred Ohms. We can therefore use this variable resistance property of the LDR to allow us to do a crude measurement of light levels.

Because the Raspberry Pi does not have any analogue inputs, it's not possible to measure the resistance of an LDR directly. However, with the addition of a couple of extra electronic components (specifically a 75R resistor and a capacitor), we can construct a circuit which allows us to do a rough measurement of the resistance of the LDR.

Because we are using an electrolytic capacitor, we do need to be careful about which way round we connect the capacitor in the circuit, since electrolytic capacitors have polarity. You will notice that there is a white stripe down the side of the capacitor, and this stripe indicates the negative terminal of the capacitor. We will be connecting this terminal of the capacitor to one of the ground pins of the Raspberry Pi via a jumper wire. It's good practice to us a black jumper wire for this connection, as this makes it obvious that this is the ground. The other terminal of the capacitor is connected to +3.3v via the LDR, so the voltage applied to the positive terminal of the capacitor will always have a value greater than or equal to 0v (or ground voltage).



You'll have noticed that there is also another resistor (75R) connected to the positive terminal of the capacitor, and that this resistor is connected to one of the GPIO pins of the Raspberry Pi. This allows us to both switch the voltage to the positive terminal of the capacitor (using the GPIO pin as an output), and also to measure the voltage at the positive terminal of the capacitor (by using the GPIO pin as an input).

The measurement of the LDR resistance is done by initially setting one of the GPIO pins (in this case, pin GPIO 25) to be an output pin, and then setting that pin to have a low voltage. This will cause the capacitor to completely discharge. Having done this, the same GPIO pin is then set as an input pin, at which point the capacitor will begin to charge again through the LDR, because one of the LDR pins is connected to one of the 3.3v GPIO pins. The higher the resistance of the LDR, the longer it will take to charge the capacitor; so measuring the time that it takes to charge the capacitor will give us a rough estimate of the resistance of the LDR. So, the less light that is shining on the LDR, the longer it will take to charge the capacitor.

By continually repeating this process of charging and discharging the capacitor, and measuring the time that the capacitor takes to charge each time, we will thus get a continuous measurement of the light level. The code that allows us to do this is shown below.

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

SENSORPIN = 25

GPIO.setmode(GPIO.BCM)

def getSensorTime (pin):
    # Set the GPIO pin to an output, and set the output to LOW
(0.0v)
    GPIO.setup(pin, GPIO.OUT)
    GPIO.output(pin, GPIO.LOW)
    time.sleep(0.1)

    # Set the GPIO pin to an input, and measure the time taken to go
    # from LOW to HIGH
    GPIO.setup(pin, GPIO.IN)
    t1 = datetime.datetime.now()
    while (GPIO.input(pin) == GPIO.LOW):
        pass
    t2 = datetime.datetime.now()
    delta = t2 - t1
    milliseconds = delta.total_seconds() * 1000
    return milliseconds

try:
    while True:
        sensorTime = getSensorTime (SENSORPIN)
        print sensorTime
        time.sleep(0.5)
finally:
    print "Cleaning up GPIO"
    GPIO.cleanup()
```

You will have noticed that we are now using a different Python library to control the GPIO of the Raspberry Pi. This is because we are now doing things with the GPIO pins that are not easily achieved by using the GPIO Zero library directly. It is useful to be able to use the GPIO Python library as well as the GPIO Zero library, since although the GPIO Zero library makes it a lot easier to do certain things, the GPIO library does actually give us a little more control when we want to do something a little more complex; such as in this project.

You'll notice also that we've defined a value which we call SENSORPIN, which we've given a value of 25. This type of code declaration is called a constant, as it defines a value which always remains the same in our program. This is a useful technique to use, as if we decide later that we want to use a different GPIO pin for our LDR we only need to change the pin number once within our code, instead of having to change it in every line of code that refers to this pin.

The number that is output by the program is the approximate time taken (in milliseconds) to charge the capacitor, and the less light that is shining on the LDR, the longer it will take to charge the capacitor. Thus, the darker the room, the higher the number will be. If we wanted to output a number that increased as the light level increased, we could do this by simply changing the line of code that prints out the sensorTime to something like:-

```
print 1000/sensorTime
```

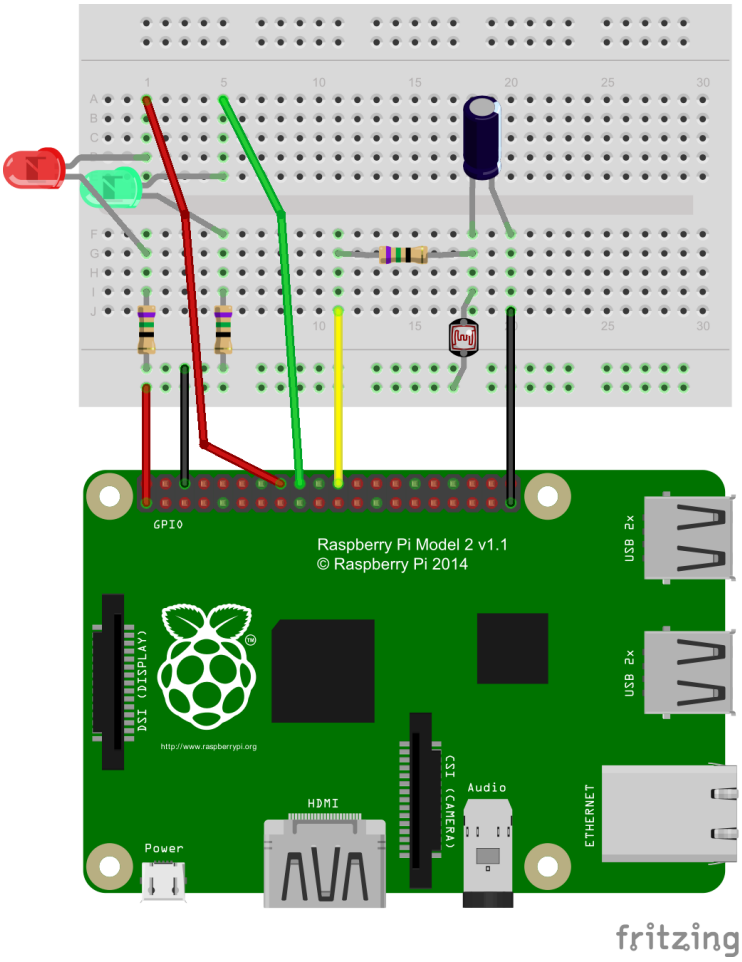
Project 8 – Indicating Light Levels with the Light Dependent Resistor (LDR) and LEDs

Introduction

Having seen how we can use the LDR, let's try extending the circuit to include a couple of LEDs that we can switch on or off depending on the light level. We already know how to use LEDs, so this will be fairly straightforward.

For this circuit, we'll use a red and green LED. Each LED will have its own resistor to limit the current flowing through the LED. As with our previous projects, we'll use a 75R resistor with each of the LEDs. The full circuit for this project is shown below.

For this circuit, we've connected the resistors to a common ground, and the cathodes (negative pins) of the LEDs to the other end of the resistors. Remember that the cathodes of the LEDs are the shorter of the 2 pins. The anodes (the longer pins) of the LEDs are connected to GPIO pins 23 and 24 by means of jumper wires. We've used the same colour wires as the LEDs, just to make the circuit easier to follow.



The code for this project is shown below. You'll notice that much of the code is very similar to the code for the previous project. This time, however, we're making use of both the GPIO and GPIO Zero Python libraries. As mentioned before, the GPIO Zero allows us to control things like LEDs really

easily, but when it comes to some of the more complex GPIO code we need to make use of the GPIO library.

```
import RPi.GPIO as GPIO
import time
import datetime
from gpiozero import LED

SENSORPIN = 25
GREENLED = 24
REDLED = 23

GPIO.setmode(GPIO.BCM)

def getSensorTime (pin):
    # Set the GPIO pin to an output, and set the output LOW (0.0v)
    GPIO.setup(pin, GPIO.OUT)
    GPIO.output(pin, GPIO.LOW)
    time.sleep(0.1)

    # Set the GPIO pin to an input, and measure the time taken to go
    # from LOW to HIGH
    GPIO.setup(pin, GPIO.IN)
    t1 = datetime.datetime.now()
    while (GPIO.input(pin) == GPIO.LOW):
        pass
    t2 = datetime.datetime.now()
    delta = t2 - t1
    milliseconds = delta.total_seconds() * 1000
    return milliseconds


def indicate (value):
    redLED.off()
    greenLED.off()

    if value < 100:
        greenLED.on()
    else:
        redLED.on()

try:
    redLED = LED(REDLED)
    greenLED = LED(GREENLED)
    redLED.off()
    greenLED.off()
    while True:
        sensorTime = getSensorTime (SENSORPIN)
        print sensorTime
        indicate (sensorTime)
        time.sleep(0.5)
finally:
    print "Cleaning up GPIO"
    GPIO.cleanup()
```

As you can see, we've defined a couple of constants to define the GPIO pins for the red and green LEDs. Again, this makes it easier to change the code later if we decide that we need to connect the LEDs to different GPIO pins.

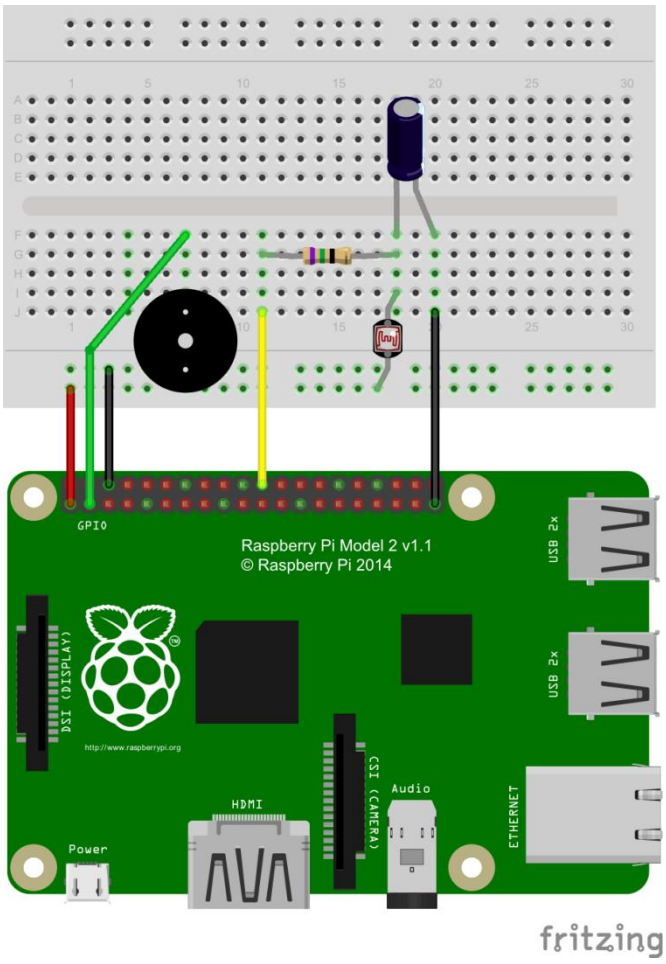
The program initially switches both LEDs off, and then switches the LEDs on or off depending on the light level detected by the LDR. We've got our program to switch the green LED on if the sensor time is less than 100ms; otherwise the red LED is switched on. You could change this value to be whatever you wish; and (as we've done with our GPIO pin numbers) you could even define your own constant to define this value.

Project 9 –Light Dependent Resistor (LDR) Controlled Noise Maker

Introduction

Our next project also makes use of the LDR. This time however, instead of controlling LEDs, we're going to have a go at controlling the buzzer. It's a sort of light controlled musical(?) instrument.

The circuit for this project makes use of the same components as for project 8, but we also need to wire in the buzzer. One of the terminals of the buzzer is connected to ground, and the other terminal is connected to GPIO pin 2. The full circuit diagram for this project is shown in the following diagram. Be careful again when placing the buzzer on the breadboard, to ensure that the terminal pins line up correctly with the correct rail and jumper wires. On our diagram you'll notice that the pin for the ground connection lines up with the rail that runs along the length of the breadboard, and this rail is also connected to one of the GPIO ground pins. We've used a black jumper wire for this, just to make things easier to follow. The green jumper wire connects the other terminal of the buzzer to GPIO pin 2.



The code for this project is as shown below.

```
import RPi.GPIO as GPIO
import time
import datetime
from gpiozero import PWMOutputDevice

SENSORPIN = 25
BUZZERPIN = 2

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

pwmBuzzer = PWMOutputDevice(BUZZERPIN)
pwmBuzzer.frequency = 5000
# Initially set duty cycle to 0 – buzzer off
pwmBuzzer.value = 0.0

def buzz(frequency):
    pwmBuzzer.frequency = frequency
    # Set duty cycle to 50% (0.5)
    pwmBuzzer.value = 0.5

def getSensorTime (pin):
    GPIO.setup(pin, GPIO.OUT)
    GPIO.output(pin, GPIO.LOW)
    time.sleep(0.1)

    GPIO.setup(pin, GPIO.IN)
    t1 = datetime.datetime.now()
    while (GPIO.input(pin) == GPIO.LOW):
        pass
    t2 = datetime.datetime.now()
    delta = t2 - t1
    microseconds = delta.total_seconds() * 1000000
    return microseconds

try:
    while True:
        sensorTime = getSensorTime (SENSORPIN)
        print sensorTime
        buzz(sensorTime/100)
finally:
    print "Cleaning up GPIO"
    GPIO.cleanup()
```

You'll see that we're again using both the GPIO and GPIO Zero Python libraries for this project.

Although, as we discussed previously, we cannot alter the loudness of the piezo buzzer, we are able to alter the frequency of vibration of the buzzer by modifying the frequency of the PWM signal. Our code modifies the PWM frequency as the light level changes. To do this, the code samples the light level, and then calls the 'buzz' function to set the PWM signal frequency. In our code, we set the frequency to be equal to the sensor time divided by 100; but you could experiment with different values to see what different sounds you can get from the buzzer.

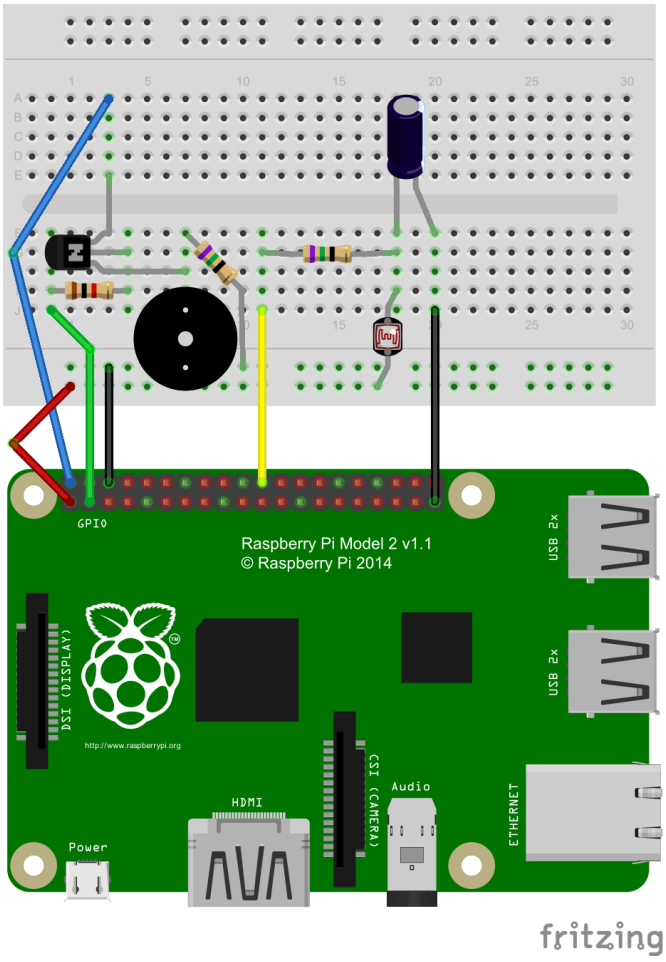
Project 10 - Light Dependent Resistor (LDR) Controlled Noise Maker – alternative version

Introduction

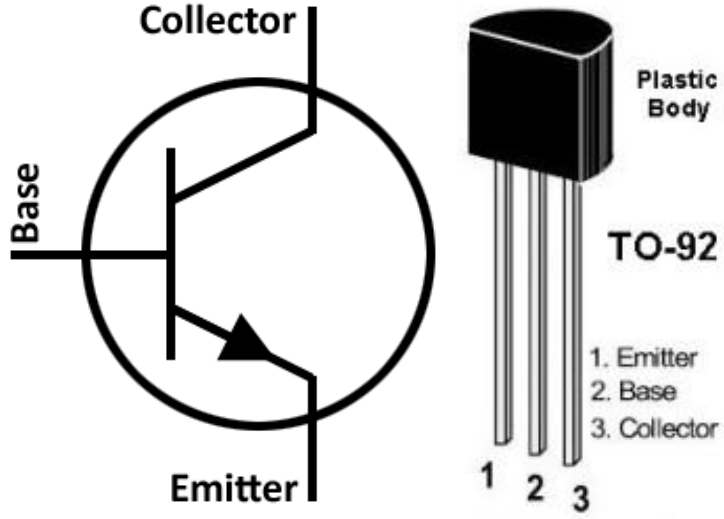
We said in project 9 that it is not possible to control the loudness of our piezo buzzer by means of altering the signal supplied to the buzzer. However, although it may not appear obvious, one thing we can do is to amplify the signal that we use to control the buzzer.

The buzzer is designed to have a minimum supply voltage of 3v, however, it will also work at higher voltages; and by increasing the voltage supplied to the buzzer we can nominally increase the loudness of the buzzer.

The Raspberry Pi GPIO outputs that can be switched on or off can only supply an output voltage of either 0v (off) or +3.3v (on), so you might think that it's not possible to supply a higher switched output voltage. However, we have included a component in our bag of bits that will allow you to do just that. The component we are referring to is the transistor, which is effectively a switching device. By connecting the transistor into the buzzer circuit, and connecting the transistor to one of the +5v fixed voltage pins, we can effectively increase the level of the signal voltage applied to the piezo buzzer from +3.3v to +5v. The circuit diagram below shows how the transistor should be wired into the buzzer circuit.



The transistor that is supplied in the kit is an NPN transistor. The standard symbol for an NPN transistor is shown below. The diagram next to it shows which of the transistor leads is which.



The transistor, as already mentioned, is a switching device that can be switched on or off by applying a small voltage to one of its terminals. For the NPN transistor that we have supplied, the Collector terminal is connected to the positive supply voltage +5v GPIO pin. The Emitter terminal is connected to one of the buzzer terminals, and the other buzzer terminal is connected to one of the Pi GPIO ground pins. When the voltage applied to the Base terminal of the transistor is set to 0v, the transistor is effectively switched off, so the voltage applied to the buzzer will be 0v. However, when the voltage applied to the Base terminal is increased, the transistor switches on, and causes the voltage applied to the buzzer to increase to the supply voltage applied to the Collector terminal; in this case +5v. So, now when we apply our PWM signal to the Base terminal of the transistor, the amplitude of the signal applied to the buzzer is effectively amplified from +3.3v to +5v. You'll notice that the Base terminal of the transistor is not connected directly to the GPIO pin, but that we have connected a 1k resistor between the GPIO pin and the Base terminal. This resistor will limit the amount of current that is drawn from the GPIO pin in order to prevent any damage to the Raspberry Pi should the transistor fail for any reason. We have also connected a 75R resistor in parallel with the buzzer to limit the current supplied by the transistor to the buzzer.

The code for this project is exactly the same as that for the previous project. The real difference in this project is the fact that rather than the Raspberry Pi sending the control signal directly to the buzzer, the control signal is now being used to control the transistor, which in turn controls a higher voltage signal to the buzzer.