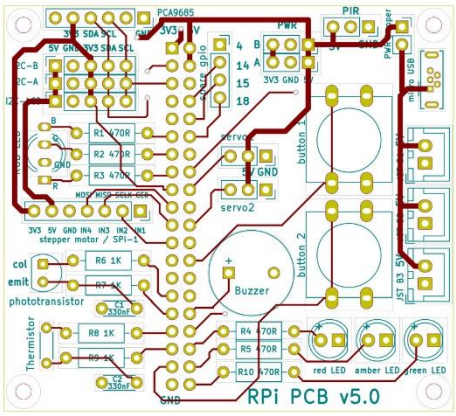
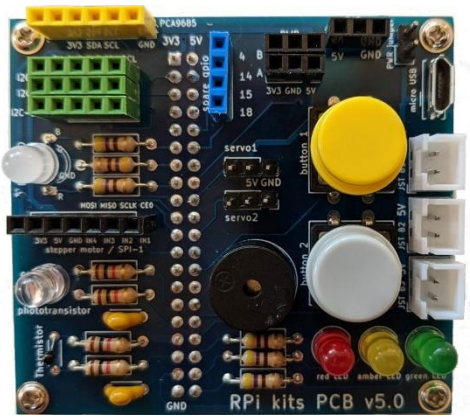
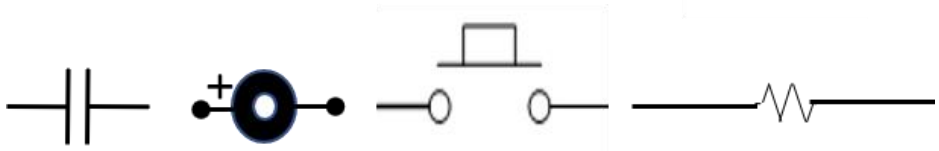
Raspberry Pi PCB







Usage Documentation

version 2.1

Table of contents

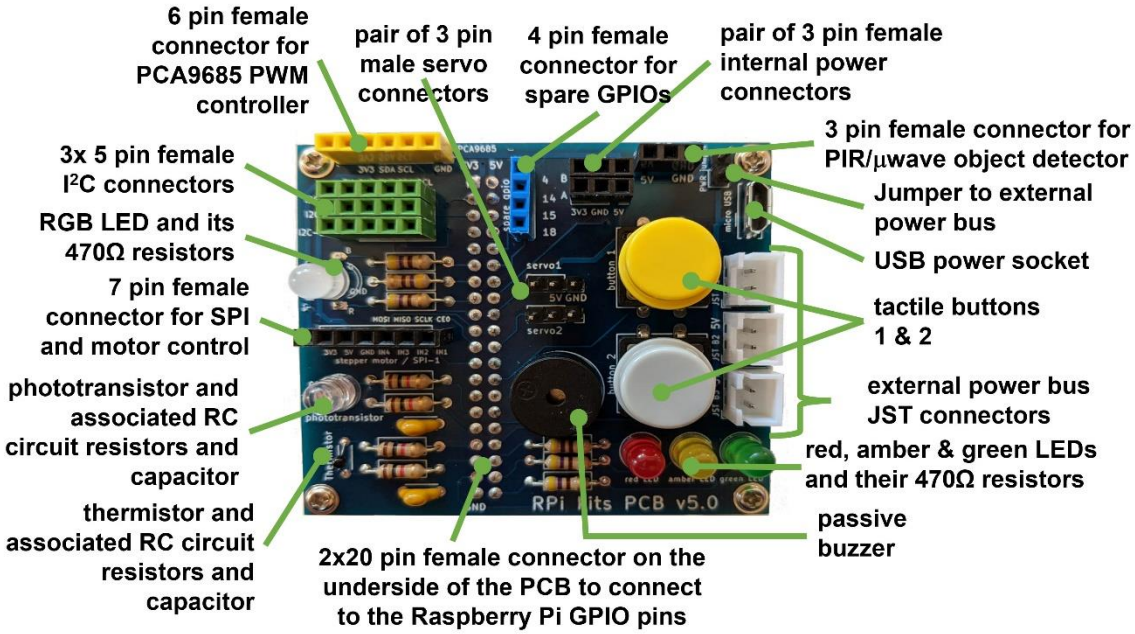
Introduction

The **Raspberry Pi PCB** enables digital makers to explore a wide range of projects and methods. The definition of 'digital making' is somewhat wide but is generally taken to be the combination of Design & Technology (DT) aspects of 'making' with the 'digital' Computer Science (CS) skills of programming and code development. It is therefore all about the creation and development of physical things that are controlled and managed by software. The activity scope is therefore broad, embracing electronics, photography, robotics/mechatronics, music, artwork/installations, and much more!

Digital making with small low-cost single board computers (SBCs) like the Raspberry Pi is not only a fun thing to do, but it is increasingly important to encourage these skills in the context of the extraordinarily rapid Fourth Industrial Revolution that is now upon us. Equipping both adults and young people for a changing society, and the 'digital intense' world of work that is emerging, is therefore becoming ever more vital. This is especially important in the context of recent reports which estimate that 90% of all new jobs will require digital skills to some degree.

The **Raspberry Pi PCB** is a custom printed circuit board (PCB) onto which a set of electronic components and connectors can be soldered. This allows a digital maker to create a 'module' that connects to the GPIO pins of a Raspberry Pi and enables a wide variety of projects and methods, all involving software control by a Raspberry Pi.

The aim of this project development is to provide access to an expanding library of example code to allow a digital maker to explore and control many different components and devices. The project started by consolidating components previously used in many separate breadboard-based projects, to which further additional options have now been added. The result is a PCB design that allows a robust, permanent assembly to be built as shown in the schematic below which illustrates a populated printed circuit board.



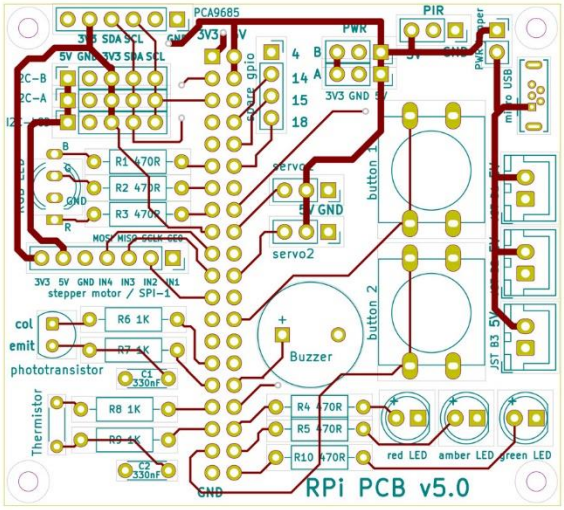
Building a more permanent assembly from a set of components that includes a custom Printed Circuit Board (PCB), not only allows an extra 'soldering' skill to be practiced, but also allows the more robust populated PCB to be used over an extended period to develop different software projects and options.

Example software has been developed to provide 'get going' capabilities that digital makers can use immediately and then build upon to extend their software development skills. Python code has been developed for all the various example projects, but in addition, for the 'electronic basics' projects, example Scratch code is also available for the equivalent set of code in Python. This 'dual approach' aims to facilitate a maker progressing from the easy-to-use Scratch 'block programming' to more functional text-based Python coding.

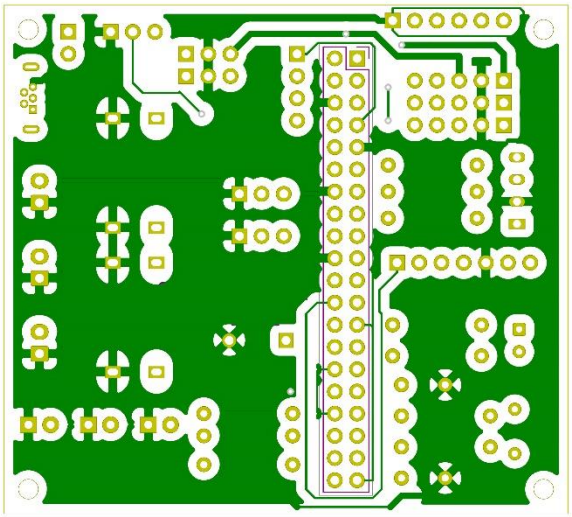
PCB development and component usage

The design of the **PCB** is now at v5.0, which reflects several iterations of the design to implement refinements and functional extensions from the start of the project in September'20. A .zip file of the 'Gerber' design files can be downloaded from **here** so that anyone can have small quantities of the PCB manufactured using one of the many low cost online suppliers that are now available. Further information is provided in *Appendix A: PCB development details.*

The PCB design process has used the opensource KiCAD software and the images below show some details of the v5.0 design generated by the software.



front of the PCB showing the component footprints and the copper power & signal interconnections, with overlaid annotation of the Raspberry Pi GPIO numbering

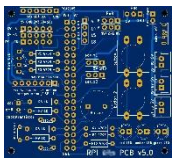
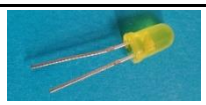
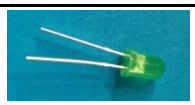
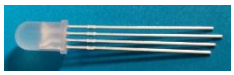
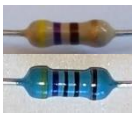
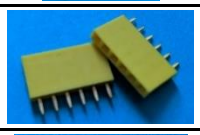
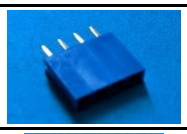
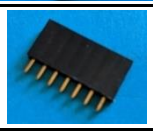
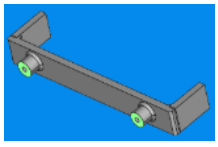


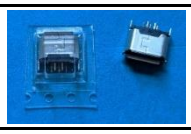
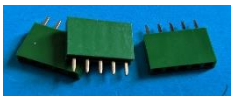
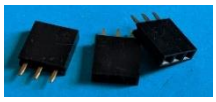
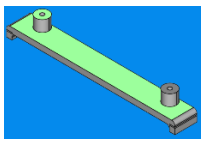
back of the PCB with the 'flooded' ground plane as well as some interconnections on this side

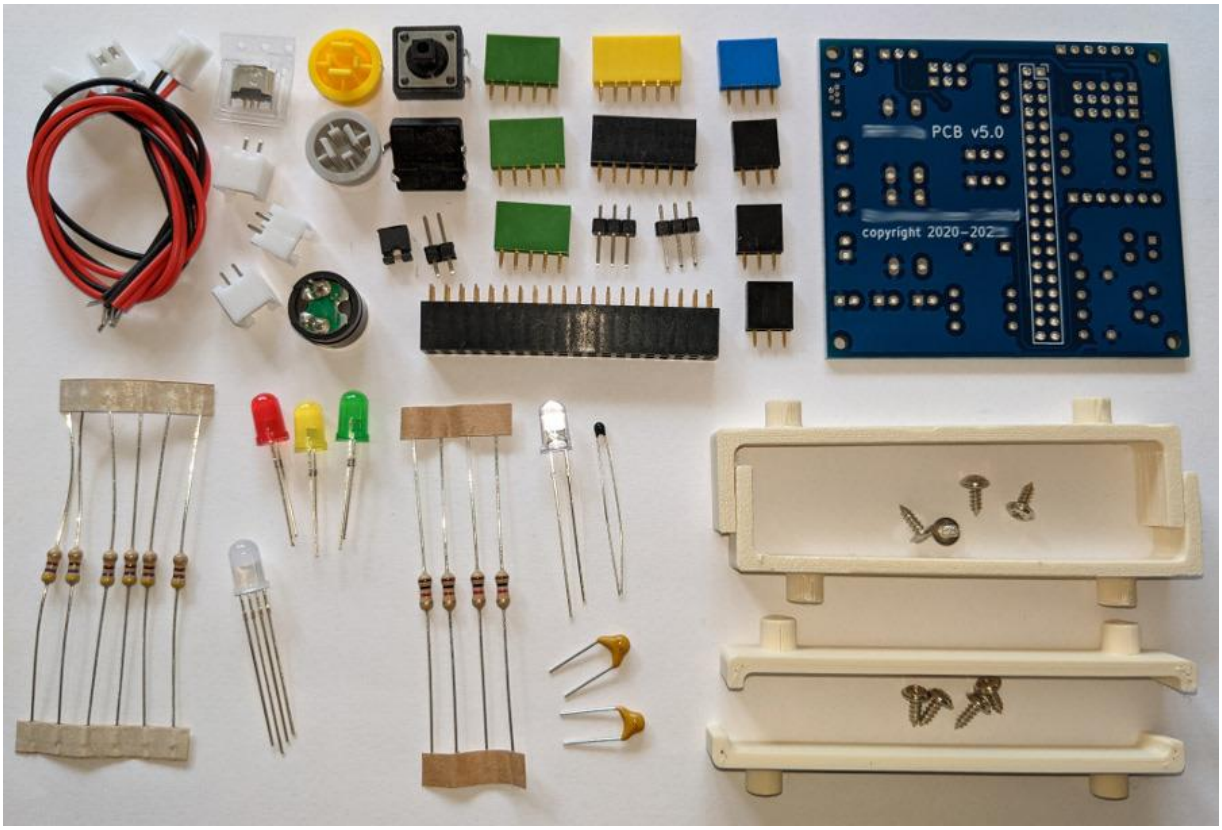
PCB component details

A **PCB** assembly uses the readily available components in the quantities set out in the table below and shown all together in the following image. A complete assembled PCB can be produced by soldering all the components in place with the *PCB assembly details* section of this document providing build description information.

Further technical information on each of the components is provided in *Appendix B: PCB component details*.

description	image	quantity
Printed circuit board (v5.0)		1
Red LED		1
Amber LED		1
Green LED		1
RGB LED		1
470Ω resistors (carbon or metal film type) bands: yellow-purple-brown or yellow-purple-black-black-brown		6
JST sockets + plugs with lead		3
1row x 6P female yellow header		1
1row x 4P female blue header		1
1row x 7P female black header		1
1row x 3P male black header		2
3D printed PCB 'stand' + 2x 6mm M2 self-tap screws		2 + 4

description	image	quantity
1kΩ resistors (carbon or metal film type) bands: brown-black-red or brown-black-black-brown		4
330nF capacitor		2
Thermistor 10kΩ NTC		1
Phototransistor		1
Tactile buttons (colour of caps to suit)		2
Passive buzzer		1
USB type B micro socket		1
1row x 5P female green headers		3
1row x 3P female black headers		3
2row x 20P female black header		1
1row x 2P male black header + jumper		1 + 1
3D printed Raspberry Pi 'stand' + 2x 6mm M2 self-tap screws		2 + 4



PCB usage summary

The **Raspberry Pi PCB** has been designed to allow a very wide (and expanding!) range of projects and methods to be explored, as summarised below, with example/starter code provided that a digital maker can develop further themselves.

The aim of all the discussed projects and methods is, not only to show how a particular component/device operates, but to also show, by using 'operational logic' set out in code, how individual physical actions can be carried out under the control of the Raspberry Pi.

Individual sections of this document are provided later for each of these project/method areas summarised below, providing the detail for connecting various devices and using the example/starter software.

Electronic basics projects:

This set of electronic projects are the starting point for all the subsequent usage of an assembled PCB. These show how 'output' components like the various LEDs, or the buzzer can be operated by an 'input', such as the pressing of a button. These basic functions introduce a key facet of 'digital' operation where something is 'switched on', not by completing a circuit by the closing of a switch, but instead by sensing something that 'happens' and using logic to decide whether to separately energise/switch something on, or to carry out some other action.

The PCB with a set of components that when assembled, allows a range of 'basic' electronic projects to be explored. Example Scratch code is available to start exploring these basic electronic projects, and similar function Python code is also provided to support the transition to a text-based programming language.

Please note that this document does not provide support for the set up and running of a Raspberry Pi, nor the general use of Scratch or Python - since there are plenty of good books and online materials available to do this.

For Raspberry Pi Scratch coding the main examples and descriptions provided are for the offline version of Scratch 3, available on a Raspberry Pi4 and Pi5 and needing a minimum of 2GB of memory, which provides custom coding blocks to support the use of the Raspberry Pi's GPIO pins. The software download does however also provide similar code examples for the old/legacy off-line version of Scratch 1.4 in order to support the use of Scratch with much older versions of the Raspberry Pi for which only this version of Scratch is available. Also provided are Scratch 2 examples, but as this version of Scratch uses Adobe Flash which is no longer supported and does not run on a Raspberry Pi from the Buster OS onwards, these examples can only run on a Raspberry Pi from Stretch onwards. In addition, as Scratch 2 does not allow a GPIO pin's pull up/down status to be set, some additional system configuration would be needed if a button is connected to any GPIO pin other than 2 to 8, since these higher numbered pins have their default as pull-down. The design of the PCB does however use GPIO pins 7 and 8 for the buttons, so for reference, details for how the system configuration for resetting the GPIO defaults can be done is provided in *Appendix H: Raspberry Pi GPIO pin default settings.*

Image taking methods:

A number of different image taking methods (single images, video clips, stop-motion, time-lapse, etc.) can be explored by using a simple USB camera that can be connected to a USB port on the Raspberry Pi.

Python code examples are available that use an assembled PCB's buttons, LEDs, and buzzer to show how images can be captured when a button is pressed, with LEDs and the buzzer used to show the progress of the image taking cycle.

Motor control:

Servo, stepper, and drive motors can all be controlled by a Raspberry Pi using an assembled PCB with additional commonly available motors and other components. Each of these motor types can be used in different ways to develop projects for 'things that move', which could be anything from a robot that runs around, to a complex, static mechatronic construction.

Each motor type can be controlled with code that runs on the Raspberry Pi and example Python code is provided to demonstrate a range of control methods for each motor type.

Display projects:

Various types of display (LCD, OLED, 7 segment LED, etc.) can be explored using the PCB where an I^2C or a SPI interface might be required. Details of commonly available low-cost displays are given, and example Python code for them is made available.

Sensor projects:

The Raspberry Pi is an excellent platform for measuring a wide variety of device performance and environmental parameters with many different types of low cost sensors being readily available. The PCB provides a means to easily connect many different types of sensor to a Raspberry Pi with example Python software made available to help a digital maker explore different ways of 'measuring the world'.

PCB assembly details

Assembling a complete Raspberry Pi PCB is a significant soldering task for which some experience at soldering is needed.

If you are just a 'beginner' when it comes to soldering then the **Starter PCB** will provide a less challenging project to practice soldering before tackling this more extensive **Raspberry Pi PCB**.

Some suggested assembly/soldering 'tips'

- As there are a large number of components to be added to the PCB, and this is a manual soldering process, a suggested component sequence is detailed below that can help avoid previously soldered components 'getting in the way' of the next component being soldered.
- A particular danger to avoid is part of the soldering iron inadvertently touching a previous component, particularly the plastic surround of a female header. This may cause sufficient damage that the header cannot be used and removing and replacing a previously soldered component like a header with multiple connection points is extremely difficult! It is therefore particularly important to avoid damaging the 2x20 GPIO header once it is soldered onto the underside of the PCB as any damage to this could prevent it from inserting properly onto the Raspberry Pi's GPIO pins, making the entire PCB assembly unusable!
- A good quality lead free solder wire with a rosin/flux core should be used. In addition, to ensure that only small amounts of solder can be carefully added to a joint in a controlled way, a very small diameter solder wire, say 0.6mm, is recommended.
- A narrow tipped soldering iron should be used and as with any soldering activity you should make sure that the soldering iron is fully up to temperature, the activity is carried out in a well-ventilated area, and all appropriate safety precautions are undertaken. Whilst a more sophisticated thermostatically controlled soldering iron is always useful, a more basic uncontrolled iron is perfectly adequate for these activities.
- Generally, the use of additional solder paste/flux should be avoided, and whilst it might be 'safely' used on say a discrete component like a resistor, it should be avoided when soldering female headers in particular, since additional flux can cause excess solder to quickly 'wick' up into the header and block the insertion of a male component pin into the female header opening.

- Once a PCB has been fully populated, whilst a final 'cleaning' step may not be necessary, especially if no additional solder paste/flux has been used, washing the complete PCB in isopropyl alcohol/isopropanol and 'solvent brushing' all the soldered joints with a small clean paint brush will ensure any soldering residues are removed which could, for example, create low resistance paths between adjacent pins on a header connector. This solvent use should of course only be carried out in a well-ventilated area observing all necessary safety precautions when using volatile solvents.
- Finally, completing all the needed soldering steps for a complete PCB will take some time, so take an occasional break!

Suggested component assembly sequence

Below is a suggested sequence for soldering all of the components onto the PCB.

As previously mentioned this is an extensive soldering exercise for which some soldering experienced is needed – but the **Starter PCB Usage Documentation** provides additional tips and illustrations that may be found useful for soldering these components onto the Raspberry Pi PCB.

Resistors x10

Soldering discrete components like resistors is perhaps the most straightforward of the assembly tasks since there are no problems if too much solder is applied since it can 'safely' wick along the wires and it is very visibly obvious if too little solder is applied.

All ten of the resistors, 6x 470Ω and 4x 1kΩ, are fitted in a similar way by inserting the resistor into the designated area on the front of the PCB – clear labelling is provided on the PCB - and then folding the leads on the underside so that the resistor is held 'snugly' against the PCB front surface. As these are simple resistive devices it does not matter which way round the resistors are inserted into the PCB.

The soldering process is completed by placing the soldering iron tip firmly on the PCB's underside opening and the resistor lead and gently touching the joint with the tip of the solder wire until it is seen to start to melt. A small amount of melting solder wire is then 'fed' into the joint and the soldering iron tip held in place for a few more seconds until it is seen that the melted solder has 'wetted' the surfaces of the resistor's lead and the PCB opening and some solder has 'wicked' into and completely filled the gap. The excess length of resistor lead can then be snipped off or 'waggled' until it breaks off.

Capacitors x2

The two ceramic capacitors are installed in exactly the same way as the resistors, inserting them into their designated areas on the front of the PCB. Just like the resistors, it also does not matter which way round these simple ceramic capacitors are inserted into the PCB.

2x20 female GPIO header

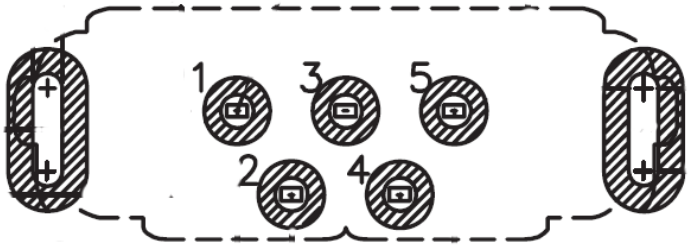
The greatest of care should be taken with the installation of the 2x20 female header, which is fitted to the underside of the PCB. It is essential that all 40 connections are electrically sound whilst no excess of solder is allowed to 'wick' up into the header which would prevent the header from fully inserting on the Raspberry Pi's GPIO pins.

A simple suggestion to help with this installation is to use some Blu Tack to hold the header firmly in place and to first solder 2 or 3 well separated connections not too near the Blu Tack to initially secure the header in place. The Blu Tack can then be easily removed (as it will not have been heated up to any extent) and the rest of the connections completed.

For each connection the same soldering technique as used for a discrete component like a resistor is used i.e., the soldering iron tip is held firmly on the PCB opening and the header connection tip whilst gently touching the joint with the solder wire until it is seen to start to melt. Only allow the most minimum amount of solder to flow into the joint whilst ensuring that the header connector and the PCB opening are properly 'wetted' by the solder.

USB micro connector

This component can also be initially held in place on the front of the PCB using some Blu Tack and secured in place by soldering the two outside metal tabs (as shown right) into their 'slots' on the PCB.



With the unit now physically secured, the Blu Tack can be removed and just the two outside small connectors (1 and 5 as shown above) can be soldered. These are just very small openings, so the soldering iron tip just needs to be held onto each opening and a very small amount of solder used to 'wet' the connection.

Tactile buttons x2

The two buttons are inserted into the front of the PCB: the four legs of each button may need to be straightened slightly so that they align with the four 'slot' openings on the PCB. Each button is firmly pushed into the PCB openings so that it 'clicks in' and sits evenly and directly on the PCB surface and is held in place. All four legs of each button are then soldered in place by the soldering iron tip being pressed firmly on the junction between the button leg and the PCB opening with solder being fed into the gap as it melts.

JST female sockets x3

All three JST sockets can be inserted into the front of the PCB (make sure they are the right way round!) and held in place with Blu Tack whilst the socket pins are soldered into the PCB openings.

Passive buzzer

Make sure this is inserted correctly with the + pin of the buzzer inserted into the PCB opening also marked with a +, then use some Blu Tack to hold the buzzer in place whilst the buzzer pins are soldered into the PCB openings.

Female and male headers

Several female and male headers can now be fitted to the front of the PCB in specific 'groups' using a similar method as used for the 2x20 female header i.e., Blu Tack is used to initially hold the headers in place while 1 or 2 connections are soldered, the Blu Tack is then removed, and the rest of the connections completed. In all instances only a minimal amount of solder should be used to avoid excess solder 'wicking' up into the header. Please note that different coloured headers are noted below just to make them more distinctive in use – but any colours can of course be used.

- ***1 row x 4P female blue header x1*** and ***1 row x 3P male black header x2***
  These headers are for the 'spare' GPIO and the two servo connections. Extra care should be taken to not let the soldering iron 'touch' the plastic sides of the 2x20 header as this 'group' of headers are in close proximity to the main 2x20 header.
- ***1 row x 3P female black header x3*** and ***1 row x 2P male black header x1***
  This 'group' is the two internal power connections, the PIR/μwave connector and the male 2P 'jumper' connector. They can all be held in place together with Blu Tack whilst 'securing' them in place with a few soldered connections before removing the Blu Tack and completing the remaining connections.
- ***1 row x 6P female yellow header x1*** and ***1 row x 5P female green header x3***
  This 'group' is the PCA9685 connector and the three I^2C connectors, which can all be Blu Tacked in place together before soldering the connections.
- ***1 row x 7P female black header x1***
  This last female header is for the SPI/motor control connections, and it can be individually Blu Tacked in place before soldering the connections.

Red, amber, and green LEDs

The LEDs are inserted onto the front of the PCB in their red, amber, green sequence, making sure that the longer +'ve anode connectors are inserted into the correct openings on the PCB, which are clearly labelled with a small +.

Even though both connectors of a LED are quite long, and they can be bent to try to hold the LED in place, because of their height the LEDs do tend to 'wobble' and not stay flush with the surface of the PCB. Small pieces of Blu Tack can therefore be used to temporarily fix the LEDs firmly in place. Each of the connectors are then soldered in place in the same way as all the previous discrete components by the soldering iron tip being pressed firmly on the junction between the LED lead and the PCB opening with a small amount of solder being fed into the gap as it melts.

All the LEDs can be positioned/held in place and soldered at the same time – which will make the installation a little quicker. Then once soldered in place the Blu Tack can be removed and the excess lengths of LED connector lead can be snipped off or 'waggled' until they break off.

Phototransistor, Thermistor and RGB LED

This last set of discrete components can all be inserted into the front of the PCB, initially held in place with Blu Tack if necessary, and soldered in place. Make sure that the long (common) lead of the RGB LED is inserted into the PCB opening marked *GND* and the long (emitter) leg of the phototransistor is inserted into the PCB opening marked *emit.*

It should be noted that all these components have PCB connections that are spaced at 2.54/5.08mm so that, instead of directly inserting them into the PCB, a female header could be soldered in place and the component connected on a long multi-wire lead to allow remote usage of the component.

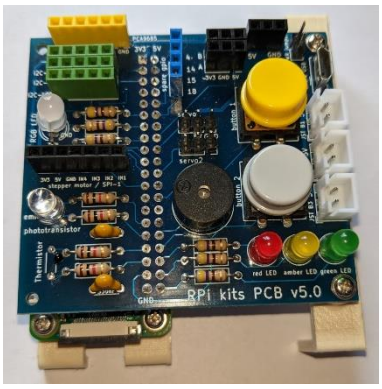
Connecting an assembled PCB to a Raspberry Pi

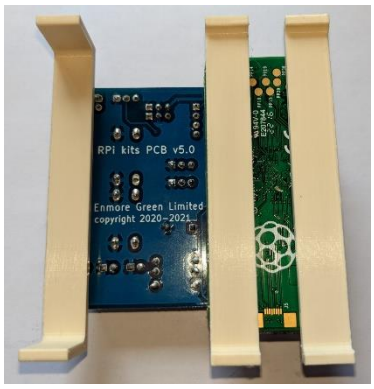
The main Raspberry Pi family of Single Board Computers (SBCs) is available in a number of different formats, from the larger format Pi5 down to the smallest version, the Pi Zero.

The **Raspberry Pi PCB** is designed to connect to any of these SBCs that have a 2x20 pin GPIO connection point, which is all of them except the Compute Modules (although the Compute Module's optional IO board does have one) and some of the very earliest versions that are no longer in production.

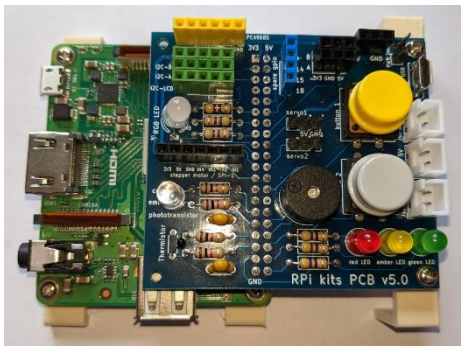
To support all these possible connection options two sets of 3D printed 'stands' have been designed that fit the fixing holes for the PCB and all the various Raspberry Pi's, as shown left and right below – the designs for these 3D prints can be downloaded from **here**.

For example, the images below show how a pair of Pi-stands (above right) are attached to a Pi Zero, and a populated PCB with one of the PCB-stands (above left) has been connected to a Pi Zero so that the PCB's 2x20 pin connector can insert into the Pi Zero's GPIO pins. A simple, complete assembly is therefore created that is stable and allows all the Pi Zero's connectors to be accessible.

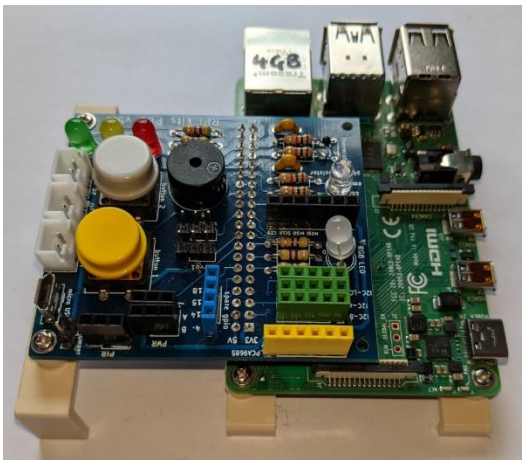


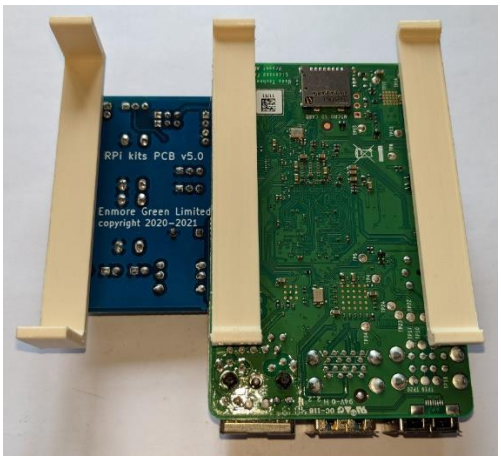


A similar method, using the same 3D printed 'stands' can be used with the mid-sized Raspberry Pi 3A as illustrated in the images shown next.

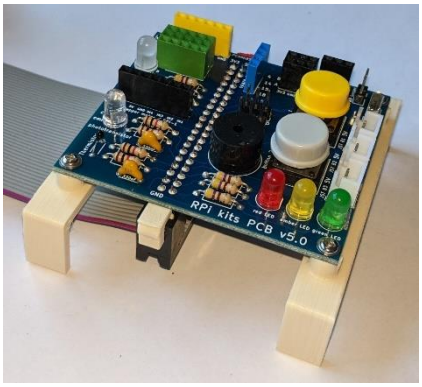


.. and similarly for the largest Pi format (for a Pi2B, Pi3B, Pi4B or a Pi5), which have the same form factor and fixing hole spacings, and is shown below for a Raspberry Pi 4B:





For a Pi 2B, a Pi 3B, a Pi5 or a Pi 4B, as shown 1st right, a PCB can also be simply inserted onto the GPIO pins with the Raspberry Pi in a 'conventional' case like the standard Raspberry Pi4 case shown, albeit without a lid or the side pieces.



Also, if a GPIO extension cable is used, as shown above/2nd right, it can be connected to any Raspberry Pi, such as the connection to a Raspberry Pi 400, as shown on the right.

For all these assembly options care should be taken to insert the PCB the right way round, and to help with this the PCB is labelled '3V3 5V' at one end of the GPIO connector and 'GND' at the other end.

Scratch and Python coding

To support the exploration of some electronic 'basics' with an assembled PCB, the projects described below can use both Scratch and Python coding.

The provided Scratch examples closely match how the corresponding Python examples work and can therefore help digital makers 'transition' from the easier to use graphical user interface of Scratch, to the more extensive and powerful functionality of the text based Python language.

For Scratch coding, repeating some previously mentioned information, the examples and descriptions provided are all for the offline version of Scratch 3 available on a Raspberry Pi4 and Pi5 with a minimum of 2GB of memory, which provides custom coding blocks to support the use of the Raspberry Pi's GPIO pins. The example software downloads do however also provide similar code examples for the old/legacy off-line version of Scratch 1.4 in order to support the use of Scratch with much older versions of the Raspberry Pi for which only this version of Scratch is available.

For all the projects and method described in this document Python code examples are provided which were all developed for Python 3 which is now the standard version on any Raspberry Pi

Details on how to download all the example code and install all the required libraries on a Raspberry Pi are provided in *Appendix C: Software download*.

It should be noted that all coding and descriptive text use GPIO BCM numbering.

Using a Virtual Environment for Raspberry Pi Python projects

A virtual environment (*venv*) is a useful, logical way of isolating Python project dependencies from each other on a Raspberry Pi and ensuring things don't break when there are conflicts.

Starting with the Bookworm OS this more structured approach is more strictly enforced, so it should be used when using Python in the various coding examples described throughout this document.

To make this easier (using the advice from this link) the following is recommended to both create a Virtual Environment (*venv*) and to automatically 'activate' it whenever a CLI window is opened from the Raspberry Pi desktop or the Pi is remotely accessed by SSH.

Using the command: *sudo nano /home/YOURUSERNAME/.bashrc*

- the following should be added at the end of the file:

```
# automatically set up a virtual environment for using python
PY_ENV_DIR=~/my_virtual_env
if [ ! -f $PY_ENV_DIR/bin/activate ]; then
    printf "Creating user Python environment in $PY_ENV_DIR, please wait...\n"
    mkdir -p $PY_ENV_DIR
    python3 -m venv --system-site-packages --prompt myenv $PY_ENV_DIR
fi
printf " ↓ ↓ ↓ ↓   Hello, we've activated a Python venv for you. To exit, type \"deactivate\".\n"
source $PY_ENV_DIR/bin/activate
```

When using the Thonny IDE for Python coding, this should also be configured to use the Virtual Environment set up from the .bashrc edit above. But the current 4.x release of Thonny for the Raspberry Pi with the Bookworm OS does not make the use of a pre–configured *venv* straightforward – so the 'work around' that seems to work OK is to use the Thonny menus to create another new Virtual Environment and then close Thonny and edit its config file at:

/home/YOURUSERNAME/.config/Thonny/configuration.ini

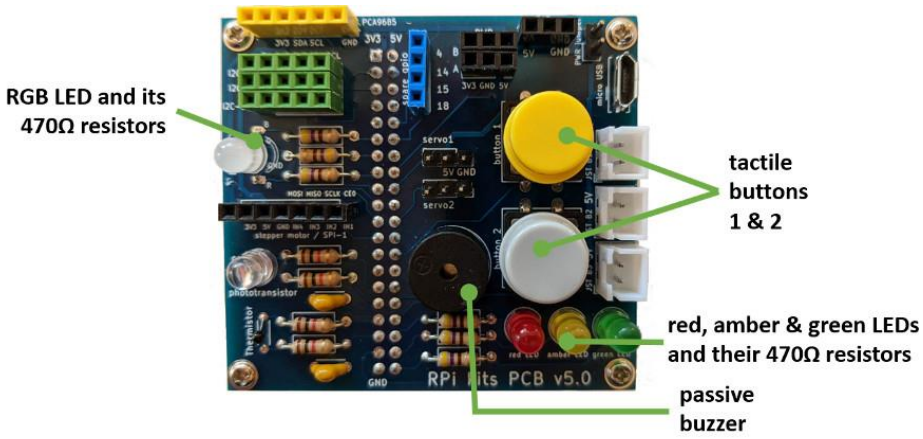
- and change the references in the file to the newly set up *venv* to the pre-existing one.

Electronic basics

As previously discussed, one of the main aims of the ***Raspberry Pi PCB*** project has been to enable a range of simple electronic projects to be explored using both Scratch and Python software.

These "electronic basics" are the starting point for all the subsequent usage of the PCB and show how 'output' components like the various LEDs or the buzzer can be operated by an 'input', such as the pressing of a button.

This aim determined the need for the chosen 'electronic basics' components to be included in the design and interconnected on a populated PCB as shown on the right and in more detail in the table below.



Component	Raspberry Pi GPIO pin connections
Red LED	GPIO#16 connected through a 470Ω resistor to the LED's +'ve anode (long leg)
Amber LED	GPIO#20 connected through a 470Ω resistor to the LED's +'ve anode (long leg)
Green LED	GPIO#21 connected through a 470Ω resistor to the LED's +'ve anode (long leg)
RGB LED	GPIOs #22, #27 & #17 connected to the RGB LED's red, green, and blue legs respectively
Tactile button 1	GPIO#07 and GND connections across the button
Tactile button 2	GPIO#26 and GND connections across the button
Passive buzzer	GPIO#12 connected to the positive terminal of the buzzer

To support the exploration of some electronic basics with an assembled PCB, the following projects using both Scratch 3 and Python 3 coding can be used.

Once the example software has been downloaded, as shown in *Appendix C: Software download*, the default folder on the Raspberry Pi for all the 'electronic basics' Python software is assumed to be:

/home/YOURUSERNAME/RPi_maker_PCB5/electronic_basics/

.. and the default folder for all the 'electronic basics' Scratch 3 software is:

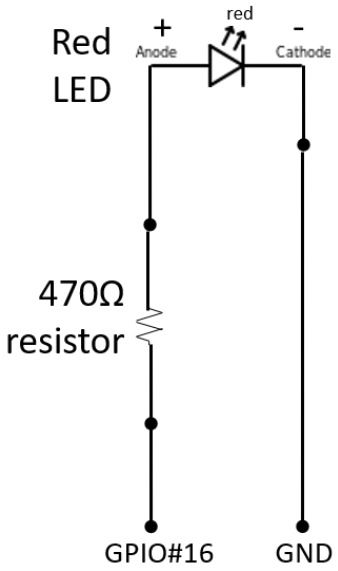
/home/YOURUSERNAME/RPi_maker_PCB5/electronic_basics/scratch3/

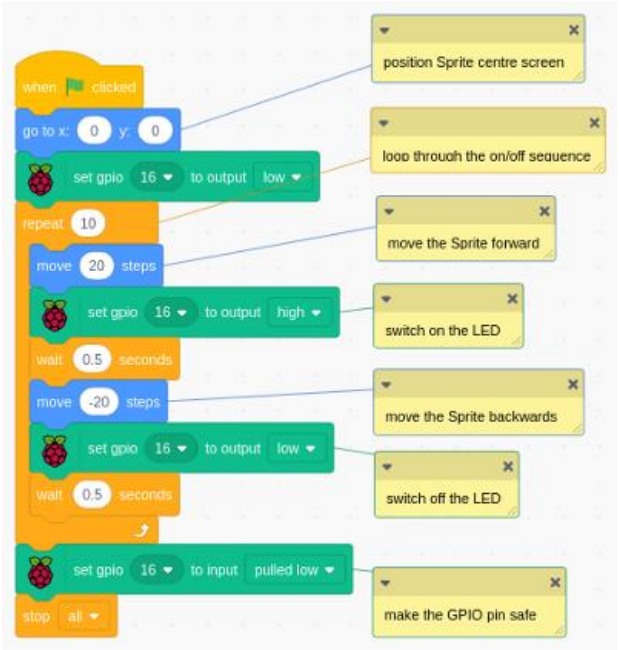
.. where YOURUSERNAME is whatever your user name is on the Raspberry Pi – but the files can of course be moved to somewhere else that may be more convenient, but these are the assumed locations for the descriptions of the individual projects below.

Single flashing LED

The circuit diagram shown on the right shows how the red LED on the PCB is connected through to ground from its (negative) cathode lead and to the Raspberry Pi's GPIO#16 pin via a 470Ω resistor from its (positive) anode lead.

The LED will be turned ON when the GPIO pin is set HIGH as this sets the pin at about 3.3V, and to avoid damaging the LED, the 470Ω resistor ensures that the current through it is limited. More details of how LEDs work is provided in *Appendix B: PCB component details.*



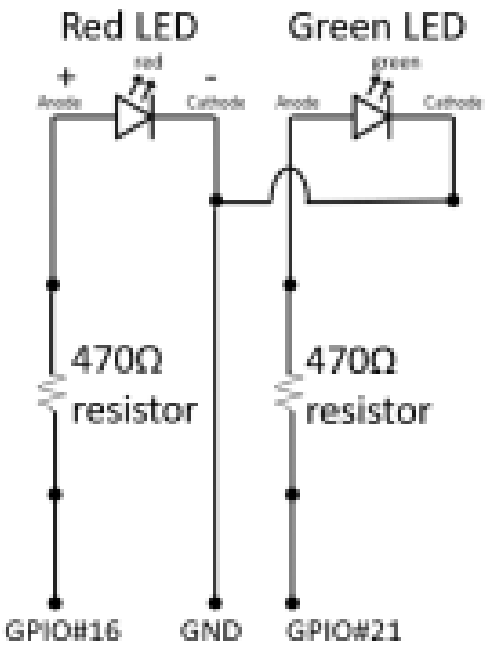


This very simple project flashes the red LED on/off and the screenshot on the left shows the example code which can be run in Scratch 3 by using the 'File->Load from your computer' menu and loading the ***LED_flash.sb3*** file from the folder it was downloaded to.

To run the Python ***LED_flash.py*** code use the Thonny IDE or alternatively type or copy/paste the full command shown at the top of the listing into a Terminal window, where a *venv* should have been automatically 'activated', i.e.

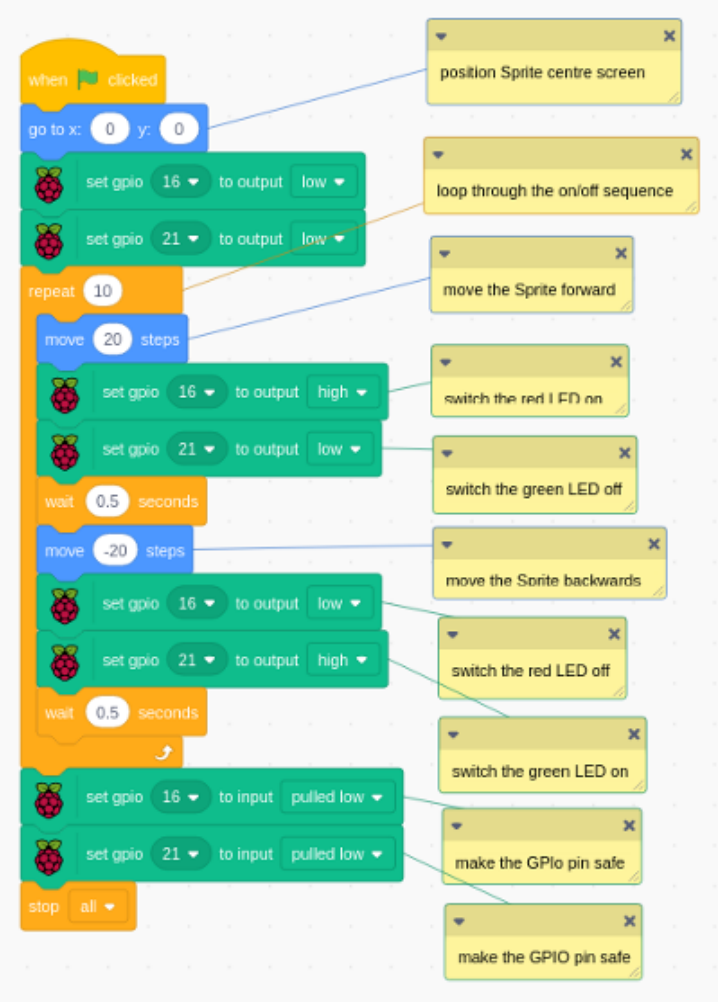
```
$ python3 ./RPi_maker_PCB5/electronic_basics/LED_flash.py
```

Red/Green flashing LEDs



The circuit diagram shown on the right shows how the red and green LEDs on the PCB are connected through to ground from their (negative) cathode leads and to the Raspberry Pi's GPIO#16 and #21 pins via 470Ω resistors from their (positive) anode leads.

As discussed above and in *Appendix B: PCB component details*, each LED will be turned ON when its GPIO pin is set HIGH and the 470Ω resistors protect the LEDs from over current damage.



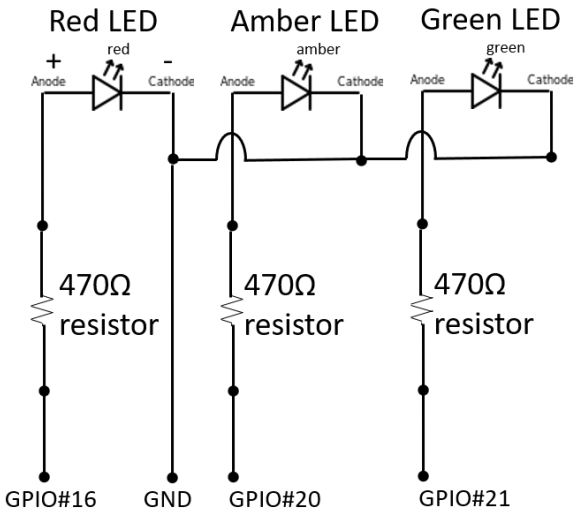
This project alternately flashes the red and green LEDs on/off and the screenshot on the left shows the example code which can be run in Scratch 3 by using the 'File->Load from your computer' menu and loading the ***LED_red_green_flash.sb3*** file from the folder it was downloaded to.

To run the Python ***LED_red_green_flash.py*** code use the Thonny IDE or alternatively type or copy/paste the full command shown at the top of the listing into a Terminal window, where a *venv* should have been automatically 'activated', i.e.

```
$ python3 ./RPi_maker_PCB5/electronic_basics/LED_red_green_flash.py
```

Red/Amber/Green flashing LEDs

The circuit diagram shown on the right shows how the red, amber, and green LEDs on the PCB are connected through to ground from their (negative) cathode leads and to the Raspberry Pi's GPIO#16, #20 and #21 pins via 470Ω resistors from their (positive) anode leads.



As discussed above and in *Appendix B: PCB component details*, each LED will be turned ON when its GPIO pin is set HIGH and the 470Ω resistors protect the LEDs from over current damage.

This project alternately flashes the red, amber, and green LEDs on/off in sequence.

Only example Python code is made available for this project allowing the digital maker to develop their own version of Scratch code, perhaps building upon the LED_red_green_flash.sb3 example and/or developing traffic light flashing sequences.

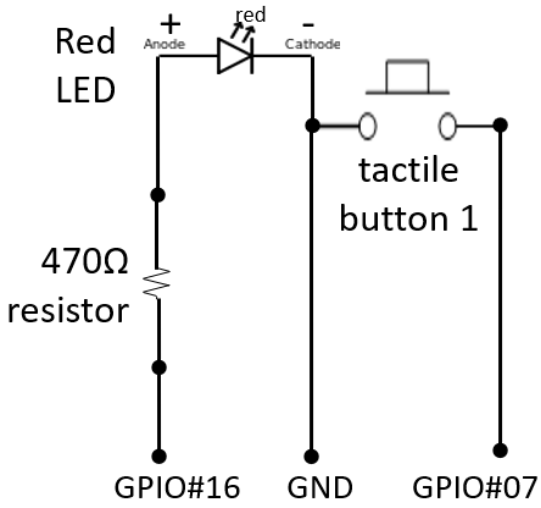
To run the Python ***LED_red_amber_green_flash.py*** code use the Thonny IDE or alternatively type or copy/paste the full command shown at the top of the listing into a Terminal window, where a *venv* should have been automatically 'activated', i.e.

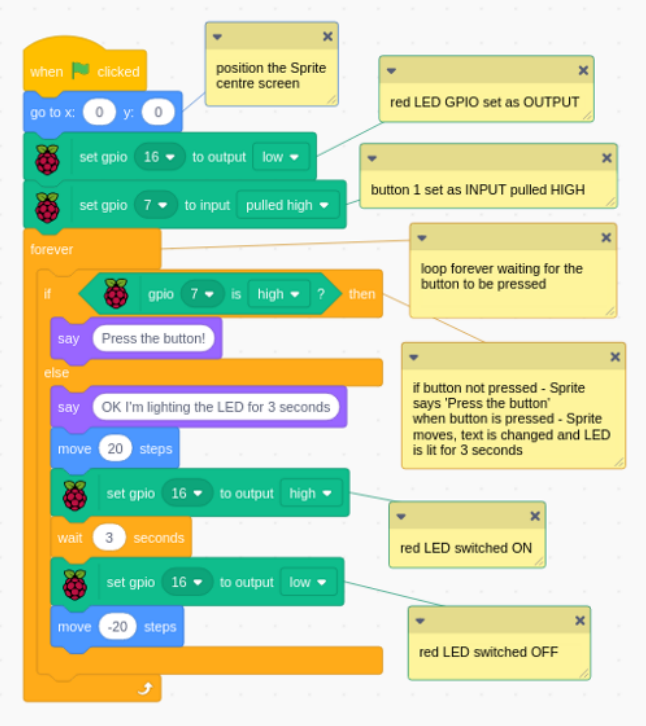
```
$ python3 ./RPi_maker_PCB5/electronic_basics/LED_red_amber_green_flash.py
```

Button switched LED

The circuit diagram shown on the right shows how the:

- red LED on the PCB is connected through to ground from its (negative) cathode lead and to the Raspberry Pi's GPIO#16 pin via a 470Ω resistor from its (positive) anode lead, and
- how button 1 is connected between ground and GPIO #7





This project turns the red LED ON for 3 seconds when button 1 is pressed and the screenshot on the left shows the example code which can be run in Scratch 3 by using the 'File->Load from your computer' menu and loading the ***LED_button_flash.sb3*** file from the folder it was downloaded to.

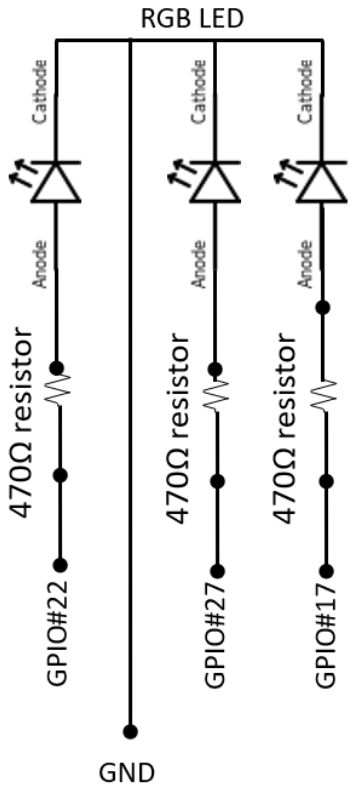
To run the Python ***LED_button_flash.py*** code use the Thonny IDE or alternatively type or copy/paste the full command shown at the top of the listing into a Terminal window, where a *venv* should have been automatically 'activated', i.e.

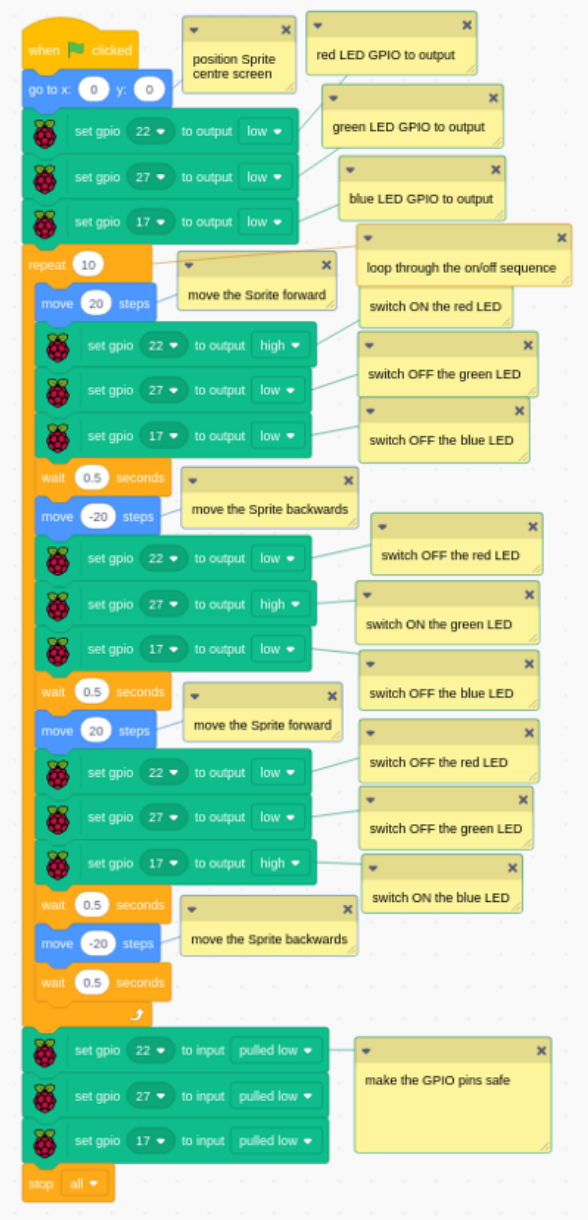
```
$ python3 ./RPi_maker_PCB5/electronic_basics/LED_button_flash.py
```

RGB flashing LED

The circuit diagram shown on the right shows how the red, green, and blue LEDs within the combined RGB LED on the PCB are connected through to ground from their common (negative) cathode lead and to the Raspberry Pi's GPIO#22, #27 and #17 pins via 470Ω resistors from their (positive) anode leads.

Each LED will be turned ON when its GPIO pin is set HIGH and the 470Ω resistors protect the LEDs from over current damage.





This project alternately flashes the red, green, and blue LEDs on/off and the screenshot on the left shows the example code which can be run in Scratch 3 by using the 'File->Load from your computer' menu and loading the ***LED_RGB_flash.sb3*** file from the folder it was downloaded to.

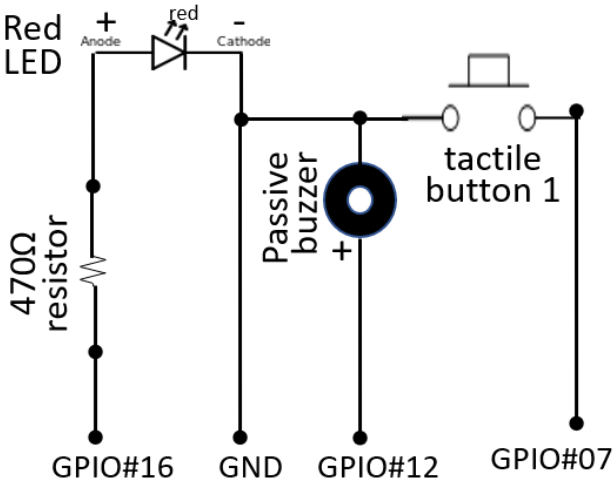
To run the Python ***LED_RGB_flash.py*** code use the Thonny IDE or alternatively type or copy/paste the full command shown at the top of the listing into a Terminal window, where a *venv* should have been automatically 'activated', i.e.

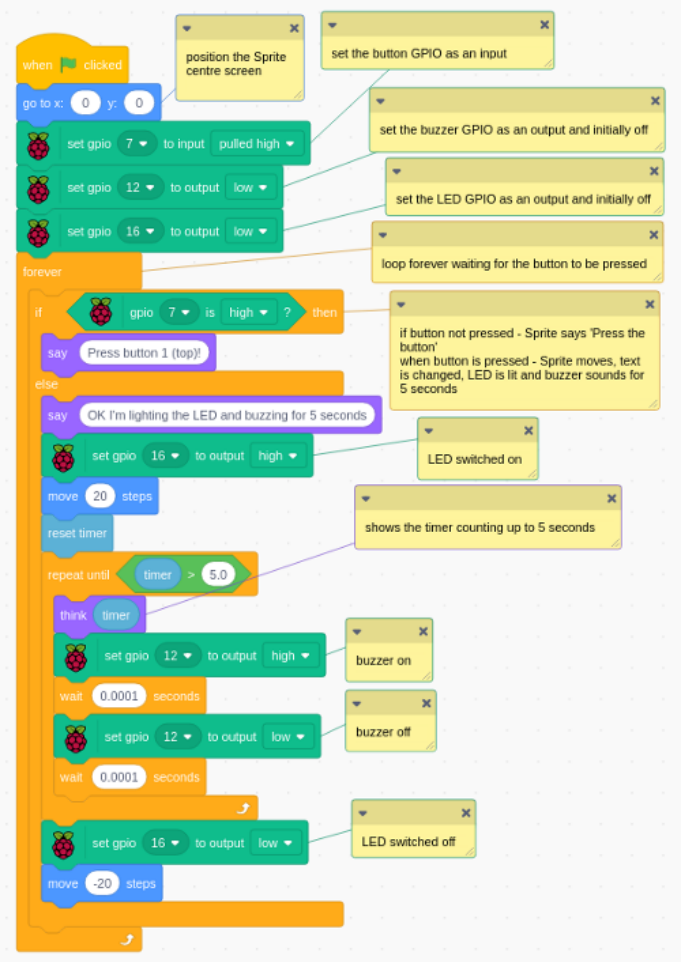
```
$ python3 ./RPi_maker_PCB5/electronic_basics/ LED_RGB_flash.py
```

Button switched LED & buzzer

The circuit diagram shown on the right shows how the:

- red LED on the PCB is connected through to ground from its (negative) cathode lead and to the Raspberry Pi's GPIO#16 pin via a 470Ω resistor from its (positive) anode lead,
- how button 1 is connected between ground and GPIO #7, and how
- the buzzer is connected between ground and GPIO#12





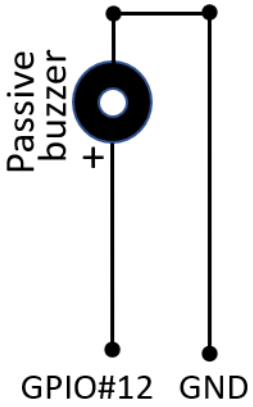
This project turns the red LED ON and sounds the buzzer for 5 seconds when button 1 is pressed and the screenshot on the left shows the example code which can be run in Scratch 3 by using the 'File->Load from your computer' menu and loading the ***LED_button_buzzer.sb3*** file from the folder it was downloaded to.

To run the Python ***LED_button_buzzer.py*** code use the Thonny IDE or alternatively type or copy/paste the full command shown at the top of the listing into a Terminal window, where a *venv* should have been automatically 'activated', i.e.

```
$ python3 ./RPi_maker_PCB5/electronic_basics/LED_button_buzzer.py
```

Buzzer 'tunes'

The circuit diagram shown on the right just shows how the buzzer is connected between ground and GPIO#12.



This project, only developed using Python software, shows how the very simple passive buzzer installed on the PCB can be programmed to 'play' quite elaborate tunes.

The code:

- defines a set of ***notes*** by their frequency,
- a series of well-known tunes defined by a ***melody*** as a short representative sequence of notes, and
- a ***tempo*** defined for each tune to signify the duration of each note in the melody.

To run the Python ***buzzer_player.py*** code use the Thonny IDE or alternatively type or copy/paste the full command shown at the top of the listing into a Terminal window, where a *venv* should have been automatically 'activated', i.e.

```
$ python3 ./RPi_maker_PCB5/electronic_basics/buzzer_player.py
```

Image taking methods

The **Raspberry Pi PCB** aims to support a number of different Image Taking methods that all use a USB camera that is connected to a USB port on the Raspberry Pi. Example/demonstration software is available that can use the components listed in the table below for many possible image taking control and 'indicator' options.

Component	GPIO pin(s)
RGB LED	GPIOs #22, #27 & #17 connected to the RGB LED's red, green, and blue legs respectively
Tactile button (2)	GPIO#26 and GND connections across the button (the 2nd tactile button is used for the Image Taking methods)
Passive buzzer	GPIO#12 connected to the buzzer's positive terminal
Passive InfraRed (PIR) detector	GPIO #23 connected to the PIR's 'OUT' terminal with a 5V/GND power supply

Whilst many USB cameras will work with a Raspberry Pi, not every camera will necessarily work, especially if the camera needs any special software drivers.

For the various image taking and demonstration software methods discussed below, a simple low cost camera is used that is widely available, along with a small tripod, as shown above/right. This only has a very low maximum resolution of 640x480 pixels - but the example/demonstration web interface software has been evolved to optionally allow for higher resolution cameras.

Image streaming

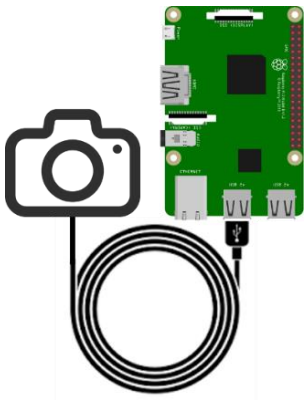
Two separate sets of Python software are available for using this streaming method, depending upon whether the 'root' user can be used (e.g., using sudo) or whether your Raspberry Pi is only set up for use by non-root users. For either method some additional set up is needed and this must be done by the root user. It should also be noted that the software used is relatively advanced and has primarily been provided as an initial set up utility. This method can therefore be returned to at any time, to check that a connected camera is working correctly and is producing images that are in focus and is properly oriented for all the other image taking methods described in this document.

It should be noted that the USB camera does need a reasonable amount of power, so if you have an early model of Raspberry Pi which uses a power supply with less than 2.5A you will need to use a powered USB hub for your camera USB connection, or you may get camera errors and unreliable/unstable performance.

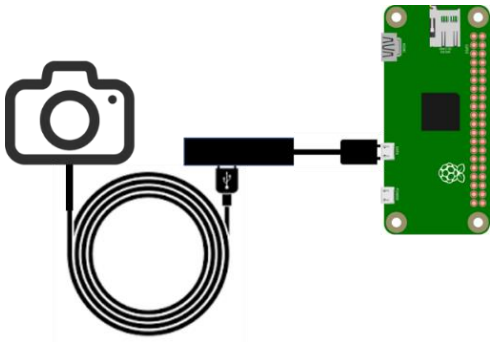
As can be seen in the schematics below if a Raspberry Pi Zero is being used or an older version of the Pi which may only have two USB ports, and you are not using a separate powered USB hub, you will need to have a multi-port USB adaptor. This is illustrated in the schematic below right where the standard sized USB plug on the camera is connected

to the micro USB socket on the Pi Zero via an adaptor, and so that you have at least one additional port to use a wireless or wired USB keyboard/mouse. An example adaptor shown above right can be sourced from **here**.

Raspberry Pi3



Raspberry Pi Zero



To check that your USB camera has been 'found' by the Pi use the 'lsusb' command in a Terminal window and, for example, you would see a GEMBIRD device like that shown below:

```
pi@rpikit01:~ $ lsusb
Bus 001 Device 050: ID 1908:2311 GEMBIRD
Bus 001 Device 003: ID 0424:ec00 Standard Microsystems Corp. SMSC9512/9514 Fast Ethernet Adapter
Bus 001 Device 002: ID 0424:9514 Standard Microsystems Corp.
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
pi@rpikit01:~ $
```

Then to check what resolutions are supported by the camera if you are using a USB camera of your own, use the following command:

```
v4l2-ctl --list-formats-ext -d /dev/video0 | grep Size
```

…. and you should see the following type of output:

```
pi@rpikit01:~ $ v4l2-ctl --list-formats-ext -d /dev/video0 | grep Size
                Size: Discrete 640x480
                Size: Discrete 352x288
                Size: Discrete 320x240
                Size: Discrete 176x144
                Size: Discrete 160x120
pi@rpikit01:~ $
```

Image streaming: software and setup

These streaming methods uses web software called 'Flask', which is what is known as a micro web framework. It is written in Python and is based on two other pieces of software called the Werkzeug toolkit and the Jinja2 template engine.

Flask provides a very flexible 'flow' arrangement that allows customised HTML web pages, created with HTML templates, to invoke specific sections of Python code, passing data to it - which in turn can 'call' other HTML web pages, passing data back to the HTML.

Details of the various software elements are provided in *Appendix D: Flask web server*, which will allow more advanced Python users to use the supplied code as a base for any further developments they may want to do. A key element of the overall approach is that

the streamed video essentially consists of a series of individual images taken by the camera that are 'presented' to the screen very rapidly.

Each image does however have to be stored, albeit on a temporary basis. To do this instead of using the Raspberry Pi's normal file system, which is typically a SD card that could rapidly deteriorate with the many thousands of read/write operations involved, a so-called 'ram drive' is used instead.

This technique creates a 'virtual drive' in computer memory that can be written to, and read from, as if it were a normal file system but is not only much faster but will not 'wear out' like a SD card. Details of how the 'ram drive' must be set up on a one-time basis by a user with root privileges are also provided in *Appendix E: Setting up a ram drive*.

What may be considered a quirk of the Flask system is that a browser that accesses the web server can only use the conventional HTTP port 80 if the main software is run by a user with root privileges. Two slightly different software programs are therefore made available for root and non-root users and for both cases it is recommended that the code is simply run from a 'Terminal' window instead of using the Thonny IDE.

Image streaming: user with root privileges

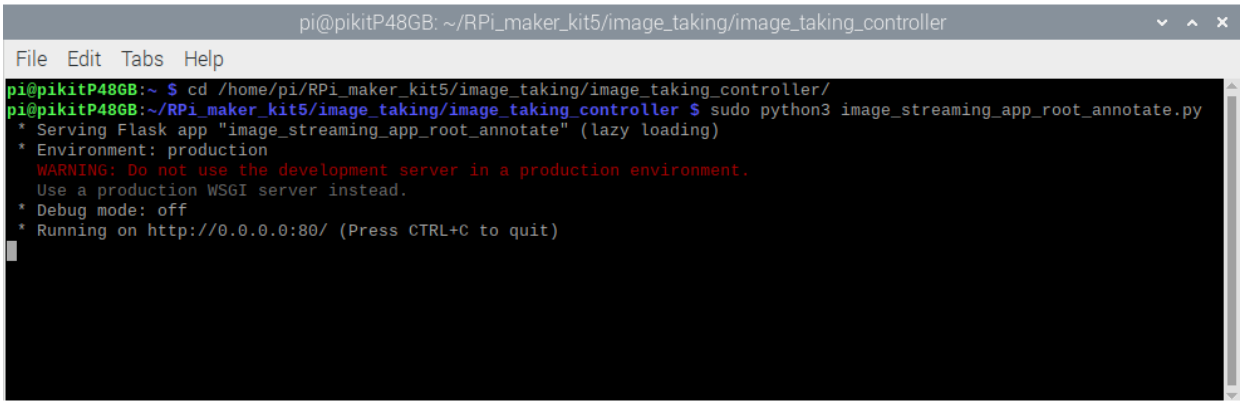
With your Raspberry Pi in 'Desktop' mode and connected to your local network, use a 'Terminal' window to run the ***image_streaming_app_root_annotate.py*** program that the software download process would have placed in the subdirectory:

./RPi_maker_PCB5/image_taking/image_taking_controller/

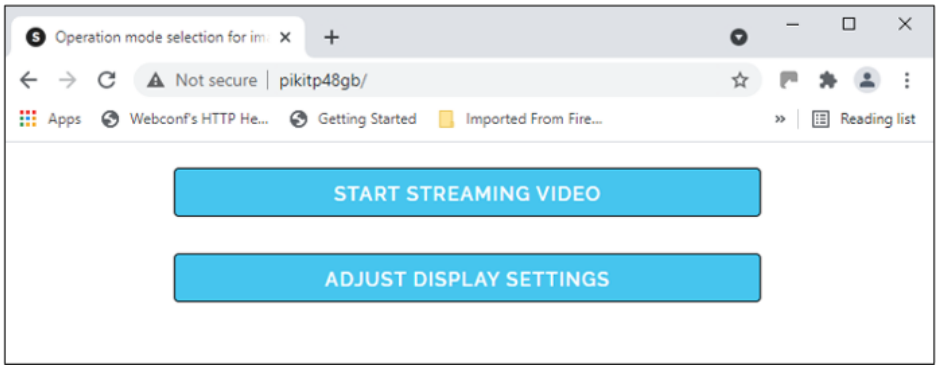
The sequence of commands to move to this directory and then run the image streaming code are as follows:

```
$ cd ./RPi_maker_PCB5/image_taking/image_taking_controller/
$ sudo python3 image_streaming_app_root_annotate.py
```

The screen shot below shows the Terminal window display when running on a Raspberry Pi with a host name of 'pikitP48GB':



Now either open the browser on your Raspberry Pi or use a browser on another computer that is also connected to your local network and go to the URL http://yourpihostname, where *yourpihostname* is whatever host name you have given your Raspberry Pi. Your local router should then connect the browser to your Pi and will show the screen shot below.



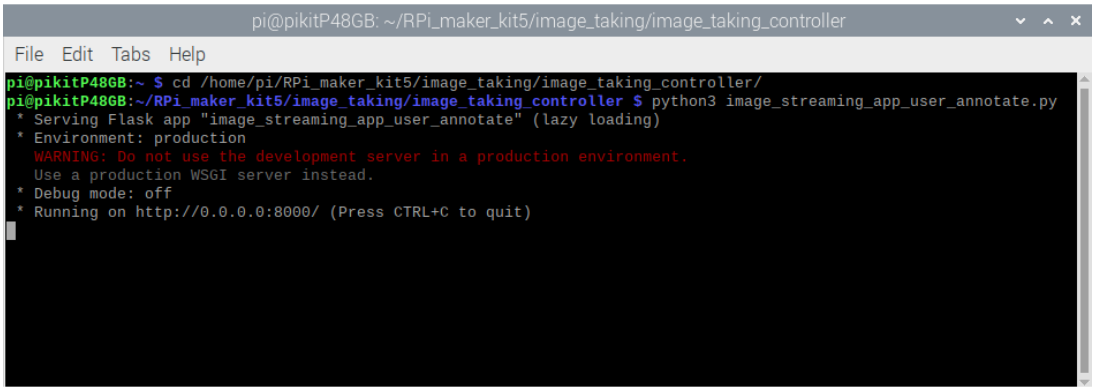
Now you can click the top blue button to start showing a streamed image from the USB camera.

Image streaming: user without root privileges

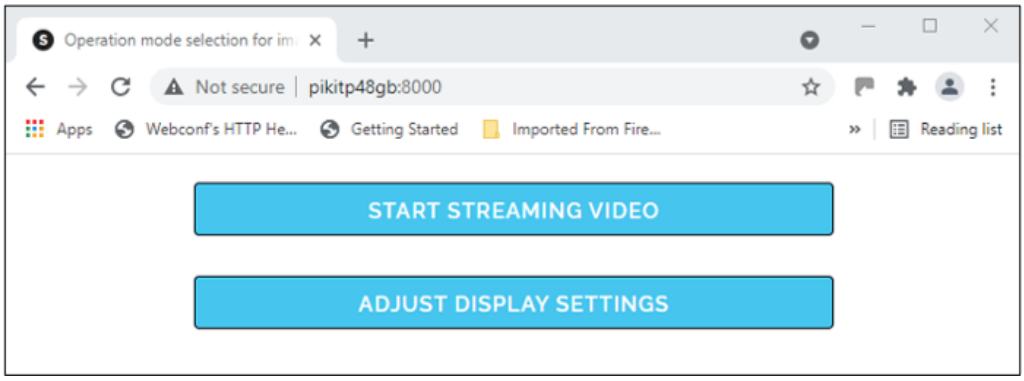
The Terminal window commands for a non-root privilege user are as below and as also shown in the screen shot below, i.e., you no longer have to use sudo:

$ cd ./RPi_maker_PCB5/image_taking/image_taking_controller/

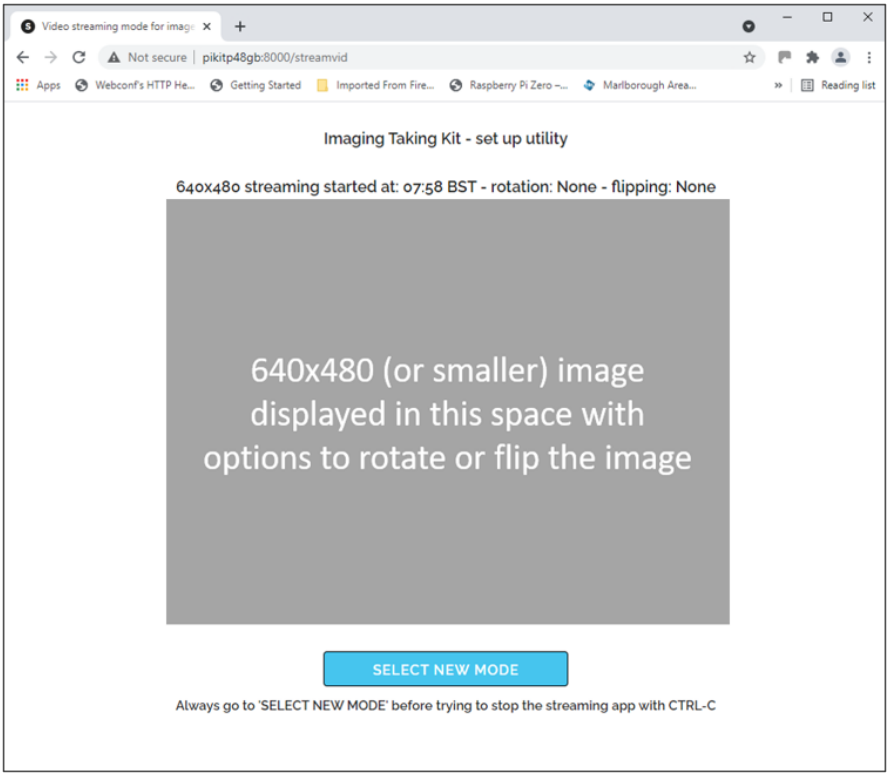
$ python3 image_streaming_app_user_annotate.py



The other significant difference for a non-root privilege user is that the resultant streamed images are now only shown in http port 8000, so the required URL to be typed into your browser is http://yourpihostname:8000, where *yourpihostname* is whatever host name you have given your Raspberry Pi, and the screen shot below shows the browser output for this non-root user case.



For either usage case (root or non-root), the screen shot below shows what the browser will display when the 'START STREAMING VIDEO' option is selected - where the grey rectangular area will be showing live streamed images from the camera.



The streamed display should be in focus – if not adjust the front bezel on the camera – it should usually be almost fully clockwise.

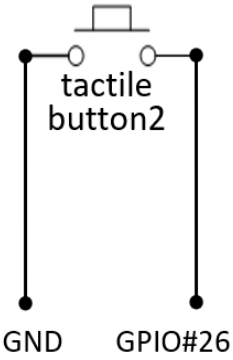
Then, depending upon how you have mounted your camera the image may need to be rotated so that it is oriented correctly in the display by either physically remounting the camera the right way round or changing the orientation 'in software'. To do this use the "SELECT NEW MODE" button to return to the main screen, and then click the "ADJUST DISPLAY SETTINGS" to go to this screen where you can set the rotation to 90°, 180° or 270°.

As you will see there are also some 'flip' options on this screen that you might want to use in some circumstances – as well as options to set different camera resolution sizes.

Once set up as you require, you should be seeing a reasonably fluid 'real time' streamed display with both the CPU temperature and the current time superimposed on the image at the bottom left and bottom right of the image, although there will be some 'lag' between what is shown and the real world.

You should note however that changing the display settings with this streaming method does not set the camera in this mode, it just adjusts what is displayed whilst streaming, so that you know what adjustments will be needed when you use the same camera set up with other image taking methods described next, most of which could be done using the Thonny IDE to run individual programs or you can continue to use direct commands in a Terminal window.

Button taking image



The circuit diagram shown on the right shows how this basic image taking method uses button 2 on the PCB, and by examining the ***button_take_image.py*** Python code you will see that it uses a software command called *fswebcam* to initiate the taking of a single image by the USB camera; the library for this command will have been installed for you with the rest of the software as part of the download process.

To run this code, use the Thonny IDE or alternatively type or copy/paste the full command shown at the top of the listing into a Terminal window, where a *venv* should have been automatically 'activated', i.e.

```
$ python3 ./RPi_maker_PCB5/image_taking/button_take_image.py
```

Once started, you will see a prompt to enter a subfolder name where the images will be stored, and then a second prompt to click button 2 to take an image or to type CTRL-C to stop the program.

When the button is clicked, the screen will then display a completion notification telling you that the image has been stored at:

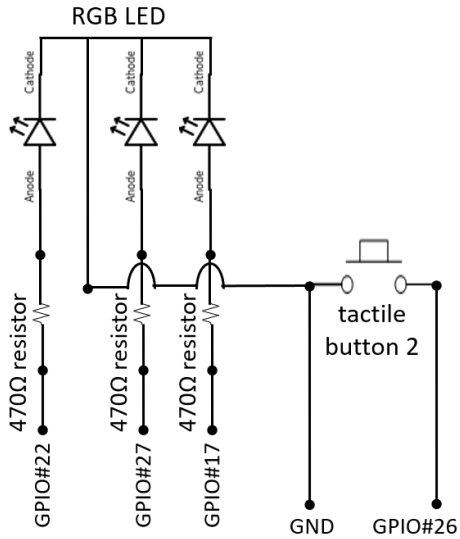
./RPi_maker_PCB5/image_taking/your_input_folder_name/

.. with a file name like *single_image_2021-05-03_10.03.25.jpg* where the numbers in the file name are the date (YYYY-MM-DD) and time (HH.MM.SS) that the image was taken.

You then have an option to briefly view the image on screen before the program repeats the '*button click prompt/take image*' cycle indefinitely until you type CTRL-C to stop it.

At any time, you can also use the Pi's File Manager and Image Viewer utilities to display on your screen any of the images taken.

Button taking image with LED indicator



The circuit diagram shown on the right shows how this image taking method uses button 2 and the RGB LED on the PCB, plus as shown in the ***button_led_take_image.py*** Python code it also uses *fswebcam* to take an image with the USB camera.

Whilst the program is starting up, the RGB LED is initially 'on' with a red light, and then once fully started, the LED turns green and you will see a prompt to enter a subfolder name where the images will be stored, and then a second prompt to click button 2 to take an image or to type CTRL-C to stop the program.

When the button is clicked, whilst the image is being taken, the LED turns blue. When image taking is complete, the LED returns to green and the screen will also display a completion notification telling you that the image has been taken and has been stored at:

./RPi_maker_PCB5/image_taking/your_input_folder_name/

.. with a file name like *single_image_2021-05-03_10.03.25.jpg* where the numbers in the file name are the date (YYYY-MM-DD) and time (HH.MM.SS) that the image was taken.

You then have an option to briefly view the image on screen before the program repeats the button click prompt/take image cycle indefinitely until you type CTRL-C to stop it.

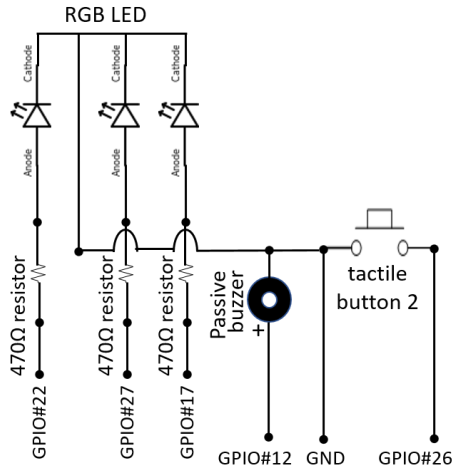
At any time, you can also use the Pi's File Manager and Image Viewer utilities to display on your screen any of the images taken.

To run this code, you can also use the Thonny IDE or alternatively type or copy/paste the full command shown at the top of the listing into a Terminal window, where a *venv* should have been automatically 'activated', i.e.

```
$ python3 ./RPi_maker_PCB5/image_taking/button_led_take_image.py
```

Timer taking image with LED and buzzer indicators

The circuit diagram shown on the right shows how this image taking method uses button 2, the RGB LED and the buzzer on the PCB, plus as shown in the ***button_timer_take_image.py*** Python code it also uses *fswebcam* to take an image with the USB camera.



Whilst the program is starting up, as with the previous method, the LED is initially 'on' with a red light, then once the program has fully started the LED turns green and you are prompted to enter a subfolder name where the images will be stored. You will then see a second prompt to enter a timer duration in seconds (which needs to be more than 5 seconds). Having entered a suitable value, you are then prompted to click the button to take an image or to type CTRL-C to stop the program.

When the button is clicked, the timer starts to count down and the LED flashes blue on and off, plus for the last 3 seconds of the timer duration the buzzer also beeps; with the LED then steadily on blue, the image is taken.

On completion of the image taking, the LED returns to green and the screen will also display a completion notification telling you that the image has been taken and has been stored at: *./RPi_maker_PCB5/image_taking/your_input_folder_name/*

.. with a file name like *single_image_2021-05-03_10.03.25.jpg* where the numbers in the file name are the date (YYYY-MM-DD) and time (HH.MM.SS) that the image was taken.

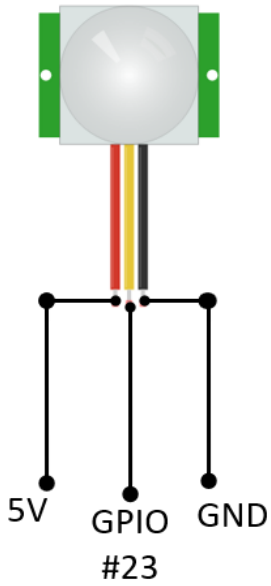
You then have an option to briefly view the image on screen before the program repeats the button click prompt/take image cycle indefinitely until you type CTRL-C to stop it.

As always, at any time, you can also use the Pi's File Manager and Image Viewer utilities to display on your screen any of the images taken.

To run this code, you can also use the Thonny IDE or alternatively type or copy/paste the full command shown at the top of the listing into a Terminal window, where a *venv* should have been automatically 'activated', i.e.

```
$ python3 ./RPi_maker_PCB5/image_taking/button_timer_take_image.py
```

PIR taking image



More details about Passive InfraRed (PIR) sensors are provided in *Appendix F: Control methods and 'plug in' component details*, but the circuit diagram shown on the right shows how this image taking method uses a PIR inserted into the dedicated 3 pin female connector on the PCB, to 'trigger' an image being captured by the USB camera when the sensor detects an object.

As shown in the ***PIR_take_image.py*** Python code this image taking method also uses *fswebcam and o*nce started you are prompted to enter a subfolder name where the images will be stored.

Then the screen shows a message that it is waiting for the electronics to settle. Thereafter whenever the PIR senses movement of a warm body within its range, an image is taken and stored in the designated folder.

There are no visual indications that an image has been taken other than the screen displaying a completion notification telling you that the taken image has been stored at:

./RPi_maker_PCB5/image_taking/your_input_folder_name/

.. with a file name like *single_image_2021-05-03_10.03.25.jpg* where the numbers in the file name are the date (YYYY-MM-DD) and time (HH.MM.SS) that the image was taken.

The program will continue cycling through its routines indefinitely, taking images whenever a warm body movement is detected until you type CTRL-C to stop it. Therefore, if you leave this program running for any length of time, you should make sure that you have sufficient storage space for all the images that may need to be stored.

You can at any time use the Pi's File Manager and Image Viewer utilities to display any of the captured images on your screen.

To run this code, you can use the Thonny IDE or alternatively type or copy/paste the full command shown at the top of the listing into a Terminal window, where a *venv* should have been automatically 'activated', i.e.

```
$ python3 ./RPi_maker_PCB5/image_taking/PIR_take_image.py
```

Video recording

Video clips can be recorded using a set of three Python programs that allow a short video to be captured where button 2, the RGB LED and buzzer are all used in a similar way to the previous single image capture methods.

The video clips can then be shown later by using the *VLC Media Player* application available from the 'Sound & Video' menu.

However, as with the use of the PIR triggered image taking method, available storage space on your SD card is something you need to consider for all the image taking methods but especially so for video capture.

Button taking video clip

The Python program ***button_take_video.py*** allows a video clip to be captured when tactile button 2 is pressed in a similar manner to how ***button_take_image.py*** is used to capture a single image.

To run this code, you can use the Thonny IDE or alternatively type or copy/paste the full command shown at the top of the listing into a Terminal window, i.e.

```
$ python3 ./RPi_maker_PCB5/image_taking/button_take_video.py
```

Button taking video clip with LED indicator

The Python program ***button_led_take_video.py*** allows a video clip to be captured using the RGB LED to indicate the video taking cycle when tactile button 2 is pressed in a similar manner to how ***button_led_take_image.py*** is used to capture a single image.

To run this code, you can use the Thonny IDE or alternatively type or copy/paste the full command shown at the top of the listing into a Terminal window, i.e.

```
$ python3 ./RPi_maker_PCB5/image_taking/button_led_take_video.py
```

Timer taking video clip with LED and buzzer indicators

The Python program ***button_timer_take_video.py*** allows a video clip to be captured after a timer countdown when tactile button 2 is pressed, with the RGB LED and the buzzer indicating the cycle progress in a similar manner to how ***button_timer_take_image.py*** is used to capture a single image.

To run this code, you can use the Thonny IDE or alternatively type or copy/paste the full command shown at the top of the listing into a Terminal window, i.e.

```
$ python3 ./RPi_maker_PCB5/image_taking/button_timer_take_video.py
```

Time lapse video

Time lapse photography is a fun way to present, in a video format, something that changes quite slowly over a period, but is shown 'speeded up' so that, for example, a duration of several days, or much longer, is shown in a few minutes. Some common examples of the uses of time lapse are capturing how plants grow, which can be shown in a fun way with a "cress egg heads" project, or the activity might be during a construction project, or just a 'day in a life' of a busy location.

A normal full speed video would typically be showing more than 30 frames per second (fps), meaning more than 30 individual images are captured and then played back every second to show fluid/jitter-free motion. In comparison, a time lapse video might consist of individual images of the scene being captured every 5 minutes and then played back at (say) 10fps. So, if for example, individual images of a scene were captured every 5 minutes for (say) 10 days this would result in a total of 2880 images – which would result in a video clip of less than 5 minutes if played back at 10fps.

Production of a time lapse video is therefore composed of the following two steps:

- capturing individual digital images of a scene, on a defined periodic basis, and then
- 'stitching' the individual images together into a digital video file format that can be 'played'

This image taking method uses the USB camera, mounted in a fixed position, and connected to the Raspberry Pi. Various software tools and methods are then used to carry out the two steps described above that are now discussed in more detail.

Individual image capture

To capture an image periodically you could run a Python program, similar to those used in the previous methods, on a continuous basis and wait for defined intervals between image taking cycles. A more reliable and consistent method however is to run a program that takes a single image and then stops – and then to set this program to run at periodic intervals for which there is a standard Linux process called 'cron' which will keep going even if the Raspberry Pi is rebooted.

To start your time lapse image capture process, you should simply connect the USB camera to the Raspberry Pi and power it up. Then set the camera up in a suitable way to 'frame' and capture the required scene. You might want to use method (1) to show a continuous initial view in a browser so that you get the camera positioned correctly and then make sure it is firmly secured since you will need to ensure that neither the camera nor the overall scene context, e.g., the flowerpot or egg cup holding a cress egg head, does not move, perhaps for several hours, or even for days, or weeks!

Two versions of the software used to take single individual images are provided in the downloaded set. The first, called *timelapse_cron_take_image.py,* is the simpler version and this just takes an image, whereas the second version called *timelapse_cron_take_annotated_image.py* adds some annotation text with a time stamp to each image.

Using the Pi's File Manager and Text Editor utility, you should look through the Python code for each of these two versions as they have additional comments throughout to explain what each section of the code does.

Next, decide whether to use the 'plain' or annotated image software version and run the program in a Terminal window to produce a single test image, by using the command that is shown towards the top of the listing, i.e., either

```
python3 ./RPi_maker_PCB5/image_taking/timelapse_cron_take_image.py
```

or

```
python3 ./RPi_maker_PCB5/image_taking/timelapse_cron_take_annotated_image.py
```

There are no visual indications that an image has been taken other than the screen displaying a completion notification telling you that the image has been taken and has been stored at *./RPi_maker_PCB5/image_taking/timelapse_image_folder/* with a file name like *single_image_2017-07-03_10.03.25.jpg* where the numbers in the file name are the date (YYYY-MM-DD) and time (HH.MM.SS) that the image was taken.

You can now use the Pi's File Manager and Image Viewer utilities to display the image on your screen to check that the individual captured image is OK.

Assuming you are happy with the camera location/set up, and the test image, we can now use 'cron' to schedule the running of this Python code on a regular basis.

Cron is a tool for configuring scheduled tasks on Linux systems like the Raspberry Pi OS (more info can be found **here**). It is used to schedule commands or scripts like our Python code to run periodically and at fixed intervals.

To add our time lapse routine to any existing scheduled items that may already be on your Pi, we can edit the root user's '*time schedule'* (or cron table shortened to crontab) by typing this command into an opened Terminal window:

```
crontab -e
```

This will open up the editor for the schedule, but if it has never been edited before you will be asked to select a specific editor (nano is recommended).

How to use an editor like nano, is not covered in this document, but there are plenty of online resources to help you with this.

To add a time lapse task just add two lines of text like those shown below (a small font has been used to show that these are continuous single lines). It should also be noted that because this command will be running in the background on the Pi it is necessary to explicitly set the full path to the Virtual Environment for the command and the full path to the code.

The first line is just some comments (signified by the line starting with a #) to document the second line.

```
# script to run every 5 minutes to take an image with a USB camera (adjust time as necessary)
*/5 * * * * /home/YOURUSERNAME/my_virtual_env/bin/python3
/home/YOURUSERNAME/RPi_maker_PCB5/image_taking/timelapse_cron_take_image.py >> /dev/null 2>> /dev/null
```

or an alternative pair of lines is:

```
# script to run every 5 minutes to take an image between 6:00am and 9:55pm (adjust times as necessary)
*/5 6-21 * * * /home/YOURUSERNAME/my_virtual_env/bin/python3
/home/YOURUSERNAME/RPi_maker_PCB5/image_taking/timelapse_cron_take_image.py >> /dev/null 2>> /dev/null
```

For each of these options, the second line is the scheduled task instruction using the format described below: where **/5* i.e., the 1st element of the five-element setting for the schedule, means every 5 minutes in the 0-59 minute range; and 6-21 or * sets the hours the command is run i.e., from 6am to 9pm or * for all hours.

```
* * * * *  command to execute
┬ ┬ ┬ ┬ ┬
│ │ │ │ └──── day of week (0 - 7) (0 to 6 are Sunday to Saturday,
│ │ │ │       or use names; 7 is Sunday, the same as 0)
│ │ │ └─────── month (1 - 12)
│ │ └──────────── day of month (1 - 31)
│ └────────────────── hour (0 - 23)
└──────────────────────── min (0 - 59)
```

*/5 * * * * therefore means every 5 minutes, in every hour, in every day of the month, in every month, in every day of the week. Please note that if for example you used */9 instead of */5 this would create a schedule that ran on the 0, 9th, 18th, 27th, 36th, 45th, and 54th minutes of the hour, and then this cycle repeats – so the last 'gap' is only 6 minutes. Therefore, to have a uniform gap between all the executed commands you do need to choose a number of minutes that can be multiplied up to 60 exactly.

*/5 6-21 *** then means every 5 minutes if the hour is anywhere from 6 to 21 i.e., 6:00am through to 9:00pm – but as */5 has also been set the last time will be 9:55pm.

The next part of the second line, the command to execute part, is obviously the command we used before when testing the code, but after this command there is also:

">> /dev/null 2>> /dev/null"

This is added at the end because the scheduled tasks will be run in the background i.e., we will not see any output that the code might produce or any errors that might be generated. In fact, the Raspberry Pi does not even need to be connected to a screen with a keyboard/mouse attached, it could be simply plugged in and powered up, which is known as running in "headless server" mode. In this context, this final additional text on the second line, tells the system to just 'dump' the program outputs and any system error outputs. This then avoids these outputs being directed to various standard system log files which could get filled up unnecessarily.

Finally, please note that time lapse files will start being produced immediately after the cron schedule is updated and will continue indefinitely until you update the schedule again to stop the scheduled task. To stop the task, just edit the crontab again and the simplest option is to just add a # to the front of the second line to make it a comment – this will then let you easily re-instate it at any time without having to retype lots of text.

You do therefore need to decide how long you want to capture time lapse images for, to make sure you stop it at the appropriate time, but you also need to make sure that your Raspberry Pi has enough spare storage capacity to store what might be many thousands of reasonably sized .jpg images.

Creating a video from individual images

You now have a set of time lapse images stored in the Raspberry Pi at the *./RPi_maker_PCB5/image_taking/timelapse_image_folder/* and they all have an explicit date/timestamp built into their file names.

To 'stitch' these together into a video we can now use the *ffmpeg* system utility which will have been installed for you as part of the download script. This utility has a set of usage parameters that are exactly for this purpose, but unfortunately the files that *ffmpeg* can stitch together need to be in time sequence order with file names that include a simple sequential number i.e., 0000, 0001, 0002, etc. To achieve this, we now need to run a small custom utility program that is provided with the downloaded files in a Terminal window with the following command:

python3 sort_number_symlink_files.py

This utility will look at the folder where all your individual time lapse images have been stored and first of all make sure they are sorted in chronological order. Then, one file at a time, it creates in a new folder, equivalent file names that are called symbolic links to each of the real files using a much-simplified file name of NNNN_image.jpg where NNNN is incremented from 0000 onwards to the last file number. In this way, we create a set of file names that satisfy the requirements of using *ffmpeg*, without copying all the existing files, which would double the required storage - or renaming the existing files, which would lose all their explicit timestamp information.

With this new set of symbolic link files, we can now use *ffmpeg* to do the 'stitching together', and as this is easily done just by executing a single command in a Terminal window, there is not any specific Python software provided to do this process.

An *ffmpeg* command has many different options but the most basic usage for the file names we are using looks like:

ffmpeg -i '/path_to_symlinks_folder/%04d_image.jpg' /path _to_output_video/yourvideoname.mp4

The small font used above is just to show that this is one contiguous set of text, but let's break the command into its different sections:

-i *indicates that the input stream is defined next*

***'/path_to_symlinks_folder/%04d_image.*jpg'** is the path location and file name 'structure' for the input stream of individual images. So, for this method the path will be something like *./RPi_maker_PCB5/image_taking/timelapse_symlink_folder,* where the set of symlinks will have been created with 4-digit sequential numbers i.e., 0000_image.jpg, 0001_image.jpg, etc., which is indicated by the ***%04d_image.*jpg** part of the command.

/path_to_output_video/yourvideoname.mp4 *is where the output video should be created and its file name*

Stop motion video recording

Stop motion video or stop motion animation, is a very creative and fun way of producing video content that technically is very similar to the time lapse method. In recent years, it has become a very popular cartoon production method e.g., Aardman Animations' Wallace & Grommet characters, and many others.

Production of a stop motion video is composed of the following two steps:

1. capturing a sequence of individual digital images of a scene, that is altered manually by a very small amount, and then
2. 'stitching' the individual images together into a digital video file format that can be 'played'

In this way, controlled motion of individual objects can be 'constructed' so that inanimate objects can be made to look as if they are moving around.

Individual image capture

To capture the individual images, either the "Button taking image" or the "Button taking image with LED indicator" method can be used to systematically collect a series of still images of a scene, storing the images in a dedicated folder for your stop motion video project.

Care needs to be taken to make sure that the camera is firmly secured and not moved when taking each image, that only the objects being 'animated' are moved a tiny amount for each image with the overall scene otherwise kept completely still, and that the lighting for each image also remains the same.

Creating a video from individual images

You now use exactly the same method to 'stitch' together the individual images, as was used for the time lapse video method, i.e., you:

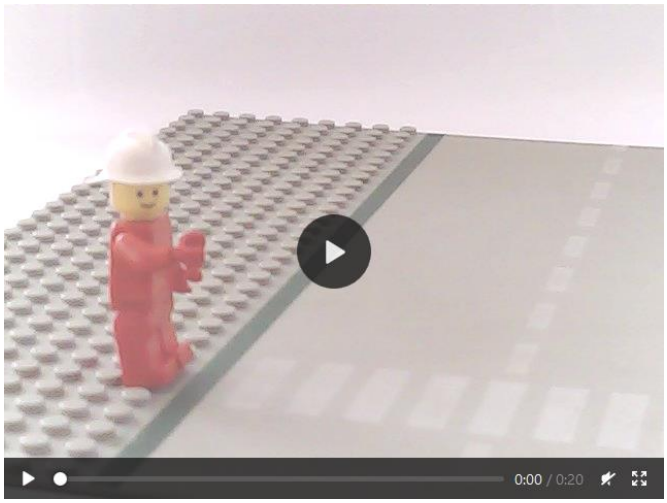
Use the *sort_number_symlink_files.py* software to 'sort' your series of individual still images and to create a new folder of corresponding symbolic link files that have a simplified file name with an ascending number element starting at 0000

Then you use *ffmpeg* to do the 'stitching together' just by running the following command in a Terminal window.

ffmpeg -i '/path_to_symlinks_folder/%04d_image.jpg' /path _to_output_video/yourvideoname.mp4

To illustrate what can be quickly produced using this method, click the image shown on the right to link through to a web page that will show a short video clip – where you will also see some of the 'errors' that are all too easily made! i.e., for this test video, the overall scene moved slightly for some individual shots and the lighting differed from shot to shot.

I'm sure you will be able to do much better!



Motor control

The Raspberry Pi PCB can support many different methods for controlling three main types of electric motor, i.e., servo, stepper, and drive motors.

The simple control examples profiled in the following sections can be used to help develop more advanced projects, some of which are illustrated in each section.

Servo motor control

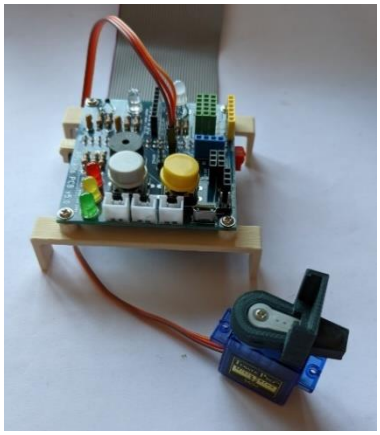
The Raspberry Pi PCB can support two main methods for controlling servo motors and the components and GPIO connections used in a populated PCB are set out in the table below.

In all the worked examples the low cost, and readily available, SG90 servo as shown on the right has been used and more detail about servos are provided in *Appendix F: Control methods and 'plug in' component details*.

Component	GPIO pin(s)
directly connected servo on the S1 servo connector	GPIO #24 plus 5V and GND connections all provided on the 3 pin male S1 connector
if a 2nd directly connected servo is needed the S2 servo connector can be used	GPIO #25 plus 5V and GND connections all provided on the 3 pin male S2 connector
Tactile button (1)	GPIO#07 and GND connections across the button
Tactile button (2)	GPIO#26 and GND connections across the button
PCA9685 PWM control board	GPIOs SDA & SCL plus 3V3 and GND connections all provided on the PCB's dedicated 6 pin female connector that 'matches' the 6 pins on the PCA9685 board

The set of components tabulated above allows many different types of movement control to be programmed for which example 'starter' Python code is made available. The details below summarise just some of the control options that are possible for which the starter code enables more complex projects to be developed.

Simple button controlled single servo



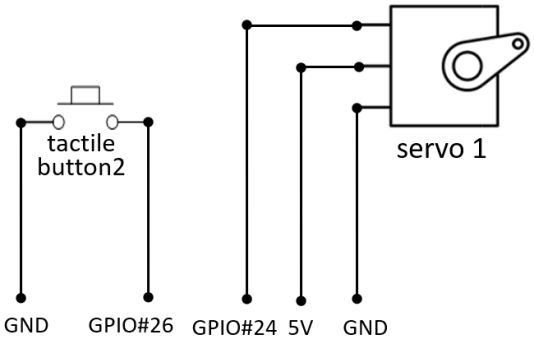
This first method shows how a servo can be controlled by directly connecting it to the Raspberry Pi, although whilst this is OK with a single, or at most two, low powered servos, it is generally not recommended since voltage fluctuations due to multiple servo operations can affect the stability of the operation of the Raspberry Pi itself.

The image on the right shows a SG90 servo connected to the dedicated S1 servo 3 pin male connector which has its 3 pins aligned to the standard SG90 servo cable female connector, i.e.,

- orange signal wire – servo1 pin
- red supply voltage +'ve wire – 5V pin
- brown supply –'ve wire – GND pin

In the image shown the servo has one of its standard 'horns' fitted to the splined gear box drive shaft to connect a 3D printed holder used in another test; this is just so that the servo movement is more clearly visible.

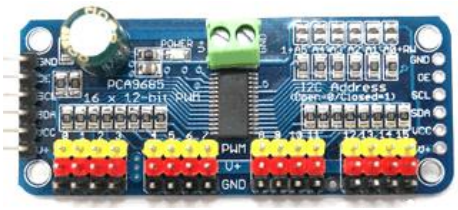
The circuit diagram shown on the right shows how this basic servo control method uses button2 on the PCB, and by examining the ***simple_servo_btn.py*** Python code you will see that a button press initiates Pulse Width Modulation (PWM) being applied to GPIO#24 to control the servo as explained in more detail in *Appendix F: Control methods and 'plug in' component details*.



The code for this simple test routine is part of the main software download and to run this code, you can use the Thonny IDE or alternatively type or copy/paste the full command shown at the top of the listing into a Terminal window, i.e.

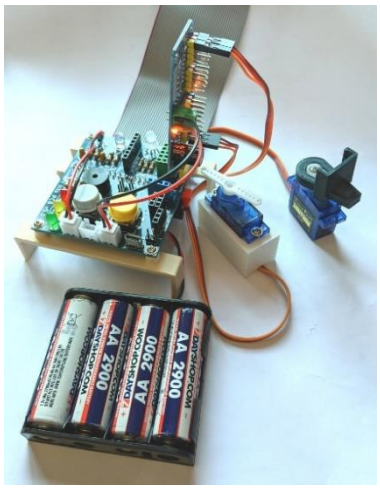
```
$ python3 ./RPi_maker_PCB5/motor_control/servo_motors/simple_servo_btn.py
```

I²C button controlled one or two servos



This second method uses a PCA9685 I²C Pulse Width Modulation (PWM) control board as shown left.

These boards are readily available and more details about this type of board are provided in *Appendix F: Control methods and 'plug in' component details*, but in summary the board takes care of the PWM in a much more electrically stable way using hardware and a separate power supply, than can be produced by the Raspberry Pi using only software and its 'internal' power lines, and it can also separately manage up to 16 individual servos.

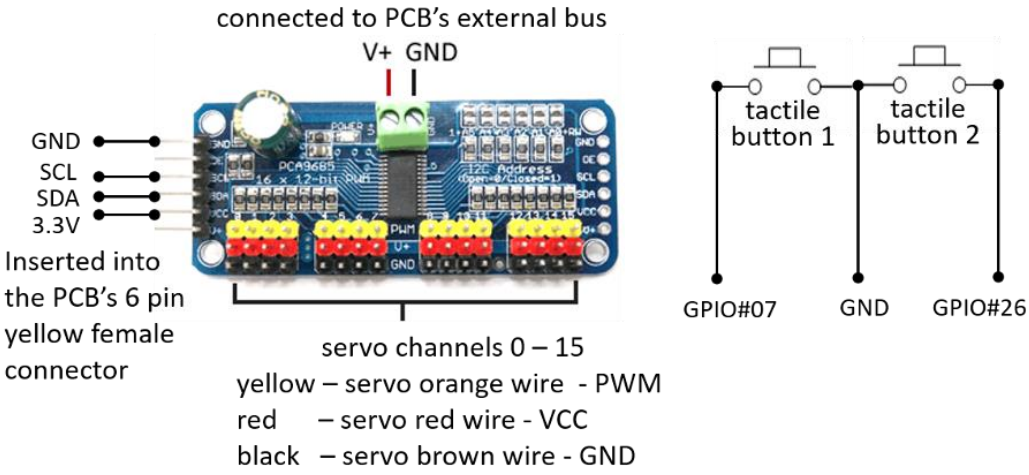


The image on the right shows how the board can be inserted into its dedicated 6-pin yellow female connector on the assembled PCB.

Servos can then be connected to the PCA9685 board (two shown in this example).

Using JST connectors, a power supply can then be connected to the 'external' power bus on the PCB (4xAA batteries in this example) and the PCA9685 board's external power connection cross-connected to the PCB's power bus.

The schematic/circuit diagram below shows in some more detail how this more sophisticated servo control method uses either button1 or button2 on the PCB, and by examining the ***I2C_servo_btn.py*** or ***I2C_2servo_btn.py*** Python code you will see that that button presses initiate Pulse Width Modulation (PWM) being applied to the servos by sending I²C control signals to the PCA9685 board.



The PCA9685 board has 6 pins which with its usual set of male connectors can be directly inserted into the dedicated 6-pin female connector on the assembled PCB, making the connections through to the Raspberry Pi as shown in the table below.

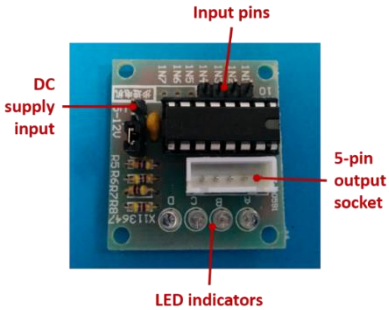
GND	Any GPIO GND pin
OE (Output Enable)	Not used
SCL (serial clock line)	SCL GPIO pin (GPIO pin#03)
SDA (serial data line)	SDA GPIO pin (GPIO pin#02)
VCC (3.3V supply for the control board)	3.3V GPIO pin
V+ (5V supply for the devices)	Not used because an external battery supply is used

The code for these I^2C test routines is part of the main software download and to run the code, you can use the Thonny IDE or alternatively type or copy/paste the full commands shown at the top of the listing into a Terminal window, i.e.

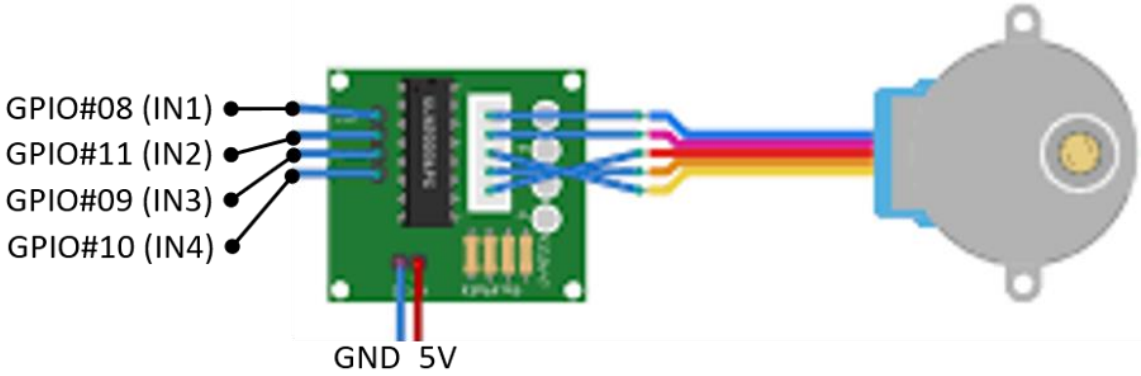
```
$ python3 ./RPi_maker_PCB5/motor_control/servo_motors/I2C_servo_btn.py
```
or
```
$ python3 ./RPi_maker_PCB5/motor_control/servo_motors/I2C_2servo_btn.py
```

Stepper motor control

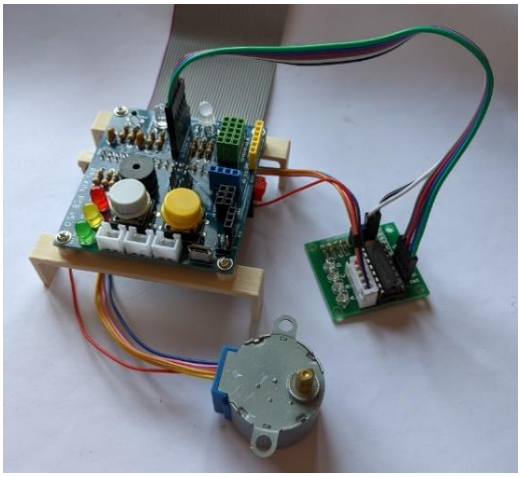
This motor control method uses the low cost 28BYJ-48 stepper motor with its associated ULN2005 control board that are shown below and for which more details of these readily available components are provided in *Appendix F: Control methods and 'plug in' component details*.



The schematic/circuit diagram below shows how this 28BYJ-48 stepper control method uses the INI-IN4 connections that are on the 7-pin black female connector on the assembled PCB, and by examining the Python code ***wave_drive_stepper.py, half_step_stepper.py*** and ***full_step_stepper.py*** you will see how the connected GPIO pins are sequenced HIGH/LOW in the ways needed to energise/de-energise the stepper's coils.



The stepper motor can usually be powered direct from the Raspberry Pi, assuming it is supplied by an adequate 5V DC power supply with the recommended 2.5A capacity for the Raspberry Pi 3, or 3A capacity for the Raspberry Pi 4.



For this direct powered arrangement, the image on the right shows how the GND-5V connections for the ULN2005 control board can also be fed from the 7-pin black female connector on the PCB.

However, if there is any doubt about the power supply stability, or problems are experienced, then the ULN2005 can be separately powered from the PCB's 'external' power bus using 4xAA batteries instead.

The code for these stepper motion test routines is part of the main software download and to run the code, you can use the Thonny IDE or alternatively type or copy/paste the full commands shown at the top of the listing into a Terminal window, i.e.

```
$ python3 ./RPi_maker_PCB5/motor_control/stepper_motors/wave_drive_stepper.py
```

or

```
$ python3 ./RPi_maker_PCB5/motor_control/stepper_motors/half_step_stepper.py
```

or

```
$ python3 ./RPi_maker_PCB5/motor_control/stepper_motors/full_step_stepper.py
```

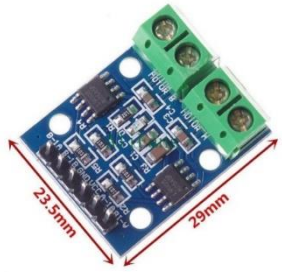
Drive motor control

Conventional 'drive' motors, controlled by a Raspberry Pi, are very commonly used in many projects e.g., the various types of 'wheeled' robots it is possible to build.

The PCB can be used to prototype/demonstrate 'drive' motor control and the examples below describe the use of very commonly used 5-6V geared motors controlled by two different types of motor controller.

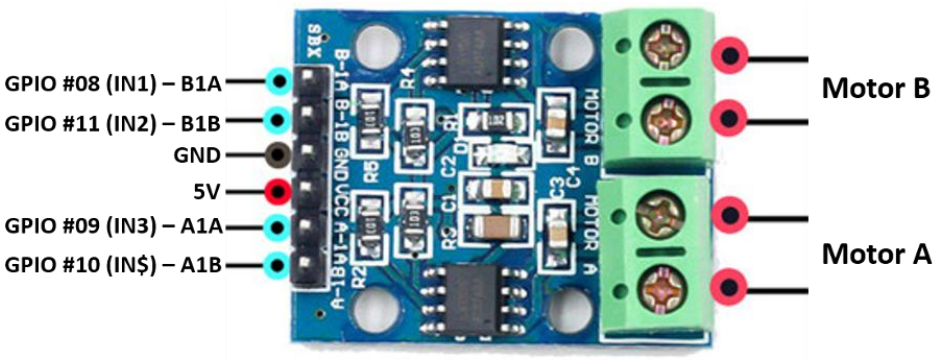
For each use case Python software is available to demonstrate all the usual motor control methods that would be needed for a 'wheeled' robotics project.

HG7881 motor controller

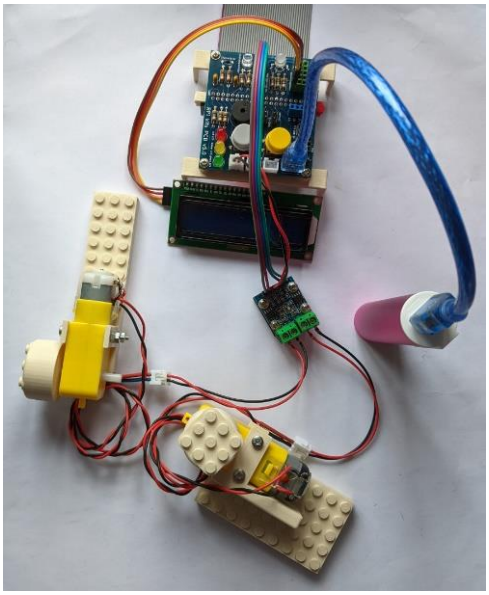


The HG7881 motor controller, as shown on the left, is a small and relatively basic 2-channel device for controlling two electric drive motors. The board and its controlled motors can be powered from a source that can supply from 2.4 to 10.0 volts DC and each motor can be supplied with a maximum of 800mA continuously.

The small size of this board, with its range of voltage/current capabilities, make it a particularly useful module for use in robotics 'making' projects.



The schematic/circuit diagram above shows how this HG7881 motor control method uses the INI-IN4 connections that are on the 7-pin black female connector on the assembled PCB



As shown in the image on the left, an HG7881 motor controller can be connected to the Raspberry Pi PCB to independently control two drive motors.

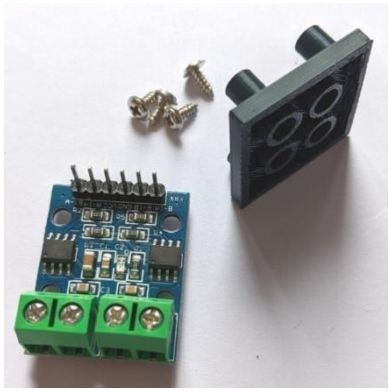
Power for the motor controller and the two drive motors can be supplied from either a 4xAA battery pack, or as shown, from a 5V USB battery bank.

By examining the two available versions of Python code, i.e., ***HG7881-motors_LCD_PWM.py*** and ***HG7881-motors_LCD_PWM.py*** you will see how motor control functions have been developed that set the IN1-IN4 GPIO pins appropriately.

For this test/demonstration build option the HG7881 control pins are 'reusing' the stepper motor IN1-IN4 input pins on the PCB, with power to the motor controller and the drive motors being fed from the PCB power bus.

The drive motors shown above have been mounted on custom 3D printed LEGO compatible blocks so that they can be incorporated into an overall LEGO 'mechatronic' construction where parts of the build can be made to move with the motors.

A 16x2 LCD has also been connected to the PCB (see later for details) so that the controlling Python code can also display the status of the 'drives' during the various running modes.



The HG7881 has also been mounted on a small 3D printed custom designed LEGO compatible 'tile', as shown on the right, so this too can be securely incorporated into an overall LEGO construct.

The code for these motor controller test routines is part of the main software download and to run the code, you can use the Thonny IDE or alternatively type or copy/paste the full commands shown at the top of each listing into a Terminal window, i.e.

For full PWM control of each motor use:

```
$ cd ./RPi_maker_PCB5/motor_control/drive_motors/HG7881_motor_controller/
```

.. followed by:

```
$ python3 HG7881-motors_LCD_just_PWM.py
```

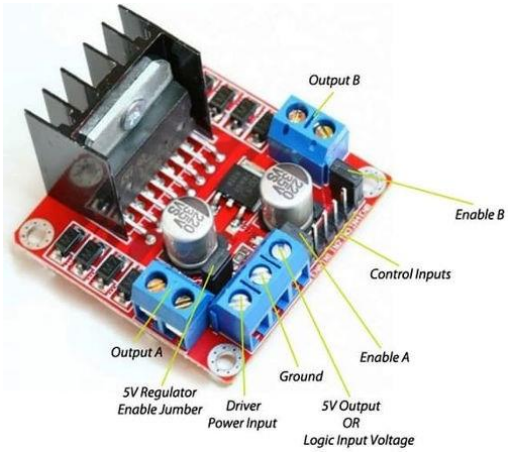
Or for simple on/off control of each motor use:

```
$ cd ./RPi_maker_PCB5/motor_control/drive_motors/HG7881_motor_controller/
```

.. followed by:

```
$ python3 HG7881-motors_LCD_on_off.py
```

L298N motor controller

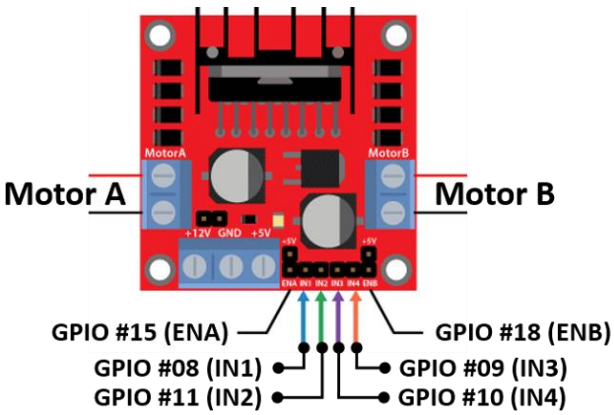


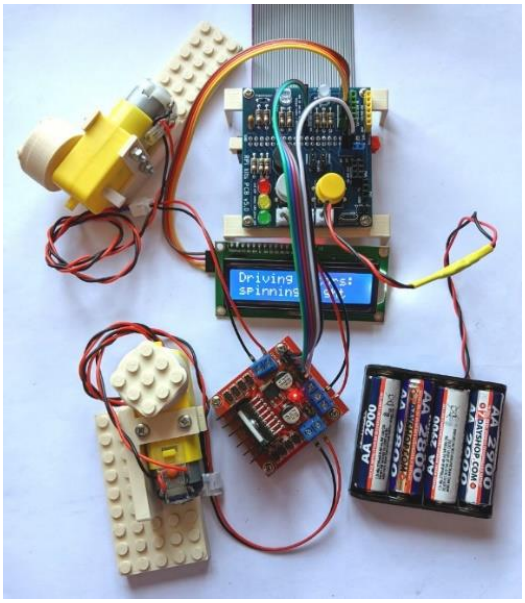
The L298N motor controller board, as shown on the left, provides a wide range of capabilities for controlling two electric motors.

The motors can be powered from a source that can supply from 5 to 35 volts DC and each motor can be supplied with 2A continuously. The board itself is powered from the same supply source with the voltage automatically stepped down to 5 volts DC and when powered correctly the L298N board has a red LED switched on.

A very useful capability is that if the input power comes from anything that supplies no more than 12 volts DC, then a board 'jumper' enables an output feed of 5 volts DC which can be used to power the controller e.g., a Raspberry Pi, Arduino board, etc.

The schematic/circuit diagram on the right shows how this L298N motor control method uses the INI-IN4 connections that are on the 7-pin black female connector plus two of the Spare GPIOs on the assembled PCB and by examining the ***L298N_motors_LCD_PWM.py*** and ***L298N_motors_LCD_on_off.py*** Python code you will see how motor control functions have been developed that set the IN1-IN4 and 'spare' GPIO pins appropriately.



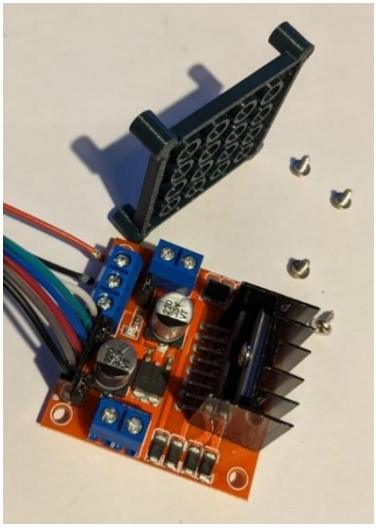


As shown in the image on the left, a L298N motor controller can be used to independently control two drive motors when connected to the PCB, with power for the motor controller and the two drive motors supplied either by a 4xAA battery pack, as shown, or from a 5V USB battery bank.

For this test/demonstration build option, the L298N IN1-IN4 control pins are 'reusing' the PCB's stepper motor IN1-IN4 input pins on the PCB and the L298N's ENA + ENB pins are connected to the PCB's 'spare' GPIO #15 and #18 connectors. Power input to the motor controller is then fed from the PCB power bus and the drive motors' power supplied from the L298N in the usual way.

A 16x2 LCD has also been connected to the PCB (see later for details) so that the controlling Python code can also display the status of the 'drives' during the various running modes.

The drive motors shown here have been mounted on custom 3D printed LEGO compatible blocks so that they can be incorporated into an overall LEGO 'mechatronic' construction where parts of the build can be made to move with the motors.



The L298N has also been mounted on a 3D printed custom designed LEGO compatible 'tile', as shown on the right, so this too can be securely incorporated into an overall LEGO construct.

The code for these motor controller test routines is part of the main software download and to run the code, you can use the Thonny IDE or alternatively type or copy/paste the full commands shown at the top of each listing into a Terminal window, i.e.

For full PWM control of each motor use:

```
$ cd ./RPi_maker_PCB5/motor_control/drive_motors/L298N_motor_controller/
```

.. followed by:

```
$ python3 L298N_motors_LCD_PWM.py
```

Or for simple on/off control of each motor use:

```
$ cd ./RPi_maker_PCB5/motor_control/drive_motors/L298N_motor_controller/
```

.. followed by:

```
$ python3 L298N_motors_LCD_on_off.py
```

Display projects

The Raspberry Pi PCB can be used to explore I²C, and SPI (Serial Peripheral Interface) controlled display devices, as well as simpler 7 segment LED displays as described in more detail in the following sections.

I²C managed displays

The Raspberry Pi PCB has been specifically designed to support multiple devices more easily when managed through the Pi's I²C interface. The assembled PCB has three 'general' 5-pin green connectors that provide access to 5V-GND-3V3-SDA-SCL, in addition to the dedicated PCA9685 6-pin yellow connector.

Both 5V and 3V3 connections have been made available on these 5-pin connectors so that devices that need either 5V or 3V3 to power them can be used - but as discussed below, some consideration and care must be taken to ensure that any device with its own pull-up resistors between its SDA and SCL connections and its 5V supply line are disabled - and if several devices are being managed on the I²C bus simultaneously, then the 'accumulative' pull-up resistance needs to be judged/calculated to ensure sufficient 'pull-up' is provided to the combined SDA and SCL lines into the Raspberry Pi.

This section describes the use of two different types of display that each use the I²C and addresses some of the pull-up issues discussed above.

16 character x 2 line LCD

A liquid crystal display (LCD) uses the light-modulating properties of liquid crystals combined with polarizers. Liquid crystals do not emit light directly, instead they use a backlight or reflector to produce images in colour or monochrome.

The commonly available monochrome 16 character x 2 line LCD, with an I²C 'backpack' for control, is powered by 5V, so some care is needed to ensure that the voltages on the SDA and SCL control lines do not exceed 3.3V as this would potentially damage the Raspberry Pi to which it is connected.

These LCD devices have internal 4.7kΩ pull-up resistors between both the SDA and SCL control lines and their supply voltage of 5V - so the safest way to use these displays when connected to and controlled by a Raspberry Pi, is to remove these pull-up resistors as illustrated in the image above. The Raspberry Pi's internal pull-up resistors are then sufficient to ensure the correct electrical operation of the I²C signalling.

The four I²C connection leads (GND, VCC, SDA and SCL) can then be plugged into one of the 5-pin I²C (green) female connectors on the PCB using the 5V port for the supply voltage, as shown in the image on the left.

Once connected to the I²C bus the address for these 16 character x 2 line LCD devices is usually 0x27 and this is the assumed default in the Python code used to demonstrate these devices.

The code for these display demonstrations is part of the main software download and to run the code, you can use the Thonny IDE or alternatively type or copy/paste the full commands shown at the top of each listing into a Terminal window, i.e.

For a simple demonstration use:

```
$ python3 ./RPi_maker_PCB5/displays/LCD_1602_i2c_display/hello_world.py
```

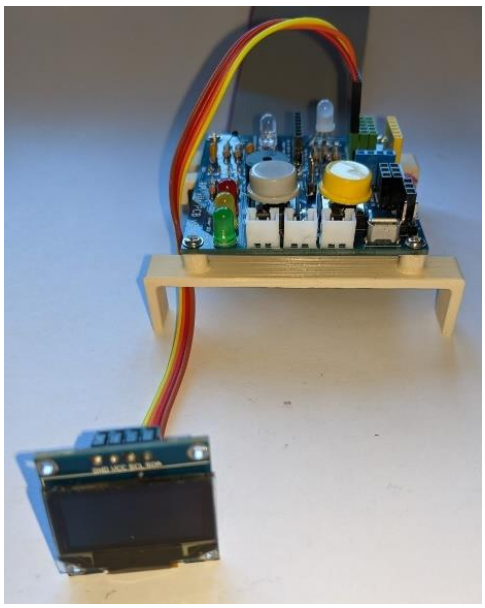
or for a more detailed demonstration use:

```
$ python3 ./RPi_maker_PCB5/displays/LCD_1602_i2c_display/ LCD_all_functions_demo.py
```

64x128 pixel OLED

An organic light-emitting diode (OLED or organic LED), also known as organic electroluminescent (organic EL) diode, is a light-emitting diode (LED) in which the emissive electroluminescent layer is a film of organic compound that emits light in response to an electric current. These displays are very attractive since, as they emit visible light, they can operate without a backlight and allow deep black levels to be displayed. They can also be made much thinner and lighter than Liquid Crystal Displays (LCDs).

The commonly available, small 128x64 pixel OLED I²C managed display, shown on both sides in the image on the right, is powered by 3V3 so does not cause any concerns about too high a voltage on the Raspberry Pi's SDA or SCL control lines.



It therefore works OK directly connected to any of the 5-pin I²C (green) female connectors on the PCB as shown in the image on the left - using only the 3V3 port of course!

The display can be used for text or images, or a combination of both, with Python software used to 'compose' a 'bit image' of whatever content is required that is then displayed as shown in the example in the image on the left.

Once connected to the I²C bus the default I²C address for these displays is usually 0x3c (and this may be the address despite what may be printed on your device!!)

The code for these display demonstrations is part of the main software download and to run the code, you can use the Thonny IDE or alternatively type or copy/paste the full commands shown at the top of each listing into a Terminal window, i.e.

```
$ python3 ./RPi_maker_PCB5/displays/OLED_128x64_I2C/simple_text.py
```

or: *$ python3 ./RPi_maker_PCB5/displays/OLED_128x64_I2C/shapes.py*

or: *$ python3 ./RPi_maker_PCB5/displays/OLED_128x64_I2C/image.py*

or: *$ python3 ./RPi_maker_PCB5/displays/OLED_128x64_I2C/animate.py*

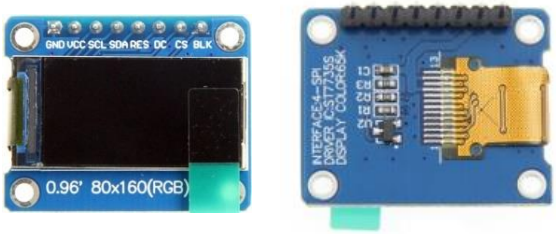
SPI managed displays

The Serial Peripheral Interface (SPI) is a synchronous serial communication interface specification used for short-distance communication that can be used to connect devices to a Raspberry Pi using the PCB. Compared to I²C, which is a relatively low speed bus, designed primarily for transferring small amounts of data, e.g., for sensors etc., SPI is a higher speed interface which requires more wires but can be better for supporting some types of display that require more data to be transmitted such as larger pixel sizes or multi-colour devices such as those described below.

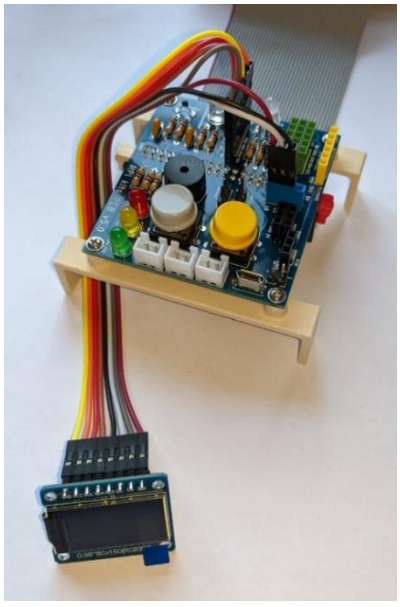
The devices below are described as IPS TFT, where:

- Thin Film Transistor (TFT) is a variant of a liquid crystal display (LCD) that uses thin-film-transistor technology to improve image qualities such as addressability and contrast, and
- In-Plane-Switching (IPS) is an additional improvement that increases the viewing angle as well as colour quality.

80x160 IPS display



This commonly available 0.96-inch and 65k colour display with an 80x160 pixel resolution, as shown front and back on the right, uses the ST7735S integrated circuit chip and can be connected to the PCB using both the 7P black female connector and three of the 'spare' GPIO pins.



Display pin	PCB connection
GND	7P connector GND
VCC	7P connector 3V3
SCL (SPI clock)	7P connector SCLK
SDA (SPI write data)	7P connector MOSI
RES (reset)	GPIO spare 14
DC (display control)	GPIO spare 15
CS (SPI control)	7P connector CE0
BLK (backlight)	GPIO spare 18

The PCB's 7P connector provides access to the SPI0 interface on the Raspberry Pi and all of the display connections are as set out in the table above with a test set up shown in the image above right.

The ***scrolling-text.py***, ***shapes.py***, ***image.py*** and ***gif.py*** code for these display demonstrations is part of the main software download and to run the code, you can use the Thonny IDE or alternatively type or copy/paste the full commands shown at the top of each listing into a Terminal window, i.e.

```
$ python3 ./RPi_maker_PCB5/displays/IPS_80x160_SPI/scrolling-text.py
```

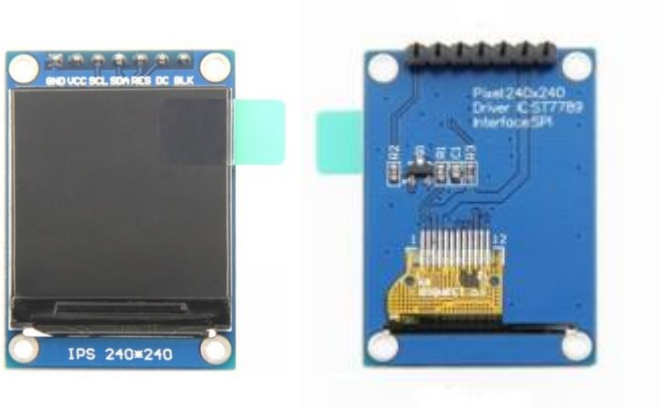
or: `$ python3 ./RPi_maker_PCB5/displays/IPS_80x160_SPI/shapes.py`

or: `$ python3 ./RPi_maker_PCB5/displays/IPS_80x160_SPI/image.py`

or: `$ python3 ./RPi_maker_PCB5/displays/IPS_80x160_SPI/gif.py`

240x240 IPS display

This commonly available 1.3-inch and 65k colour display with a 240x240 pixel resolution, as shown front and back on the right, uses the ST7789 integrated circuit chip and can be connected to the PCB using both the 7P connector and three of the 'spare' GPIO pins.



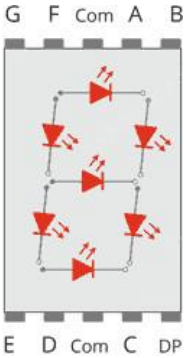
Display pin	PCB connection
GND	7P connector GND
VCC	7P connector 3V3
SCL (SPI clock)	7P connector SCLK
SDA (SPI write data)	7P connector MOSI
RES (reset)	GPIO spare 14
DC (display control)	GPIO spare 15
BLK (backlight)	GPIO spare 18

The PCB's 7P connector provides access to the SPI0 interface on the Raspberry Pi and all of the display connections are as set out in the table above with a test set up shown in the image on the right.

The ***1.3_IPS_LCD.py*** code for this display demonstration is part of the main software download and to run the code, you can use the Thonny IDE or alternatively type or copy/paste the full commands shown at the top of each listing into a Terminal window, i.e.

```
$ python3 ./RPi_maker_PCB5/displays/IPS_240X240_SPI/source/1.3_IPS_LCD.py
```

7 segment LED displays



7 segment LED displays are one of the simplest types of display that are commonly used, and they have been in use for many decades in digital clocks, basic calculators and any devices that need a basic numerical display.

Each digit of a display is made up of 7 LEDs, as shown in the image on the left, where an eighth LED may be used for a decimal point or for a separate colon in a multi-digit construct, and all the LED's anodes (or the cathodes depending upon the display type) are grouped together to a common connector.

An individual digit segment can be 'lit' by turning its supply pin HIGH just as is shown in the previous *Electronic basics* section of this document, but obviously this becomes quite complex when forming individual characters by lighting sets of segments and this complexity multiplies when multiple sets of 7 segment digits are constructed.

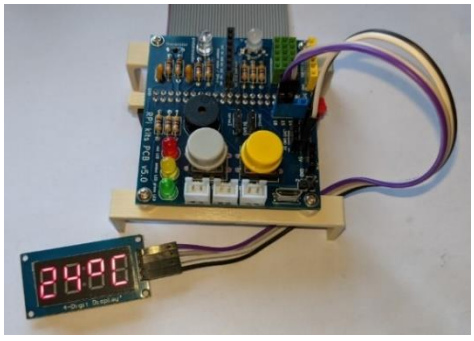
To simplify this complexity challenge, sets of 7 segment digits are therefore usually packaged with their own dedicated driver/control chip and there are two commonly used integrated circuits (ICs) that are used for this, namely the Titan Micro Electronics TM1637 and the Maxim Integrated MAX7219 which are discussed in the sections below.

TM1637 LED display

The images on the right show a front and back view of a 4 digit 7 segment LED display, with an additional 'colon' LED, that is packaged with and controlled by a TM1637 IC.

The four pins of the display can be connected to the PCB as shown in the table below and in the image below right: a custom Python library is available to simplify the coding for displaying text in many different and useful ways.

Display pin	PCB connection
CLK	spare GPIO #18
DIO	spare GPIO #15
VCC	5V from a PWR connector
GND	GND from a PWR connector



Details about the Python library used to demonstrate this display, and how it can be installed on a Raspberry Pi, can be found **here**.

The ***TM1637_test01.py*** code for this display demonstration is part of the main software download and to run the code, you can use the Thonny IDE or alternatively type or copy/paste the full commands shown at the top of each listing into a Terminal window, i.e.

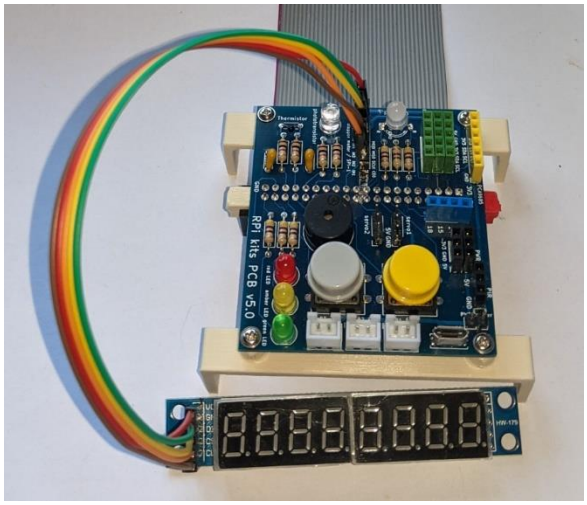
```
$ python3 ./RPi_maker_PCB5/displays/TM1637_7segment_LED/TM1637_test01.py
```

MAX7219 LED display

The images on the right show a front and back view of an 8 digit 7 segment LED display, with additional 'decimal point' LEDs, that is packaged with and controlled by a MAX7219 IC.

The display uses a SPI protocol for connection to a Raspberry Pi and the five pins of the display can be connected to the PCB as shown in the table and image below: a custom Python library is available to simplify the coding for displaying text in many different and useful ways.

Display pin	PCB connection
VCC	7P connector 5V
GND	7P connector GND
DIN	7P connector MOSI
CS	7P connector CE0
CLK	7P connector SCLK



Details about the Python library used to demonstrate this display, and how it can be installed on a Raspberry Pi, can be found **here**.

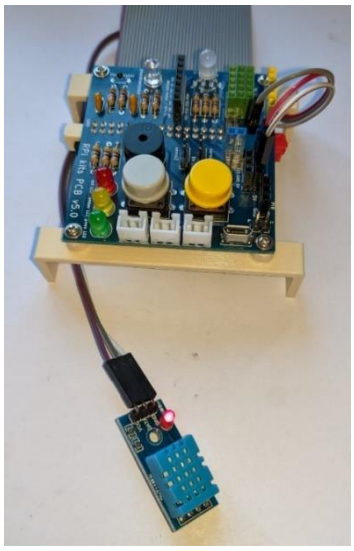
The ***sevensegment_demo.py*** code for this display demonstration is part of the main software download and to run the code, you can use the Thonny IDE or alternatively type or copy/paste the full commands shown at the top of each listing into a Terminal window, i.e.

```
$ python3 ./RPi_maker_PCB5/displays/MAX7219_7segment_LED/sevensegment_demo.py
```

Sensor projects

The PCB supports the connection of a wide range of sensors that can be managed by the Raspberry Pi and the following sections describe just some of the many ways that the Raspberry Pi can be used to measure the performance of devices and the general environment.

Temperature & humidity sensing

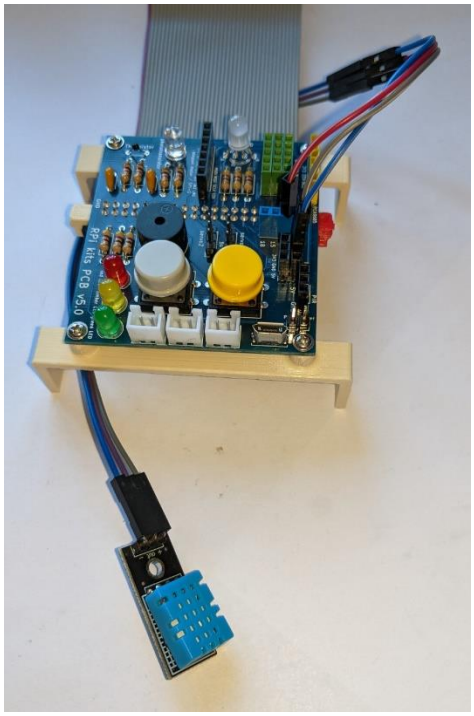


The DHT11 sensor is a commonly available and low-cost sensor of both temperature and relative humidity (RH).

The sensor can be obtained in a number of different 'packages' with one incorporating a LED to indicate it is operating shown in the image on the right and a more basic one shown below.

Whilst not shown in these images, as the PCB supports the connection of an I^2C controlled 16x2 LCD, the available test/demonstration code includes Python software that not only continuously collects the sensor data, but also displays the output on the LCD.

The DHT11 sensors all work in much the same way, providing a 40-bit (5 byte) digital data output when it is triggered by an external 'start signal', with the humidity and temperature data encoded in this output data.



Whilst not providing very high accuracy, i.e., ± 5% within a 20-80% RH range and ± 2°C in a 0-50°C range, this simple to use sensor is quite reliable and provides fast responses from a resistive-type humidity measurement component and an NTC (negative temperature coefficient) temperature measurement component connected to an 8-bit microcontroller.

The units shown here both have the sensor and processing elements mounted on small module boards with just 3 connectors labelled: *VCC*, *DATA* and *GND* or +, *out* and –

The sensor module can be powered by 3V3 (therefore ensuring a safe output voltage level for the Raspberry Pi on the *DATA/out* line), so for the test arrangements shown here the '*DATA*' or '*out*' pins are connected to one of the 'spare' GPIOs on the PCB (GPIO #14) and the 3V3/GND connections are provided from the nearby PWR-B 3-pin female connectors.

Importantly, built-in to the 'LED board' is a 1kΩ pull up resistor between the VCC and DATA connectors, and on the 'non-LED board' is a 5kΩ pull up resistor between the + and out connectors. Both 'levels of pull-up' satisfactorily ensure that the '*DATA*' or '*out*' signals operate at a stable voltage level and do not 'float'.

The ***dht11_test_example.py*** and ***dht11_test_LCD_example.py*** code for this temperature and relative humidity measurement demonstration is part of the main software download and to run the code, you can use the Thonny IDE or alternatively type or copy/paste the full commands shown at the top of each listing into a Terminal window, i.e.

```
$ python3 ./RPi_maker_PCB5/sensors/temp_sensors/dht11_test_example.py
```

or: `$ python3 ./RPi_maker_PCB5/sensors/temp_sensors/dht11_test_LCD_example.py`

Thermistor temperature sensing

The defined PCB electronic components include a thermistor, packaged as a small black ethoxyline resin coated ceramic bead with a pair of uninsulated tinned copper wire leads.

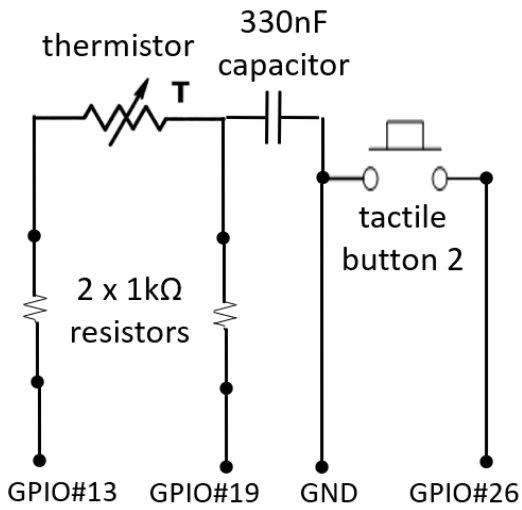
The thermistor used in the PCB assembly has a reference resistance of 10kΩ at 25°C and is suitable for measuring temperature over a range of -30°C to +125°C, although obviously the overall PCB assembly should not be subjected to such a temperature range!! The thermistor with the defined pair of 1kΩ resistors and the 330nF ceramic capacitor are configured on the PCB (as shown above left) as a "RC charging" circuit that can be used to 'deduce' the thermistor resistance from which the surrounding temperature can then also be 'deduced'.

The electronic component interconnections on a populated Raspberry Pi PCB are as summarised below:

Component	**GPIO pin(s)**
Tactile button 2	GPIO#26 and GND connections across the button- this 2nd button is used to initiate the temperature measurement cycle
Thermistor	GPIO connections: #13 and capacitor side #19, each through a 1kΩ resistor with a 330nF ceramic capacitor used to form a charging/discharging 'RC circuit' triggered by the tactile button press

The circuit diagram shown on the right shows how pressing button2 on the PCB can be used to initiate an 'RC charging circuit' cycle which consists of:

- Discharging the capacitor by setting GPIO#19 as an OUTPUT that is LOW
- Measuring the recharge time by setting GPIO#19 as an INPUT, GPIO#13 as an OUTPUT that is HIGH, and then 'clocking' how long it takes for GPIO #19 to go HIGH
- This time is then used to deduce the thermistor resistance, from which the surrounding temperature can then be deduced.



A more detailed description of this 'deduction analysis' is provided in *Appendix G: RC charging circuit analysis*.

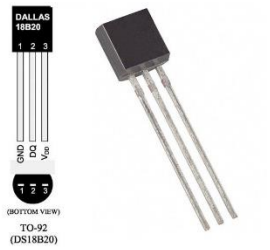
Python code is available as part of the main software download to support the use of this 'RC charging' circuit and to run the ***thermistor_sensor.py*** code, you can use the Thonny IDE or alternatively type or copy/paste the full commands shown at the top of each listing into a Terminal window, i.e.

```
$ python3 ./RPi_maker_PCB5/sensors/temp_sensors/thermistor_sensor.py
```

1-wire temperature sensor

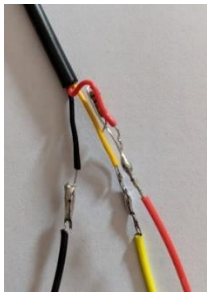
As shown below in the first two images, the DS18B20 temperature sensor is available either as a simple 'chip package' or as a waterproof assembly in a sealed metal tube. For either packaging type the sensor connects to a Raspberry Pi using the 1-wire interface, although there are usually 3 wires, i.e., power and ground as well as the single data line.

For the PCB test set up described here, a waterproof tube sensor is used, and the 3rd and 4th images below show that in order to ensure the data line voltage does not 'float', a 10kΩ pull-up resistor has been connected across the power and data lines when soldering male jumper leads to the wires.

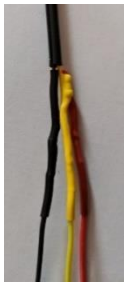


The DS18B20 device in a TO-92 'transistor-like' chip package

Waterproof version of the DS18B20 embedded in a metal tube with a long connection lead

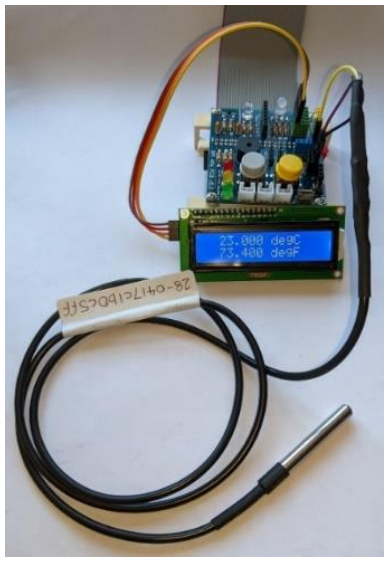


10kΩ pull-up resistor soldered between the power and data lines



heat shrink tubing used to insulate the soldered joints

The sensor is powered by 3.3V (therefore ensuring a safe output voltage level for the Raspberry Pi), so for the test arrangement shown in the image on the right, the sensor's (yellow) data line is connected to the 'spare' GPIO#4 on the PCB - which is the default GPIO used for the Raspberry Pi 1-wire interface. The 3V3 and GND connections are then provided from the nearby PWR-B 3-pin female connectors on the PCB.



The DS18B20 is a very functional sensor, providing configurable °C temperature measurements with 9-12 bit resolution that defaults to the full 12 bit resolution. It also has an alarm function with non-volatile user-programmable upper and lower trigger points and a measurement accuracy of ±0.5°C over a -10°C to +85°C range.

The 1-wire interface is a device communications bus system designed by Dallas Semiconductor Corp. that provides low-speed (16.3kbps) data with associated signalling and it can also power devices over the single data conductor so long as there is also a ground connection; this is called parasitic supply, although as we have used in this test set up, providing a separate power connection is usually more satisfactory.

As it is a bus system there is always a 1-wire master that initiates and controls the communication with one or more 1-wire slave devices on the bus which on a general basis may carry out a variety of functions other than temperature measurement.

Each 1-wire slave device has a unique, unalterable, factory-programmed, 64-bit ID (identification number), which serves as a device address on the 1-wire bus. The 8-bit family code, that is a subset of the 64-bit ID, identifies the device type and functionality.

Typically, 1-wire slave devices operate over the voltage range of 2.8V (min) to 5.25V (max). More detailed information on the 1-wire bus system is available at **this link**.

The Raspberry Pi must have its 1-wire interface enabled and it then acts as the 1-wire master automatically polling for 1-wire slave devices that have a data connection to the default GPIO#04. There is a way to configure the Pi to use a different GPIO pin for the 1-wire interface, or indeed multiple pins, but for simplicity using the default GPIO#04 makes the connection of 1-wire slave devices extremely easy.

If 1-wire slaves such as the DS18B20 are detected, then the Raspberry Pi 1-wire master automatically creates individual files for each slave. The files contain the latest data for each slave that has been detected and the address of each such file is as follows:

/sys/bus/w1/devices/64-bitID/w1_slave - where 64-bitID is the unique 64-bit ID for the 1-wire slave; an example ID for a DS18B20 device is 28-0417c1b0c5ff

Once the 1-wire interface has been enabled on the Raspberry Pi and a DS18B20 connected as described above, no further 'system' configuration is needed since data will automatically be loaded into the appropriate file.

Software can therefore be written to interpret the contents of this file and the test/demonstration Python code developed for this test set up continuously 'parses' the updated data file and displays the results as either °C or °F on a 16x2 LCD which is also connected to the PCB using one of the I^2C connections.

The ***DS18B20_LCD_demo.py*** and the non-LCD ***DS18B20_demo.py*** code for this temperature measurement demonstration is part of the main software download and to run the code, you can use the Thonny IDE or alternatively type or copy/paste the full commands shown at the top of each listing into a Terminal window, i.e.

```
$ python3 ./RPi_maker_PCB5/sensors/temp_sensors/DS18B20_LCD_demo.py
```

or:

```
$ python3 ./RPi_maker_PCB5/sensors/temp_sensors/DS18B20_demo.py
```

Object detection

An important 'class' of sensor is one that can be used to detect objects and the following series of sections describe the uses of various sensors that use heat, a radar doppler effect or sound to indicate whether an object is within the 'range' of a sensor.

Each sensor type can connect to the PCB so that it can be controlled by a Raspberry Pi.

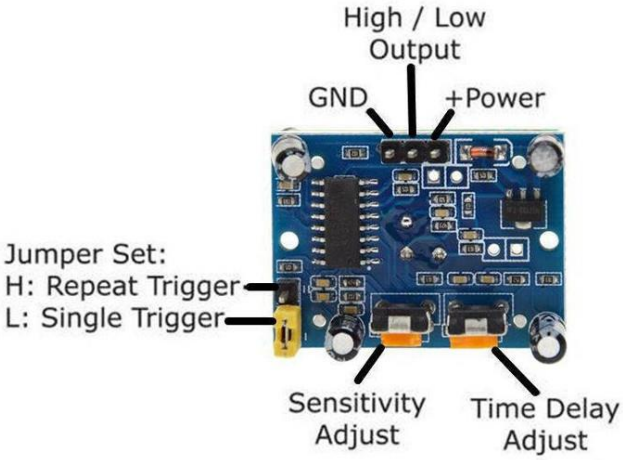
Passive Infrared object detection

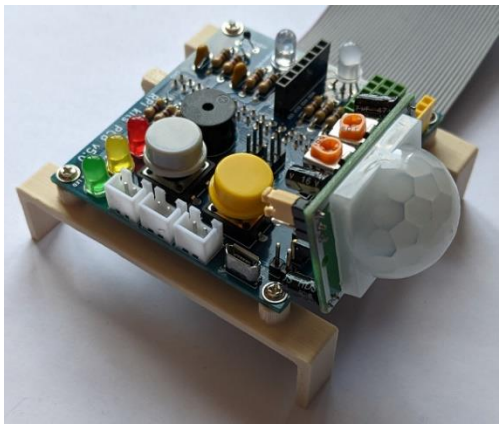
A passive infrared sensor (PIR sensor) is a sensor that measures infrared (IR) radiation, i.e., heat that radiates from objects in the sensor's field of view.

This type of sensor is readily available in a number of different 'packages' and the following details a few of the available types.

HC-SR501, HC-SR312 and HC-SR602 PIR sensors

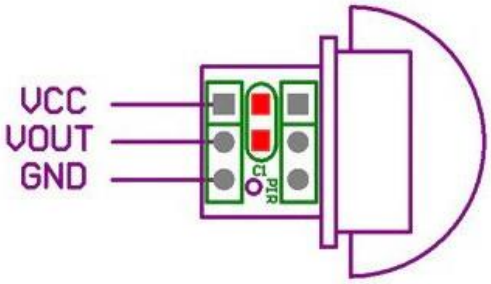
The **HC-SR501**, shown on the right, is the most commonly available PIR sensor and as the schematic shown next of the underside shows there are a number of configuration options that can be set.

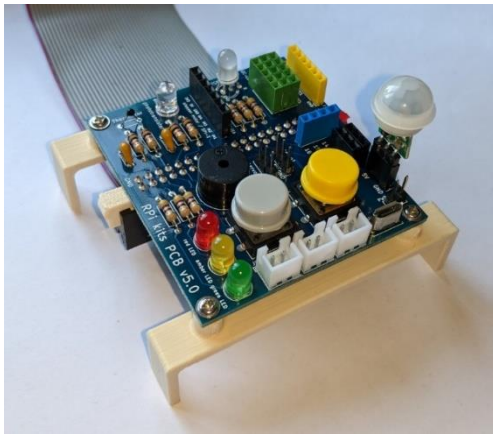




Using a right-angled adaptor, this sensor can be directly inserted into the PCB, as shown above right, and Python software is then used to monitor the 'Output' pin which connects through to GPIO #23.

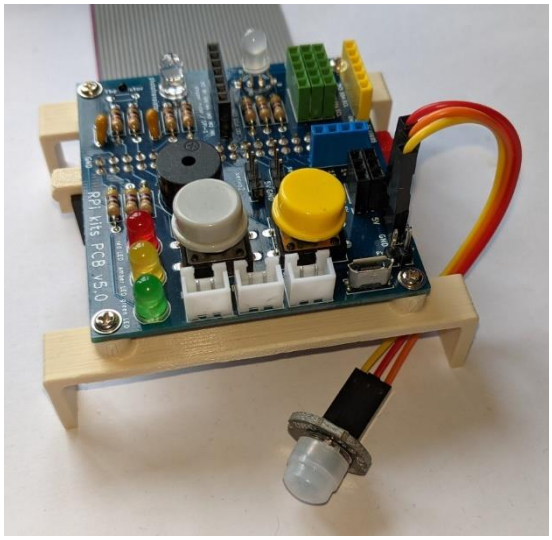
The **HC-SR312**, shown on the right, is a less commonly available 'mini' PIR that has exactly the same pin configuration as the HC-SR501, as shown in the schematic below.





This means it can also be inserted direct into the PCB as shown left, taking care to align the pins correctly (!!) and exactly the same Python software can be used to collect object detection data.

The **HC-SR602**, shown on the right, is an even less commonly available 'mini' PIR, but whilst it still has 3 pins (OUT, +, -) they are not in the same order as the HC-SR501 or the HC-SR312.



Therefore, as shown on the left, jumper leads must be used to correctly connect it to the 3 pin female PIR on the PCB.

As all the above PIRs behave in the same way, ***PIR_detect01.py*** is a general set of Python code used for this PIR object detection demonstration and is part of the main software download.

To run the code, you can use the Thonny IDE or alternatively type or copy/paste the full commands shown at the top of each listing into a Terminal window, i.e.

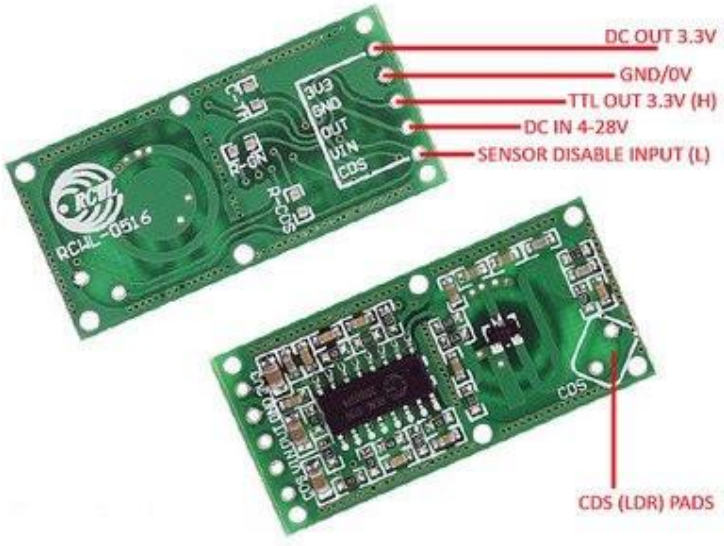
```
$ python3 ./RPi_maker_PCB5/sensors/object_detection/PIR_detection/PIR_detect01.py
```

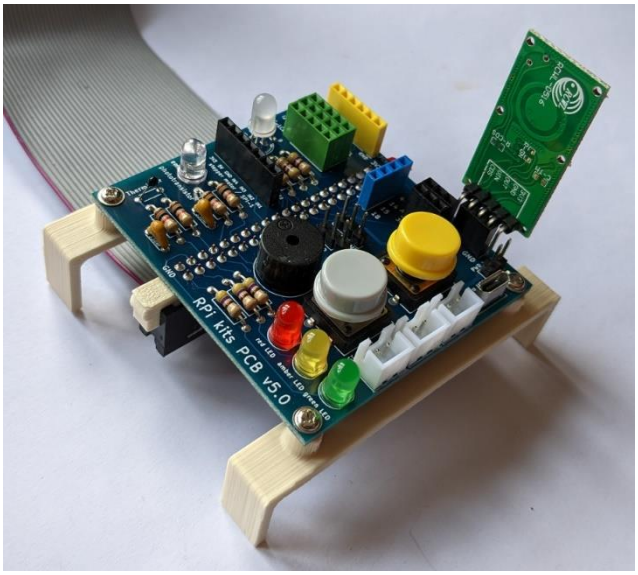
Microwave doppler radar object detection

A Doppler radar is a specialised radar that uses the Doppler effect to produce velocity data about objects at a distance. It does this by bouncing a microwave signal off the object and analysing how the object's motion has altered the frequency of the returned signal.

Typically, this measurement technique is used for high accuracy measurements in applications such as 'radar guns' for vehicle speed assessment.

The image on the right however shows a low cost (and obviously less accurate) but now widely available RCWL-0516 module that can be used in many different digital making projects.





With an input voltage range of 4-28VDC and an output signal voltage of 3.3VDC, the board can easily be connected to a Raspberry Pi for object detection.

In addition, whilst the RCWL-0516 has 5 pins, the main 3 middle pins are in the same order as required by the PIR connector on the PCB. This means it can be directly inserted into this connector as shown in the image on the left.

The unused 'outer' pins are a regulated 3.3V output and sensor 'disable' input pin, neither of which are needed for the exploratory testing described here.

Some other points to note about using the RCWL-0516 are:

- Its operating frequency is at or about 3.181GHz, so non-metallic structures will NOT block the broadcast signal i.e., it will detect through walls!
- The 'forward' side of the board (the side with all the components) is the main object detection direction and should not be obstructed by anything metallic.
- The default detection range is 7m, adding a 1MΩ resistor across the R-GN pads on the 'back' side of the module reduces it to 5m. Lower resistance values might further reduce the range, perhaps down to less than 1m.
- The 'back' side of the module should have clearance of more than 1cm from anything metallic.
- Operating multiple modules in close proximity is not recommended as their broadcast signals will interfere with each other.
- The default 'repeat trigger time', i.e., the period of time that the object detection pin will remain HIGH, is 2 seconds which can be adjusted by adding a capacitor across the C-TM pads on the 'back' side of the module.
- The module includes an option to fit a photoresistor that will enable/disable the sensor when the background is either light or dark. The area on the module where this is fitted is labelled CDS because inexpensive CdS or Cadmium Sulphide devices have often been used – BUT this is a material that is now severely restricted throughout Europe due to the RoHS ban on cadmium. The use of a phototransistor as a light sensor is a better substitute.

As this device produces a similar detection output as the PIR devices 'derivative' Python code ***microwave_detect01.py*** is available as a basic object detection demonstration and is part of the main software download.

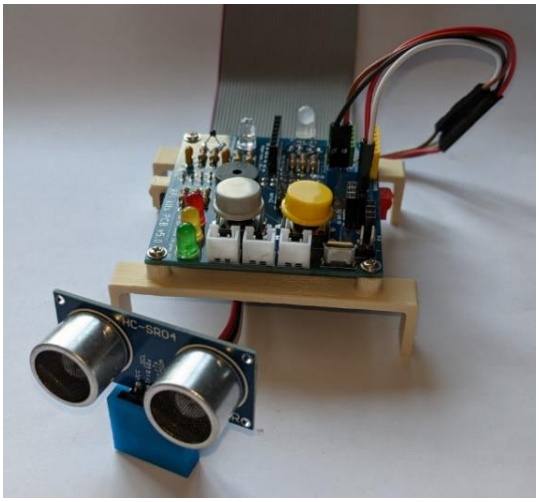
To run the code, you can use the Thonny IDE or alternatively type or copy/paste the full commands shown at the top of each listing into a Terminal window, i.e.

```
$ python3 ./RPi_maker_PCB5/sensors/object_detection/microwave_detection/microwave_detect01.py
```

Ultrasonic sound object detection

The HC-SR04P ultrasonic sensor shown front and back on the right (wrongly labelled on the front face) is a newer version of the original HC-SR04. This newer version can be powered with a wider range of input voltage, i.e., from 3.0V to 5.5V – but in all other respects it is the same as the original.

This type of sensor can be programmed to emit a high frequency sound wave (c. 40,000 Hz which is well above the highest frequency that a human can hear) from its 'transmission speaker' (labelled T). This sound will 'bounce back' to the module if it hits a solid object in its path and can be 'heard' by the modules 'receiving microphone' (Labelled R).



As the speed of sound in air is known, the time interval between the sound being emitted and its reflection being 'heard' by the sensor allows the object's distance to be calculated.

The sensor module has a 4 pin male connector labelled: VCC, Trig, Echo, GND

- and the image on the left shows how the pins are connected to the PCB using male-female jumper leads

One of the PCB's 3 pin black female PWR connectors is used to provide the 3V3 and GND connections.

Then:

- Trig is connected to the 'spare' GPIO#15 and configured as an OUTPUT pin, and
- Echo is connected to the 'spare' GPIO#18 and configured as an INPUT pin.

The continuous measurement cycle programmed in Python then consists of the following steps:

- Trig is set LOW for 0.5 seconds and then set HIGH for 10 µseconds which initiates the sending of the ultrasound signal.
- A timer is started and stopped when the Echo pin goes HIGH, which indicates that the reflected sound has been detected – but this timing cycle is curtailed if it exceeds 0.04 seconds as that is 'deemed' to show that the 'reflection' was missed as it came back too quickly for the electronics/software to 'catch' it, i.e., an object was too near the sensor.
- Each successful timed response is then used to calculate the distance to the detected object.

Python code ***ultrasonic_detect01.py*** is available as a basic object detection demonstration and is part of the main software download.

To run the code, you can use the Thonny IDE or alternatively type or copy/paste the full commands shown at the top of each listing into a Terminal window, i.e.

```
$ python3 ./RPi_maker_PCB5/sensors/object_detection/ultrasonic_detection/ultrasonic_detect01.py
```

Light sensing

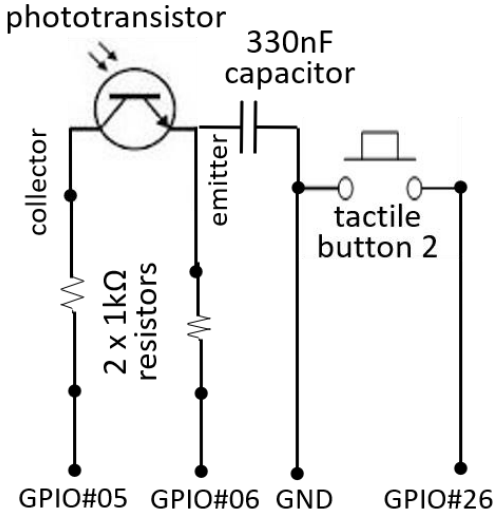
Phototransistor light sensing

The design of PCB includes the use of a phototransistor of the type that has its output tuned to the visible light range.

The phototransistor with the defined pair of 1kΩ resistors and the 330nF ceramic capacitor are configured on the PCB (as shown on the left) as a "RC charging" circuit that can be used to 'deduce' the phototransistor's resistance which can thenbe used as a simple 'proxy' for the prevailing light level.

The electronic component interconnections on a populated PCB are as summarised below:

Component	GPIO pin(s)
Tactile button 2	GPIO#26 and GND connections across the button- this button is used to initiate the light sensing cycle.
Phototransistor	GPIO connections: collector #05 and emitter #06, each through a 1kΩ resistor with a 330nF ceramic capacitor used to form a charging/discharging 'RC circuit' triggered by a tactile button press.



The circuit diagram shown on the right shows how pressing button2 on the PCB can be used to initiate an 'RC charging circuit' cycle which consists of:

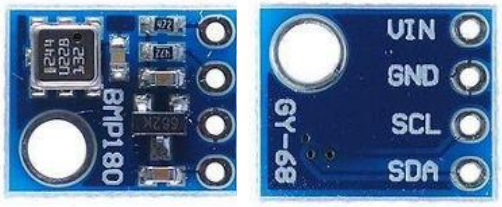
- Discharging the capacitor by setting GPIO#06 as an OUTPUT that is LOW
- Measuring the recharge time by setting GPIO#06 as an INPUT, GPIO#05 as an OUTPUT that is HIGH, and then 'clocking' how long it takes for GPIO #06 to go HIGH
- As this time depends on the voltage at the phototransistor's collector, which moves from high to low when the light level increases, this then increases the current flow, which decreases the capacitor charging time. A short capacitor charging time is therefore a 'proxy' for a high light level and a long time for low light level. These times could then be calibrated on a one-time basis against an accurate light meter.

A more detailed description of this 'deduction analysis' is provided in *Appendix G: RC charging circuit analysis*.

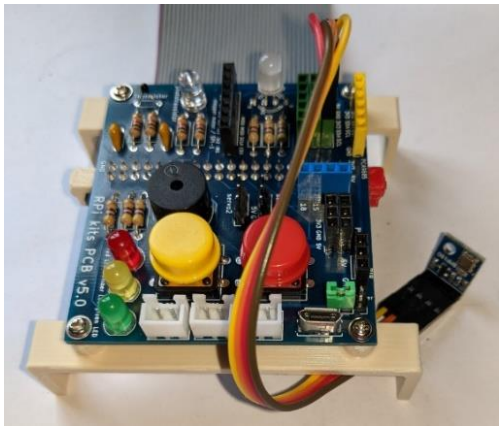
Python code is available as part of the main software download to support the use of this 'RC charging' circuit and to run the ***phototran_light_sensor.py*** code, you can use the Thonny IDE or alternatively type or copy/paste the full commands shown at the top of each listing into a Terminal window, i.e.

```
$ python3 ./RPi_maker_PCB5/sensors/ light_sensors/phototran_light_sensor.py
```

Pressure sensing



A Bosch BMP180 pressure sensor in a very small module (13mmx10mm), as shown front and back on the left, provides a high-precision measurement capability for both pressure, range 300-1100 hPa (mbar), and temperature, range -40°C to +85 °C.



It has a standard I²C interface and can be powered by 1.8-3.6V, so connection to the PCB can be made using the I²C green female headers and the 3V3 power line.

This provides a simple arrangement, as shown on the right, for data collection and analysis using Python code with a default I²C address of 0x77.

Python code ***bmp180.py*** is available as a basic demonstration and is part of the main software download.

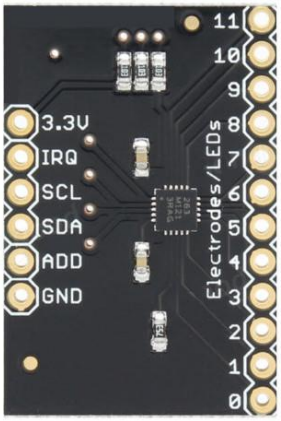
To run the code, you can use the Thonny IDE or alternatively type or copy/paste the full commands shown at the top of the listing into a Terminal window, i.e.

```
$ python3 ./RPi_maker_PCB5/sensors/pressure_sensors/bmp180.py
```

The screen shot below also shows how the *i2cdetect* CLI command can be used to check the module's I²C address, and also illustrates the output from the demonstration code.

```
pi@pikitP48GB:~ $ i2cdetect -y -r 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: 20 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- 77
pi@pikitP48GB:~ $ python3 /home/pi/RPi_maker_kit5/sensors/pressure_sensors/bmp180.py
Chip ID     : 85
Version     : 2
Temperature : 22.4 C
Pressure    : 996.37 mbar
pi@pikitP48GB:~ $
```

Capacitive touch sensing



An MPR121 module, as shown on the left, is a breakout board for Freescale's MPR121QR2 chip providing a capacitive touch sensor controller driven by an I²C interface.

The chip and module can control up to twelve individual electrodes, as well as a simulated thirteenth electrode, and the module provides twelve input pins (0-11) which can each be used as a 'capacitive touch' input (or indeed to control up to 8 LEDs).

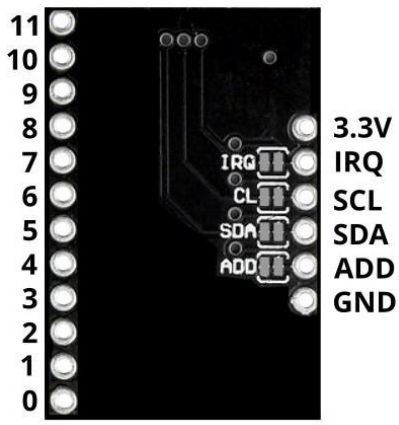
The module is powered by 3V3 and has internal 10kΩ pull-up resistors for the SDA, SCL and IRQ lines, so generally there are

no concerns about too high a voltage level on these I²C control lines when connected to a Raspberry Pi. However, consideration needs to be given as to whether these module pull-up resistors should be 'disconnected' (see later) depending upon how many pull-up resistors are 'on the bus' at the same time and may therefore prevent sufficient 'pull' to occur to set the SDA and SCL pins high enough: a Raspberry Pi already has 1.8kΩ internal pull-up resistors on its SDA and SCL GPIO pins so does not normally need pull-ups on attached devices.

The module is connected to the Raspberry Pi PCB via one of the I²C (green) 5-pin female connectors - but obviously only using the 3V3 port on the connector to power the device.

When connected to the Raspberry Pi the default I²C address for the device is 0x5a which conveniently does not conflict with any of the other common I²C devices that may be used with the PCB, so the use of a single MPR121 device may not need to have its I²C address changed.

The device I²C address can however be changed by a process that cuts the small 'trace' between the two 'pads' next to the ADD connector on the underside of the module (see the image on the right). This removes the internal connection from the ADD pin to GND (which can obviously be restored by reapplying a 'solder bridge' across the pads).



To assign an alternative I²C address, the ADD pin is 'shorted' to a control pin as follows:

ADD's default connection to GND	address = 0x5A
ADD tied to 3V	address = 0x5B
ADD tied to SDA	address = 0x5C
ADD tied to SCL	address = 0x5D

With four different addresses available this means that up to 48 capacitive touch inputs could be enabled using four separate modules.

To disconnect the module's internal pull-up resistors on any of the control pins, the small 'trace' between the two pads next to the IRQ, SCL or SDA connectors on the underside of the module should be cut (see the image above right).

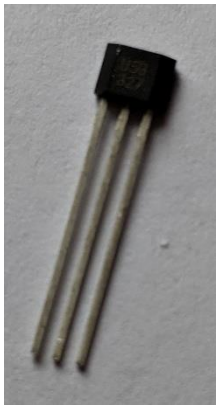
Some simple Python code is available as part of the main software download to support the use of this capacitive touch sensor and to run ***simpletest.py*** you can use the Thonny IDE or alternatively type or copy/paste the full commands shown at the top of each listing into a Terminal window, i.e.

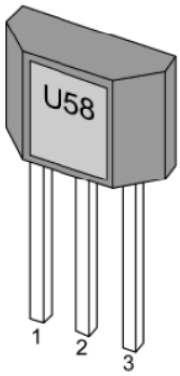
```
$ python3 ./RPi_maker_PCB5/sensors/MPR121_touch_sensor/simpletest.py
```

Magnetic sensing

The “Hall effect” is the main method used to sense the presence of a magnetic field. With this method, discovered by Edwin Hall in 1879, a sensor produces a voltage difference (the Hall voltage) across an electrical conductor that is transverse to an electric current in the conductor and to an applied magnetic field perpendicular to the current.

The Melexis US5881 sensor, shown in the images on the right, is a unipolar Hall-effect switch designed in mixed signal CMOS technology.

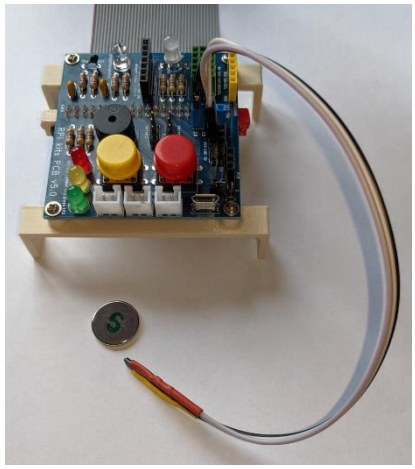




The device integrates a voltage regulator, Hall sensor with dynamic offset cancellation system, Schmitt trigger and an open-drain output driver, all in a single package.

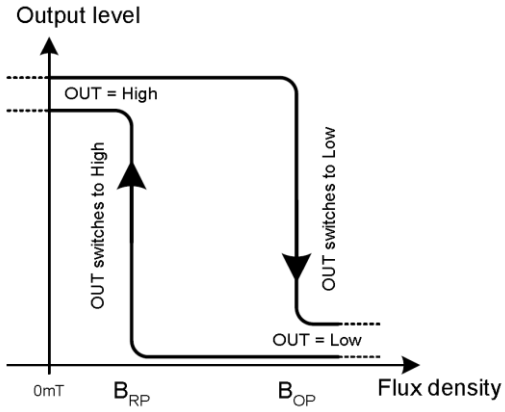
The device can be packaged as a Thin Small Outline Transistor (TSOT) for surface mounting, or as used here, as a Plastic Single In Line (TO-92 flat) for through hole mounting.

For use with the PCB, jumper leads are soldered to the three connection wires with heat shrink tubing used to strain relieve and insulate the joints, as shown in the image on the right.



Connections to the PCB are as shown in the table below.

Sensor pin	PCB connection
1 - VDD	3V3 from a PWR connector
2 - GND	GND from a PWR connector
3 - OUT	GPIO #18 'spare' pin



Unipolar switch characteristic

The device will detect the proximity of a magnetic south pole (typically in a 10-15mm range) by sending the OUT signal pin LOW, as illustrated in the schematic on the left.

Python code is available as part of the main software download to support the use of this Hall effect sensor and to run ***US5881_hall.py*** you can use the Thonny IDE or alternatively type or copy/paste the full commands shown at the top of each listing into a Terminal window, i.e.

```
$ python3 ./RPi_maker_PCB5/sensors/magnetic_sensors/US5881_hall.py
```

433MHz RF communication

A number of low-cost and readily available transmitter and receiver modules allow a Raspberry Pi to send and receive simple messages using a 433MHz radio frequency broadcast method.

The messages are digitally encoded into the signal, and whilst some quite complex and encrypted/secure methods are available, the basic modules considered here use the following simple modulation protocols:

ASK: *Amplitude Shift-Keying* is a popular modulation technique used in digital data communication for a large number of low-frequency RF applications. The source transmits a large amplitude carrier when it wants to send a '1' and it sends a small amplitude carrier when it wants to send a '0' in its simplest form.

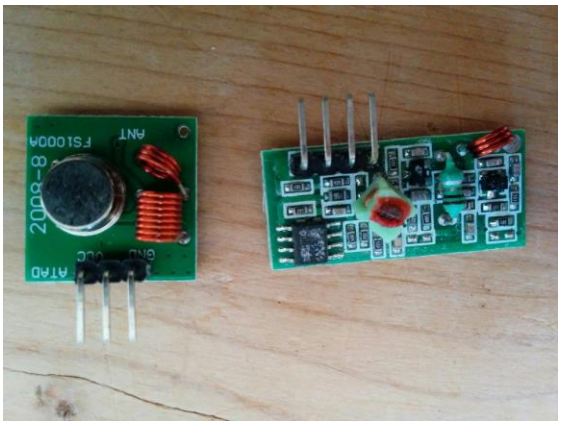
OOK: *On-Off Keying* modulation is a further simplification of the ASK method, where the source sends NO carrier when it wants to send a '0'.

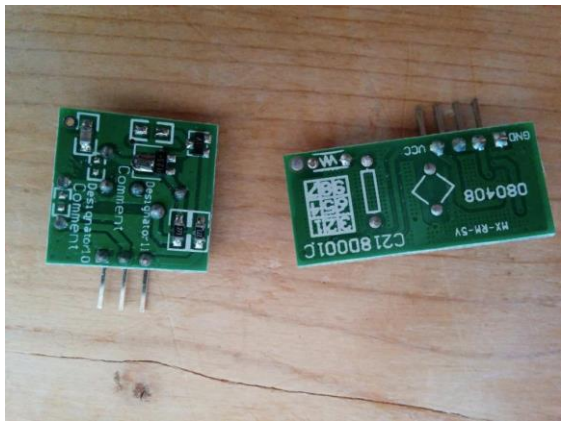
FSK: *Frequency-Shift Keying* is a type of amplitude modulation that assigns bit values to discrete amplitude levels. The carrier signal is then modulated among the members of a set of discrete values to transmit information.

A common usage area for this type of RF communication is to send ON/OFF signals to remote electrical equipment as part of a home automation system, but it should be noted that because the methods explored here are not secure, this means that they should only be used for applications where it would not matter if the equipment was inadvertently (or maliciously!) switched ON or OFF incorrectly.

FS1000A & C218D001C RF communications

The FS1000A transmitter and C218D001C receiver boards illustrated below are extremely low cost generic 433MHz RF transmitter and receiver boards that support simple ASK/OOK modulation methods.



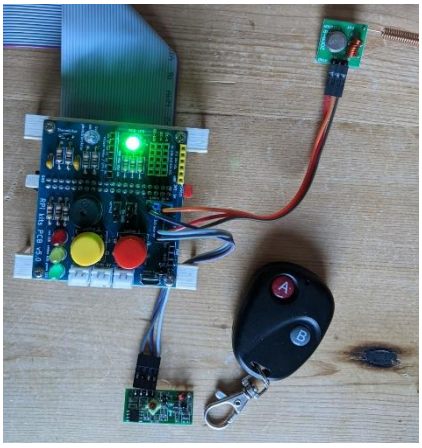


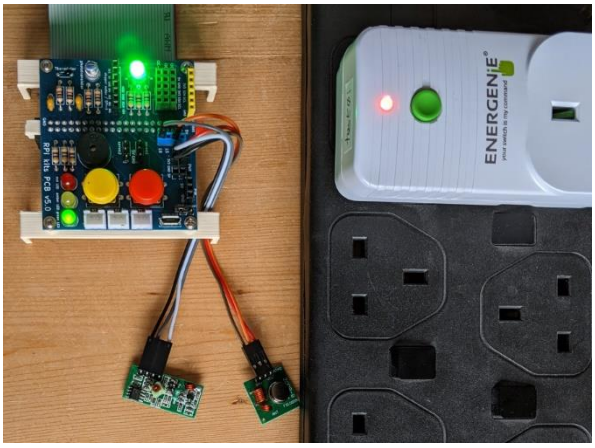
The smaller/square FS1000A transmitter board has 3 pins i.e., when viewed from the front of the board (left/left view above) the printed labels are:

- left pin: signal (ATAD i.e., DATA printed backwards for some reason!)
- middle pin: VCC i.e., 3V3 power input (for maximum safety)
- right pin: GND i.e., power ground

The slightly larger oblong C218D001C receiver board has 4 pins i.e., when viewed from the back of the board (right/right view above) the printed labels are:

- left pin: VCC i.e., 3.3V power input
- 2 middle pins: unlabelled signal/data pins - either can be used as they are simply bridged together on the PCB
- right pin: GND i.e., power ground





The images above show the FS1000A and the C218D001C connected to a Raspberry Pi PCB for two different test methods with the table below showing the pin connections for each module.

FS1000A connections	
FS1000A pin	PCB connection
DATA	spare GPIO #14
VCC	PWR B 3V3
GND	PWR B GND

C218D001C connections	
C218D001C pin	PCB connection
either middle data pin	spare GPIO #18
VCC	PWR A 3V3
GND	PWR A GND

Demonstration Python software is available as part of the main software download to:

1. show that the C218D001C receiver board can capture the 'signals' sent by a button press from a key fob pair of buttons. To run ***Key_Fob_RX_plot_C218D001C.py*** you can use the Thonny IDE or alternatively type or copy/paste the full commands shown at the top of each listing into a Terminal window, i.e.

```
$ python3 ./RPi_maker_PCB5/RF_communication/C218D001C+FS1000A_comms/ Key_Fob_RX_plot_C218D001C.py
```

and

2. to show how the FS1000A transmitter board can be used to switch a basic Energenie socket ON/OFF. To run ***switch_socket02.py*** you can use the Thonny IDE or alternatively type or copy/paste the full commands shown at the top of each listing into a Terminal window, i.e.

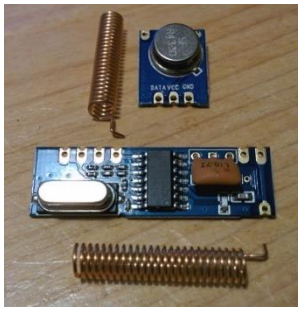
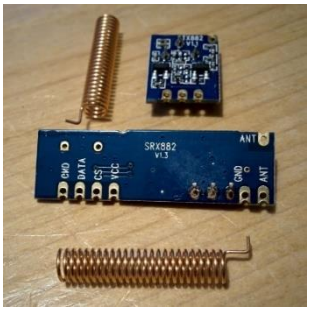
```
$ python3 ./RPi_maker_PCB5/RF_communication/C218D001C+FS1000A_comms/ switch_socket02.py
```

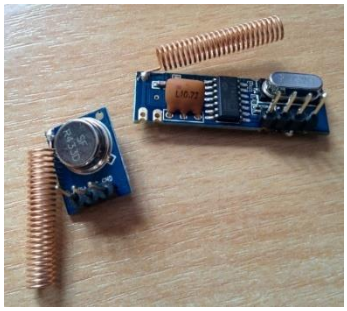
For both demo programs the LEDs and passive buzzer that are built-in to an assembled PCB, are also used to indicate the progress of the various signal transmissions and receptions along with their decoding cycles, and the built-in tactile button 1 is used to trigger the Energenie socket ON/OFF toggling.

It should be noted that the performance of the C218D001C receiver board was not that reliable/consistent during the tests, but it was not clear whether this was due to poor module design/build quality or perhaps intermittent physical connections due to poor quality leads.

SRX882 and STX882 RF communications

The STX882 and SRX882 are low cost and (supposedly) high-power, long range RF boards operating at 433MHz, that support simple ASK/OOK modulation methods.

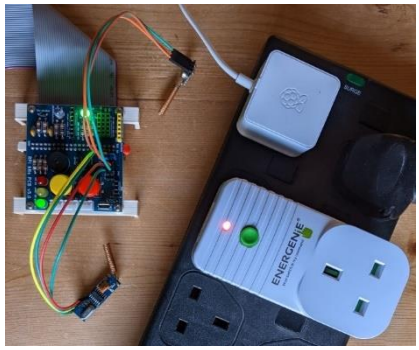





The smaller STX882 transmitter and the larger oblong SRX882 receiver, shown top and rear views above, are often supplied with simple antennae that can be soldered to a connection point provided on each board, as shown in the final image above right.

The images on the right show the SRX882 and the STX882 connected to a PCB.

This pair of boards are in many ways similar to the previously described FS1000A transmitter and the C218D001C receiver, although the SRX882 receiver has only the one data pin with the other 'middle' pin, labelled CS, used as a 'mode setting' pin that is set HIGH for normal receive mode operation.

The tables below show the pin connections for each module.

STX882 connections	
STX882 pin	PCB connection
DATA	spare GPIO #14
VCC	PWR B 3V3
GND	PWR B GND

SRX882 connections	
SRX882 pin	PCB connection
VCC	PWR A 3V3
DATA	spare GPIO #18
CS	spare GPIO #15
GND	PWR A GND

Very similar demonstration Python software to the previous modules is available as part of the main software download to:

1. show that the SRX882 receiver board can capture the 'signals' sent by a button press from a key fob pair of buttons. To run ***Key_Fob_RX_plot.py*** you can use the Thonny IDE or alternatively type or copy/paste the full commands shown at the top of each listing into a Terminal window, i.e.

```
$ python3 ./RPi_maker_PCB5/RF_communication/SRX882+STX882_comms/ Key_Fob_RX_plot.py
```

and

2. to show how the STX882 transmitter board can be used to switch a basic Energenie socket ON/OFF. To run ***switch_socket02.py*** you can use the Thonny IDE or alternatively type or copy/paste the full commands shown at the top of each listing into a Terminal window, i.e.

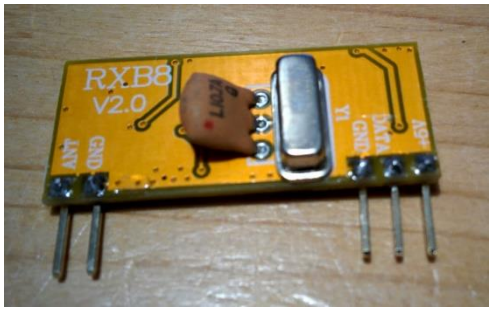
```
$ python3 ./RPi_maker_PCB5/RF_communication/SRX882+STX882_comms/ switch_socket02.py
```

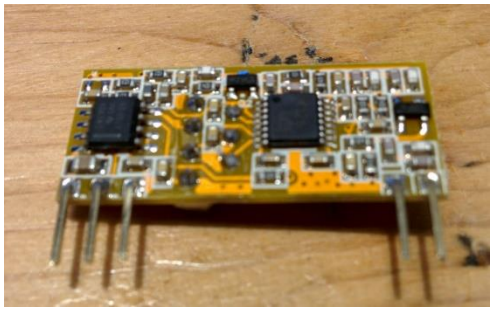
As before, for both demo programs the LEDs and passive buzzer that are built-in to an assembled PCB, are used to indicate the progress of the various signal transmissions and receptions along with their decoding cycles, and the built-in tactile button 1 is used to trigger the Energenie socket ON/OFF toggling.

It should be noted that the performance of both the SRX882 and STX882 boards was very reliable and consistent during the tests, and this may, in part, be due to the use of the soldered on antennae.

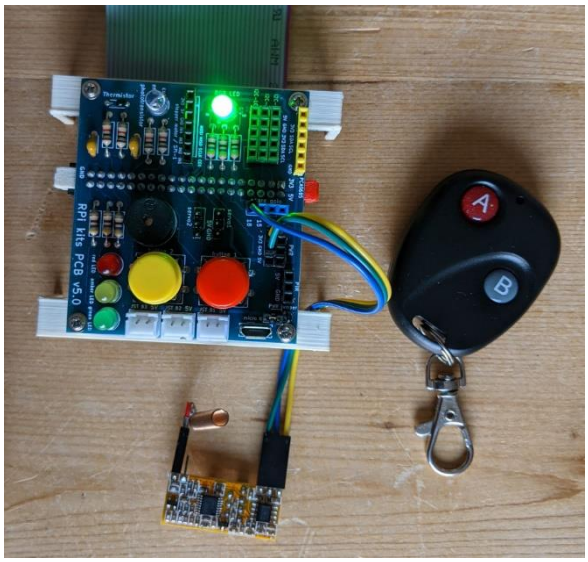
RXB8 RF communications

The RXB8 receiver shown in front and rear views below, specifies a 4.5 - 5.5 supply voltage range in its data sheet - which would cause a problem with the GPIO signal voltage level on a Raspberry Pi - so all tests were carried out with the 3.3V PCB voltage supply - which all seemed to work well.





The image on the right shows the RXB8 receiver connected to a Raspberry Pi PCB with the table below showing the pin connections for the module.



As the module also has a separate pin marked ANT, a simple antenna was created that could be a push-fit to this antenna pin .

RXB8 connections	
RXB8 pin	PCB connection
DATA	spare GPIO #18
VCC	PWR B 3V3
GND	PWR B GND

Again, very similar demonstration Python software to the previous modules is available as part of the main software download to:

1. show that the RXB8 receiver board can capture the 'signals' sent by a button press from a key fob pair of buttons. To run ***Key_Fob_RX_plot_RXB8.py*** you can use the Thonny IDE or alternatively type or copy/paste the full commands shown at the top of each listing into a Terminal window, i.e.

```
$ python3 ./RPi_maker_PCB5/RF_communication/RXB8_comms/ Key_Fob_RX_plot_RXB8.py
```

2. but the second available demo code is another 'receiving' program that can be run continuously to 'sniff' whatever signals it can detect. To run ***rpi-rf_receive.py*** you can use the Thonny IDE or alternatively type or copy/paste the full commands shown at the top of each listing into a Terminal window, i.e.

```
$ python3 ./RPi_maker_PCB5/RF_communication/RXB8_comms/rpi-rf_receive.py
```

As before, for both demo programs the LEDs and passive buzzer that are built-in to an assembled PCB, are used to indicate the progress of the various signal receptions along with their decoding cycles.

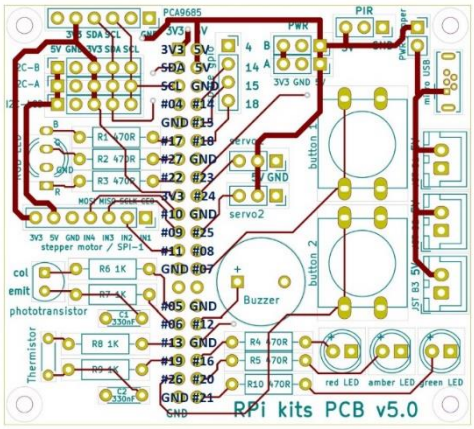
It should also be noted that the performance of the RXB8 board was very reliable and consistent during the tests. In addition, for the 'sniffer' code additional filtering has been added to the code to show how specific types of signal can be detected.

Appendix A: PCB development details

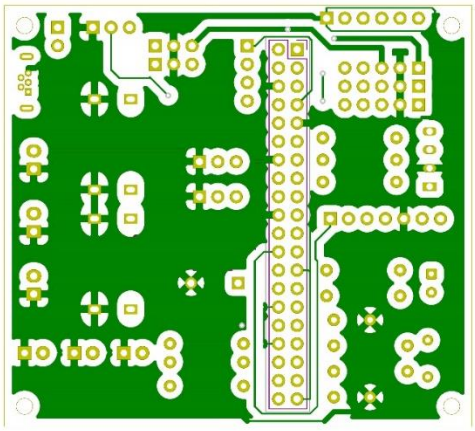
Development cycle

The design of the PCB is now at v5.0 which reflects several iterations to implement refinements and functional extensions from the start of the development.

The PCB design process has used the KiCAD software, and the images below show some details of the v5.0 design generated by the software.



front of the PCB showing the component footprints and the copper power & signal interconnections, with overlaid annotation of the Raspberry Pi GPIO numbering



back of the PCB with the 'flooded' ground plane as well as some interconnections on this side

The key principles of the design are as follows:

- A 2x20 female header on the underside of the PCB allows the fully assembled unit to be directly inserted onto a Raspberry Pi's GPIO pins or connected via an extension ribbon cable;
- to make the '*component to GPIO pin*' connections as straightforward and non-circuitous as possible, the components needed to be laid out on both sides of the 2x20 female header and as far as possible to use GPIO pins that were nearest the component and grouped together appropriately;
- the component layout had to also ensure that the 'inboard overhang' i.e., that area of the PCB that would overlap the Raspberry Pi board if the PCB were inserted directly onto the Raspberry Pi GPIOs, would not 'conflict' with any of the Raspberry Pi fittings/components;
- a small number of GPIO pins were not needed to interconnect defined components, and these were grouped together so that they could still be used and connections on the PCB provided for them (labelled as spare pins);
- GPIO#4 was specifically included in the 'spare pin' allocation since this is the default GPIO pin used by the Raspberry Pi 1-wire interface, so making this separately available allows the PCB to be easily used for additional 1-wire component connections and further associated project developments;

- GPIO#14 (TX) and GPIO#15 (RX) were also included in the 'spare pin' allocation so that the standard Raspberry Pi TX/RX functions could also be used in further associated project developments;

- to provide external power for a PCA9685 PWM module to manage servos and to provide power for other uses, a separate 'power bus', using JST connectors plus a USB type B micro connector with a common ground to the rest of the PCB, was added so that a 4xAA battery pack, 5V rechargeable battery bank or a 5V mains power supply could be connected as 'input' to this 'power bus', and additional connectors used for outputs to various devices;

- Inclusion of a 'jumper' connection between the separate 'power bus', fed by the JST and micro USB connectors, with the main 5V power lines from the Raspberry Pi allowing the unit to be powered in a stand-alone mode - although this usage mode must take care to not use a voltage supply greater than the 5V required by the Pi!!

- to provide power output options from the internal Raspberry Pi power lines, two 3 pin female connectors (PWR-A and PWR-B) were added to provide 3V3-GND-5V outputs;

- to make the connection of a pair of servos, directly powered by the Raspberry Pi (two servos probably being a reasonable limit from the Pi's internal power) as easy as possible, two 3 pin male connectors have been added labelled servo1-5V-GND and servo2-5V-GND;

- whilst a dedicated 6-pin female connector is used to provide I^2C control for a PCA9685 PWM control board for servo management, three additional 5-pin I^2C female connectors are provided to allow further components to be connected to the Raspberry Pi's I^2C bus, so that yet more project developments are enabled; and

- finally, a 7-pin connector, using various pin combinations is optionally used as:
  - a connector for a stepper motor;
  - a connector for direct drive motor controllers; and
  - as a general purpose SPI connector (using SPI0 and CE0)

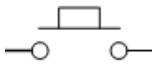
Finalised design

The v5.0 design was finalised in April'21 and a .zip file of the 'Gerber' design files can be downloaded from **here** so that anyone can have small quantities of the PCB manufactured using one of the many low cost online suppliers that are now available.

Appendix B: PCB component details

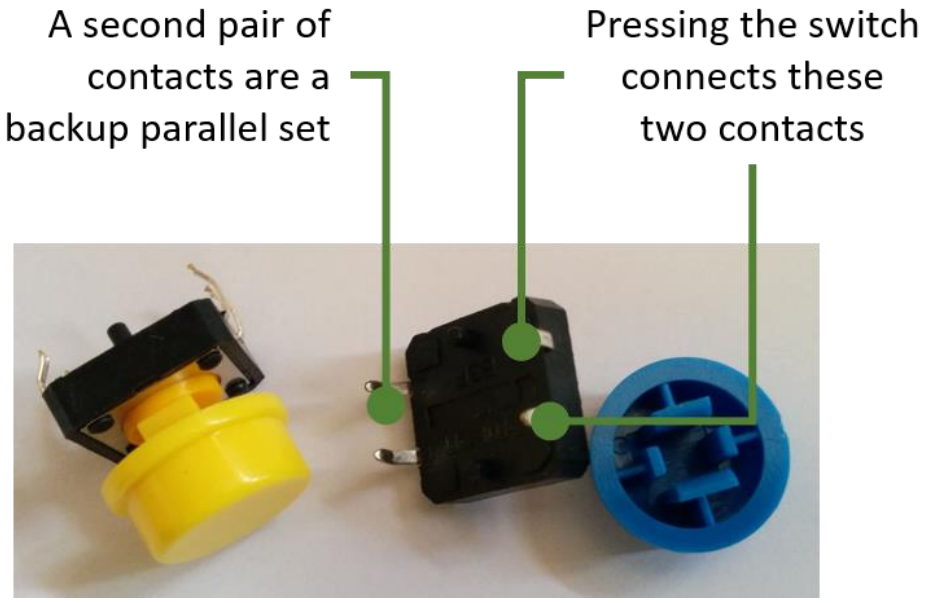
This appendix provides more detail for the various components used to assemble a populated PCB.

Tactile buttons



A tactile push button is a switch that makes momentary contact when it is pressed and may also make a 'tactile' click.

The buttons defined for use with the **Raspberry Pi PCB** are 12mm wide with a coloured push on cap (colours can vary) where the PCB footprint design allows each button to be gently pushed and held in their insert points the correct way round, prior to soldering the connections on the underside of the PCB.

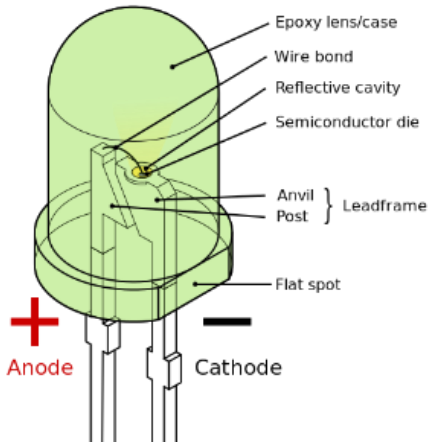


As shown above, it is the two contacts on the same side of the button that are connected when the button is pressed.

Light Emitting Diodes (LEDs)

Light Emitting Diodes, or LEDs, are simple, low energy consumption light sources that are powered by low voltage direct current (DC) and are available in a range of colours.

A very detailed description of their history and technical performance is available in this Wikipedia link.

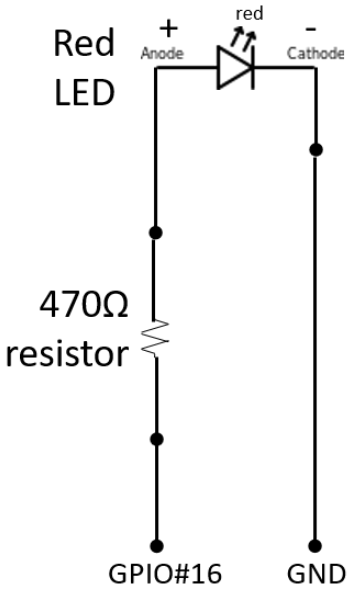


Specified for use with the **Raspberry Pi PCB** are one each of red, amber, and green coloured LEDs packaged in either a 5mm or 3mm potted plastic lenses, as shown in the diagram on the left.

There is also one so-called RGB LED which has separate red (R), green (G) and blue (B) light emitting diodes packaged all together in a single plastic lens. This allows each of the RGB colours to be independently set on or off, or to different light intensities and by 'mixing' light in this way, a range of different coloured light can be produced.

For the defined single colour LEDs, the anode or positive power lead is longer (NB the schematic above shows a flat spot can also be used to denote the cathode side but this may not be present on some LEDs). It is very important that the +'ve DC voltage is correctly applied to the longer anode connection by inserting the LED into the PCB holes so that the longer lead is nearest the resistors for each of the LEDs. The insertion points for the long legs on the PCB are each labelled with a small + to ensure the correct orientation is used.

For the RGB LED, the negative (cathode) leads for the three colours are all interconnected within the plastic lens with a single 'common' ground lead that is longer than all the separate anode leads. The sequence of the four leads from a RGB LED is: RED, common, GREEN, BLUE so it is important that the RGB LED is inserted into the PCB holes correctly with the insertion points labelled R, GND, G and B on the PCB to ensure the correct orientation is used.



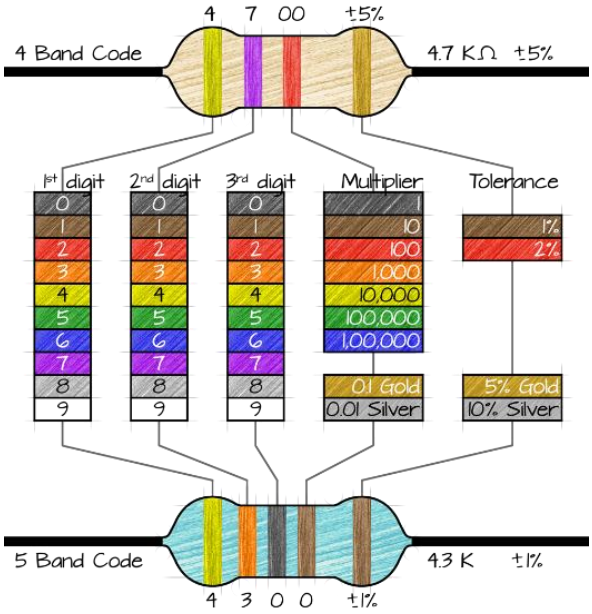
In operation, each LED type has a slightly different forward voltage drop (V_f) that should not be exceeded, otherwise its maximum current limit could be exceeded which would cause damage.

V_f for red and amber LEDS is typically 1.8V and for blue, green, and white it is typically 3.3V.

To avoid over current damage, all the LEDs on the PCB are therefore put in series with the defined 470Ω resistors, as illustrated in the circuit diagram on the right, with the resistor having been sized to achieve a safe level of the applied voltage for all the LED types.

Resistors

Resistors are small passive electrical devices used to provide electrical resistance measured in ohms (Ω).



The resistors defined for use with the **Raspberry Pi PCB** are so-called axial-lead devices, meaning the inbound and outbound leads to and from the device are in line with the component.

All axial resistors are marked with coloured bands, as shown in the diagram on the left, that indicate their resistance value in ohms (Ω), as well as their manufacturing tolerance as a percentage.

The resistors to be used with the PCB are either carbon or metal film with 4 or 5 band coding as follows:

1kΩ resistor coloured bands:

brown–black–red–gold which means:

1 0 x100 5% i.e. 1000Ω or 1kΩ ±5%

470Ω resistor coloured bands:

4 band: yellow–purple–brown–gold which means:

4 7 x10 5% i.e. 470Ω ±5%

5 band: yellow–purple–black-black–brown which means:

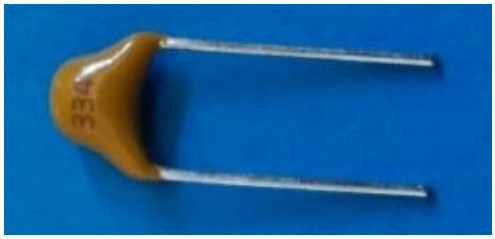
4 7 0 x1 1% i.e. 470Ω ±1%

Capacitors

A capacitor is a passive electrical component, available in a range of shapes and sizes, as shown right, that is designed to store electrical energy.

As these devices store energy, if a capacitor has a high value of capacitance i.e., it can store a lot of energy, it does need to be treated with some caution since the abrupt discharging of all the stored energy can cause damage or even personal injury!

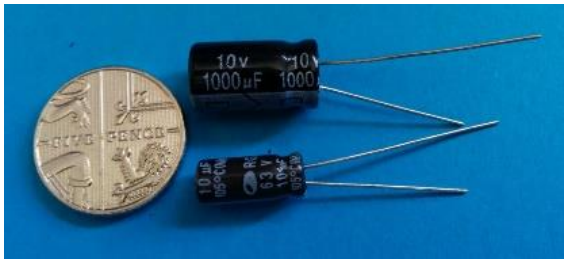
The unit of capacitance is called a farad (F) which is the number of coulombs per volt (C/V), where a coulomb is a measure of electrical charge. Capacitors used in general electronics have typical vales from 1 pF (10^{-12} F) to about 1 mF (10^{-3} F).



The type of capacitor used with the **PCB**, as shown left, is a ceramic disc capacitor with a nominal value of 330nF ($330x10^{-9}$F) and it can operate up to 50V.

This type of capacitor is quite common as they are generally small and relatively cheap although they are not particularly robust.

Another common type is an electrolytic capacitor as shown in the image on the right. These capacitors must be connected with the right polarity and the longer lead will usually be the +'ve and the –'ve lead may be indicated by a stripe on the main body.



Electrolytic capacitors are generally physically larger than ceramics, as can be seen above compared to the 5p coin. They can also have higher capacitance values i.e., the two shown in the image are 1000µF and 10µF respectively, so as mentioned before they should be used carefully when being discharged.

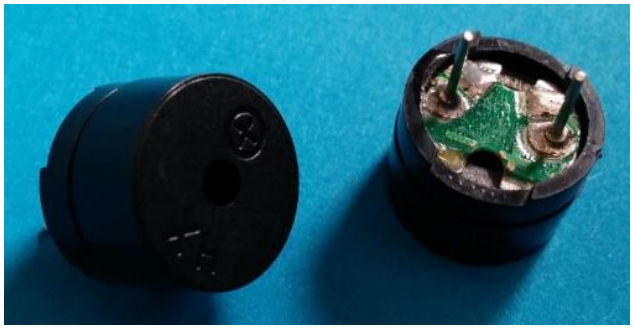
Buzzers

There are two types of small piezoelectric buzzer or beeper that are commonly available, usually described as either passive or active.

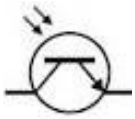
They are called piezoelectric buzzers as they use a small electromechanical element made from a material that exhibits the piezoelectric effect i.e., it generates electricity when deformed, or for a buzzer, it will deform when electricity is applied to it.

An active piezo buzzer, as shown left, contains not only the piezo element but also some additional internal electronic components so that it only requires a DC voltage to be applied to it to make a sound, which will be at a single defined pitch. This type of device will usually have the underside 'potted' so that the internals are not visible, they may have a longer pin to designate the +'ve input as well as a + sign being stamped on the top, and as shown above are often shipped with a small sticky label to protect the buzzer's 'sound opening'.

The **Raspberry Pi PCB** however uses a passive piezo buzzer type, as shown right. This type of device is simpler than the active type, not having any additional electronic components and it is not potted on the underside. But does have the +'ve pin indicated on both the top and under sides.



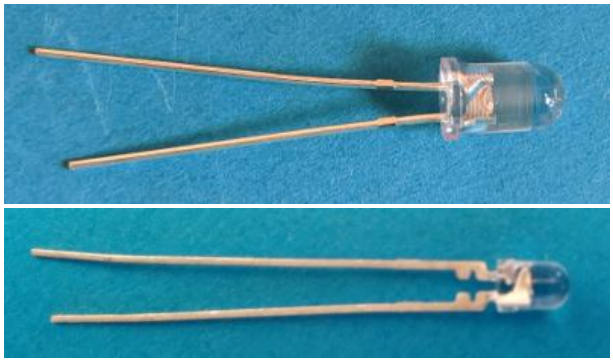
Without the additional electronics a passive buzzer requires an alternating voltage input to generate a sound, and whilst this does require more 'effort' in writing the software to achieve any sound, it means that sounds with a different pitch can be generated from the one device.

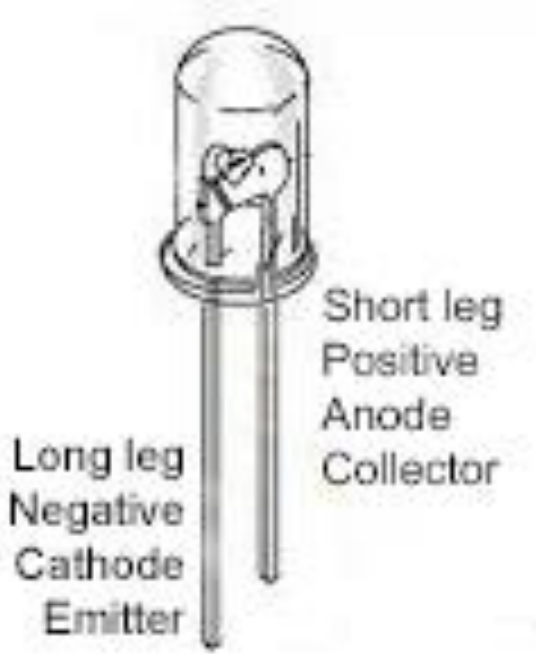
Phototransistors 

A phototransistor is a semiconductor light sensor formed from a transistor set inside a transparent cover. It can be thought of as the opposite of a light emitting diode (LED) i.e., instead of lighting up when some current flows through it, it will generate a current when light shines on the device.

In the same way that LEDs have been 'tuned' to emit light at different wavelengths (colours), phototransistors are designed to be sensitive to particular wavelengths.

The phototransistor defined for use with the **Raspberry Pi PCB** is one of the types shown right and has its output tuned to the visible light range.





Phototransistors can be easily confused with LEDs as they can be of very similar construction, so take care to correctly identify the appropriate component which will usually have a transparent casing.

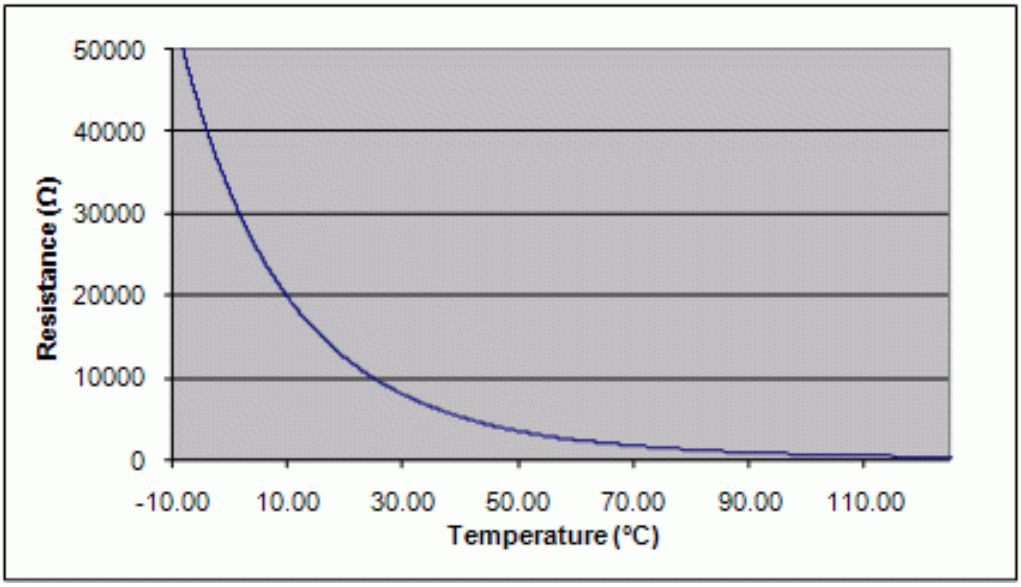
Also, as shown left, it should be noted that the longer lead is the emitter i.e., the equivalent of a –'ve or cathode connection.

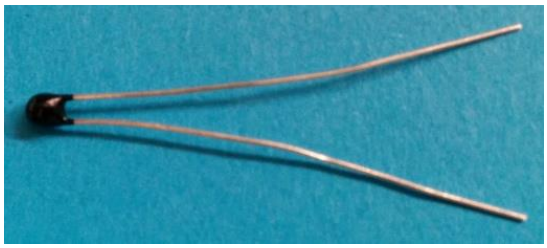
Phototransistors are not usually precise devices, so they cannot be used for any application that requires an exact measurement of light level, but with some empirical calibration they are perfectly adequate for detecting different degrees of it 'getting dark' and can be successfully used in, say, a home automation project to decide when to switch lights on or off.

Thermistors

A thermistor is a passive electronic component that, with temperature, changes its electrical resistance over a wide range and is very commonly used as: a temperature sensor, a current limiter; or a self-regulating heater element.

For a temperature sensor, a thermistor with a Negative Temperature Coefficient (NTC type) would usually be employed. This means that the resistance of the device decreases (usually quite substantially) with an increase in temperature as illustrated in the graph below.





The thermistor used with the **PCB**, as shown left, is a small black ethoxyline resin coated ceramic bead with a pair of uninsulated tinned copper wire leads.

This thermistor has a reference resistance of 10kΩ at 25°C and is suitable for measuring temperature over a range of -30°C to +125°C.

Appendix C: Software download

Any model of Raspberry Pi with 2x20 GPIO pins can be used with the PCB, but the Raspberry Pi 4 or Pi 5 with at least 2GB or memory is recommended as well as a SD card that is ideally at least 32GB.

You should also be using at least the latest Bookworm version of the operating system – if you have an earlier operating system version, you should carry out an update or create/install a new SD card before downloading the software and documentation for use of this PCB.

For Scratch and Python coding, whilst some knowledge is useful this is not essential since the PCB provides an opportunity to explore and start to understand coding with both Scratch and Python.

With your Raspberry Pi started in 'Desktop' mode and connected to the Internet, this document, plus additional support documents and all the software for the various project and method explorations can be downloaded to the Raspberry Pi by executing the following commands in an opened 'Terminal' window.

N.B. the $ sign in the command lines below signifies the prompt character in your terminal application, which you do not need to type.

First run the following command to download an initial control script where you substitute your username for YOURUSERNAME:

```
$ wget -O /home/YOURUSERNAME/RPi_maker_PCB5.sh https://onlinedevices.co.uk/dl1389
```

(take great care to type this correctly and if you get an error then recheck what you have typed)
note: -O above is an upper case letter O and the last set of characters are lower case DL1389

The control script contains all the 'commands' to set up the required folders on the Raspberry Pi's file system, to download the documentation and Scratch + Python exploration scripts and to also install the various required system functions/libraries.

You then run the following two commands to prepare and run the downloaded control script which when run, will then download all the files, and store them on your Raspberry Pi in a main folder and various subfolders starting at *./RPi_maker_PCB5*.

```
$ chmod +x RPi_maker_PCB5.sh
$ ./RPi_maker_PCB5.sh
```

This overall process can be repeated at any time, and it will then download any updated or new material, but obviously if any customisations have already been carried out these should be stored under separate file names/folders since repeat downloads will overwrite the original files.

Finally, please note that the ***RPi_maker_PCB5_readme.txt*** file that is downloaded as part of the documentation is a version control file that lists all the download material. Please review this file once downloaded as it will indicate any additional documents/code that may have been added to the set of support materials.

Appendix D: Flask web server

Flask is a lightweight web server that integrates with Python to allow a browser based interface to be developed for a Python application.

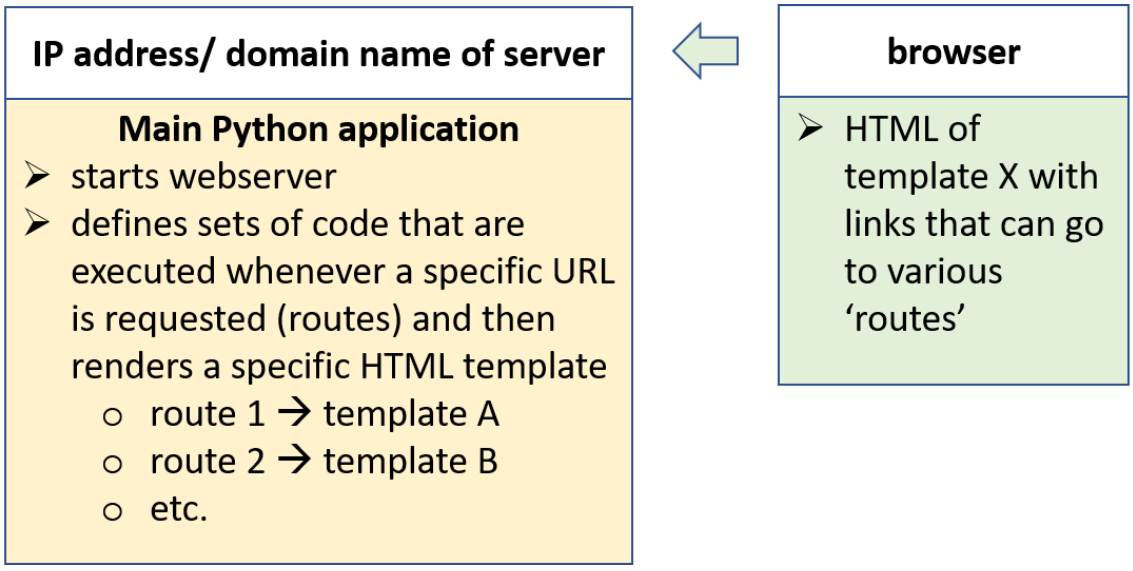
With the PCB this capability is used in:

- an Electronics demonstration example to show how a LED can be controlled through a web interface, and
- in image streaming, to allow not only a real-time video to be streamed from a USB camera to a browser, but also for a number of camera parameters to be changed through a web interface.

This appendix provides some short notes on the Flask set up, but for an in-depth understanding of Flask, the reader is referred to http://flask.pocoo.org/, and also recommended is the excellent book by Miguel Grinberg, "Flask Web Development" O'Reilly ISBN: 978-1-449-37262-0

How it all works

Flask provides a very flexible 'flow' arrangement that allows customised HTML web pages, created with HTML templates, to invoke specific sections of Python code, passing data to it - which in turn can 'call' other HTML web pages, passing data back to the HTML being viewed in a browser – as illustrated in the schematic below.



In the context of use on a Raspberry Pi, that is connected to a local network, any browser on another device, also connected to the same local network, can therefore 'connect' to the Raspberry Pi's IP address/host name. This then provides a web interface to the Python application that is running on the Raspberry Pi.

The 'Electronics' demonstration software components

File name	Folder path	Brief description/ purpose
LED1_flash_web.py	./RPi_maker_PCB5/electronic_basics/ebasics_web_controller/	Main Python application when code is run as YOURUSERNAME
electronics_layout.html	./RPi_maker_PCB5/electronic_basics/ebasics_web_controller/templates	HTML template providing an overall layout
electronics_header_insert.html	./RPi_maker_PCB5/electronic_basics/ebasics_web_controller/templates	HTML template providing a common header content
electronics_select_mode1.html	./RPi_maker_PCB5/electronic_basics/ebasics_web_controller/templates	HTML template providing a web interface for the streamed video
led1_setup_mode.html	./RPi_maker_PCB5/electronic_basics/ebasics_web_controller/templates	HTML template providing a web interface for the main options
run_led1.html	./RPi_maker_PCB5/electronic_basics/ebasics_web_controller/templates	HTML template providing a web interface for LED operations
normalize_advanced.css	./RPi_maker_PCB5/electronic_basics/ebasics_web_controller/static/css	CSS file for 'styling' the HTML
skeleton_advanced.css	./RPi_maker_PCB5/electronic_basics/ebasics_web_controller/static/css	CSS file for 'styling' the HTML
favicon.png	./RPi_maker_PCB5/electronic_basics/ebasics_web_controller/static/images	FAVICON image for the browser tab
RPi_PCB05_20210419_160549156_900w.jpg	./RPi_maker_PCB5/electronic_basics/ebasics_web_controller/static/images	Image used in the web page header area
RPi_PCB05_front_image01.png	./RPi_maker_PCB5/electronic_basics/ebasics_web_controller/static/images	Image used in the web page header area

The 'Image Taking' demonstration software components

File name	Folder path	Brief description/ purpose
image_streaming_app_root_annotate.py	./RPi_maker_PCB5/image_taking/image_taking_controller	Main Python application when code is run using 'sudo'
image_streaming_app_user_annotate.py	./RPi_maker_PCB5/image_taking/image_taking_controller	Main Python application when code is run as YOURUSERNAME
image_camera_usb_opencv_annotate.py	./RPi_maker_PCB5/image_taking/image_taking_controller	Custom Python class for the USB camera that also adds text annotations to each image
cam_setup_mode.html	./RPi_maker_PCB5/image_taking/image_taking_controller/templates	HTML template providing a web interface for the camera set up
select_mode.html	./RPi_maker_PCB5/image_taking/image_taking_controller/templates	HTML template providing a web interface for the main options
stream_video_mode.html	./RPi_maker_PCB5/image_taking/image_taking_controller/templates	HTML template providing a web interface for the streamed video
normalize_advanced.css	./RPi_maker_PCB5/image_taking/image_taking_controller/static/css	CSS file for 'styling' the HTML
skeleton_advanced.css	./RPi_maker_PCB5/image_taking/image_taking_controller/static/css	CSS file for 'styling' the HTML
favicon.png	./RPi_maker_PCB5/image_taking/image_taking_controller/static/images	FAVICON image for the browser tab

Appendix E: Setting up a ram drive

Creating a 'ram drive' is a way of using the computer's memory 'as if' it was the type of media that would be used as a 'drive' to read/write files. This type of drive obviously does not create a permanent record of the data in a file since the data is lost whenever the computer is powered down. It is however a way to store/retrieve data quickly on a temporary basis and in usage situations where there is an extremely high/frequent read/write cycle it avoids premature aging/failure of a conventional drive medium such as a SD card.

A ram drive is configured by editing the */etc/fstab* file (which needs to be done as sudo) which will then establish the drive whenever the computer is powered up.

As an example, for the image streaming method where a .jpg file is stored/retrieved very rapidly (i.e., many times/second) the following line should be added to the */etc/fstab* file:

```
none    /mnt/ramimage    ramfs    noauto,user,size=2M,mode=0770    0    0
```

where:

- `/mnt/ramimage` is the mount point folder, where the *ramfs* filesystem will be mounted and this should have previously been created;
- the `noauto` option prevents the drive from being mounted automatically (e.g., at system's boot up);
- the `user` makes this mountable by individual regular users (as well as root);
- the `size` sets this "ramdisk's" size, e.g., just 2M in this example since just a single image file is to be stored, and;
- the `mode` is very important: by using the octal code 0770 only root and the user who mounted this filesystem, will be able to read and write to the drive, not any other users.

It should also be noted that only one user can use the ram drive at any one time!

Appendix F: Control methods and 'plug in' component details

This appendix provides information about various control methods and further details on various components that can be 'plugged in' to the assembled PCB, and for which example software is made available for their use.

Passive Infra-Red sensors (PIRs)

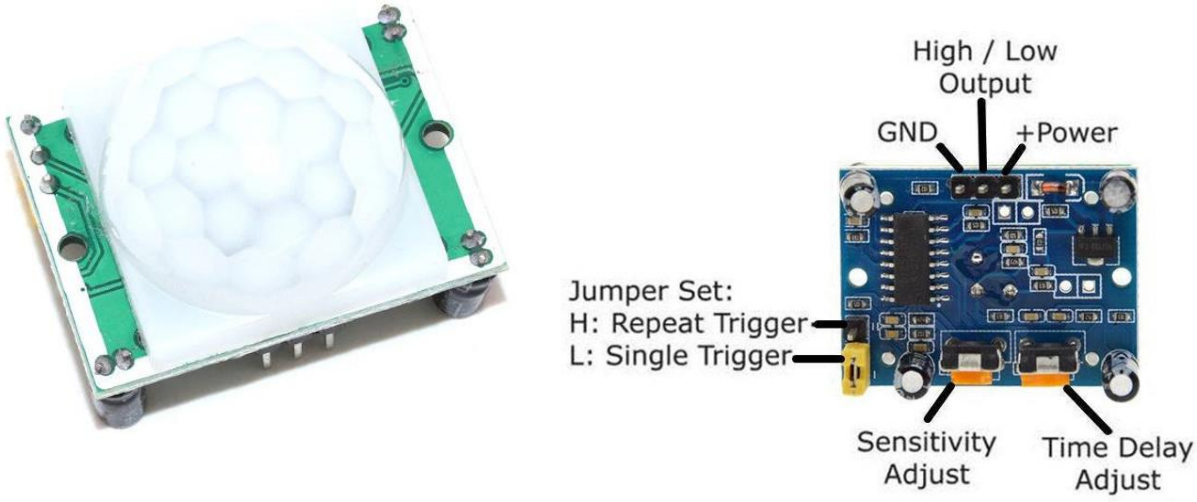
A Passive InfraRed sensor, or PIR sensor, is an electronic module that measures infrared (IR) light radiating from objects in its field of view. They are commonly used in PIR-based motion sensors packaged as part of alarm or automated lighting systems and have become a mass produced, commodity component in recent years.

They are described as 'passive' because they do not generate or radiate any energy of their own for detection purposes, but instead simply rely upon detection of energy given off by other bodies. Therefore, they cannot detect the movement of bodies that do not radiate heat energy, unlike an 'active' detection method such as radar or sonar which generates a signal and interprets a reflected return signal.

All objects that are at a temperature above absolute zero will emit heat energy i.e., they will be emitting radiation in the infrared spectrum, which is not visible to the human eye, but is easily detected by appropriate electronics. These normally comprise a solid state pyroelectric sensing element and electronics that detect a change in the overall sensed level of radiation, e.g., being caused by a warm body moving in and out of the field of view. The sensor will convert this change in radiation level to an output voltage level that can be used to trigger further processes.

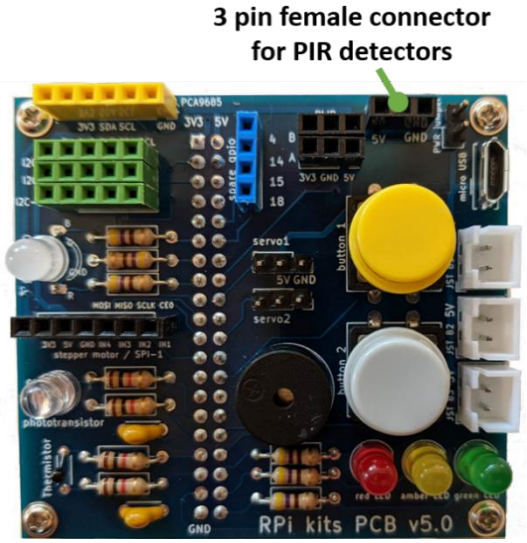
More details on PIRs can be found at this **Wikipedia link**.

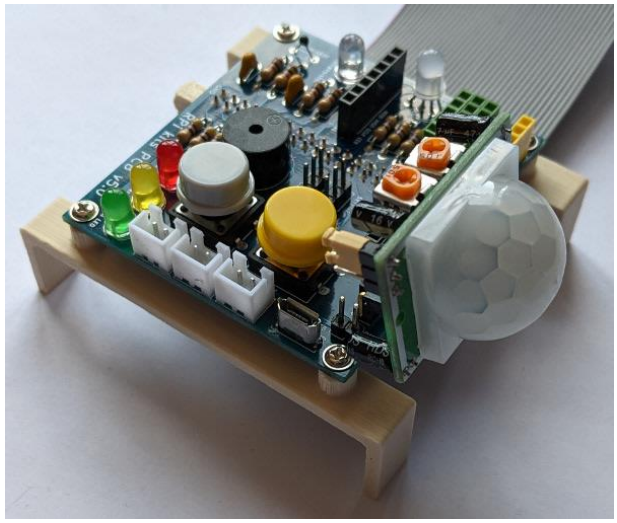
A very commonly available PIR sensor module, often described as HC-SR501 but may be called something else, is shown below along with a schematic showing its various connections and ways of adjusting its performance.



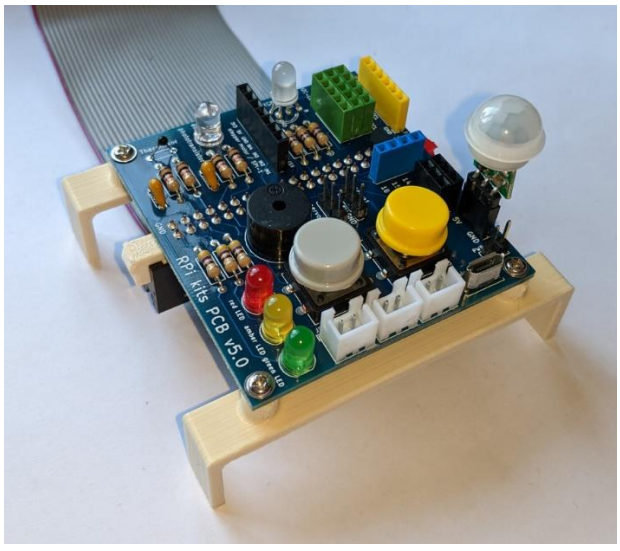
The +Power and GND pins are used to provide a 5V DC power supply to the electronics and the High / Low Output pin will provide a 3V3 'detection' signal, i.e. whenever there is a variation in the infrared radiation this pin will go HIGH.

The assembled PCB has a dedicated 3-pin female connector, highlighted in the schematic below left with its connections aligned to suit the HC-SR501 PIR. Therefore, using a right angled 3-pin male-female adaptor it can simply be inserted into the PCB as shown below right (ensuring it is connected the right way round! - the PCB PIR 1st and 3rd connection points are labelled 5V and GND to ensure the correct orientation is used)





Other types of PIR are also commonly available and if they have a similarly oriented set of three conectors then they can also be directly inserted into the PCB as illustrated with the mini HC-SR312 shown below.



Servo motors

A servo motor is a packaged motor and gearbox with control circuitry that acts as a rotary or linear actuator with control of angular or linear position. Servos are used extensively in industrial control applications.

Importantly the motor has an in-built sensor that gives position feedback to its controller, so the actuator is a so-called *closed-loop* servomechanism i.e., the current position of the actuator is continuously compared to the command position and any difference is fed back to the controller so that the motor is made to move further towards its required position. A servo motor will consume power as it moves to its command position but once there it stops, unlike a stepper motor as discussed later.

The low cost SG90 micro servos, shown right, are simple servos commonly used in radio controlled models and only provide position feedback with so-called bang-bang control, i.e., the motor is either at full speed or stopped.

More sophisticated servos used in industrial control will often use both position and speed feedback and can vary both the motor speed and direction to provide more accurate and smooth control.

The SG90 servos can rotate approximately 180°, (90° in either direction) and are supplied with a selection of plastic 'horns' that fit on to the splined gear box shaft. Control of the servo is provided by applying a Pulse Width Modulation (PWM) signal to the orange signal wire, where the PWM Frequency is set to 50Hz and the middle, full left and full right positions are set by the PWM Duty Cycle being set between 1 and 2 milliseconds (ms). A fuller description of Pulse Width Modulation and its various parameters is provided next.

The starter code available for the PCB provides some basic methods of controlling servos to enable you to go on and use them in projects of your own. Python code for a fun semaphore demonstration is also provided, and additional interesting ways of using servos are illustrated below (click each image to link through to online videos that are available on the web):

hexabot robots

automated marionettes

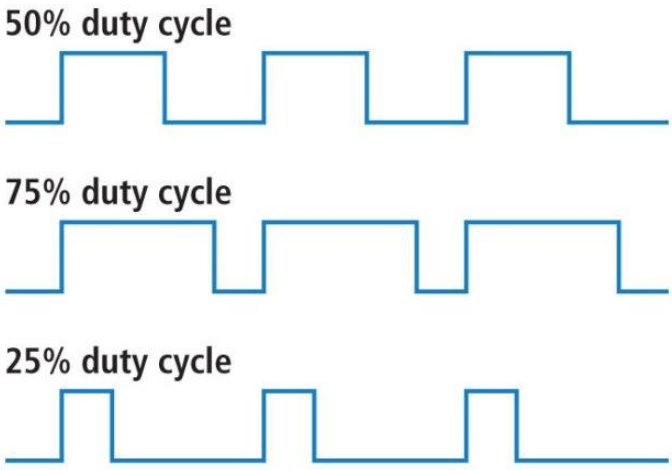
automated xylophones

Pulse width modulation

Pulse-width modulation (PWM) simply means a digital signal that is on/off (pulsed) for a period of time, where the **frequency** at which the on/off switching occurs can be set, and where the amount of time spent on, versus off, known as the **duty cycle** and expressed either as a time or as a percentage, can also be set.

PWM can be used to encode a message into a pulsing signal, but it is also commonly used to control the amount of power supplied to electrical devices, such as electric drive motors and LEDs.

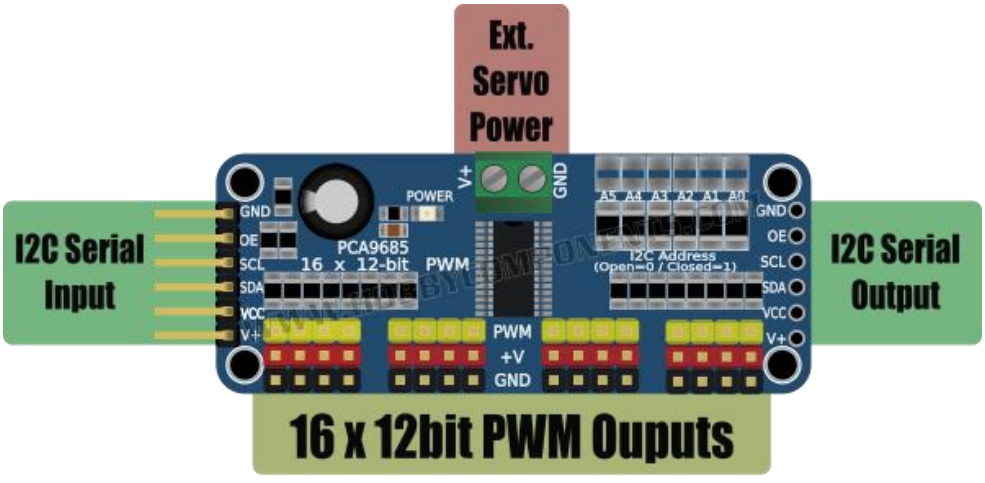
By switching the applied voltage on/off, the average voltage (and therefore current) that is applied to the electrical device is controlled by using different duty cycles, e.g., a duty cycle of 0% means no power and a duty cycle of 100% means full power. The schematic on the right illustrates some other duty cycles where the frequency is exactly the same for all three examples.



The frequency at which the switching occurs needs to be appropriate for the electrical device being powered, so that there are no negative effects i.e., the resultant wave form that is applied to the device is 'perceived' as something that is very smooth. For example, if the frequency used to control a LED was too low you might see the light varying in intensity, or for a motor it might not rotate smoothly. A typical frequency for use with a drive motor might be 50Hz, whereas you would need about 500Hz for a LED to avoid visible 'flicker'.

PCA9685 control module

The PCA9685 board is a I^2C controlled device that can apply Pulse Width Modulated power to 16 devices such as LEDs or servo motors.



As shown in the schematic above, there are six input pins, GND, OE, SCL, SDA, VCC and V+, and these are mirrored as outputs (but no pins inserted) so that additional devices could be added downstream on the bus. GND and VCC are a 5V DC input to power the board and V+ is power input for the PWM devices if this can be supplied down the bus. The SCL and SDA pins are the two main I^2C control lines and the OE (Output Enable) pin is effectively an overall on/off switch which has a default LOW setting for 'on'. For the PCB's servo methods, the V+ is not used, as the Pi cannot support sufficient power for multiple servos, nor is the OE used, as this is unnecessary.

There is also a pair of screw terminals to provide a separate power source for the 16 PWM controlled devices and 16 sets of 3 pins for the PWM devices, designated as channels 0 to 15 (left to right on the diagram above).

I^2C bus

The I^2C bus [1] was originally designed by Philips in the early 1980s to allow easy communication between components which reside on the same circuit board, and the name stood for "Inter IC" i.e., Inter Integrated Circuit, sometimes shown as IIC but more commonly as I^2C.

I^2C is now, not only used on single boards, but also to connect components which are linked via cable, and it is its simplicity and flexibility that make this bus attractive for many applications.

The most significant features of I^2C include:

- Only two bus lines are required for the data transport and signalling:
  - SDA: serial data (I^2C data line)
  - SCL : serial clock (I^2C clock line)
- A simple master/slave relationship can be defined between all components on the bus with multiple masters being possible.
- Each device connected to the bus is software-addressable by a unique address.
- There are no strict data exchange speeds (baud rate) requirements like for instance with RS232, as the master generates a bus clock speed.
- I^2C is a true multi-master bus providing arbitration and collision detection.

Stepper motors

Stepper motors have many industrial uses and come in many sizes and shapes.

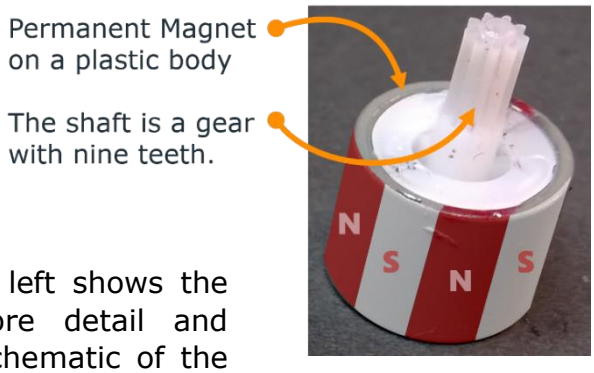
The image on the right shows a dis-assembled stepper commonly used by digital makers that is low cost and widely available.

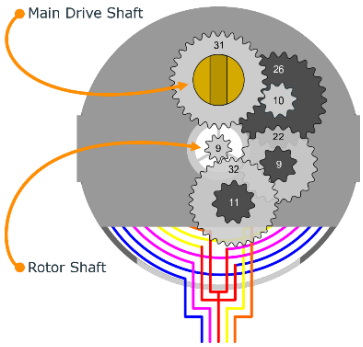
[1] *a bus is a shared digital pathway between multiple resources and devices in an electronic system where signals and data can be exchanged between everything connected to the bus using defined protocols.*

This 28BYJ-48 stepper motor's separate components consist of:

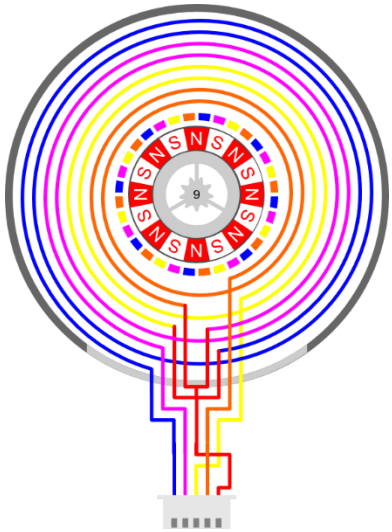
- An outer case with 8 teeth at the base and a centre post for the rotor.
- Two wire coils (windings) in white plastic rings with 3 plates that also have 8 teeth.
- Each coil acts as an electromagnet but they also have a common 'centre tap', so each coil has three ends or is effectively two coils and there are therefore 5 leads:
  - Blue/yellow are the ends of one coil,
  - Pink/orange are for the other, and
  - The red lead is for the common centre tap.
- A rotor body with a magnet wrapped around outside.
- Gears that finally attach to a motor shaft that goes through a top plate which produces a 64:1 gear ratio.

The image on the right illustrates the rotor's permanent magnet poles and it also shows the attached 9-tooth gear that is part of the overall 64:1 gear train.





The image on the left shows the gear train in more detail and below right is a schematic of the four coils that magnetise the 32 teeth which will align with the rotor's magnetic poles.



Rotor movement is achieved by energising the coils in a specific sequence which will magnetise the metal teeth that are set around the coils, causing them to align to the rotor's magnetic poles.

Each energising of a coil therefore produces a 'step' movement of the rotor. With 32 teeth, 32 steps will rotate the rotor one full revolution, and with a 64:1 gear train, 2048 steps will rotate the drive shaft once.

Therefore, depending upon the sequencing of the energisation of the coils, and the frequency that this is carried out, this will determine the overall drive shaft rotation amount and direction, as well as its speed of movement.

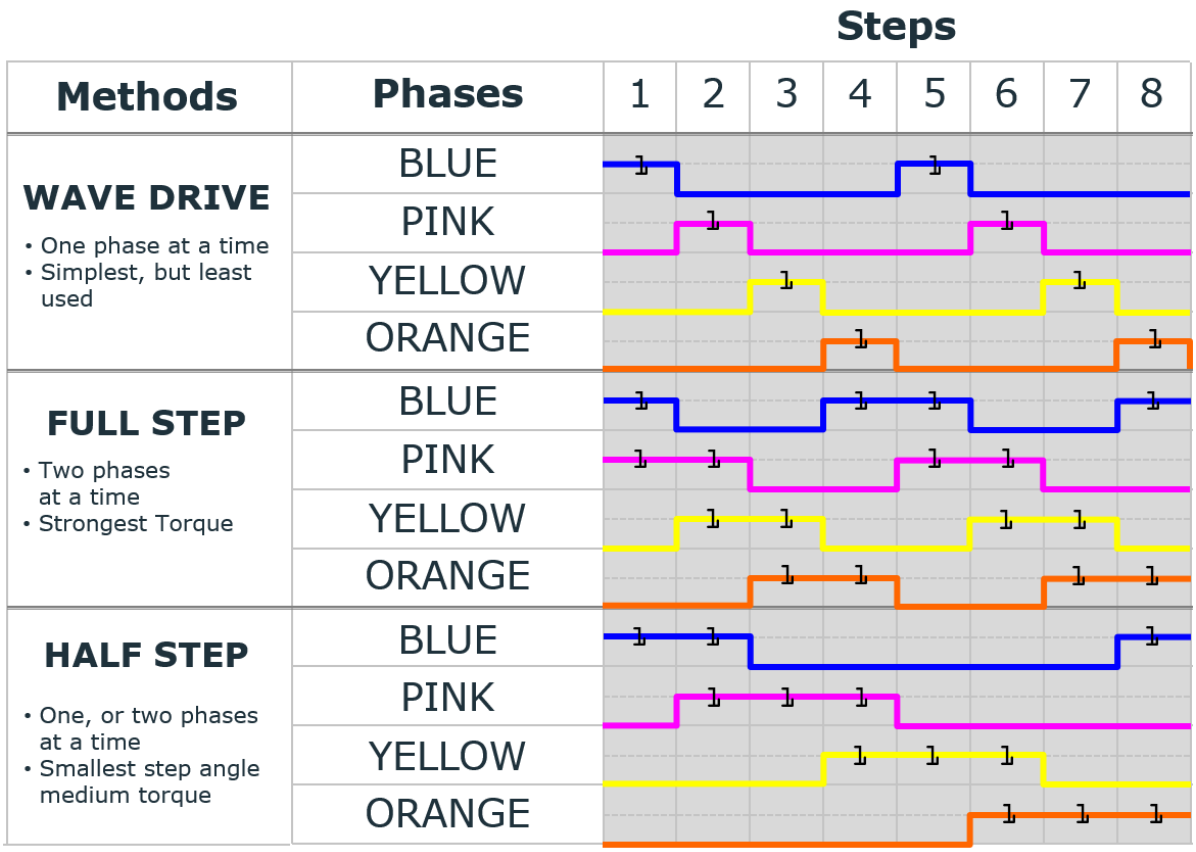
The 28BYJ-48 stepper motor plugs directly into its control board, which uses a ULN2003 integrated circuit (described in more detail in the next section), and a set of four GPIO pins on the Raspberry Pi can then 'apply' the energisation sequencing to achieve the required stepper motor movement.

With this overall physical arrangement, several 'stepping' methods are possible which are summarised below.

A **Wave Drive** approach is the simplest method as this just sets each GPIO pin HIGH for a period of time, which therefore energises and de-energises each individual coil one at a time producing a single 'step', and after every 4-step 'sequence' the pattern is repeated to achieve continuous motion. This method, although simple, is however not used very often since the next two methods have superior characteristics.

A **Full Step** approach energises two coils at a time which means that twice the power is applied to the rotor to achieve a single step, producing a stronger drive torque. As with the Wave Drive, after every 4-step 'sequence' the pattern is repeated to achieve continuous motion.

A **Half-Step** approach energises either one or two coils at one time which has the effect of creating a different single step that is half the amount moved in the previous two methods i.e., 4096 'half' steps are needed to achieve one full drive shaft rotation. This does not produce as much torque as the Full Step method, but it obviously achieves a finer resolution of movement.

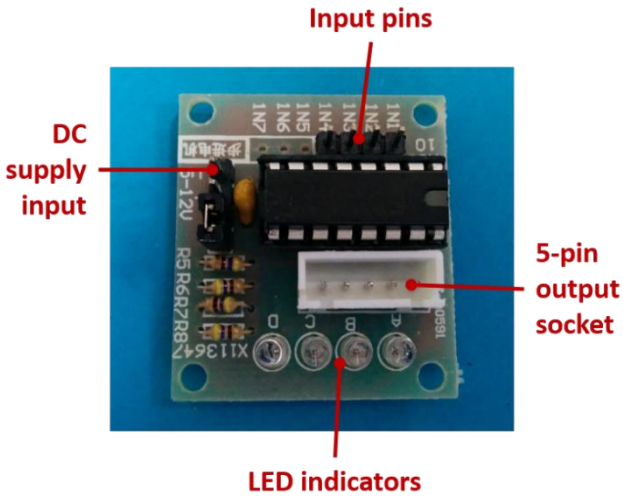


ULN2003 control board and Darlington arrays

The primary component of the ULN2005 control board is a ULN2003 integrated circuit (IC) with seven high voltage and high current Darlington transistor pairs in an array.

Each Darlington pair is essentially a current amplifier and each one can be used independently to switch and amplify an input signal current to a higher level so that it can be used to drive a variety of devices such as relays, LEDs and of course stepper motors.

The control board, as illustrated on the right, packages the IC with input pins (IN1 – IN4), an output socket that the stepper motor can plug in to, LED indicators for the four input/output streams, and a pair of DC supply input pins.



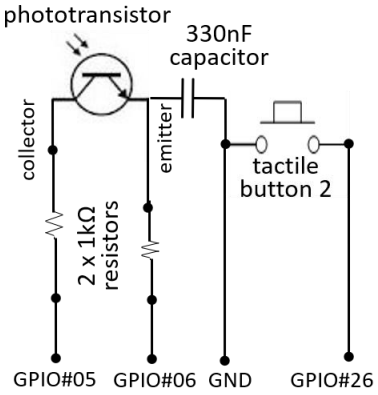
Only four of the Darlington pairs are exposed with input pins – as this is all that are needed for driving the 28BYJ-48 stepper motor. These input pins are used to amplify input electrical pulses applied to the IN1 – IN4 pins, by connecting them to Raspberry Pi GPIO pins which are set HIGH/LOW in a set sequence and frequency.

These input signals are also used to light the four LEDS (A – D). It should be noted however that for stepper motor control, the electrical pulses are usually at such a high frequency that the LEDs will appear to be permanently lit.

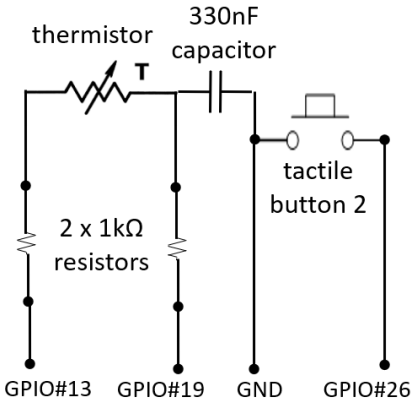
Appendix G: RC charging circuit analysis

A so-called RC charging circuit is used with both a phototransistor and a thermistor in the assembled PCB to provide approximate measurement techniques for light level and temperature.

The phototransistor circuit shown on the right connects the phototransistor to two GPIO pins via 1kΩ resistors, along with a 330nF ceramic capacitor connected between the phototransistor's emitter and ground.



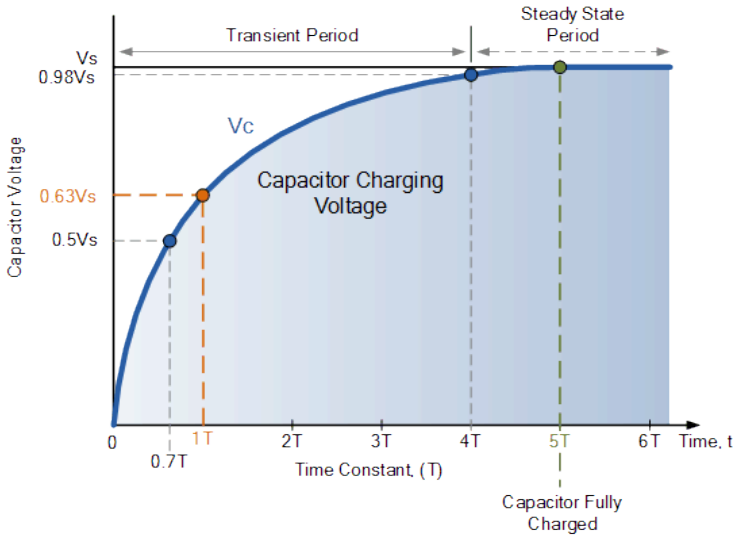
This arrangement allows a simple measurement technique by successively discharging the capacitor and timing how long it takes to recharge.



In a similar way, the thermistor circuit shown on the left connects a thermistor to two GPIO pins via 1kΩ resistors, along with a 330nF ceramic capacitor connected between one side of the thermistor and ground.

This arrangement also uses the technique of successively discharging the capacitor and timing how long it takes to recharge.

Both measurement methods rely on the capacitor charging rate being of the form as shown on the right where the time constant T (tau) equals R*C, and R is the combined resistance of the 1kΩ resistor and either the phototransistor or the thermistor, and C is the capacitance of the ceramic capacitor.



A next assumption is that when charging at 3.3V the sensing GPIO pin will go HIGH at 1.25V which is when the capacitor will stop being charged and is therefore the time that is measured t_m.

A further assumption is that the initial portion of the curve up to 0.7 T can be approximated by a straight line so T (tau) can therefore be calculated as a simple ratio:

$$T = (t_m * 0.5 * 3.3) / (1.25 * 0.7)$$

So, the combined resistance is therefore T/C from which the individual phototransistor or thermistor resistance can be calculated.

For a light level measurement the phototransistor resistance can be used as a simple proxy but once the thermistor resistance is known the temperature can be deduced from a relationship such as the Steinhart-Hart equation (see **this ref**).

Appendix H: Raspberry Pi GPIO pin default settings

PLEASE NOTE: *this information is only relevant for older versions of the Raspberry Pi OS that can support Scratch 2 and will generally be irrelevant to most users today.*

On power up/reboot all the Raspberry Pi GPIO pins default to being INPUTs and:

- Pins 2-8 have their pull resistor set to PULL UP, with
- All the other pins having their pull resistors set to PULL DOWN.

When programming with Python, Scratch 1.4, and Scratch 3 the 'pull' status of a Raspberry Pi GPIO pin can be explicitly set by the code, and all the provided example code does this appropriately.

However, Scratch 2 on the Pi does not have an option to set the pull up/down status for a GPIO pin, so a tactile button that needs to be pulled-up should either be connected to any of the pins 2-8, or the default status of its pin needs to be changed by editing the /boot/config.txt file as shown below.

From a PCB layout point of view, it made more sense for the buttons on the assembled PCB to use pins 7 and 26, so if Scratch 2 needs to be used the following reconfiguration of the Raspberry Pi can be carried out on a one-time basis:

Use the following command in Terminal window to edit the config.txt file

sudo nano /boot/config.txt

… and add the following two lines to the file

 # change the pull on (input) pin 26
gpio=26=pu

END OF DOCUMENT