Macintosh HD:Users:ecstv:Documents:Matt Gussin:Media peices/logos/patterns:Logos:University Logos:university_southampton_black.epsMacintosh HD:Users:ecstv:Documents:Matt Gussin:DOCUMENTS:Open Day Posters and banners etc:2016:5275_UoS_OpenDayTimetable_PSE Folder:Links:black.epsLAB

Hardware, I/O and Interrupts

This practical lab session will teach you how to use the input and output on the Raspberry Pi, how to use a Breadboard, and the differences between interrupt and polling based software. You will use Python code to program the Pi but you are not expected to be a Python expert. Some simple electronics techniques are involved but these will be fully explained in this handout. Please make sure you read all these instructions first.

Warnings about the electronics are shown in red boxes

Details of the exercises you need to complete are shown in green boxes. In order to complete the exercises you will need to read and understand the other text on the page.

At the end of each section please show a demonstrator your work.

Code blocks are shown in grey boxes

At the end of the lab, please make sure that you:

• Shutdown the Pi

• Remove the jump wires, GPIO breakout, LEDs & Switch from the breadboard and return all parts to the box

• Leave the desk in a clean and tidy state.

# Getting Started: Connecting the Raspberry Pi

* Plug in the micro-SD card into the Raspberry Pi. Make sure it is firmly inserted – it is possible to accidentally knock it out.
* Plug the network cable in.
* Plug in your keyboard and mouse.
* Plug the HDMI lead.
* Plug the power supply into the mains socket. **Do not** plug the power supply into the Raspberry Pi yet.
* Make sure the work area is clear of conductive mess.
* Finally plug the micro USB power connection into your Pi. You will see lights on the board and the boot sequence displayed on the monitor as it boots.

Once the Pi has booted you will be presented with the interface shown in Figure 1.



Figure 1: Raspberry Pi GUI (your desktop picture might vary)

Make sure to shut down the Raspberry Pi any time you want to power it off. Removing the power without shutting down can corrupt the SD card. To shut down the Pi either select “Shutdown…” from the Raspberry Pi menu at the top-left of the screen or, in a terminal window, use:

sudo poweroff

Connect & power on the Raspberry Pi as described at the start of this section. Open a terminal window by selecting the terminal icon at the top of the screen and enter the following command:

htop

The htop command shows what processes the Pi is running. The horizontal bar charts next to the numbers 1-4 at the top of the htop output show the percentage utilisation of the Pi’s four CPU cores. Later in the lab we’ll be using htop to investigate how much the CPU is being used by the code you write.

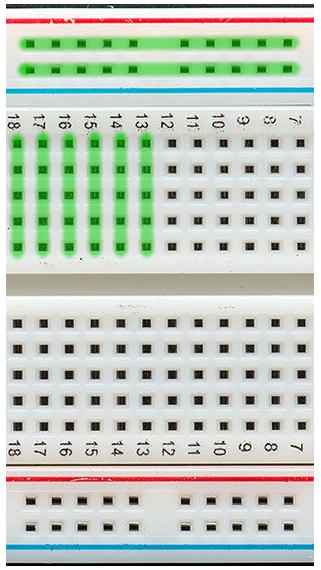
During this lab you will need to create and edit files on the Raspberry Pi for each experiment. There are several ways that you can do this with the easiest being from the GUI of the Pi.

You can edit or create files using Geany or IDLE on the Pi by clicking on the Raspberry Pi menu icon at the top-left of the screen and navigating to the “Programming” menu. Note: you will still need a terminal to execute the code you write.

# Using the Breadboard

A breadboard allows for prototyping of electronic circuits without having to do any soldering. Connections are made by inserting component leads or jump wires through the holes on the top of the breadboard into the internal connection rails.

Figure : Breadboard internal connections



The internal rails are connected as shown by the green lines in Figure 3. The long rails along each edge are usually used for power and ground connections.

The power rails are not always contiguous along the length of the breadboard – if there is a break in the red and blue lines on your breadboard, then there is a break in the internal connections.

The numbered rows are split into two separate strips with five connection holes for each strip. The rows are not contiguous across the cut-out on the middle of the breadboard.

When making connections, do not try to insert more than one wire or component lead into a hole – doing so will cause damage to the breadboard.

Shorting pins on the Raspberry Pi while it is on can cause a lock-up or even damage! Avoid this by powering off Pi before carrying out wiring by selecting “Shutdown…” from the Raspberry Pi menu or opening a terminal and running:

sudo poweroff

and then removing the power cable once the Pi has shut down.

The “T-Cobbler” GPIO breakout connects the breadboard to a Raspberry Pi with a 40-pin ribbon cable that needs to be connected the correct way round as shown in Figure 4. The T-Cobbler should be connected to the end of the breadboard such that the pins nearest the top of the “T” are in the first numbered row, and the pins straddle the centre of the breadboard as shown in Figure 7.

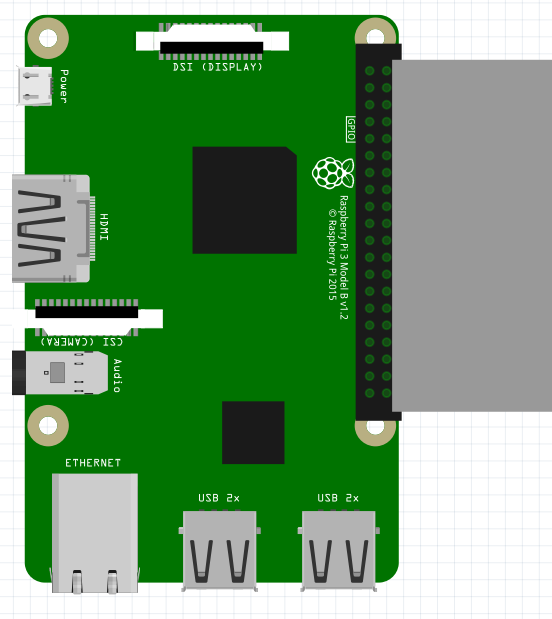


Figure : Raspberry Pi ribbon cable connection.

Shutdown your Raspberry Pi, disconnect the power and then connect the 40-pin ribbon cable to the Raspberry Pi’s expansion header, and the T-Cobbler GPIO breakout to the breadboard.

Connecting the ribbon cable the wrong way round will stop the Pi from booting and can cause damage. Check that the ribbon cable is the correct way around and not offset before turning the Pi back on.

# Using the GPIO: LED Control

In this experiment you will be using three GPIO (General Purpose I/O) pins to control three different LEDs. LEDs are semiconductor devices that emit light when powered. They are polarised so need to be connected the correct way around – the short leg is ground as shown in Figure 5.

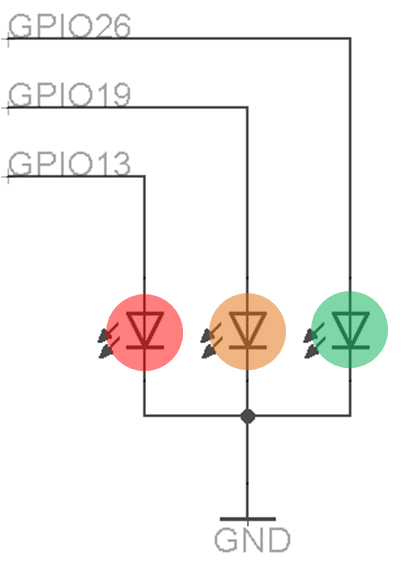


Figure : LED pins

LEDs will only work if they are connected the correct way around.

Figure 6 shows a schematic of how the LEDs will be connected to the breakout board during this exercise and Figure 7 shows how you should lay it out on the breadboard.

Figure : Schematic of LED connections



DO NOT TRY TO WORK OUT THE CONNECTIONS FROM THE PICTURES ALONE. USE THE PRINTED LEGEND ON THE BREAKOUT AND THE REFERENCE IN APPENDIX 1.

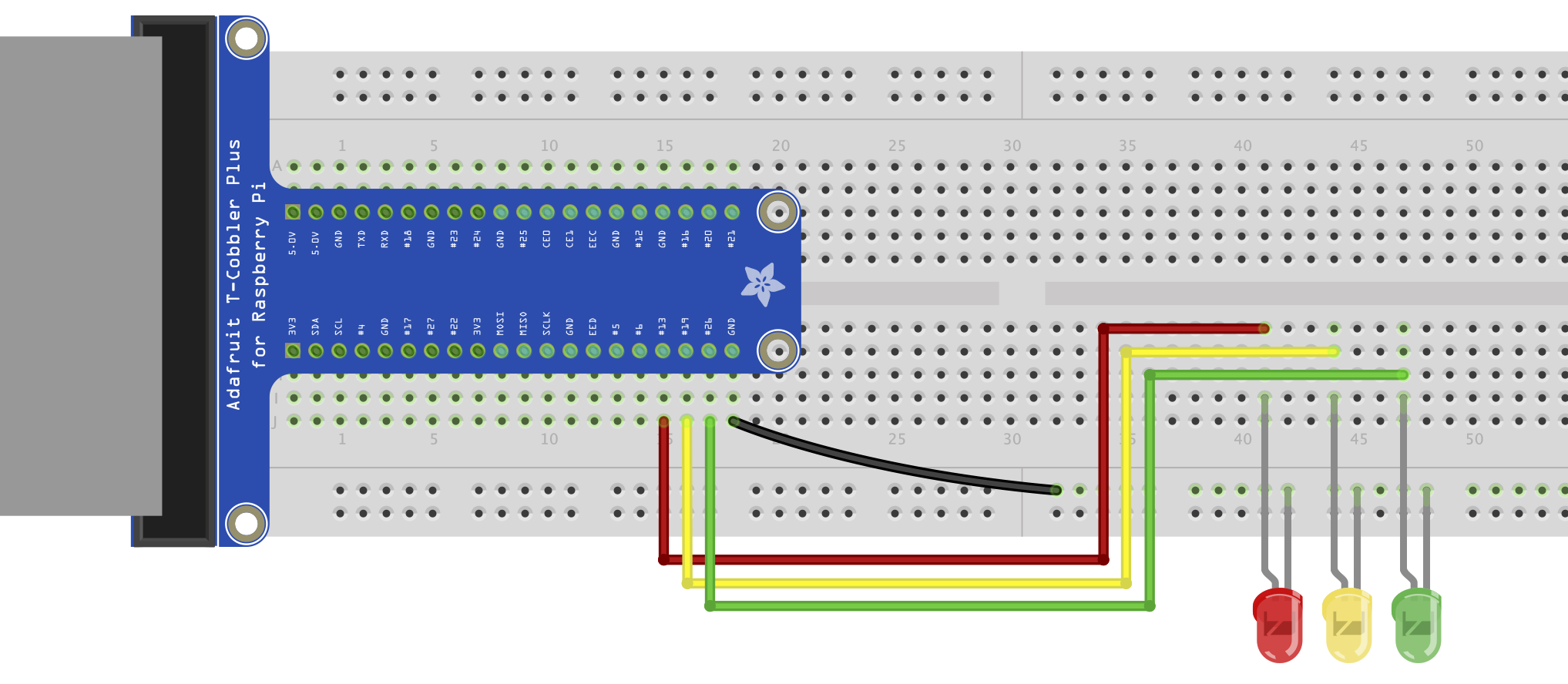


Figure 7: Breadboard layout for the LED exercise.

The LEDs you have been given are specialised 3.3V ones that already have a resistor inside. With normal LEDs you would need a resistor in series.

Plug the LEDs into the breadboard and make the connections as shown above. Check that your wiring is correct with the pin out diagram in Appendix 1.

Once you have your LEDs connected, plug the power back in. Once the Pi has booted open an editor (e.g. Geany or IDLE) to create a new Python program with the following code (note that you don’t need to type the comments [words after the # symbol] if you don’t want to) and save it as ‘leds.py’ in the home directory (/home/pi):

from time import sleep

import RPi.GPIO as GPIO #load the GPIO library

GPIO.setmode(GPIO.BCM) #ensure the pin numbering matches the cobbler

RED = 13 #Red led is connected to pin 13

YELLOW = 19 #Yellow to 19

GREEN = 26 #Green to 26

LEDS=[RED, YELLOW, GREEN] #create an array of all pins

ONTIME = 0.3

GPIO.setup(LEDS, GPIO.OUT) #Set all the led pins to be outputs

try:

while True:

for led in LEDS:

GPIO.output(led, GPIO.HIGH) #Setting high turns on

sleep(ONTIME) #Wait for some time

GPIO.output(led, GPIO.LOW) #Setting low turns off

finally:

GPIO.cleanup() #Release all resources

Note that in Python code it is important that you get the code indentation correct and don’t mix spaces and tabs.

leds.py can be run by typing the following in a terminal:

python leds.py

To stop the program running and return to the command prompt press Control+C.

Execute leds.py to flash the LEDS. This should cycle through the LEDs fairly quickly. If one of the LEDs does not light stop your program and check your wiring.

Experiment with the code for 10 minutes. Try changing the “ONTIME” or the flashing pattern. What do you observe?

# Using the GPIO: Switch Input

In this experiment you will use a GPIO pin to read the status of a button and influence the output from the Pi.

Figure 9 shows the schematic of how the “press to make” switch will be connected. The output to the right connects to your INPUT GPIO pin. When the switch is open the GPIO is held at 0V so returns “0” when the switch is pressed the input is connected to 3.3V and the GPIO becomes “1”.

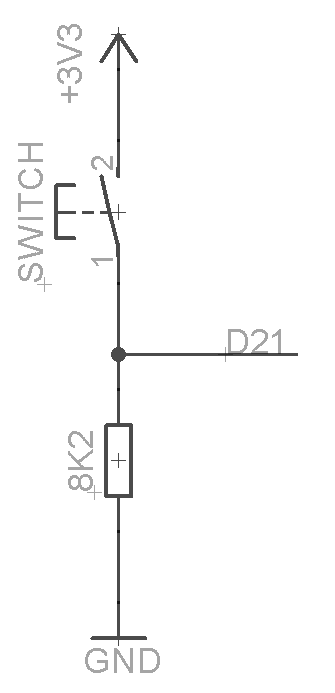


Figure : Schematic of connected switch and pull-down resistor

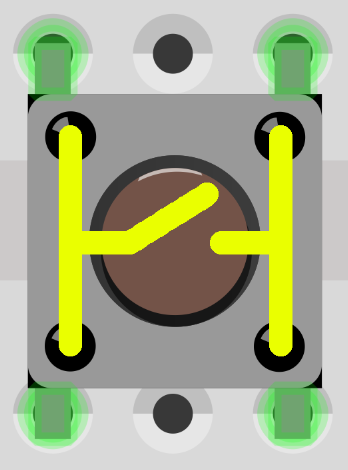


Figure 8: Internal switch connections

The switch you are using has 4 connections as shown in Figure 8 however the switch on the schematic only has two. This is because each pin in the schematic is connected to two pins in the physical package.

If your switch does not work make sure that you have the pins connected correctly.

Figure 10 shows the breadboard layout for this exercise – you should leave the LEDs from the previous exercise connected. The orientation of the switch is the same as in Figure 8.

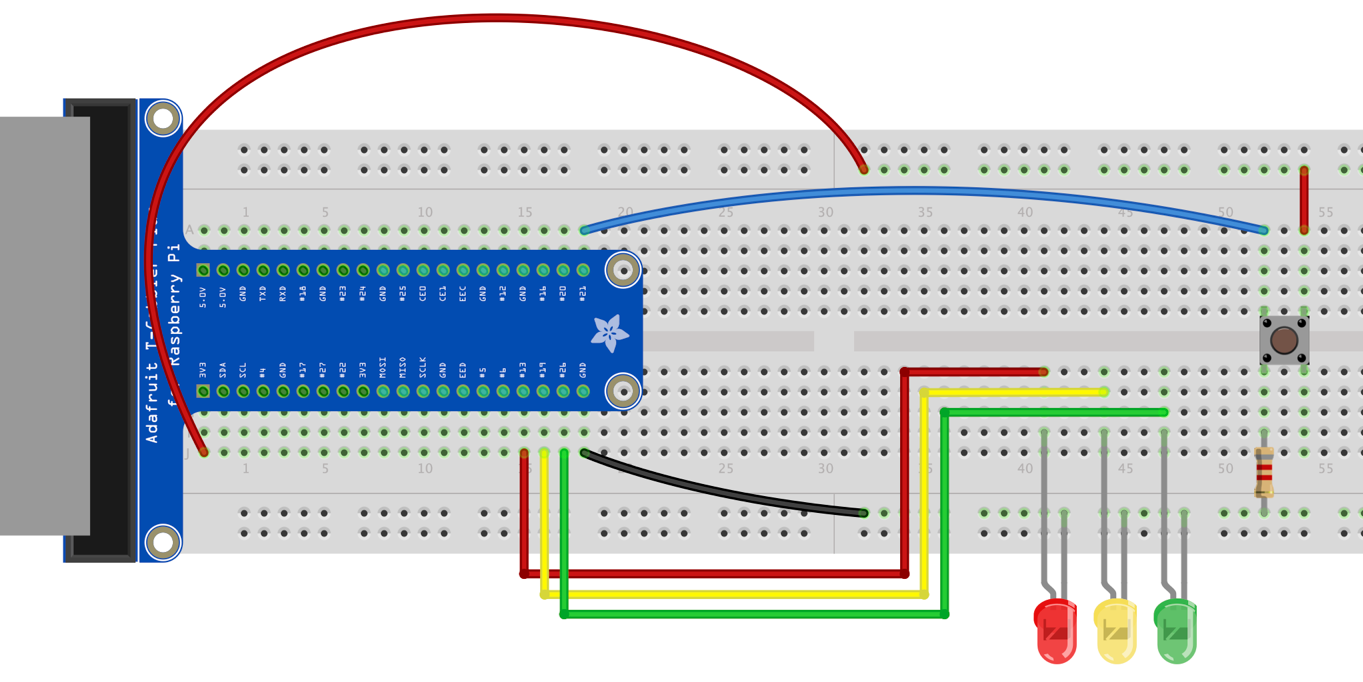


Figure : Breadboard layout for the Switch exercise.

You must use a resistor rather than a wire link – if you use a wire link, you will cause a “short” which will crash the Pi.

Shutdown the Raspberry Pi and connect the switch as shown. Once you have connected the switch, reconnect power to the Pi. Once the Pi has booted open an editor to create a new Python program with the following code and save it as ‘switch.py’ in the home directory:

import RPi.GPIO as GPIO #load the GPIO library

GPIO.setmode(GPIO.BCM) #ensure the pin numbering matches the cobbler

RED = 13 #Red led is connected to pin 13

YELLOW = 19 #Yellow to 19

GREEN = 26 #Green to 26

LEDS=[RED, YELLOW, GREEN] #create an array of all pins

SWITCH = 21 #switch is connected to pin 21

GPIO.setup(LEDS, GPIO.OUT) #Set all the led pins to be outputs

GPIO.setup(SWITCH, GPIO.IN) #Set the switch pin to be input

GPIO.output(GREEN, GPIO.HIGH) #Turn on green led

try:

while True:

state = GPIO.input(SWITCH) #Read the switch state

GPIO.output(RED, state) #Set to current state

GPIO.output(YELLOW, 1-state) #Set to not current state

finally:

GPIO.cleanup() #Release all resources

The code has an infinite while loop that runs and reads the state of the switch using GPIO.input(SWITCH). The result is stored in the “state” variable and it will contain either True (when the switch is pressed giving a “1”) or False (when the switch is not pressed, giving a “0”). A loop that continually reads the state of an input is said to be polling that input.

Execute switch.py and note what happens when the switch is pressed. Leave the switch.py file running and open a second terminal window. In that terminal run the htop command. What do you see?

Polling an input is quite an expensive operation as the htop output should illustrate. The reason for this is that the infinite while loop will consume all the processor cycles it can to run as fast as possible.

It is possible to reduce the CPU usage by polling less often by including a sleep call in the loop. The disadvantage of doing this is that you will introduce a lag between when the button is physically pressed and when the change in input is sensed and acted on.

Try adding a sleep into the code. What do you observe?

# Interrupt driven events

In this experiment, we’ll switch from using polling of the switch to an interrupt driven approach. An interrupt is a signal to the CPU that is emitted by hardware or software indicating an event that needs immediate attention. An interrupt alerts the processor to a high-priority condition requiring the interruption of the current code the processor is executing. The processor responds by suspending its current activity and executing a function called an interrupt handler, callback or interrupt service routine (ISR) to deal with the event. This interruption is temporary, and, after the interrupt handler finishes, the processor resumes the activity it was performing before the interruption.

Hardware interrupts, like those generated by a switch, are generated when the state of the hardware changes. When the switch in our circuit is pressed, the signal going to the switch GPIO pin goes from low (ground) to high (+3.3V). This transition is called a rising edge. Conversely, when the switch is released, the input to the GPIO pin drops from high to low; this transition is called a falling edge. The Raspberry Pi’s GPIO library lets us attach interrupt handlers to the detection of either or both rising and falling edges for a particular GPIO pin.

If you haven’t already, stop the switch.py program. Open an editor to create a new Python program with the following code and save it as ‘interrupt.py’ in the home directory:

import RPi.GPIO as GPIO #load the GPIO library

GPIO.setmode(GPIO.BCM) #ensure the pin numbering matches the cobbler

RED = 13 #Red led is connected to pin 13

YELLOW = 19 #Yellow to 19

GREEN = 26 #Green to 26

LEDS=[RED, YELLOW, GREEN] #create an array of all pins

SWITCH = 21 #switch is connected to pin 21

GPIO.setup(LEDS, GPIO.OUT) #Set all the led pins to be outputs

GPIO.setup(SWITCH, GPIO.IN) #Set the switch pin to be input

GPIO.output(GREEN, GPIO.HIGH) #Turn on green led

#This defines a function to be used as the callback when

#the interrupt is generated.

def switchStateChanged(channel):

state = GPIO.input(SWITCH) #Read the switch state

GPIO.output(RED, state) #Set to current state

GPIO.output(YELLOW, 1-state) #Set to not current state

#start listening for changes in state, and set the callback function

GPIO.add\_event\_detect(SWITCH, GPIO.BOTH)

GPIO.add\_event\_callback(SWITCH, switchStateChanged)

try:

#main loop

while True:

input()

finally:

GPIO.cleanup() #Release all resources

This code uses interrupts to change the state of the LEDs when the switch is pressed. The code still has a while loop, but this doesn’t consume CPU cycles because the “input” call (which waits for a keyboard press) will block until it receives a signal from the OS as a result of a hardware interrupt generated by the user pressing a key.

Execute interrupt.py and note what happens when the switch is pressed. Leave the switch.py file running and open a second terminal window. In that terminal run the htop command. What do you see? How does the CPU usage compare to when you controlled your LEDs with polling?

# Additional Tasks

These tasks are to be completed if you have time left over at the end of the lab.

Make an LED turn on and remain on for 5 seconds after a button press.

Edit leds.py so that there are 2 LEDs on simultaneously for 0.5 seconds before the first one is turned off.

Implement traffic lights for a pedestrian crossing. At some point after the button is pressed, the lights should go through the sequence from green to yellow to red, then wait 1 minute, and then go to both red and yellow together to green. Once the lights are green, there should be a minimum of a two-minute wait before the lights change again (assuming the button has been pressed). Pressing the button multiple times should have no effect after the initial press that triggers the sequence.

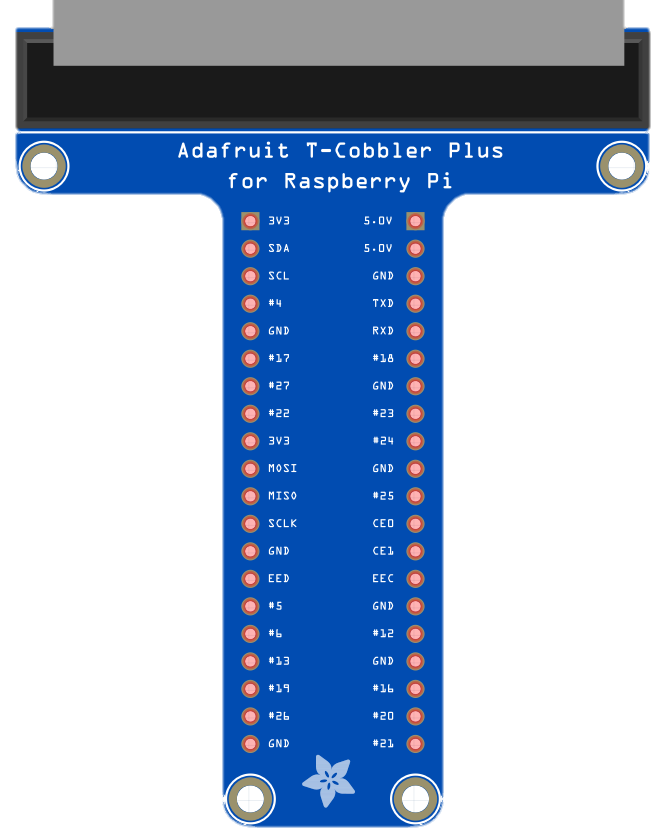
At the end of the lab, please make sure that you:

• Shutdown the Pi

• Remove the jump wires, GPIO breakout, LEDs & Switch from the breadboard and return all parts to the box

• Leave the desk in a clean and tidy state.

1. Pinout Reference



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  | 3.3V |  | 5V |  |
|  | I2C SDA |  | 5V |  |
|  | I2C SCL |  | GND |  |
|  | GPIO 4 |  | Serial TX |  |
|  | GND |  | Serial RX |  |
|  | GPIO 17 |  | GPIO 18 |  |
|  | GPIO 27 |  | GND |  |
|  | GPIO 22 |  | GPIO 23 |  |
|  | 3.3V |  | GPIO 24 |  |
|  | SPI MOSI |  | GND |  |
|  | SPI MISO |  | GPIO 25 |  |
|  | SPI SCLK |  | SPI CE0 |  |
|  | GND |  | SPI CE1 |  |
|  | EED |  | EEC |  |
|  | GPIO 5 |  | GND |  |
|  | GPIO 6 |  | GPIO 12 |  |
| Red LED | GPIO 13 |  | GND |  |
| Orange LED | GPIO 19 |  | GPIO 16 |  |
| Green LED | GPIO 26 |  | GPIO 20 |  |
|  | GND |  | GPIO 21 | Switch Input |
|  |  |  |  |  |