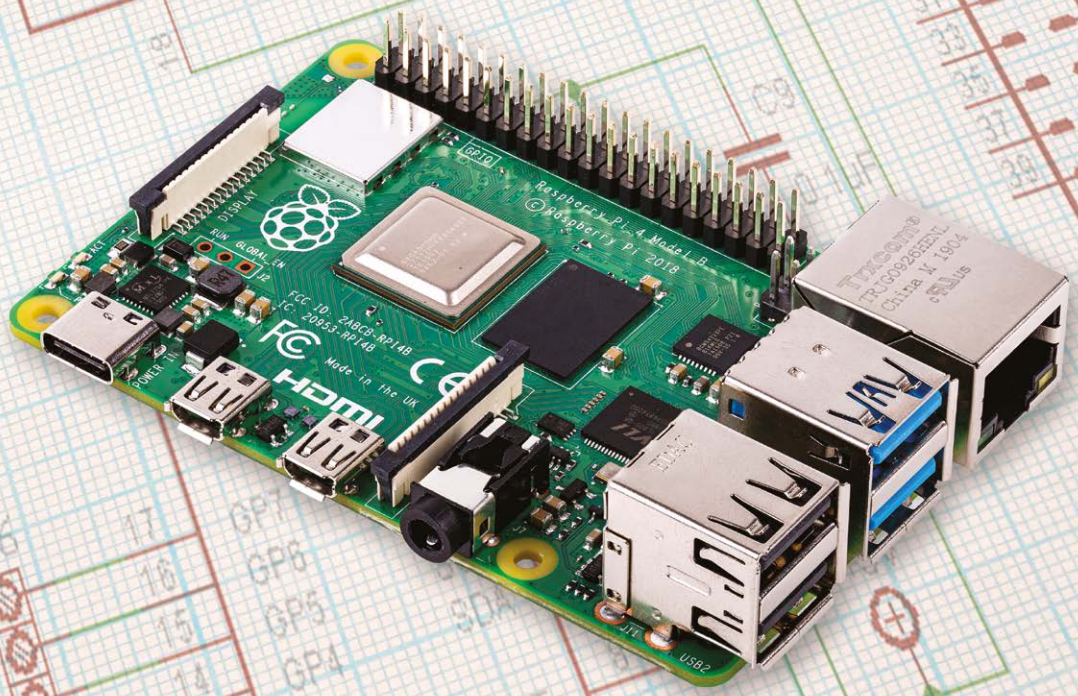
Explore the Raspberry Pi
in 45 Electronics Projects

3rd Edition
Fully updated for
Raspberry Pi 4



Bert van Dam

elektor

LEARN › DESIGN › SHARE

Explore the Raspberry Pi in 45 Electronics Projects

Bert van Dam

elektor

LEARN DESIGN SHARE

● This is an Elektor Publication. Elektor is the media brand of
Elektor International Media B.V.
78 York Street, London W1H 1DP, UK
Phone: (+44) (0)20 7692 8344

● Declaration

The author and publisher have used their best efforts in ensuring the correctness of the information contained in this book. They do not assume, or hereby disclaim, any liability to any party for any loss or damage caused by errors or omissions in this book, whether such errors or omissions result from negligence, accident or any other cause..

● British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

● **ISBN 978-1-907920-82-0**

Prepress Production: D-Vision, Julian van den Berg
3rd Edition. Fully updated for Raspberry Pi 4
First published in the United Kingdom 2020

Introduction

In this book, we concentrate on one of the more powerful aspects of the Raspberry Pi: the combination of programming and electronics. The Raspberry Pi costs just a handful of dollars, yet you get a complete computer, which you can easily connect to all kinds of electronics.

After a short introduction to the Raspberry Pi we install the required software. If you bought the SD card for this book you'll find that all the Raspberry Pi software you need is already installed on the card. All the software for the (optional) Windows PC is free and can be downloaded in a single, convenient package.

This is followed by a quick overview of Linux (the operating system), and we start programming in Bash, Python and JavaScript. We concentrate mostly on Python, but we keep it short for all of them. We look only at the most important details, and then get involved with the projects straight away.

There are over 45 exciting and fun projects in this book. It varies from flashing lights to a motor speed controller; from processing and creating analog signals to a CGI web server and client-server programs. You can use this book as a project book, building the projects and using them in practical applications. The clear explanations, circuit diagrams, and photos of the setup on a breadboard make building the projects an enjoyable experience. When you're not building the projects in sequence, you should read section 5.1 first, since it contains important safety instructions.

You can also use this book as a study guide. For each project an explanation is given why it was designed in a certain way. This teaches you a lot about the Raspberry Pi and the components used, enabling you to enhance the projects or even combine several projects.

The book is also very handy as a reference guide. You can easily find projects via the index, which can then be used as a starting point for your own projects. Even if you've built all the projects, it still deserves a place next to your Raspberry Pi for this reason.

I would like to take this opportunity to thank Bert Oudshoorn for his help with the final editing of the original Dutch version of the book, and for testing the projects.

Bert van Dam

Chapter 1 • Requirements

You will obviously need a Raspberry Pi, but that's not all. As a minimum, you will also need the following components:

- Power supply
- SD card with operating system
- Solderless breadboard
- Discrete components
- Software oscilloscope (optional)

More information on these components can be found later in this chapter. The other components needed depend on the way in which you intend to use the Raspberry Pi.

1. Normal operation. In other words, you'll sit "behind" the Raspberry Pi and use it like a PC. In this case you'll need:

    - Keyboard
    - Mouse
    - Monitor
    - USB hub (optional)
    - Loudspeaker (optional)
    - Internet connection (optional)

2. Headless operation. In other words, you control the Raspberry Pi remotely from a PC over an intranet. Both your PC and your Raspberry Pi have to be connected to your router and you have to install some specific software on your PC (as described in section 3.2). In this case you'll need:

    - Internet connection

3. Both normal and headless operation. In this case you'll need:

    - Everything listed in options 1 and 2.

Please read the rest of this chapter before you go out and buy anything, so you have a good idea of what's best to buy.

1.1 Raspberry Pi

You will obviously need a Raspberry Pi. Although you can use the SD Card and this book with all current models, our advice is simple: buy the most expensive model. It is not much more expensive than the other models while offering more speed and peripherals. For example, model 4B has WiFi built in and boasts a Bluetooth adapter. Plus it is fast enough to act as a modest Linux desktop PC. Table 1.1 and Table 1.2 show a comparison of diferent models.

Model	**1A**	**2B**	**Zero**	**3B**
Processor	ARM1176JZ-F	Cortex A7	ARM11	Cortex A53
Clock speed	700 MHz	900 MHz	1 GHz	1.2 GHz
# of Cores	1	4	1	4
USB	1 full	4 full	1 micro	4 full
Header	26	40	401	40
GPIO on header	8	17	17	17
SD slot	standard	micro	micro	micro
Memory	256 MB	1 GB	512 MB	1 GB
Ethernet	no[2]	1 x 10/100	no	1 x 10/100
Power supply	500 mA	1000 mA	500 mA	1000 mA3
PAL/NTSC4	RCA plug	3.5 mm jack	3.5 mm jack	3.5 mm jack
Onboard WiFi	no	no	no	yes
Onboard Bluetooth	no	no	no	yes
[1] header is not supplied; you have to buy it separately and mount it				
[2] a connection to the internet can only be established via a WiFi adapter (to be acquired separately) whoch connects to USB. Obviously such an adapter can also be used with other models.				
[3] 2.5 A is recommended				
[4] we advise to use the HDMI connection, providing a much better image quality (and embedded audio)				

Table 1.1 Comparison of different models (early models)

Model	Zero W	3B+	3A+	4
Processor	ARM1176JZF-S	Cortex A53	Cortex A53	Cortex A72
Clock speed	1 GHz	1.4 GHz	1.4 GHz	1.5 GHz
# of Cores	1	4	1	4
USB	1 full	4 full	1 full	4 full
Header	40	40	40	40
GPIO on header	17	17	17	17
SD slot	micro	micro	micro	micro
Memory	512 MB	1 GB	512 MB	1,2,4 GB
Ethernet	1 x 10/100	1 x 10/100	no	1 x 10/100
Power supply	180 mA	1130 mA	470 mA	2500 mA
PAL/NTSC4	RCA plug	3.5 mm jack	3.5 mm jack	3.5 mm jack
Onboard WiFi	yes	yes	yes	yes
Onboard Bluetooth	yes	yes	yes	yes

Table 1.2 Comparison of different models (later models)

The Raspberry Pi is usually supplied on its own, without any peripheral components.

1.2 SD Card

The Raspberry Pi doesn't use a hard drive, but uses an SD card instead. This card has to contain the operating system and have enough room to store your own files. You can buy an SD card along with this book that already has the full operating system installed. It also contains all of the software required with this book, such as drivers for controlling the GPIO pins, I^2C, SPI, PWM, sound and serial communications. All of the source code from this book is also on the card, so you won't have to type any of it into the Raspberry Pi. It is a fast 16 GB card, which is suitable for all of the projects in this book.

We'll describe how to make a backup of this card in section 3.3.1. Once you have your backup, you can always revert to the original if something goes wrong.

Pro tip:
Backup the SD card **before** you use it!

1.3 Keyboard

You will need a keyboard with a USB connection. It is best not to use a wireless type to avoid potential problems with drivers. Once you have gained some experience with the Raspberry Pi you can always change over to a wireless keyboard if you prefer this. The use of a Bluetooth keyboard is optional (see Chapter 10), but you will still need a wired keyboard for the installation.

1.4 Mouse

You will need a mouse with a USB connection. It is best not to use a wireless type to avoid potential problems with drivers. Once you have gained some experience with the Raspberry Pi, you can always change over to a wireless mouse if you like.

1.5 Display

Option 1: HDMI

The standard monitor connection on the Raspberry Pi is for an HDMI monitor. The latest model Raspberry Pi 4 includes two micro size HDMI ports. If you already have such a monitor, this will be the easiest option to use. Apart from the monitor, you will also need an HDMI cable. If you are using a Raspberry Pi 4, you will also need an HDMI adapter to convert between standard size HDMI and micro HDMI. You should first turn on the monitor, then wait until the message "no HDMI signal" (or similar) appears, and only then turn on the Raspberry Pi. If you still need to buy an HDMI monitor, you should check beforehand if it is suitable for use with an Raspberry Pi. In most cases, it should be possible to use an LCD television with an HDMI connector as monitor. [1]

Option 2: Analog TV

It is also possible to use an analog TV as monitor. In that case, you will need a composite video cable (yellow plug). There is no composite connection for the sound, so you will need to connect separate loudspeakers for this. The disadvantage of an analog TV is that the resolution is fairly low, which means you won't get a very sharp picture. Another disadvantage is that large windows won't fit on the screen. In most cases, it is easy to reduce the size of the windows, but this is not always possible. For example, the windows for the Thonny (the Python editor) and the WiFi settings can't be resized. In practice you'll find an analog TV unsuitable if you're interested in programming. In context of the material provided in this book we advise against the use of an analog TV.

1 Most HDMI monitors and TVs will work straight away, but you may have to change the settings in config.txt. Details for these settings can be found at: "http://elinux.org/RPiconfig". When your display doesn't work, you won't be able to type anything in, since you can't see what you're doing. We therefore need to gain access to the Raspbery Pi in some other way, which will be via an intranet. Connect the Raspberry Pi to your router as described in section 1.6. Install the program Putty on your PC, as described in section 3.1.6. Now log in using Putty, and issue the command "`sudo nano /boot/config.txt`". You should now type in the required parameters that you found on the website we just mentioned. Save the file by holding down the Ctrl key and momentarily pressing "`x`". Then press "`y`", followed by the Enter key. The editor then stops. Type in "`sudo reboot`" to restart the Raspberry Pi. Make sure that your HDMI monitor is turned on and connected before you restart the Raspberry Pi.

Option 3: HDMI-VGA Converter

Figure 1.1 On the left is a Ligawo converter, on the right an Edimax WiFi adaptor.

If you still have an old VGA monitor, you could use this instead. However, you do need to obtain an HDMI to VGA converter (for Raspberry Pi 4 you need a micro HDMI connector). This has to be a proper converter with its own power supply. On the Internet you'll sometimes come across advice that suggests using a simple conversion cable without a power supply. Although it may work, the conversion chip in the cable then needs to get its power from the Raspberry Pi, which hasn't been considered by the designers. If you're lucky, you'll only blow up the protection diode (sometimes after a while). In that case, you have to desolder it and replace it with a new one. However, if you're unlucky you may cause some internal damage to the Raspberry Pi.

One possible converter you could use is the "HDMI zu VGA Converter Wandler digital zu analog - HDCP, 1080p, 1:1, DAC" (see Figure 1.1) made by Ligawo, which is available via Amazon.[2] You use a normal VGA cable to connect the VGA monitor to the convert er, which is connected to the Raspberry Pi via an HDMI cable (and with its own power supply).[3] You should always turn on the monitor and converter first, wait a while (say 5 seconds) and only then turn on the Raspberry Pi. If you turn on everything at the same time, you won't get a picture. [4]

2 Amazon tends to be one of the cheapest suppliers wherever you live. Make sure that you buy one with the correct AC line plug and voltage.

3 Note that you need a VGA cable where all 15 pins are connected. Some plugs have pin 9 missing (sometimes pins 11, 12 and 15 as well). Without all the pins present, you may find that you won't get a picture and/or sound.

4 This converter has been tested on the following VGA screens: Samsung SyncMaster E1920 (1280x1024, 19"), Samsung SyncMaster LS22B150 (1920x1080, 22"), Sima/Medion MD7315TB (1024x768, 15") and Philips 150 S4 (1024x768, 15"), which all had a perfect picture. You don't need to change any settings for these screens: they "just" work. However, this isn't a guarantee that your old VGA screen will also work without changing any settings.

When you use the converter for sound, you will get a better quality than from the audio socket on the RASPBERRY PI. However, you first need to turn this option on. Carry out the following instructions:

1. Open the Terminal (via the icon on the desktop).

2. Type in the following command:

```
sudo nano /boot/config.txt
```

3. Scroll down with the arrow key until you get to the following line:

```
#hdmi_drive=2
```

4. Change it so it looks as follows (i.e. take away the hash symbol):

```
hdmi_drive=2
```

5. Then press Ctrl-O (hold down the control key, press the letter O briefly, then let go of the control key) to save the file and accept using the Enter key.

6. Then press Ctrl-X to close nano. When nano has closed the LXterminal will still be open.

7. Now type in the following command:

```
sudo reboot
```

The Raspberry Pi will now reboot and will take the modified setting into account. We will cover audio in more depth in section 5.10, where you'll also be shown how to switch between HDMI and the socket on the Raspberry Pi. [5]

1.6 Internet Connection

It is not necessary to connect the Raspberry Pi to the Internet. However, when you want to try out the web server and client-server projects it speaks for itself that you *will* need an Internet connection. You will obviously also need an Internet connection if you want to use the Raspberry Pi in a headless configuration. Note that in this context by 'Internet' we mean a connection to your router.

Option 1: using a cable

The easiest way is to use a network cable. Plug the network cable into the Raspberry Pi and router **before** you turn on the Raspberry Pi and the connection will be established automatically.

5 There are some VGA screens that can't produce sound from the HDMI-VGA converter. In that case, you have no choice but to use the socket on the Raspberry Pi.

Option 2: wireless (WiFi)
You can also make a wireless WiFi connection. Most Raspberry Pi models have integrated WiFi modules, but if you have a model with no WiFi module then you will need a USB WiFi adapter. The recommended adaptor is the Edimax EW-7811Un made by Edimax (see Figure 1.1). You should plug the WiFi adapter into a USB port of the Raspberry Pi **before** you turn it on.

Figure 1.2. The WiFi symbol is on the left.

Left-click the WiFi symbol in the top right of the screen (see Figure 1.2). You will see a list of available networks. Pick the preferred one by clicking it and enter your data. "Preshared Key" is the WiFi password.

When a connection is established you can see the assigned IP address by hovering the mouse over the WiFi symbol.

1.7 Power Supply

The power supply should deliver a precisely regulated 5 V DC. The minimum required current depends on the model, but a minimum of 1 A is recommended for the older models. With model 4 this jumps to 2.5 A. Raspberry Pi 4 official power supply is recommended which provides up to 3A. It is therefore not recommended or even possible to use a USB connection on your PC to supply the Raspbeery Pi since it can supply only 500 mA. If nothing else is connected to the Raspberry Pi and it's "not too busy", you may find that earlier models may work for a while. However, you'll soon start to get sudden resets when the voltage drops (also known as brown-outs). It is even possible that you could damage the USB port on your PC if it isn't properly protected against overloads. A separate power supply is therefore the best solution.

1.8 Loudspeaker (Or Earphones)

You can connect earphones to the Raspberry Pi, or a small loudspeaker set that is also sold for use with a PC. In the latter case we recommend that you buy a type that comes with an external power supply, since it's not such a good idea to power loudspeakers via a USB connection on the Raspberry Pi. The Raspberry Pi doesn't really have much "spare" current left over. When you use an HDMI-VGA converter, it is best to connect the earphones or loudspeakers to this converter since it has a better sound quality. More detailed instructions can be found in section 1.5.

When you use an HDMI monitor you can use its built-in loudspeakers for the sound reproduction, so you won't need any extra loudspeakers or earphones.

1.9 Breadboard

For all projects we will be using a breadboard. This can be used to quickly build and dismantle a circuit without soldering. A breadboard consists of two or more strips with supply connections and a large central area. The holes in the central area are connected together

in each numbered column up to the gap in the middle. The supply strips are usually connected together horizontally, with a gap in the middle.

First of all, we have to prepare the breadboard for use. This means that we have to connect the four supply strips together. We also put a small 0.1 µF capacitor at each of the corners of the breadboard, which will prevent interference on the supply strips. A number of projects won't work (well) without these capacitors. You can see what the breadboard should look like in Figure 1.3.[6]

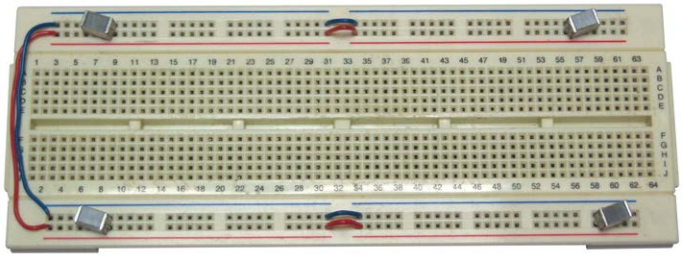


Figure 1.3. Breadboard with wire links and capacitors.

Whenever you use the breadboard, you should set it up like this, even if it isn't explicitly mentioned. Always remember this: you won't be the first to wonder why a circuit doesn't work, only to find out that part of it simply doesn't have power.

When you want to put a circuit to permanent use and decide to make a printed circuit board for it, you should remember to include sufficient decoupling capacitors. You should at least have a 100 nF capacitor across the supply next to each IC, as close as possible to the IC. Any other capacitors that are explicitly shown in the circuit must also be included, of course.

1.10 Components

Depending on what you want to build you will need some electronic components. You should ask for components that fit on a breadboard or stripboard. Some components will have pins that are too thick for a breadboard. Also, some chips such as SMD types cannot be mounted on a breadboard without special adapters. You should solder thinner leads to these pins. Don't force them into the breadboard as you could bend the internal contacts and cause permanent damage to the breadboard.

You will only need a small number of components for this book. You can buy a kit of parts from Elektor or you can use the parts list in the appendix as a shopping list. Note that the GPIO connector on the Raspberry Pi consists of a set of vertical pins, for which you need to use single way male-female cables. These are obviously included in the kit of parts.

6 The capacitors may look different, depending which type you have. They are marked as "µ1"or "104".

1.11 Software Oscilloscope

When you're experimenting with PWM, you'll find that an oscilloscope comes in very handy. It doesn't have to be an expensive piece of equipment though. A simple program that runs on your PC is more than sufficient for the projects in this book. The program doesn't even have to be installed on the PC and can be run from a memory stick.

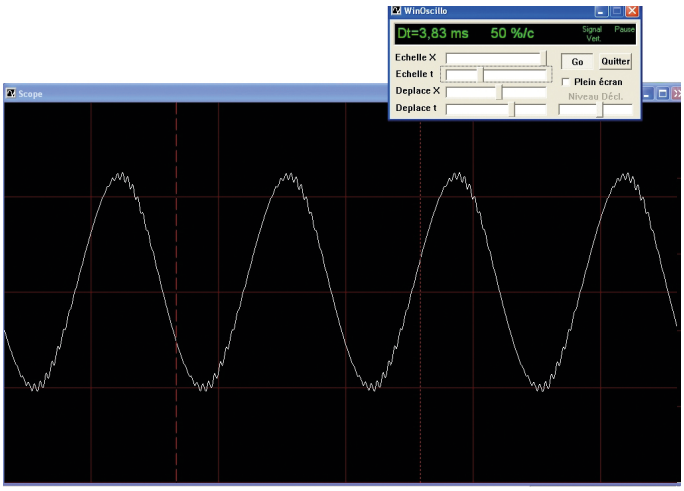


Figure 1.4. Software Oscilloscope WinOscillo.

There are several PC based oscilloscope software available free of charge on the Internet. Links to some popular ones are:

http://www.zen22142.zen.co.uk/Prac/winscope.htm

https://windowsreport.com/oscilloscope-software-pc-laptop/

http://www.winoscillo.com

The one used in this book is the WinOscillo. As shown in Figure 1.4, this is a software based oscilloscope that uses the sound card in your Windows PC to capture the signals, which has its limitations. You'll need to build a circuit similar to the one given in Figure 1.5 to go between the PC and your circuit, which protects the input of the sound card.

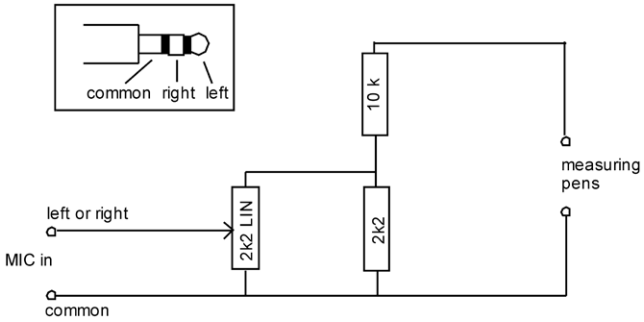


Figure 1.5. Interface for the mic or line input of the PC sound card.

The circuit in the figure above limits the signal level at the microphone or line input of the PC to less than 18% of the original input level. In this book we use 3.3 V, which is therefore reduced to 0.6 V., Make sure that the earth of the microphone plug really is connected to earth. Also, make sure that when you're using this interface, the earth is connected to the earth of your circuit and that you use the other wire for the measurement. To avoid mistakes it's best if you use a black wire for the earth and a red wire for the measurement. All the earths inside your PC (microphone, RS232, USB etc.) are connected together. If you connect the wires the wrong way round you'll create a short circuit, which could cost you a fuse, or worse.

You must obviously turn on the microphone in the Windows Sound Mixer and set it to maximum. Any noise suppression or similar options should be turned off.

WinOscillo can also record the signal to a file and make copies of the display so you can save them. You will come across these regularly in this book. Figure 1.6 shows a saved display from the oscilloscope.

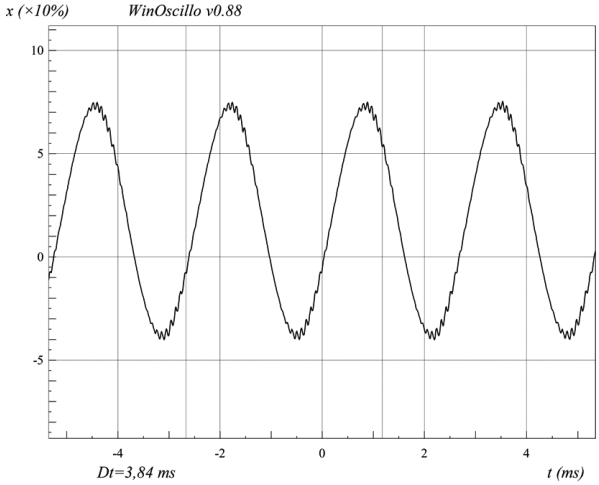


Figure 1.6. A saved display from WinOscillo, showing cursor lines and interval.

Unfortunately, the program can only be operated in French. Table 1.2 shows the translations for the commands used most often. In practice you'll find that you only need a few and those are easy to remember.

Command	Description
F2	Select oscilloscope...
F3	... or spectrum analyzer.
alt-enter	Full screen (the controls stay in the foreground).
Ctrl-S	Save the signal (as a short .wav, or as .csv for Excel).
Ctrl-L	Load a signal (first select pause, otherwise it takes very long!).
Ctrl-M	Save screen capture as .wmf.
Ctrl-V	Vertical cursor line.
Ctrl-H	Horizontal cursor line.
Ctrl-W	Switch between them (if both are on).
Ctrl-G	Tone generator (enter level with decimal comma).

Table 1.3. Using WinOscillo.

The text on the windows is also in French. Table 1.3 shows the translations for the most important terms.

Screen text	Translation
echelle	Scale.
deplace	Pan.
niveau decl.	Trigger level (if turned on).
Dt	Horizontal time.
%/c	Percentage of the maximum signal level per square.

Table 1.4. Overview of screen text.

The two cursor lines can be operated using the mouse. With the left mouse button you can move the dashed line and with the right mouse button you can move the dotted line. They jump to the mouse cursor, so you can first position the mouse cursor at the exact position you want and then click on the mouse button. The window shows which of the lines is currently active (Hor. or Vert.)

Since WinOscillo uses the sound card, the frequency range is fairly limited: from about 20 Hz to 20 kHz. But at this price you can't really complain (the program is free). It's part of the download package.

Chapter 2 • A Quick Tour Of The Raspberry Pi

When you turn on a Raspberry Pi with the SD card that can be ordered along with this book, you go straight to the Raspberry Pi command mode where you are prompted to enter the username and password. The default username and password are **pi** and **raspberry** respectively. The command mode is identified by the following prompt:

```
pi@raspberrypi:~ $
```

By default you are located at the home directory which is `/home/pi`. Your current directory is shown by entering the command `pwd`. There are many commands that can be enterd in the command mode. Some of these commands are summarized in chapter 4. Interested readers can find detailed information on the Internet on all the valid commands.

Some commands that you may find interesting in addition to the ones given in Chapter 4 are given in the next section.

2.1 The Nano Text Editor

Nano is a small, fast editor that can only be controlled via the keyboard. To start this program type **nano** in the terminal, followed by the Enter key.

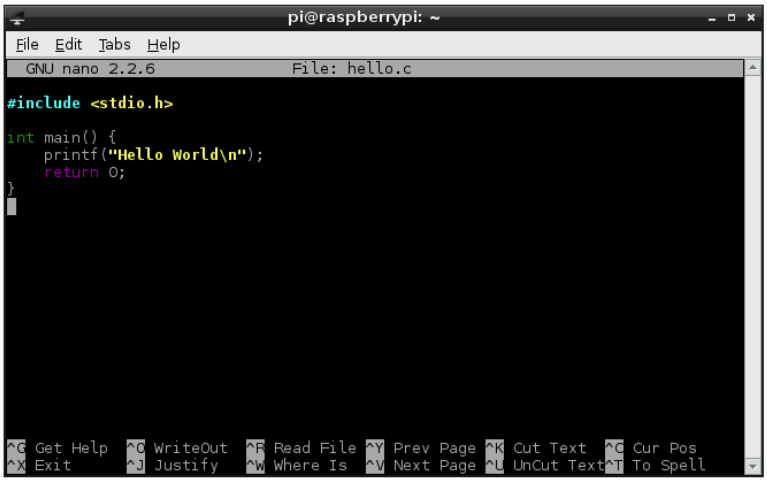


Figure 2.1. Nano, with a C program.

Its functions are accessed via the control key, in conjunction with certain other keys. Ctrl-O, for example, means that you have to hold down the control key, briefly press the O key, and then let go of the control key. The PgUp and PgDn keys (page up and page down) scroll the text up and down by one page when it's too large to fit on one page. You can't scroll with the mouse since the scrollbar on the right has been turned off. Some useful commands of nano are:

Ctrl-O	Save your work (press Enter to save with the given file name). If need be, press Y to overwrite the file.
Ctrl-X	Exit the program.
Ctrl-G	Help screen (use Ctrl-X to close the help screen).
Ctrl-K	Cut a whole line.
Ctrl-U	Paste a whole line.

In this book we use Nano mainly for editing system files, although you can also use it to write programs. It's not very useful for Python since it does not provide automatic indentation, but it is perfectly adequate for C. The traditional "Hello World" could be written in C as follows:

1. Start LXterminal, and type in the following:

```
nano hello.c
```

2. Nano now starts with the (empty) hello.c program.

3. Type in this C program:

```
#include <stdio.h>

int main()
{
    printf("Hello World\n");
    return 0;
}
```

4. Save the program with Ctrl-O followed by Enter, and close Nano with Ctrl-X.

5. Compile the program with gcc :

```
gcc –o hello hello.c
```

6. Nothing appears to happen for a moment, but then the command prompt will return.

7. Type in `./hello` to run the new program. The text "Hello World" will now appear on the screen.

2.2 Some Useful Commands

Now that we have created a text file in our default directory, we can have a look at some useful commands.

- Enter command `ls` to display the files in our default directory.

- Enter command `cat hello.c` to display the contents of file hello.c.
- Enter command `head -3 hello.c` to display the fist 3 lines of file hello.c.
- Enter command `tail -4 hello.c` to display the last 4 lines of file hello.c.

```
pi@raspberrypi:~ $ ls
Desktop    Downloads  MagPi  Pictures  Templates
Documents  hello.c    Music  Public    Videos
pi@raspberrypi:~ $ cat hello.c
#include <stdio.h>

int main()
{
    printf("Hello World\n");
    return 0;
}

pi@raspberrypi:~ $ head -3 hello.c
#include <stdio.h>

int main()
pi@raspberrypi:~ $ tail -4 hello.c
    printf("Hello World\n");
    return 0;
}
```

Figure 2.2 Some useful file commands

- Enter command `mkdir test` to create a directory under our default directory. Check with command `tree` that a new directory has been created under our default directory.
- Change the default directory to test by entering the command `cd test`. Enter command `pwd` to show the new default directory. Enter command `cd ~` to move back to the default login directory.

```
pi@raspberrypi:~ $ mkdir test
pi@raspberrypi:~ $ tree
.
├── Desktop
│   └── New
├── Documents
├── Downloads
├── hello.c
├── MagPi
│   └── MagPi83.pdf
├── Music
├── Pictures
├── Public
├── Templates
├── test
└── Videos

11 directories, 2 files
pi@raspberrypi:~ $ cd test
pi@raspberrypi:~/test $ pwd
/home/pi/test
pi@raspberrypi:~/test $ cd ~
pi@raspberrypi:~ $
```

Figure 2.3 Some useful directory commands

- Enter command `cp hello.c test/hello2.c` to copy file hello.c to directory test and change the filename to hello2.c. Enter command `tree` to show the directory structure.

- Enter command `rm test/hello.2` to delete file hello.2 from directory test. Enter command `cd test` to move to directory test and enter ls to display the contents of this directory.

```
pi@raspberrypi:~ $ cp hello.c test/hello2.c
pi@raspberrypi:~ $ tree
.
├── Desktop
│   └── New
├── Documents
├── Downloads
├── hello.c
├── MagPi
│   └── MagPi83.pdf
├── Music
├── Pictures
├── Public
├── Templates
├── test
│   └── hello2.c
└── Videos

11 directories, 3 files
pi@raspberrypi:~ $ rm test/hello2.c
pi@raspberrypi:~ $ cd test
pi@raspberrypi:~/test $ ls
pi@raspberrypi:~/test $
```

Figure 2.4 Some useful file and directory commands

- Enter command `rmdir test` to delete directory test. Enter command `tree` to show the directory structure.

- Enter command `date` to display the current date and time.

- Enter command `cal` to display the current calendar.

```
pi@raspberrypi:~ $ rmdir test
pi@raspberrypi:~ $ tree
.
├── Desktop
│   └── New
├── Documents
├── Downloads
├── hello.c
├── MagPi
│   └── MagPi83.pdf
├── Music
├── Pictures
├── Public
├── Templates
└── Videos

10 directories, 2 files
pi@raspberrypi:~ $ date
Wed 18 Sep 18:44:22 BST 2019
pi@raspberrypi:~ $ cal
   September 2019
Su Mo Tu We Th Fr Sa
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30
```

Figure 2.5 Some useful commands

2.3 Raspberry Pi Desktop

The desktop provides GUI based graphical interface to the user. If you have a monitor directly connected to your Raspberry Pi then you can move to the Desktop mode by entering command `startx`.

If you are connected to your Raspberry Pi over a WiFi link using your PC, then you can move to the Desktop mode using the **VNC** tool or **Xming** as described fully in chapter 4. This section assumes that you are already in Desktop mode. Figure 2.6 shows the top part of the Raspberry Pi desktop.

Figure 2.6 Top part of the Raspberry Pi desktop

At the top left hand side we have the Applications Menu. Next to it is the Web Browser, File Manager, and Terminal. At the top right hand side we have the VNC Sever icon, Bluetooth icon, WiFi icon, Volume control, and the current time. The Waste Basket is located on the desktop screen.

2.3.1 Application Menu

This menu has the options shown in Figure 2.9. The Programming options includes several programming tools such as Java, Mathematica, Mu, Node-Red, Scratch, Scratch2, Sonic Pi, Thonny, Wolfram and so on. In this book we will be using the Thonny in our Python programs.

Scratch

This program can be used to easily create some simple games. The program has been designed for use by children and as such makes use of a simple graphical programming language. Despite its simplicity, you can create surprisingly remarkable games with it. If you are a child, or if you have one, then this is certainly an interesting program with which to familiarize yourself. As an adult, you could still use this program to create some great animations. Figure 2.7 shows the Scratch screen.

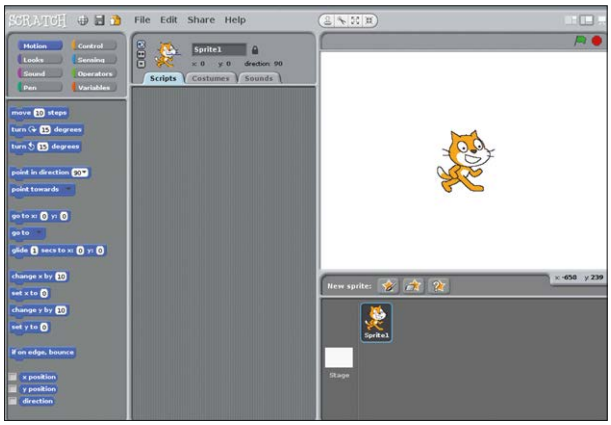


Figure 2.7 Scratch screen

The Education menu includes the SmartSim.

The Office menu includes a number of programs for office use, such as Libre Office Base (database), Libre Office Calc (spreadsheet), Libre Office Draw (drawing), Libre Office Writer (word processing), and so on. As an example clicking on Libre Office Writer starts the word processing package as shown in Figure 2.8.

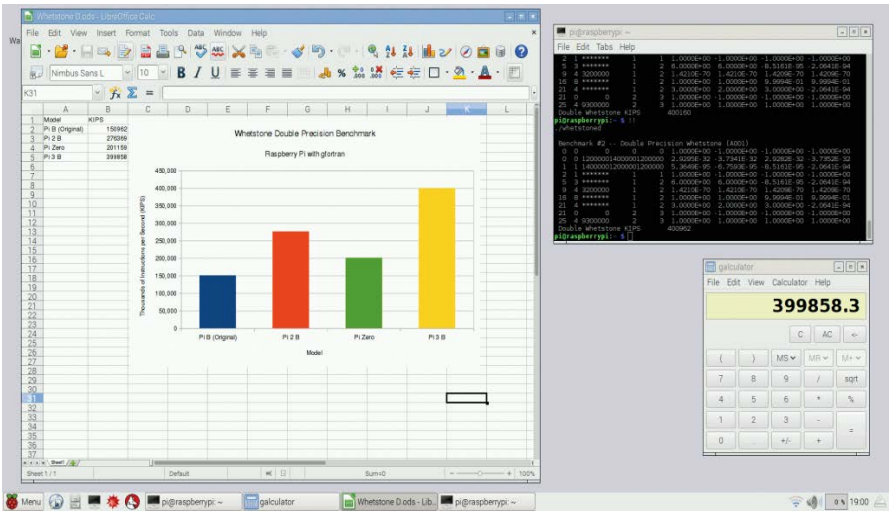


Figure 2.8 Libre Office Calc spreadsheet screen

The Internet menu includes the web browser such as Chromium web browser, Claws Mail and so on.

The Sound & Video menu includes the VLC Media Player.

The Graphics menu includes the Image Viewer.

The Games menu includes the Minecraft Pi and Python games.

The Accessories menu includes useful tools such as Archiver, Calculator, File Manager, PDF Viewer, Task Manager and so on.

The Help menu includes help on various Raspberry Pi based topics

The Preferences menu includes tools such as Add/Remove software, Audio Device Settings, Mouse and Keyboard Settings, Screen Configuration and so on.

The RUN menu runs the entered command.

Finally, the Shutdown menu includes tools such as Shutdown, Reboot, and Logout.

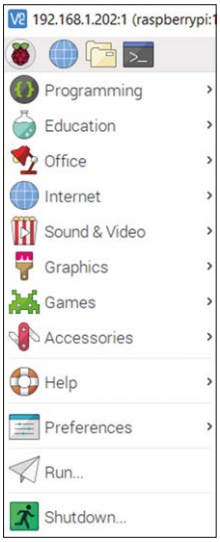


Figure 2.9 Application Menu options

2.3.2 Web Browser Menu

This menu includes the Chromium web browser

2.3.3 File Manager Menu

This menu includes various file management options found in standard Windows applications such as File, Edit, Sort, Go and Tools. You could consider this menu as a simpler version of Windows Explorer. You can browse graphically through the directories, open files and run programs. With the right-hand mouse button you can access the file's properties. From here you can modify the permission settings for files etc.

2.3.4 Terminal Menu

This menu enables commands to be entered to the Raspberry Pi as if we are in command mode. Figure 2.10 shows the screen when this option is selected.

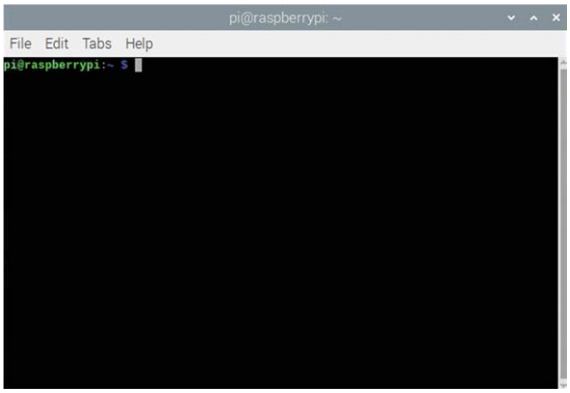


Figure 2.10 Terminal menu option

A useful command that is available while in this option is the `Scrot` which is described below.

Scrot

`Scrot` is a program which is used to capture screen images. If you enter the following command in Terminal menu, you will capture the screen contents and store the result in myscreen.bmp:

```
scrot myscreen.bmp
```

The file will be stored in the directory from which you issued the command. You can also take a screen capture after five seconds:

```
scrot -d 5 myscreen.bmp
```

If you don't want to capture the whole screen, you can use the mouse to select only the part that interests you. First you enter the command, and then you use the mouse to select part of the screen. The screen capture will take place as soon as you let go of the mouse button:

```
scrot myscreen.bmp -s
```

You can close the Terminal menu with the exit command.

2.3.5 Bluetooth Icon

Clicking on this icon enables the Bluetooth settings to be configured. The options are shown in Figure 2.11.

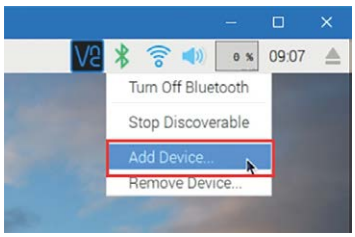


Figure 2.11 Bluetooth options

2.3.6 WiFi Icon

Clicking on this icon enables the WiFi settings to be configured. The options are shown in Figure 2.12.

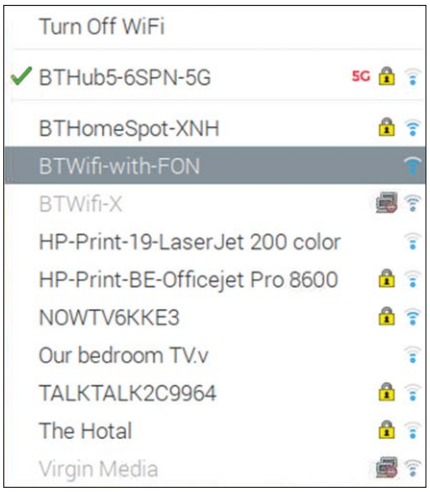


Figure 2.12 WiFi options

2.3.7 Volume Control

This is a slide control that enables the audio volume to changed.

Chapter 3 • Installing The Software

3.1 Raspberry Pi

All of the software for the Raspberry Pi is already installed on the SD card that you can buy for this book, so you don't really have to do anything. However, it's recommended that you make a backup of the SD card onto your PC before you start using it in the Raspberry Pi. Once this has been done you can always go back to the original configuration if you accidentally corrupt the card. You can make the backup with Disk Imager, which is covered in section 3.3.1.

3.1.1 Keyboard

The SD card for this book is configured for use with a keyboard as shown in Figure 3.1. It has an ampersand (@) above the number 2 and a question mark (?) above the forward slash /.

Figure 3.1. "101 keys USA International" keyboard.

This can be easily changed if you're using a different keyboard. Start your Raspberry Pi and then type in the following command :

```
sudo raspi-config
```

You should see the raspi-config menu as shown in Figure 3.2. Select "Localisation Options" and then select "Change Keyboard Layout". Raspberry Pi will select the keyboard from whom you have issued the command as your default keyboard. The keyboard shown above is widely sold throughout Europe, but it has a "101 keys USA International" layout. If you're not sure what type of keyboard you have, you should have a look on Wikipedia (search for "keyboard layout") and compare the layout of your keyboard with those shown.

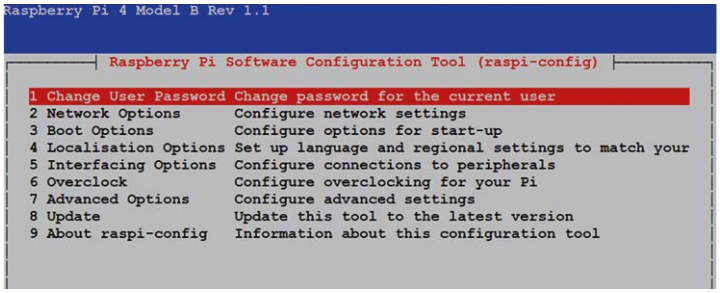


Figure 3.2. Screen-shot of raspi-config.

If you want to verify that the choice you made was the correct one, you should type a few special characters such as @ / ? #. If they come out right it means that Raspberry Pi has selected the correct keyboard layout for you.

3.1.2 Larger SD Card

The same program can be used to increase the size of the file system. This function is important when you've bought a larger SD card, otherwise you wouldn't be able to use all the extra space. The following procedure should be used when you change over to a larger SD card.

1. Make a backup of your old SD card.
2. Put this backup onto the new SD card.
3. Start the Raspberry Pi using the new card.
4. Type in: `sudo raspi-config`
5. Select the option "Expand root partition to fill SD card".
6. Restart the Raspberry Pi.

Note that you can increase the size of the file system but you can't reduce it. This procedure is therefore non-reversible.

3.1.3 Changing The Memory Allocation (Memory Split)

Another important setting in raspi-config is "Change memory split". The Raspberry Pi has two processors onboard. One is for calculations and one is for graphics. You can change the amount of memory reserved for each processor. This is very useful when you know you will be using very intensive (or very few) graphical tasks such as video and games. You have to restart the Raspberry Pi before the changed settings take effect. You can change these proportions as often as you like.

3.1.4 Raspbian Buster Installation Steps On Raspberry Pi 4

Raspbian Buster is the latest operating system of the Raspberry Pi. The SD card for this book is pre-loaded with the Raspbian Buster operating system. This section gives the steps for installing this operating system on a new SD card for the readers who have not purchased the SD card of the book. You will need a micro SD card with a capacity of at least 8GB (16 GB is even better) before installing the operating system on it.
The steps are as follows:

- Download the Buster image to a folder on your PC (e.g. C:\RPIBuster) from the following link by clicking Download ZIP under section **Raspbian Buster with desktop and recommended software** (see Figure 3.3). At the time of writing this book the file was called: **2019-07-10-raspbian-buster-full.img**. You may have to use the Windows 7 Zip software to unzip the download since some of the features are not supported by older unzip software.

  https://www.raspberrypi.org/downloads/raspbian/

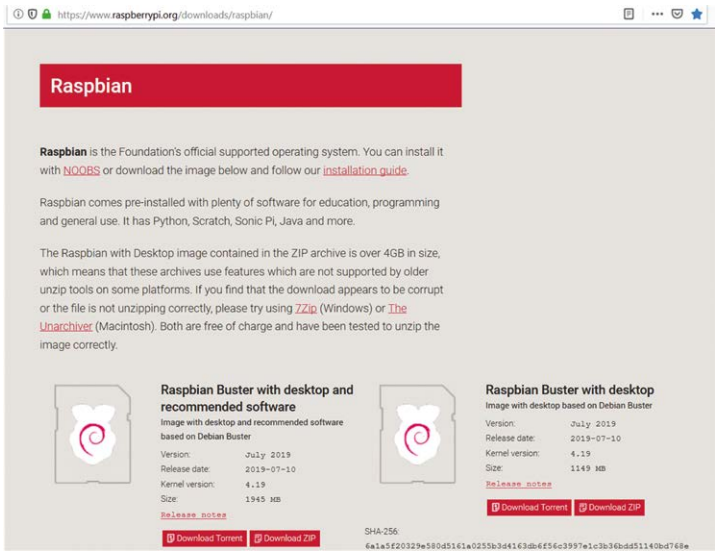


Figure 3.3 Raspbian Buster download page

- Put your micro SD card into the card slot on your computer. You may need to use an adapter to do this

- Download the Etcher program on your PC to flash the disk image. The link is (see Figure 3.4):

  https://www.balena.io/etcher/

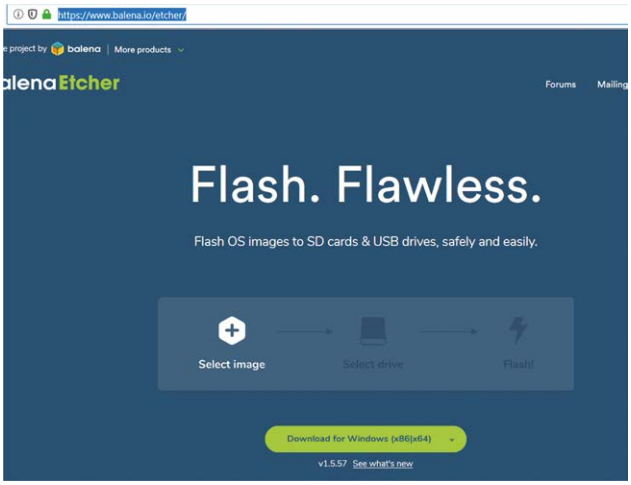


Figure 3.4 Download Etcher

- Double click to Open Etcher, and click Select image. Select the Raspbian Buster file you just downloaded.

- Click **Select target** and select the micro SD card

- Click **Flash** (see Figure 3.5). This may take several minutes, wait until it is finished. The program will then validate and unmount the micro SD card. You can remove your micro SD card after it is unmounted.

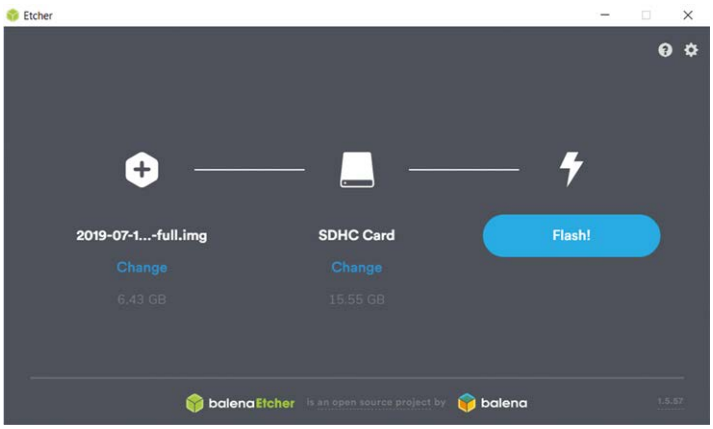


Figure 3.5 Click Flash to flash the disk image

- You are now ready to use your micro SD card on your Raspberry Pi 4.

- Connect your Raspberry Pi to an HDMI monitor (you may need to use an adapter cable for mini HDMI to standard HDMI conversion), connect a USB keyboard, and power up the Raspberry pi.

- You will see the startup menu displayed on the monitor. Click Next to get started.
- Select WiFi network and enter the password of your WiFi router
- Click on the WiFi icon at the top right hand side of the screen and note the Wireless IP address of your Raspberry Pi (notice that this can change next time you power-up your Raspberry Pi).
- You should now be ready to use your Raspberry Pi 4 (see Desktop in Figure 3.6)

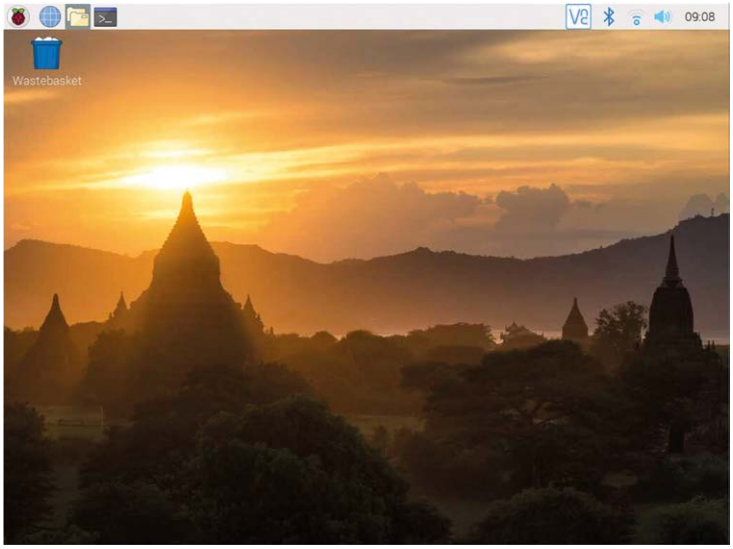


Figure 3.6 Raspberry Pi 4 desktop

Notice that the IP address of your Raspberry Pi can also be seen in your router. You can also get the IP address of your Raspberry Pi 4 using your mobile phone. There are several programs free of charge that can be installed on your mobile phone that will show you the IP addresses of all the devices connected to your router. In this book, we use an example program on an Android mobile phone called the "Who's On My WiFi – Network Scanner" by Magdalm. Running this program will display the Raspberry Pi Wireless IP address under the heading Raspberry Pi Trading Ltd. In addition to the IP address, other parameters such as the MAC address, gateway address, IP mask etc are all displayed by this program.

3.1.5 Remote Access

It is much easier to access the Raspberry Pi remotely over the Internet, for example using a PC rather than connecting a keyboard, mouse, and display to it. Before being able to access the Raspberry Pi remotely, we have to enable the SSH and the VNC by entering the following command at a terminal session:

```
pi$raspberrypi:~ $ sudo raspi-config
```

Go to the configuration menu and select Interface Options. Go down to P2 SSH (see Figure 3.7) and enable SSH. Click <Finish> to exit the menu.

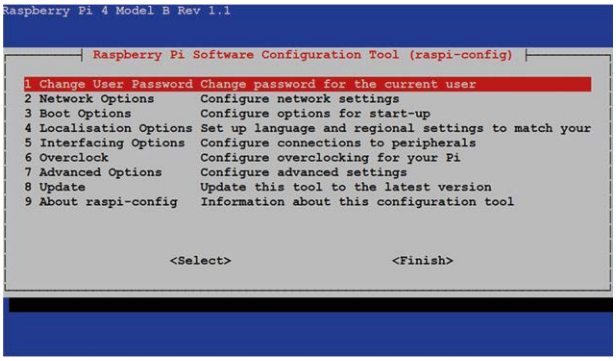


Figure 3.7 Enable SSH

You should also enable VNC so that the Raspberry Pi can be accessed graphically over the Internet. This can be done by entering the following command at a terminal session:

```
pi$raspberrypi:~ $ sudo raspi-config
```

Go to the configuration menu and select Interface Options. Go down to P3 VNC and enable VNC. Click <Finish> to exit the menu.

At this stage you may want shutdown your Raspberry Pi by clicking the Applications Menu on Desktop and selecting the Shutdown option.

3.1.6 Using Putty

Putty is a communications program that is used to create a connection between your PC and the Raspberry Pi. This connection uses a secure protocol called SSH (Secure Shell). Putty doesnt need to be installed as it can just be stored in any folder of your choice and run from there.

Putty can be downloaded from the following web site:

https://www.putty.org/

Simply double click to run it and the Putty startup screen will be displayed. Click SSH and enter the Raspberry Pi IP address, then click Open (see Figure 3.8). The message shown in Figure 3.9 will be displayed the first time you access the Raspberry Pi. Click Yes to accept this security alert.

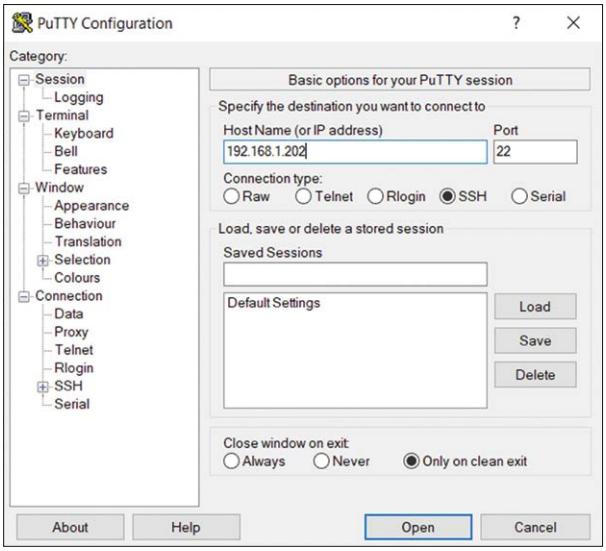


Figure 3.8 Putty startup screen

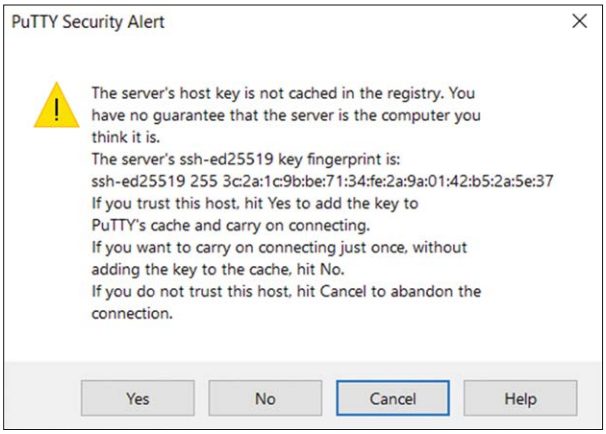


Figure 3.9 Click Yes to accept

You will be prompted to enter the username and password. Notice that the default username and passsword are:

username: pi
password: raspberry

You now have a terminal connection with the Raspberry Pi and you can type in commands, including sudo commands. You can use the cursor keys to scroll up and down through the commands you've previously entered in the same session. You can also run programs although not graphical programs.

3.1.7 Configuring the Putty

By default, the Putty screen background is black with white foreground characters. In this book we used white background with black foreground characters, with the character size set to 12 points bold. The steps to configure the Putty with these settings are given below. Notice that in this example these settings are saved with the name **RPI4** so that they can be recalled whenever the Putty is re-started:

- Restart Putty
- Select SSH and enter the Raspberry Pi IP address
- Click Colours under Window
- Set the Default Foreground and Default Bold Foreground colours to black (Red:0, Green:0, Blue:0)
- Set the Default Background and Default Bold Background to white (Red:255, Green:255, Blue:255)
- Set the Cursor Text and Cursor Colour to black (Red:0, Green:0, Blue:0)
- Select Appearance under Window and click Change in Font settings. Set the font to Bold 12.
- Select Session and give a name to the session (e.g. RPI4) and click Save.
- Click Open to open the Putty session with the saved configuration
- Next time you re-start the Putty, select the saved session and click Load followed by Open to start a session with the saved configuration

3.1.8 Using the Windows 10 SSH Client

We can also use the Windows 10 SSH client instead of Putty. The SSH client is a part of Windows 10, but it's an "optional feature" that isn't installed by default. The steps to install SSH Client on Windows 10 are as follows:

- Go to Settings -> Apps
- Click Manage optional features under Apps & features (see Figure 3.10)

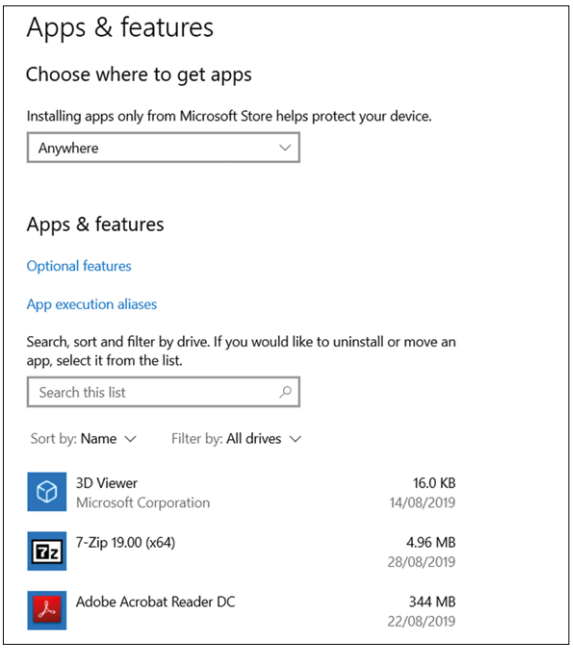


Figure 3.10 Apps & Features

- Click Add a feature at the top of the list of installed features. If you already have SSH Client installed, it will appear in the list.

- Scroll down and click Open SSH Client, and click to install

You can now use the SSH client by running the `ssh` command. This works in either a PowerShell window or a Command Prompt window.

To quickly open a PowerShell window, right-click the Start button or press Windows+X and choose **Windows PowerShell** from the menu (see Figure 3.11).

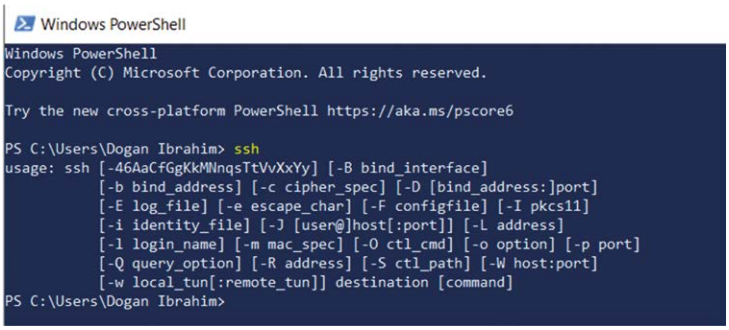


Figure 3.11 SSH menu on Windows 10

Enter the following command to Windows PowerShell to login to your Raspberry Pi (note that the character below is lower case L and not number 1):

```
ssh 192.168.1.202 –l pi
```

You will be prompted for the password. Enter **raspberry**

Figure 3.12 shows a typical session using the Windows 10 SSH Client.

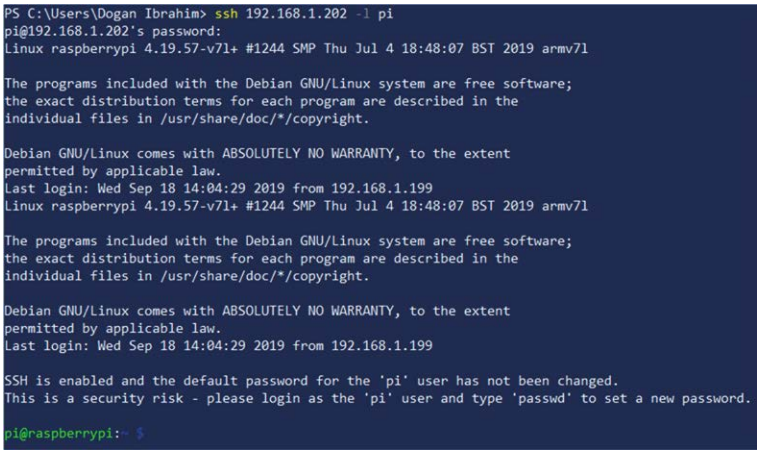


Figure 3.12 Using the Windows 10 SSH Client

3.1.9 Running Graphical Programs – Remote Access of the Desktop

You can control your Raspberry Pi via Putty, and run programs on it from your Windows PC. This won't work with graphical programs because Windows doesn't know how to represent the display. The Raspberry Pi runs under Linux and that uses Xwindows. This may sound very much like Windows, but it works in a totally different way. We can get round this problem using some extra software. Two popular software used for this purpose are: VNC (Virtual Network Connection), and Xming.

Installing and Using VNC

VNC consists of two parts: VNC Server and the VNC Viewer. VNC Server runs on the Raspberry Pi, and the VNC Viewer runs on the PC. VNC server is already installed on your Raspberry Pi. You can start the server by entering the following command in the command mode:

```
vncserver :1
```

The steps to install and use the VNC Viewer onto your PC are given below:

- There are many VNC Viewers available, but the recommended one is the Tight-VNC which can be downloaded from the following web site:

https://www.tightvnc.com/download.php

- Download and install the TightVNC software for your PC. You will have to choose a password during the installation.

- Start the TightVNC Viewer on your PC and enter the Raspberry Pi IP address (see Figure 3.13) followed by :1. Click Connect to connect to your Raspberry Pi.

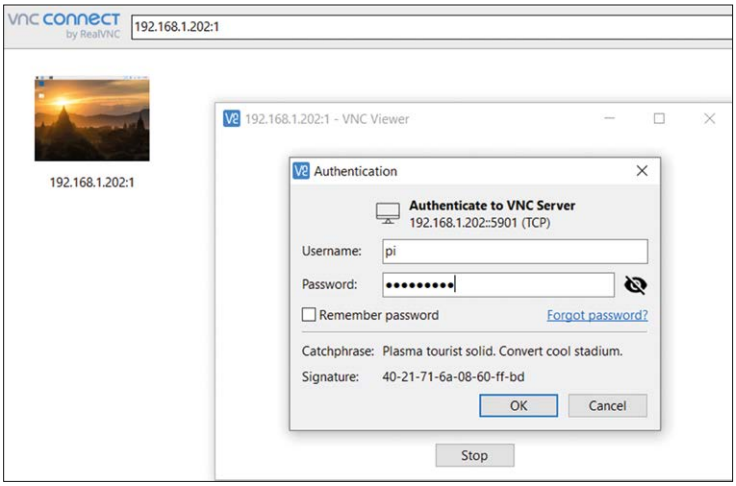


Figure 3.13 Start the TightVNC and enter the IP address

Figure 3.14 shows the Raspberry Pi Desktop displayed on the PC screen.

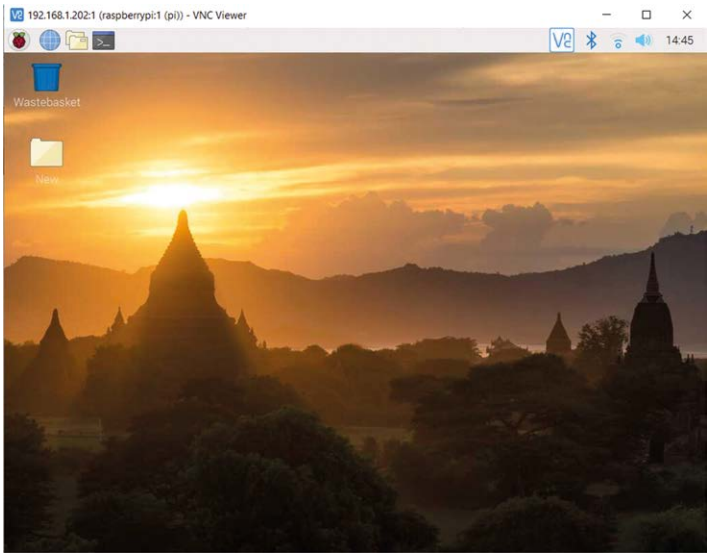


Figure 3.14 Raspberry Pi Desktop on the PC screen

Installing and Using Xming

The Raspberry Pi runs under Linux and that uses Xwindows. This may sound very much like Windows, but it works in a totally different way. We can get round this problem using some extra software: Xming. Xming is available at the following web site (see Figure 3.15):

https://sourceforge.net/projects/xming/

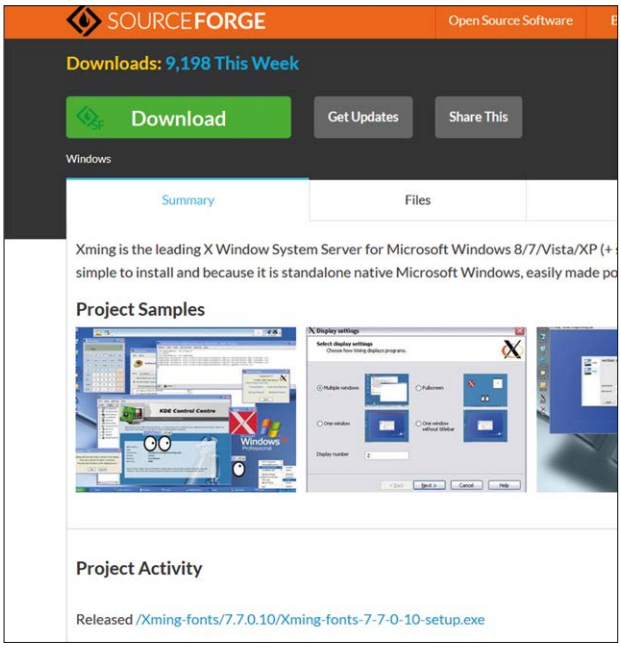


Figure 3.15 Xming web site

- Click the following button to install the fonts:

/Xming-fonts/7.7.0.10/Xming-fonts-7-7-0-10-setup.exe

- Accept the defaults: Bitmap, 75dpi and TrueType fonts.
- Click the **Download** button to install Xming-6-9-0-31-setup. During the installation, accept all the options except the following:
- On the Select Components screen you should select the option Don't install an SSH client since you already have SSH client
- On the **Select Additional Tasks** screen you should select the option **Create a desktop icon** (unless you never use desktop icons)

The Xming icon appears on the PC desktop as shown in Figure 3.16.

Figure 3.16 Xming icon

Start Xming and its icon appears in the taskbar at the bottom-right. If your PC has a firewall, you may get a warning. Select "Permit in Private Networks". Don't forget to start up Xming before you run any graphical applications.

Next start Putty. We now add X11 forwarding to this session (X11 is short for Xwindows). Make the following changes to Putty (see Figure 3.17):

1. Go to Connection at the left hand side and expand SSH
2. Click X11 and tick to Enable X11 Forwarding.
3. Type localhost:0 into X display location.
4. The remote authentication should be set to MIT-Magic-Cookie-1.
5. Go back to Session.
6. Save these settings (this way you don't have to enter them every time)
7. Then open the connection with Putty.

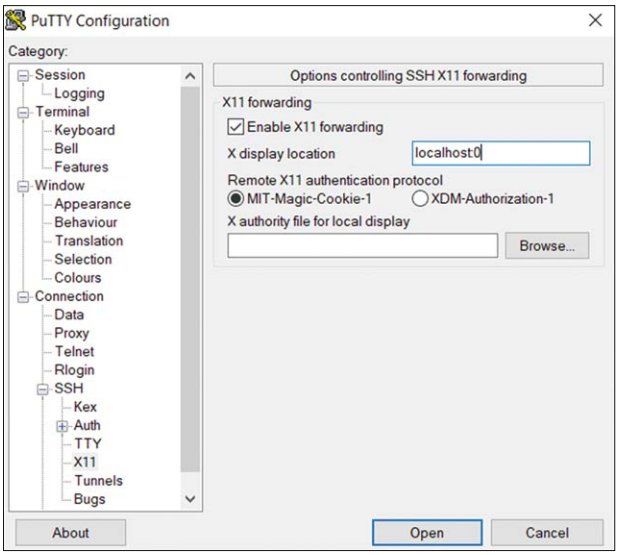


Figure 3.17 Settings in Putty for X11 communication

Start Xming, and log in using Putty. Now type in the command to start the graphical shell for the Raspberry Pi:

```
startlxde
```

You'll see the following messages:

```
pi@raspberry:~ $ startlxde-pi
** Message: main.vala:99: Session is LXDE-pi
** Message: main.vala:100: DE is LXDE
** Message: main.vala:131: log directory: /home/pi/.cache/lxsession/LXDE-pi
** Message: main.vala:132: log path: /home/pi/.cache/lxsession/LXDE-pi/run.
log
```

This is normal and you should just wait a bit longer. After a while, the normal Raspberry Pi environment starts up, but on your Windows PC (see Figure 3.18). You can now operate the Raspberry Pi as if you were sitting in front of it. The screen may update a bit slower than usual, depending on the speed of the network connection between the Raspberry Pi and router, and between the router and your PC.

Figure 3.18 Xwindows on a Windows PC

Click the pcmanfm button (on the Windows Taskbar) to dispose of the background. The Xwindows taskbar is on top of the screen and occupies the full width. This can be changed by right-clicking on the bar and then moving the mouse away from the bar while still keeping the right mouse button pressed down. The smaller menu now remains and via Panel Settings it is downsized or moved.

You can start Raspberry Pi programs via the Raspberry Pi taskbar and start the Windows PC programs via the Windows taskbar, Start Menu or desktop icons. In the screenshot in Figure 3.19, of my PC, you can see a Python program in Idle on the Raspberry Pi and next to it, a Piccolino program in JALedit on the Windows PC. This is a very neat way in which to work with Windows and Linux programs at the same time!

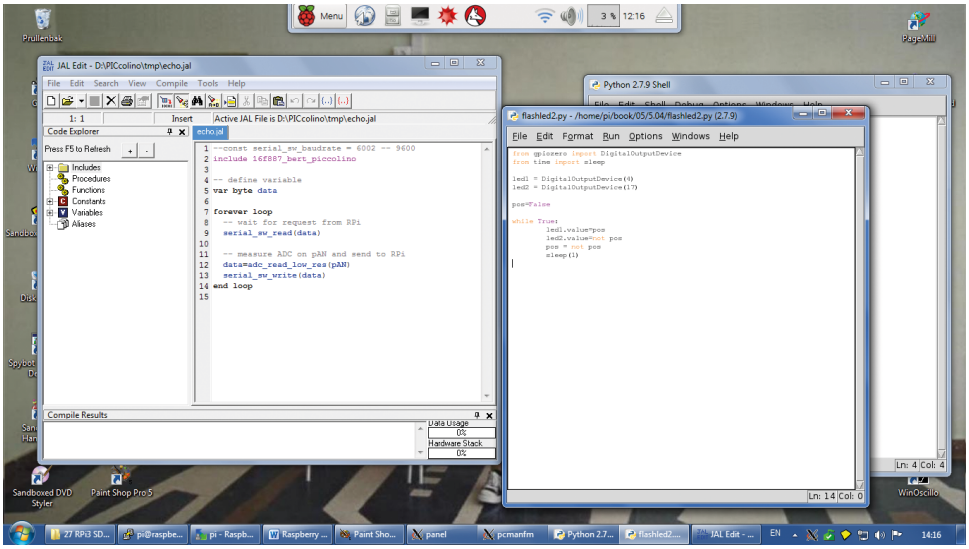


Figure 3.19. Xwindows and Windows programs running simultaneously

When you have finished, you should double-click on the Xming icon at the bottom right to close the server. If Xming reports that there are still some windows open it's best to close these first.

Possible error messages:

- `No error message, but nothing happens`

You tried to start lxde without starting Xming. First start Xming and then lxde.

- `Unable to access the X Display, is $DISPLAY set properly?`

You have forgotten to start Xming. Do that now and try it again.

- `Only one xsession can be established at a time`

You haven't turned on X11 forwarding in Putty. Go back to the text following Figure 3.16, and remember to save the settings after you've made the modifications.

- `Obt-Message: Xinerama extension is not present on the server`
  `Openbox-Message: A window manager is already running on screen 0`

These aren't error messages; the text is normal and you should wait for the command to complete.

3.1.10 WinSCP

WinSCP is a program that's used to copy files from your PC to the Raspberry Pi, and vice versa. You can also use it to view files, change permissions, and browse the folders graphically. The program doesn't have to be installed. It can just be copied to a folder of your choice and used from there. There are two programs with the same name in the folder, but with different extensions. We use the program with the .exe extension, which is winscp.exe.

1. Start the program.
2. Go to "preferences" and select Commander.
3. Go back to "sessions" and type in the IP address of your Raspberry Pi in the 'Hostname" box.
4. Verify that port 22 has been selected.
5. You can (optionally) enter your user id (pi) and password (raspberry).

When you use an IP address for the first time you may get a message stating that you haven't used this IP address before and whether it is correct. Click on "OK", and after a while you'll see the screen in Figure 3.20.

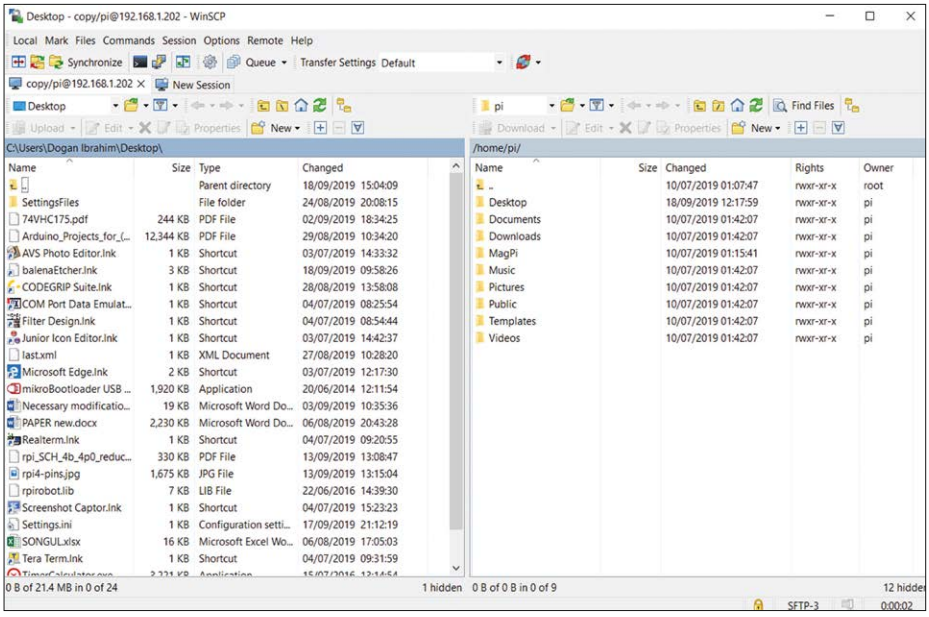


Figure 3.20 WinSCP, Windows PC on the left, Raspberry Pi on the right

If your screen shows only two windows with just the files in them, but no overview of the directories, you can change this using the option "Hide/show directory tree". The button for this is circled in Figure 3.21. Note that each window has its own button.

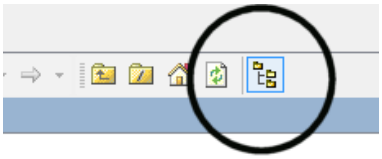


Figure 3.21. Hide/show directory tree button.

On the left you can see the directories of your Windows PC with the files underneath. On the right are the directories of the Raspberry Pi with the files underneath. You will obviously have different directories and files, but the principle is the same. Below are just some of the useful functions offered by WinSCP:

1. Copy files by dragging them with the mouse to another directory or to another machine.

2. Edit files by clicking on them with the mouse and then clicking F4. You'll be asked for your password (raspberry) the first time you want to save a file in a session, but not afterwards.

3. Create new files by clicking in the directory where you want to create the file, then from the menu click on "Files", "New" and "File".

4. Create a new directory by clicking in the directory where you want to create the directory, then from the menu click on "Files", "New" and "Directory".

5. Change the permissions for files on the Raspberry Pi by right-clicking on the file and selecting properties. A window like the one below will pop up from which you can change the permissions using the tick boxes.

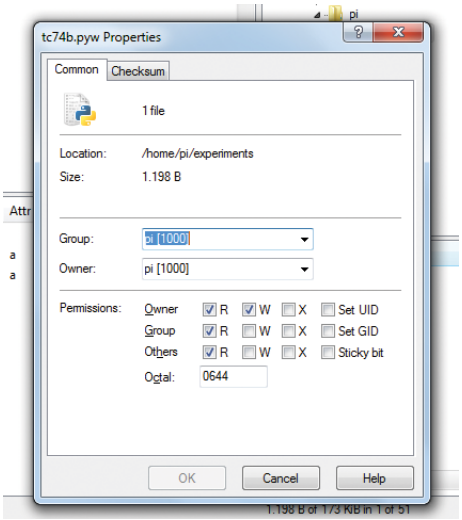


Figure 3.22. Setting file permissions.

3.1.11 Windows 10 SCP Client

We can also use the Windows 10 built-in scp (Secure Copy Protocol) client to copy files between different computers. It uses Secure Shell (SSH) to keep the files protected while in transit. Since scp uses SSH, the transferred data will be encrypted. This keeps your information from being compromised during transit

scp is a command line utility, meaning you will have to use the Command Prompt in Windows. As such, it is not very easy to use the scp utility. scp is operated by writing single commands into the command line, making it a good option for repeatedly updated files. Interested readers can get further information on scp from the Internet.

3.2 Static IP Address

In order to try out the web server and client-server projects in this book you need at least a Raspberry Pi and a PC, connected to an Internet router. Figure 3.23 shows a typical network configuration with the Raspberry Pi and PC connected to a router.

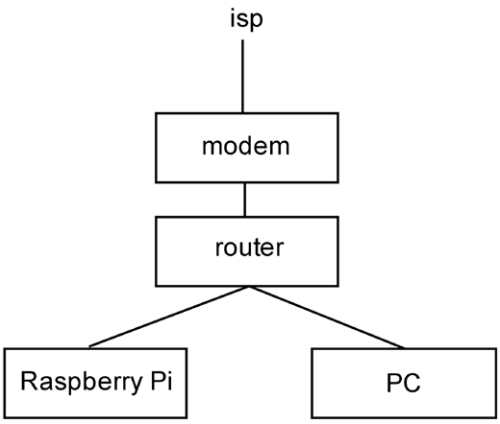
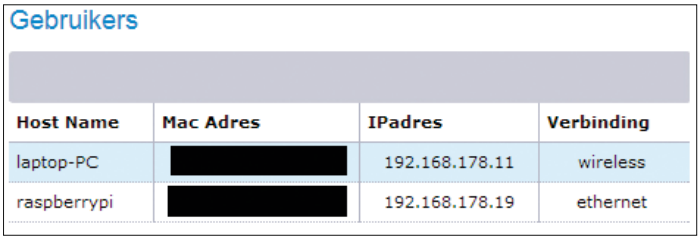


Figure 3.23. Network configuration.

The modem and router can also be combined into one device. If you can connect several computers to your device then you have a router, or a modem with a built-in router. If you can connect only one computer then you have a modem. In this case you have to buy a separate router and connect it between the modem and your PC. The instruction manual should tell you how this should be done, although it's often a matter of plugging in the network cables and turning the router on.

Before you can create a connection between your PC and the Raspberry Pi you need to find out what IP address the Raspberry Pi uses. An IP address is a unique number required for every device connected to the Internet or an intranet. For example, your modem will have been given an IP address by your ISP (ISP is short for Internet Service Provider, the company that provides your Internet connection). Local IP addresses are used on your own network, which are allocated by your router. These local IP addresses are valid only on your own network and they're not visible on the wider Internet. Everybody who has the same make of router with the same settings will therefore use the same local IP addresses.

To find out which IP address your Raspberry Pi uses, you have to log in to your router. If you don't know how to do this, you should consult its instruction manual or contact your ISP. Once you are logged in, you can see an overview of all devices that are currently connected to your router.


Gebruikers

Host Name	Mac Adres	IPadres	Verbinding
laptop-PC	[redacted]	192.168.178.11	wireless
raspberrypi	[redacted]	192.168.178.19	ethernet

Figure 3.24. IP addresses in the router.

In the above example in Figure 3.24, for my router, you can see that a laptop and Raspberry Pi are connected to the router, where the laptop is connected wirelessly (via WiFi) and the Pi is connected via the Ethernet (a cable). Make a note of the IP address of your Raspberry Pi, since you'll need this later on.

Note that the IP address of the Raspberry Pi can change if you turn it off and then turn it on again later. This is because the router picks the address itself. There are two methods to ensure that you always have the same IP address:

1. Static lease. Your router remembers that the Raspberry Pi is always allocated the same IP address, and which address that is.

2. Static IP. Your Raspberry Pi tells your router which IP address it is going to use.

The first method is preferable because you don't have to make any changes to the Raspberry Pi. This means that your Raspberry Pi will also work elsewhere, for example when you visit friends to show them what you've done with the Raspberry Pi. This method works for both wired and wireless connections.

A static lease means that your router remembers that your Raspberry Pi is always allocated the same IP address, and what that address should be. To make this happen, the router has to be able to recognize your Raspberry Pi, which it does using its MAC address. This is a series of 6 two-digit hexadecimal numbers, which are unique. Every device that can be connected to the Internet has its own MAC address. In this context, by 'device' we really mean the adapter, which is the hardware that makes the actual connection to the Internet. When your Raspberry Pi has both a wired and a wireless connection to your router, the router will see two MAC addresses. One from the Internet adapter for the wired connection (built-in on the Raspberry Pi) and one from the WiFi Internet adapter (in the USB port). In this case you can allocate a fixed IP address to both MAC addresses.

The first step is to log in to your router to find out what the MAC address of the Raspberry Pi is. In my router this could be found in the DHCP Clients List. Make a note of the address and then go to the section in your router where you can set up a static lease. In my router this was in Advanced Settings - Static Lease.[7] In here you should fill in the MAC address, and choose an IP address. In the example below you can see that two addresses have been allocated, one for the wired connection with the Raspberry Pi and one for the WiFi connection.

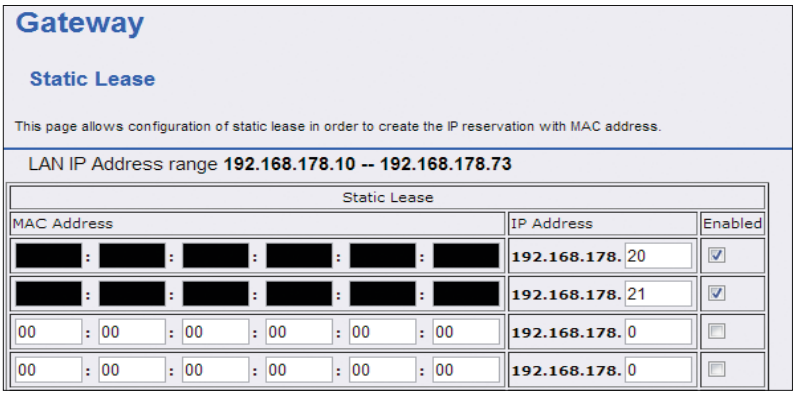


Figure 3.25. Specifying the static lease IP address in the router.

You don't need to make any changes to the Raspberry Pi. Your Raspberry Pi will have its correct IP addresses allocated automatically when it is connected to your router. When you're visiting friends, your Raspberry Pi will be given an IP address by their router.

In general, you won't have a reliable network connection if you try to use both types of connection at the same time. Before you turn on your Raspberry Pi, you must have either the network cable or the WiFi adapter plugged in, but never both.

3.3 Windows PC (Optional)

When you have access to a Windows PC, you can try out a number of interesting projects where the Raspberry Pi acts as a web server or as a server in client-server projects. Client-server means that part of the program runs on the Raspberry Pi, and another part runs on the Windows PC. It is possible to control the Raspberry Pi via your PC, without the need to connect a display, keyboard and mouse to the Raspberry Pi. This is also known as headless control. [8]

3.3.1 Disk Imager

This program is used to make a backup of the SD card for the Raspberry Pi, and to restore the backup when necessary. The best time to make a backup is before you use the SD card for the first time, which means you can always revert to the original. In order to use this program you obviously need an SD card reader in your computer, or buy a USB card reader.

7 Some routers don't use the official term "static lease", whereas the functionality is there. On the ZyXel router for example, you have to go through the menu via Network, LAN, Client List to get an overview of current connections. You can assign a static lease on this screen by ticking the "Reserve" box.

8 You can also use an Apple or Linux PC instead of a Windows PC, but then you'll have to find the necessary software on the Internet yourself.

These aren't expensive and it is very useful to be able to make a backup from time to time.

Pro tip:
Backup the SD card **before** you use it!

There is no need to install the program. When you run the program, you'll probably get a message from Windows asking you if the program may make changes to your computer, to which the answer is "yes". There is a possibility that an error message pops up in a new window, with some text that ends in "Error1: incorrect function". This error message can be ignored, so you can just click on OK. The program still functions correctly despite the error message. Choose a suitable name for your backup (which hasn't been used yet), for example "new out of the box", and check that the drive letter for the device corresponds to that for your SD card. Next, click on "Read". The program doesn't check if anything is stored on the card; it just copies the whole card to the file.[9]

You can restore the backup to the card using the same program. This time you have to select the backup file on the hard drive by typing in its name, or by browsing for it by clicking on the icon with a blue folder on it. Check that the drive letter for the Device is that of the SD card. This is very important since if you accidentally selected your hard drive this would be overwritten and made unusable. Once you have verified that the drive letter is correct you can click on "Write" to restore the backup to the card.

3.3.2 Python, wxPython and IdleX

In order to try out the web server and client-server projects in this book, or to connect to the Raspberry Pi remotely, you need at least a Raspberry Pi and a PC. They should both be connected to an Internet router.

Python is the programming language that comes installed on the Raspberry Pi, and which is used in this book to program the Raspberry Pi. You can also run these programs on your PC, which is something we'll do in Chapter 12, which covers client-server applications. For this reason you have to install Python on your Windows PC as well. The second installation is wxPython. This is an add-on for making graphical programs. For the Raspberry Pi this is already on the SD card for this book, but for the PC you will obviously need to install it. Python has two popular versions: Version 2 and Version 3. Both versions are already installed on the micro SD card. Python 2 is started with the command python, and Python 3 with the command python 3. For Python on the PC, you should navigate to this website:

www.python.org/download/

From this page you should choose the installation program that is appropriate for your operating system. Note that you should download the new Python version 3 and not version 2.7.

9 That can have surprising side effects. Say that you put a 4 Gb image onto an 8 Gb card, and then make a backup of the new card. Your backup has now become 8 Gb, of which 4 Gb is empty, but Disk Imager doesn't know this.

For wxPython you should navigate to this website:

www.wxpython.org/download.php#stable

From this page you should choose the installation program that is appropriate for your operating system. Make sure that you download the wxPython version for Python 3.

You should first install Python and then wxPython. Accept all default settings.

The standard development environment for Python is Thonny as we will see in detail in the next chapter.

Chapter 4 • Short Introductions To...

Most Raspberry Pi users won't have come across Linux, Bash and Python before. In this chapter we'll give a short introduction to each of them, which makes it easier for you to understand the projects. If you don't (yet) want to program, but just want to build some of the projects you can skip the sections on Bash, Python and JavaScript for the time being.

4.1 Debian Linux

The operating system for the Raspberry Pi is Debian Linux. Linux is an operating system that's been created and maintained by volunteers. There are many variations of Linux, one of which is Debian. And Debian in turn has several different implementations, of which the Buster version is used on the Raspberry Pi 4. The version of the operating system running on the Raspberry Pi can be shown by entering the command: `cat /etc/os-release`. The micro SD card used in this book is loaded with the operating system shown in Figure 4.1.

```
pi@raspberrypi:~ $ cat /etc/os-release
PRETTY_NAME="Raspbian GNU/Linux 10 (buster)"
NAME="Raspbian GNU/Linux"
VERSION_ID="10"
VERSION="10 (buster)"
VERSION_CODENAME=buster
ID=raspbian
ID_LIKE=debian
HOME_URL="http://www.raspbian.org/"
SUPPORT_URL="http://www.raspbian.org/RaspbianForums"
BUG_REPORT_URL="http://www.raspbian.org/RaspbianBugs"
```

Figure 4.1 Version of the operating system running

The principles behind the development of Linux and Windows are quite different. This doesn't mean that one system is better than the other. It's just that there are clear differences between them, which you'll discover when you start working with the Raspberry Pi. The most important differences are listed in the table below:

Windows	Linux
In principle there is only one user, which is you. It isn't called a Personal Computer (PC) for nothing. Although it is possible to create other user accounts, this facility was added at a later stage. One of the reasons was to stop children from changing various settings when they used their parents' PC.	There are several users by default. Every file, every directory, everything stored on the Raspberry Pi belongs to somebody. And only this 'somebody' can use them, unless the owner has given permission. This 'somebody' doesn't necessarily have to be a person; machines and programs also have their own rights. This is a fundamental aspect of Linux.
Since there was only one user, who could do anything, Windows was very vulnerable to viruses, which also had full access rights. To prevent this, an Administrator account was created, which was the only one that could install software and modify system files. This required the user to input the Administrator password, if it was set.	There is one user who has rights to do anything, which is the root. A normal user can use the sudo command to temporarily get the same rights as the root. This command can only be used if the root user has previously given permission for this. Since most users don't have many permissions the sudo command is needed in some circumstances when you're working with the Raspberry Pi.
The operation of Windows is based on a graphical user interface. The PC is operated via graphical programs in combination with a mouse or touch screen. The reason behind this is to keep the operation as simple as possible so that everybody can use it, even those who aren't computer savvy or have much interest in them.	The operation of Linux is based on the terminal, where commands are typed in. The commands have been kept short to keep the amount of typing to a minimum, with the result that they can be very confusing and even intimidating to newcomers. A number of Windows-like Linux variants have been made to make it more attractive to less technical users, such as LXDE which runs on the Raspberry Pi. These variants are called Xwindows.
Windows is used on more than 80% of all PCs. The result of this is that you can find a vast amount of software, both free and commercial, for Windows. Whatever you can think of, there will be a program for it. It is also easy to exchange knowledge and information with other users.	Linux has a market share on the PC of less than 2%. The result of this is that there is less software available for Linux, and in particular, very little commercial software because the market share is so small. Since it is mostly used by enthusiasts, the level of support and the quality of information on the Internet is often better than usual.
Windows requires a powerful processor, a lot of memory and a large hard drive. The pretty user interface requires a lot of processing power from the PC. One consequence is that a Windows PC will never be cheap, since all this power has to be paid for.	Linux runs on virtually any hardware, especially if an Xwindows variant isn't running. The Raspberry Pi is a good example, since it only has the power of a microcontroller but it still works very well. For more intensive applications, the Raspberry Pi is less suitable, although a faster SD card can help a lot.
Executable files (programs) are defined by their extension.	Whether a file is an executable or not is defined by its (modifiable) properties.

Table 4.1. Some noticeable differences between Windows and Linux.

Terminal
In contrast to Windows, where most operations are carried out graphically, in Linux you have to type in most of the commands. These commands are typed in using a terminal.[10] You can use the cursor keys to scroll "up" and "down" through the commands that you've previously typed in during this session. For this we use LXterminal, which can be started via the desktop or the menu. It's definitely NOT the intention that you use the Root Terminal.

Root user
Most Linux systems have a root user and one or more normal users. The root user can do absolutely anything, which can be very dangerous. One small typing error and half the system could be deleted. For this reason it's very uncommon for somebody to log in as a root user. On the Raspberry Pi there isn't even a root user set up. You can of course create one yourself, but it's best to follow the advice of the designers and refrain from doing so. An alternative would be the Root Terminal, which gives you full access rights, but again you shouldn't use this.

A normal user can temporarily get root privileges by using the sudo command. This is a combination of su, the command to run a program as another user, and do. This can of course only be used if the relevant user has been given permission to use the sudo command. On the Raspberry Pi the user pi (which is you) does have the right to use this command. Depending on the circumstances, it may be that Linux will ask you for your password. This should be your own password rather than the one for the root user (the latter doesn't even exist because there isn't a root user). During normal use (when you're working directly with the Raspberry Pi) or via SSH on a PC, you will be asked for your username and password.

Permissions
The permissions for files and directories are assigned in three distinct user types: [11]

User — This is a bit confusing since everybody is a user. In this context, it is meant the user who is the owner of this file. This would normally be the user who created the file. When you create and save a file you are its owner. You can do anything you like with this file, unless you change its permissions yourself. [12]

Group — You can assign permissions to certain groups of users, who then all have the same rights.

Other — This is everybody who isn't the owner or who isn't in a group. Effectively it's the rest of the world, who has access to your Raspberry Pi in one way or another. When you have a file server that is connected to the Internet you should take "rest of the world" literally.

10 A Linux terminal is therefore very different from a Windows terminal. A terminal in Linux is where you input commands. In Windows a terminal is a program that lets you communicate with other computers.
11 We'll use the word 'file' from now on in this section. A file and a directory are the same for Linux, so when we use 'file' it includes 'directory'.
12 Strictly speaking, this isn't true, since the 'execute' permission isn't set by default.

The permissions themselves are subdivided into three sections:

Read	You may inspect the contents of the file.
Write	You may modify the contents of the file. Where it concerns a directory, you may create and delete sub-directories and files within it.
Execute	You can run the file as a program.

You can use the command ls -l to display the status of files. This returns an overview of all files and directories in the current directory, such as this example:

```
drwxr-xr-x 2 pi pi 4096 Oct 28 23:54 Desktop
-rw-r--r-- 1 pi pi    5 Jan 11 13:26 myprogram.py -
```

The first symbol indicates if it is a directory (d) or file (-). This is followed by nine symbols that show the permissions in sets of three symbols for the owner, the group and others. The three symbols are for read (r), write (w) and execute (x). When a letter is shown in any location then the permission for that location is active, if it shows a dash then it isn't active.

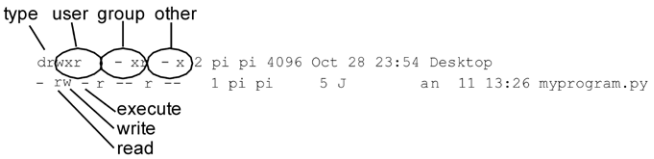


Figure 4.2. Grouping of permissions.

The first line is therefore a directory. The owner has full permissions (rxw), the group can read and execute (r-x), and others may also read and execute (r-x). The second line is for a file. The owner may read and write (rw-), the group and others may only read (r--).

The `chmod` command is used to change file permissions. Along with this command you should state whom it applies to: u for user, g for group, or o for other. Then whether you want to add a permission (+) or remove it (-), and then which type of permission it is for (r, w or x) and finally the file it applies to.

Say that you want to give yourself execute rights to the file myprogram.py in the previous figure. You can do this with the following instruction:

```
chmod u+x myprogram.py
```

When you check the result with `ls -l` the output will be:

```
drwxr-xr-x 2 pi pi 4096 Oct 28 23:54 Desktop
-rwxr--r-- 1 pi pi    5 Jan 11 13:26 myprogram.py
```

You can change the permissions of the group and others in the same way.

Tip: When a file doesn't do what you expect it to, it is often a problem with permissions.

On the Internet you may come across commands such as `cmod 715 myprogram.py`. This looks like a totally different command but it is in fact the same. The number code is used to update the permissions for all three user types at the same time. It works as follows: each of the three digits corresponds to one of the groups. The first digit (in this case 7) is for the user, the second digit (the 1) is for the group, and the third digit (the 5) is for the others. Each digit is arrived at as follows:

```
4 = r
2 = w
1 = x
0 = -
```

The number 7 can only be made by 4+2+1, so the permissions are r, w and x. The number 1 is 0+0+1, so the permissions are --x. The number 5 is 4+0+1, so the permissions are r-x. When we use `cmod 715 myprogram.py` the result will be:

```
drwxr-xr-x 2 pi pi 4096 Oct 28 23:54 Desktop
-rwx--xr-x 1 pi pi    5 Jan 11 13:26 myprogram.py
```

You can see that this is a lot quicker than using a `chmod` command for each of the user types in turn. For this reason this method is most commonly used.

If you're used to Windows you'll find it easier to use File Manager (the Linux equivalent of Windows Explorer) to modify the permissions for a file. You can find this program via the menu, Accessories and then File Manager. Right-click on the file and select Properties. From there click on the Permissions tab. You can now make the required changes and save them by clicking on OK.

Another way to give somebody permissions is to make that user a member of a group that has certain privileges. Say that there is a group called i2c, which has permissions to work with I^2C. You can use the command adduser to add yourself to this group and therefore gain its privileges. The command is followed by your own user name (pi) and the group you want to become a member of (i2c). There is a slight catch-22 here though: since you're not a member of this group you don't have the privileges to add yourself as a member. You therefore have to issue the command as a root user with the help of the sudo command:

```
sudo adduser pi i2c
```

Way of working

For Windows users these permissions come across as very clumsy. The temptation is to work as root and give everybody permission to do anything they want. When you're setting up an Internet server with SPI this would make everything a lot simpler. However, you're better off learning how to use permissions properly. Once you've got used to it you'll find it's not too difficult in practice, and it makes for a more secure environment.

Linux commands

In this section we'll cover the Linux commands that are used in this book. This is therefore not a full overview: there are many more commands and those commands that are discussed here have many more options.

Command	Result
ls	Display the contents of a directory.
cat x	Display the contents of file x on the screen.
mkdir x	Create directory x.
cd x	Go to directory x.
cd x/y	Go to directory y in directory x.
cd ..	Go up one directory (note the space between cd and ..).
cd ~/x	Go to directory x in the home directory. The home directory is the directory where you start when you use LXterminal (or Putty), and is represented by the tilde (~). The prompt is then as follows: `pi@raspberrypi ~ $` The working directory is your user directory. It appears as if it's empty, but that's not the case. The user directory is called /home/pi, which contains other directories although these can only be accessed if you're a root user. You can see the path of your user directory using the pwd command.
rmdir x	Delete directory x, but only if it is empty.
rm x	Delete file x, without asking for confirmation. Extra flags: f = also delete write-protected files i = ask for confirmation For example: rm -i x Delete file x, but ask for confirmation first (answer y or n).
rm -r x	Delete file x, and if x is a directory then delete directory x and any files and directories within it, without asking for confirmation. [13]
cp -i x y	Copy file x to y. The i flag ensures that you get a warning when file y already exists. If y is not a file, but a directory, then file x will be copied to directory y.
mv -i x y	Ditto, but this time file x isn't copied, but moved. You can also use this command to rename a file (or directory). When file y doesn't exist then the effect of mv is to change the name of the file from x into y.
df -h	Shows how much space is used on the drives. In our case, these will be virtual drives that are stored on the SD card, since the Raspberry Pi doesn't have any real drives.
du -sh x	Show how much space is used by directory x.
man x	Ask for a user manual for command x. If it exists you'll get a text file that you can scroll through using the cursor keys. Pressing h gets you more help and q exits.

Table 4.2. Some of the Linux commands used in this book.

13 You may well come across the term "rm -rf /" in stories about disasters in Linux. This command deletes the whole directory, and if you're the root user you can even delete everything on your Raspberry Pi! You may now understand why it's unwise to log on as a root user.

Installing applications
This is done in a different way than under Windows. First of all, the repository in the Raspberry Pi has to be updated so that the Raspberry Pi knows which software is found where.

```
sudo apt-get update
```

Next, we have to search for the software we want to install. As an example we'll search for the software "aprogram":

```
sudo apt-cache search aprogram
```

When the software has been found, it can be downloaded and installed:

```
sudo apt-get install aprogram
```

When you no longer need the software, you can also remove it:

```
sudo apt-get remove aprogram
```

It may be that the software you're looking for has to be downloaded and installed in a different way. In that case you should follow the instructions from the supplier.

It's recommended that you make a backup of the SD card before installing new software. You can use the program Disk Imager for this, as described in section 3.3.1. It can happen that the new software introduces some problems. If you have a backup then you don't have to rely on the "remove" function to remove the program without any further problems.

4.2 Programming in Bash
Bash is the Linux version of Batch files, which you'll probably have come across in Windows. It is a simple language with which you can automate many common tasks on the Raspberry Pi and which can be used to output text to the screen.

In this chapter we look at only those Bash commands that you'll need in this book. Bash itself is a lot more extensive and powerful than these examples would suggest!

As an example we'll create a simple Bash program that displays "Hello World" on the screen. The program is as follows:

```
echo Hello World
```

It could hardly be any simpler. `echo` means "display on the screen", and everything following the command is shown on the screen. We use an editor called Text Editor to create the program on the Raspberry Pi. This is a simple graphical text editor, similar to those in Windows. You can run this program from the menu (Accessories – Text Editor).

Figure 4.3. Text Editor

Many Linux users aren't very keen on graphical applications and they prefer to use Nano. You can run this program from the terminal by typing nano, followed by Enter. Nano is a fast, small editor that can only be operated via the keyboard..

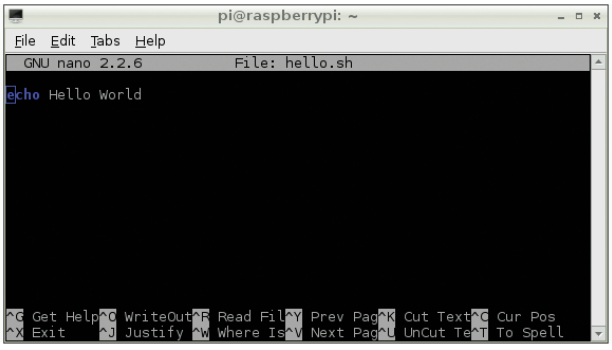


Figure 4.4. Nano.

If you're using an SSH connection you could also use the built-in editor of WinSCP. Refer to section 3.1.7, 3.1.8, and 3.1.10 for details on installing SSH and WinSCP.

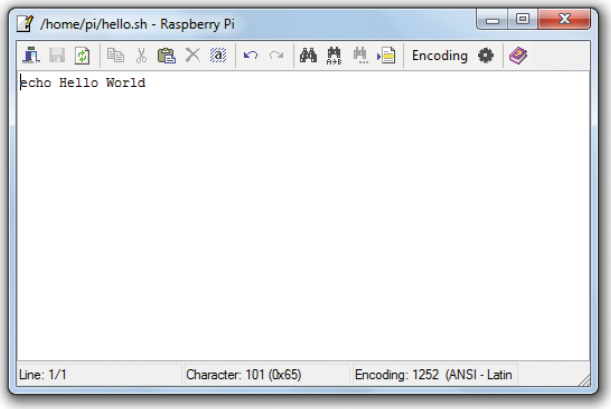


Figure 4.5. WinSCP editor.

Save the program as hello.sh in the directory pi. Since you are the user "pi" this is your home directory. To run this program type the following from the terminal:

```
bash hello.sh
```

You'll see that the text "Hello World" appears on the screen. We can improve the program by asking for the user's name and replying with a personalized greeting.

```
echo What is your name
read name
echo Hello $name
```

The second line of the program looks at what the user has typed in and stores it in a variable called 'name'. In the third line the text Hello is printed, followed by the contents of the variable 'name'. The $ sign is used to indicate that we want to display the contents of the variable 'name' rather than the word name itself.

You can work with numbers in the same way. The following program asks for a whole number and then displays double its value:

```
echo Enter a number
read number
echo Two times $number equals $(($number*2))
```

You'll have noticed that quite a lot of parentheses are needed to get Bash to perform the actual calculation. Note that this program will only work with whole numbers.

Bash is particularly useful for carrying out a series of commands in one go. Imagine that you wanted to play back the file blonde.mp3 at three different volume levels. The following Bash program would achieve this:

```
amixer set PCM 20%
mpg321 blonde.mp3
amixer set PCM 60%
mpg321 blonde.mp3
amixer set PCM 80%
mpg321 blonde.mp3
```

The amixer program is used to set the volume level, and the mpg321 program plays back the blonde.mp3 file.

Up to now, all of the commands have been executed once only. It is also possible to execute commands several times, for example for as long as a particular condition is met. This can be done using the `for` command. Its structure is as follows:

```
for [variable] in [ range ]
do
   commands that
   have to be repeated
done
```

The variable takes on the value of each of the numbers in the range in turn, and for each value the commands between do and done are executed. The commands that are repeated are in what is called a loop. You can use the variable as a simple counter, for example when you want to play back the file blonde.mp3 five times in succession:

```
for i in 1 2 3 4 5
do
   mpg321 blonde.mp3
done
```

In this case the name of the variable is 'i'. Programmers tend to dislike typing, so they choose variable names that are as short as possible.

You can also use the variable inside the loop. In the following program, the variable is used to set the volume. The loop is executed three times, where 'i' takes on the value 20, 60 and 80. You can see that it takes very little extra typing to increase the number of different volume levels. It's tasks like these where Bash programs come in very useful.

```
for i in 20 60 80
do
   echo Volume is $i
   amixer set PCM $i%
   mpg321 blonde.mp3
done
```

The commands within a loop are normally indented. This makes it much easier to see where the loop begins and where it ends.

It could be possible that we don't know in advance how many times the loop has to be executed, but we do know that a particular condition has to be met for the loop to continue. An example of such a command is the `while` command. This commands repeats a certain section of the program while a certain condition is met. A condition is met if it is true. The structure is as follows:

```
while [ condition ]
do
   commands that
   have to be repeated
done
```

In the next program you have to guess which number the Raspberry Pi is thinking of. Your input is compared with the solution (2) and if you gave the wrong answer, the Raspberry Pi will repeat the question.

```
number=0
while [ $number -ne 2 ]
do
   echo Have a guess
   read number
done
echo Correct!
```

In the first line, the variable `number` is given the value `0`, so that the program knows that the next line concerns a number. Without this line, the program would assume that 'number' was a text string and you would immediately get an error message in line 2. In this line a check is made to see if the value of the variable `number` is not equal (`-ne`, for not equal) to 2. If it isn't equal then you haven't guessed the correct answer and you have to try again. At this point in time the program has just started and you haven't yet had the chance to input your guess. The part of the program that is repeated in the while loop is between the words do and done. The program then asks you to input a number (Have a guess), and stores your answer in the variable `number`. The program then jumps back to the start of the `while` loop and compares your answer with the correct answer. If it is correct then the `while` loop is skipped and you'll see 'Correct' on the screen. If it is wrong then the while loop will be executed again.

A special case of this loop is the `while true` loops. Since true is always true this loop will be repeated forever, or in any case until you stop the program with Ctrl-C or you switch off the Raspberry Pi. [14]

14 Ctrl-C means that you hold down the Ctrl key and then briefly press the C key.

```
while true
do
    echo Press Ctrl-C to stop!
done
```

4.3 Programming in Python

The standard programming language for the Raspberry Pi is Python.[15] All software is already installed on the Raspberry Pi and you can get started straight away. If you also want to use Python on your Windows PC you first have to install it. Since Python is a multi-platform programming language it can be used on both the PC and the Raspberry Pi. It's even possible to run exactly the same program on both computers! [16] In this book we make use of Python for client-server applications on the Windows PC. This means that part of the program runs on your Windows PC and part runs on your Raspberry Pi.

The easiest way to write Python programs is with Thonny.

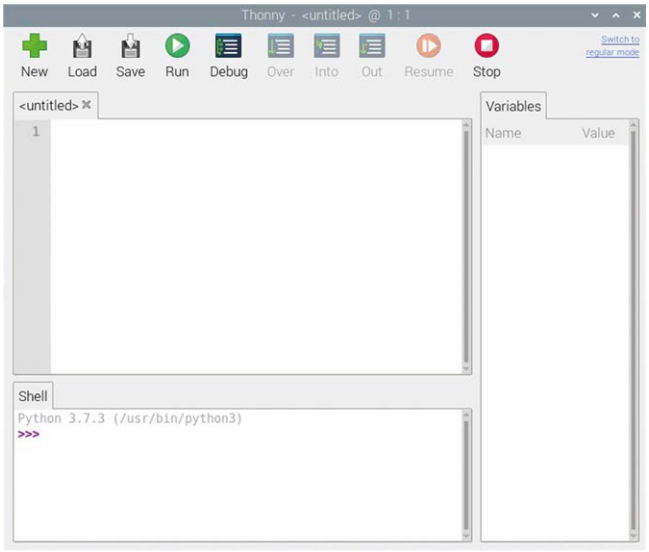


Figure 4.6. Thonny startup screen

You can start Thonny on the Raspberry Pi via the desktop menu (Programming → Thonny Python IDE). As you can see from Figure 4.6, Thonny is based on Python 3. At the bottom part of the screen you will see the Python promt as ">>>", where you can enter interative command here directly. Figure 4.7 shows the famous Hello World message displayed at the bottom part of the screen. The nice thing about Thonny is that you can enter commands directly in the bottom part of the screen to see how various commands work, or to test your ideas while programming.

15 There are two versions of Python on the Raspberry Pi: version 2 and 3. In this book only the latest version Python 3 is used.
16 Unless you use hardware that doesn't exist on a computer. Your PC won't have a GPIO port, which means that a program using that port can't work on the PC.

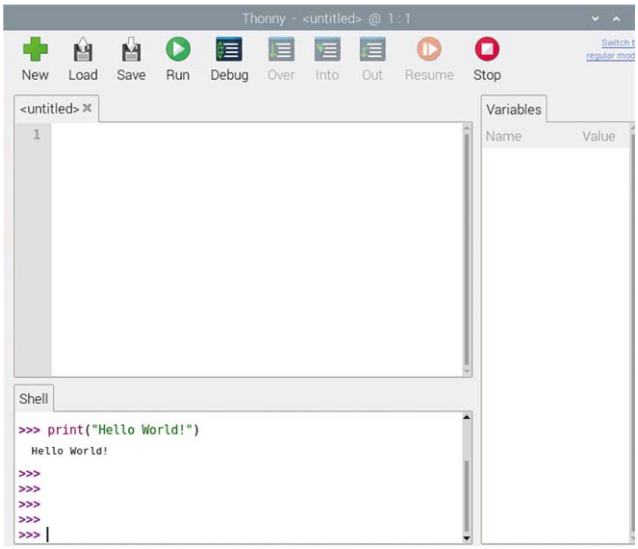


Figure 4.7 Entering a command interactively

The top part of the screen is used for creating new Python programs. Here, we have the following meny options:

New: Create a new program

Load: Load an existing program

Save: Save a program

Run: Run a saved program

Debug: Debug a program

Over: Used while debugging a program

Into: Used while debugging a program

Out: Used while debugging a program

Resume: Used while debugging a program

Stop: Used to stop a running program

If you're working via an SSH connection from a Windows PC you can create a program for example using the nano text editor in command mode, and then run the program by entering the command `python3` followed by the program name. The program names must

have extensions ".py". For example if the program name is hello.py, the following command should be entered to run the program:

```
python3 hello.c
```

Thonny is a programming environment as well as a Shell from which you can type in Python commands directly. We'll give that a try straight away by entering the following statement at the bottom of the screen:

```
print("Hello")
```

The response from Python appears immediately: `Hello`.

Since the text is enclosed by quotes, it's not possible to display a quote, since Python would think that the quote signified the end of the text. The way round this is to use a backslash. This backslash is a "command" for Python so it won't interpret the following character, but just display it. A consequence of this is that you can't display a backslash, but this is resolved by adding a second backslash after the first.

`\'`	single quote
`\"`	double quote
`\\`	backslash

The statement

```
print("A single \' double \" and backslash \\ symbol")
```

Will display the following:

```
A single ' double " and backslash \ symbol
```

We can also use Thonny to write programs. Select File and then New to open an empty window. We type in our program (again just a single line: `print("Hello World")`. Before you can run the program, it first needs to be saved. Select Save to save the program. You can use any suitable name, for example myprogram.py. It is important that the file has the extension .py otherwise you won't be able to tell at a later date in which language a program was written. The extension also tells Thonny that this is a Python program, which makes it display your source code with clear color codes. [17] Once the program has been saved you can run it by clicking the Run button with a green arrow. The output of the program will appear in the Shell window as shown in Figure 4.8.

17 It is customary for graphical programs that use the wxPython module to be saved with a .pyw extension. We'll come back to this later in the chapter.

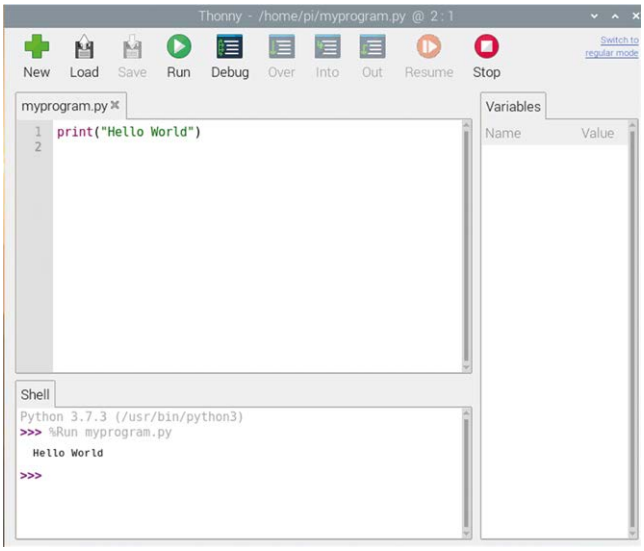


Figure 4.8 Running a program using Thonny

When you have finished developing your program, you can close Thonny. Just as with the Bash program, we can ask for the user's name and use it in a greeting.

```
name=input("What is your name ")
print("Hello " + name)
```

The command input is used to ask the user a question. The answer to the question is stored in the variable 'name'. In the next line, the text Hello is displayed followed by the contents of the variable. [18] It is also possible to type this program into the bottom part of Thonny (i.e. in the Shell) one line at a time. This is a smart way to quickly try out some commands.

Python knows about two types of numbers: integers and floats. Integers are whole numbers, such as 1 or 507. Floats are numbers that have decimal places, such as 2.5 or 78.253. Python also uses strings and lists. A string is a line of text, such as "Hello", and a list consists of a number of items. These can be numbers [2,67,190], but also strings ["hello","book","bicycle"]. Since Python wasn't originally written for use on microcontrollers the variable types bits and bytes weren't included.

In Python 3, the `input()` statement always returns a string, even if the user enters numeric data (this behavior is exactly same as the `raw_input` statement in Python 2). Therefore, for example in the following code, the data returned by the `input()` statement is string type '10' and not numeric 10:

```
Age = input("Enter your age: ")
print(type(Age))
```

18 In Python version 2, the raw_input and input statements are used instead of the input statement.

The output will be as follows. Statement `type()` displays the type of a variable:

Enter your age: 10

`<class 'str'>` As you can see, the type of variable Age is string. i.e. `str`.

Suppose that we wish to receive two numbers from the keyboard, add them and then display the result. The following code will give wrong answer since it is trying to add two strings.

```
N1 = input("Enter number 1: ")
N2 = input("Enter number 2: ")
Sum = N1 + N2
print(Sum)
```

Running the program gives:

```
Enter number 1: 5
Enter number 2: 3
53
```

The result is 53 where the program is combining the strings '5' and '3' to give string '53'.

The correct way will be to convert the inputted numbers from string into integer or floating point format using the built-in functions `int()` of `float()` respectively as shown in the following code:

```
N1 = float(input("Enter number 1: "))
N2 = float(input("Enter number 2: "))
Sum = N1 + N2
print(Sum)
```

Running the program now gives:

```
Enter number 1: 5
Enter number 2: 3
8.0
```

In Python verion 2, text and binary data (bytes) are mixed and can be used interchangeably. In Python verion 3 however, all text is Unicode and encoded Unicode is represented as binary data. Binary data can be converted to strings using the decode function. Similarly, strings can be converted to binary data using the encode finction. By default, the decode/encode function used in known as the `utf-8`. For example, we have to use str.encode('utf-8') to go from str to bytes, and use bytes.decode('utf-8') to go from bytes to str.

Binary data (bytes) in Python version 3 is represented using 'b' followed by the data. The example shown below assigns binary data 4 to variable a:

```
a = b'4'
print(type(a))
<class 'bytes'>
```

An example of converting a string into bytes is shown below:

```
a = "hello"
b = a.encode('utf-8')
```

now, printing b gives:

```
print(b)
b'hello'
```

Similarly, the following example converts a byte into a Unicode string:

```
a = b'3'
x = a.decode('utf-8')
print(x)
3

print(type(x))
<type 'unicode'>
```

Whenever you use a variable Python tries to figure out what type it should be. It usually does this very well, but you have to be careful when numbers are involved. Suppose that we have a variable called temp and we assign a value of 5 to it.

```
temp=5
```

If we now divide temp by 2 the answer will be 2.5 as shown below:

```
>>> temp=5
>>> temp=temp/2
>>> print(temp)
  2.5
```

We can also enter `temp` as a floating point nuber as shown below. The answer will be the same:

```
>>> temp=5.0
>>> temp=temp/2
>>> print(temp)
  2.5
```

An integer variable can be converted into floating point by using the built-in function `float()` as shown below:

```
>>> temp=float(temp)
>>> print(temp/2)
 2.5
```

It is also possible to convert a floating point number into an integer number by using the built-in function `int()`. This is shown in the following example code:

```
>>> temp=2.5
>>> temp=int(temp)
>>> print(temp)
 2
```

There are two ways in which to combine text and variables. The first way uses a comma.

```
>>> temp=2.5
>>> print("The value is",temp,"points")
 The value is 2.5 points
```

The second way uses substitution. You put `%s` in the position where you want the variable to appear, and you add the variable to the end of the line. This is very similar to the way it's done in the C programming language, which is why it's so common (most programmers also "speak" C).

```
>>> print("The value  is %s points" % temp)
 The value is 2.5 points
```

When there are several variables you put them between parentheses after the percent sign, separated by commas. In the text you still use `%s` for every variable. They will be displayed in the order in which they occur after the percent sign.

```
>>> temp2=3.4
>>> print("The values are %s and %s points" % (temp,temp2))
 The values are 2.5 and 3.4 points
```

The advantage of the C method is that you can define how the numbers should be displayed:

`%s`	Convert it to a string and display it.
`%d`	Display it as an integer.
`%c`	Display it as a character.

You can also specify formatting options:

`%4s`	A minimum of 4 characters, filled from the left with spaces.
`%.2f`	A float with two decimal places.

A quick example of a float with formatting:

```
>>> print("A float with two decimal places %.2f" % temp)
A float with two decimal places 2.50
```

You can easily experiment with the different display options in Thonny before you apply them in your program.

The list is a special type of variable. In many other programming languages the term array would be used instead of list. A list is a number of items that share a common name: the name of the list. You can create a list by putting all elements of the list inside square brackets and assigning it to the variable:

```
L=[34,6,128]
```

The list is called L and contains three integers, which are 34, 6 and 128. To keep the elements in a list apart they're all given a number. This number is called the index. Note that the index starts at 0 (the same as in arrays). The number 34 in list L therefore has an index of 0; the number 6 has an index of 1. You can select an element from the list by adding its index between square brackets.

```
>>> L=[34,6,128]
>>> print(L[0])
 34
```

Incidentally, in Python a string is also a list, where each character is automatically given an index number.

```
>>> name="Bert"
>>> print(name[3])
 t
```

At first sight the list seems just like an interesting feature to have, but Python has a number of very powerful functions for lists: [19]

19 Have you noticed that I've typed various commands directly into Thonny to try them out? By all means try it yourself. It is very unusual to be able to do this in a programming language. No doubt there is something you'd like to try out. This is how you learn the quickest!

print
The print statement can be used to display the contents of the whole list in one instruction (see the example for append)

append
This instruction is used to add an element to the end of the list. Suppose we want to add the number 300 to the end of list L. This can be done with the instruction `L.append(300)`

```
>>> L = [34, 6, 128]
>>> L.append(300)
>>> print(L)
 [34, 6, 128, 300]
```

pop
The opposite of append is `pop`, since this removes an element from the end of the list.

```
>>> L.pop()
300
>>> print(L)
 [34, 6, 128]
```

You can also remove any other element from the list by specifying its index number between the parentheses. With `L.pop(0)` you will remove the first element from list L.

With clever use of the append and `pop(0)` instructions you can maintain a list that has the last three numbers that have been input. With append you add the last number entered to the list and with `pop(0)` you remove the first number from the list. A good example of this can be found in project 9.4 where this method is used to calculate a moving average.

sum
The `sum` instruction is used to add up all elements of a list, assuming that all of them are integers or floats. When you try to add strings Python stops with an error message.

```
>>> print(L)
 [34, 6, 128]
>>> print(sum(L))
 168
```

len
The `len` instruction returns the number of elements in a string. Since Python also treats strings as lists this instruction can also be used to determine the length of a string. As an example, the len instruction can be used to calculate the average of a list.

```
>>> print("The average of L is ",sum(L)/len(L)*1.0)
 The average of L is 56.0
```

Don't forget to multiply by 1.0 to ensure that Python works with floats. All elements in list L are integers. In this case it makes no difference because the average happens to be an integer as well, but you would only know this if you did the calculation yourself, and that's not why you've acquired an expensive computer. Always make sure that Python calculates with floats if you're not sure what type the result is.

slice

Slice is used to extract parts from a list. The syntax is `Listname[index1:index2]`. The first index is the first element you want to select, the second index is for the element that you no longer want to select. If you leave out an index it means "everything".

```
>>> print(L)
 [34, 6, 128]
```

We now display L from index 0 and stop before index 1:

```
>>> print(L[0:1])
 [34]
```

We display L from an index to the end:

```
>>> print(L[0:])
 [34, 6, 128]
>>> print(L[1:])
 [6, 128]
```

If you use a negative index you can display the list up to a certain number of elements from the end. The following example will show the whole list except the last element:

```
>>> print(L[:-1])
 [34, 6]
```

replace

With `replace` you can replace certain elements in a list with other elements. This will only work for a list of strings and not for numbers. We'll first create a list with four strings and then replace all occurrences of "a" with "x". The structure of the command isn't immediately clear because it is really a one-line program where we create a string (w) and use that each time to perform the replacement.

```
>>> L=["a","b","a","c"]
>>> L=[w.replace("a","x") for w in L]
>>> print(L)
 ['x', 'b', 'x', 'c']
```

The command becomes a lot clearer when we use it on a single string:

```
>>> temp="Some text with five words"
>>> print(temp.replace("five","four"))
 Some text with four words
```

There are many more functions that can be used with strings (such as min, max, remove), but these aren't used in this book.

Operators

In principle there are two types of operators: arithmetic and logical. An example of an arithmetic operator is the +, which stands for "addition". The result of an arithmetic operator is always a number.

Another example of an arithmetic operator is &. This symbol stands for the arithmetic AND which adds two numbers together according to the following truth table:

&	input1	input2	output = input1 & input2
	0	0	0
	0	1	0
	1	0	0
	1	1	1

Table 4.3 And (&) truth table.

The AND operator is performed at the binary level for the whole variable. If the variable is a byte then the operation is performed on all 8 bits. For example:

```
3 & 7 = 3
```

This is easier to understand in binary:

```
0000 0011 & 0000 0111 = 0000 0011
```

The Windows calculator can also perform arithmetic AND operations.

A logical operator doesn't give a number as result, but instead returns either true or false. The logical AND operator can be used to check two conditions simultaneously. When both conditions are true the result of the AND operation is also true.

An example of a logical operator is 'and'. This looks at two conditions and returns true or false according to the following truth table:

and	input1	input2	output = input1 and input2
	false	false	false
	false	true	false
	true	false	false
	true	true	true

Table 4.4. AND truth table.

The following table gives an overview of several operators that can be logical as well as arithmetic, which may sometimes be confusing.

Operator	Logical	Arithmetic
AND	and	&
OR	or	|
NOT	not	~
XOR		^
equal to	==	=

Table 4.5. Arithmetic and logical operators.

Note that the "equal to" logical operator consists of a double "=" sign. The instruction `i==7` returns "true" if i was equal to 7 and "false" if this was not the case.

A single "=" sign is an arithmetic operator. The instruction `i=7` results in the value 7 being assigned to the variable i.

Logical operator	Description
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
!=	not equal to

Table 4.6. Logical operators.

Arithmetic operator	Description
+	addition
-	subtraction
=	assignment
*	multiplication
**	exponent
/	division
%	modulus
>>	shift 1 bit to the right
<<	shift 1 bit to the left

Table 4.7. Arithmetic operators.

Arithmetic abbreviations

In many programs you will come across arithmetic abbreviations. These are often used when you want to increase a counter by 1, where you can use `i+=1` instead of the usual `i=i+1`. The following table shows the most common abbreviations:

Abbreviation	Meaning	Description
i+=5	i = i + 5	Add 5 to i.
i-=5	i = i - 5	Subtract 5 from i.
i*=5	i = i * 5	Multiply i by 5
i/=5	i = i / 5	Divide i by 5

Table 4.8. Arithmetic abbreviations.

For

Python can also have loops, just like Bash. One point to note is that you must indent the code in the loop; it's not optional. This is because Python doesn't have any instructions for the beginning or end of a loop. The line where you start the indentation is the beginning of the loop and the line where you stop is the end of the loop. For example:

```
for i in range(1, 5):
   print(i)
print("Done")
```

Only the statement `print i` is part of the loop. Can you work out what this program will display? The variable "i" is assigned every (whole) number in the range from 1 to 5 (but not including 5), so 1,2,3 and 4. The program will display these numbers, starting with 1. The next statement `print("Done")` is not indented, so isn't part of the loop. The output of this program will therefore be:

```
1
2
3
4
Done
```

Did you notice that there is a colon at the end of the `for` line? This is always required at the beginning of a loop.

You must always make sure that you use either spaces or tabs to indent the lines, but not a mixture of the two. This would just confuse Python and the program may not work. It is a pity that Python can't cope with a mixture of tabs and spaces since this makes it more difficult to copy snippets of code from another program or the Internet. The following program looks ok, but it won't work:

```
for i in range(1, 5):
    print(i)
    print(i+3)
print("Done")
```

This is because the first indentation consists of a tab and the second consists of 7 spaces. You can't see the difference, but Python does. It gives an error message "unexpected indent".

While

The Python while loop has the following format:

```
while [ condition ]:
    statements that
    have to be repeated
```

An example of this is in the following program, which keeps asking you for a number until you enter one which is greater than 10.

```
value=0
while value <= 10:
    value=int(raw_input("Enter a number ") )
print("Done")
```

Note that the result of input is always a string (text). Therefore, when you input 4 the program doesn't see it as a 4 on which it can carry out calculations, but as text. Before it can be used in calculations, it has to be converted to an integer, using the instruction `int()`.

When using a `While True` loop in Python you have to bear in mind that `True` should always start with a capital letter. This is the opposite of Bash, where it has to start with a small letter.

```
while True:
    print("This never stops")
```

Pass

Sometimes you may want to have a while loop, but without any instructions within the loop. This could apply when the condition changes externally, for example. This program waits for somebody to release a switch connected to a GPIO. [20]

```
while switch.value:
    pass
```

The program doesn't need to do anything to check for this, so the loop is empty. However, an empty loop is not permitted. We therefore put a pass statement in the loop, which does nothing other than satisfy the syntax requirements.

If-then

The `if-then` statement is a decision-making statement. `if` certain condition is met, "then" the program should do something. The keyword "then" isn't actually used in Python. Instead, a colon is used.

```
if i==5:
    # do this if i equals 5
```

Sometimes you may want the program to do something if the condition isn't met. To do this you have to add an `else` section. `if` a certain condition is met, 'then' the program should do something, 'else' do something different.

```
if i==5:
    # do this if i equals 5
else:
    # otherwise do this
```

Note that there is also a colon after the word else.

20 We will discuss in chapter 5 what GPIO is and how it works.

Binary

People generally use decimal numbers. For example, the number 135 means one hundred and thirty-five. It is in fact just a sequence of digits, but we have agreed that the position of each digit determines what its value is. The digit 5 is furthest right, and is therefore equal to five. The digit 3 is in the second position (we're counting from right to left) and equals thirty. The digit 1 is in the third position and equals one hundred. We're effectively using the following table, where it should be noted that we start counting from 1 and computers from 0.

position	3	2	1	0
factor	times a thousand	times a hundred	times ten	times one

This is called the decimal system (deci = 10) since each factor is different factor of 10:

position	3	2	1	0
factor	times 103	times 102	times 101	times 100

For computers it's easier to work with two digits, 0 and 1, off and on, low and high. This is the binary system .

position	3	2	1	0
factor	times 23	times 22	times 21	times 20

The binary number 101 is $1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 5$ decimal. The problem is that you can't tell that this number is binary. (That comes in very handy for jokes: "There are 10 types of people in the world, those who understand binary and those who don't".) To avoid this confusion we use a prefix, which for binary numbers is 0b. Note that in contrast to many other programming languages, Python can't display binary numbers.

It is very useful if you can convert between binary and decimal in your head. However, if mathematics isn't your strongest point you can use the free Windows calculator. You have to set it to the scientific or programmer mode before these features become available.

Using extensions

Apart from all the standard Python instructions, you can also use extra instructions by loading specific modules. This is done using the `import` statement. For example, Python doesn't have an instruction for pausing the program. The time module does have such an instruction. This example program will display the letter "a" every 0.3 seconds:

```
import time
while True:
   print("a")
   time.sleep(0.3)
```

The prefix `time` in front of sleep indicates that the instruction `sleep` can be found in the time module. During the loading of a module you can give it a different name if you like:

```
import time as t
while True:
    print("a")
    t.sleep(0.3)
```

If you require a single instruction from the module, you can load just that instruction. In that case you no longer have to precede the instruction with the module name:

```
from time import sleep
while True:
    print("a")
    sleep(0.3)
```

We use both of these methods throughout the book. If you want to know which instructions are in a module, you can use the `dir()` function. Before you use this, you must first load the module.

```
>>> import time
>>> dir(time)
['__doc__', '__name__', '__package__', 'accept2dyear', 'altzone', 'asctime',
'clock', 'ctime', 'daylight', 'gmtime', 'localtime', 'mktime', 'sleep',
'strftime', 'strptime', 'struct_time', 'time', 'timezone', 'tzname']
```

You can get more detailed information using the `help()` function, for example:

```
help(time)
```

The result is an extensive description of all the features of this module. In practice you'll find that the depth of the descriptions can vary a lot between modules. It is noticeable that modules written by hobbyists often have a lack of help functions.

Files

Python considers files to be objects. When they are opened, they have to be assigned immediately to a variable. The following instruction opens the file counter.dat in the cgi-bin directory as read-only ("r").

```
myfile = open('cgi-bin/counter.dat','r')
```

Note that Linux uses the forward slash / in path names, whereas Windows uses the backslash \. Except for reading ("r"), a file can also be opened for (over)writing ("w"), or appending ("a"). The contents of a file always consists of strings.

Now that the file has been opened, you can start reading it. The following instruction reads one line:

```
data=myfile.readline()
```

Information in a file is always interpreted as text. You can convert it into numbers using `int()` or `float()`, assuming that the data from the file are indeed numbers, otherwise you'll get an error message. Other options are:

Read all lines into a list	`data=myfile.readlines()`
Read n characters	`data=myfile.read(n)`
Write a string	`myfile.write("hallo")`
Ditto, with a linefeed	`myfile.write("hallo\n")`
Write a list	`myfile.writelines(data)`

Without any linefeeds you end up with all strings on one line. This is not a problem, but it does make it more difficult to read back the data. It is therefore recommended to use linefeed characters. Since Windows and Linux use different characters to signify the end of a line, \n is replaced by the character that is in use on your computer. You can also use the \ options that were discussed previously in the `print` section, such as \".

When you have finished with the file, you can close it:

```
myfile.close()
```

The following example opens a file, writes "bert" to it including a linefeed, then writes 34 to it and finally closes the file.

```
myfile = open('mydata.dat','w')
myfile.write("bert\n")
myfile.write(str(34))
myfile.close()
```

We'll now read back these two items and display them. Note that we use slicing to prevent the end-of-line character {\n} from being shown:

```
myfile = open('mydata.dat','r')
print(myfile.readline()[:-1])
print(int(myfile.readline()))
myfile.close()
```

We'll read the file again, but this time put the contents into a list, which is then displayed.

```
myfile = open('mydata.dat','r')
L= myfile.readlines()
print(L)
myfile.close()
```

The result:

```
bert
34
['bert\n', '34']
```

Try/except
Try/except is a neat way to trap errors without stopping the program and showing the user an error message. In effect, you're telling Python to try something and if it doesn't work, carry out the instructions following 'except'.

As an example, we'll look at part of the program from project 11.3.2. You can see we first try to open the file, read a number from it and store it in the variable number. If we don't succeed (if the file doesn't exist, for example, or the line doesn't contain anything that can be converted into a number by `int()`), the program jumps to except. Here the variable gets a value of zero, after which the program carries on without a break.

```
try:
    # Read a number if the file exists
    myfile = open('cgi-bin/counter.dat','r')
    number=int(myfile.readline())
    myfile.close()
except:
    # otherwise give it a value of zero
    number=0
```

You can use the same technique to trap Ctrl-C presses by users who try to stop the Python program. In Python this is called a keyboard interrupt.

```
try:
    # the normal program is here
except KeyboardInterrupt:
    # we arrive here when the user has pressed Ctrl-C
```

In section 12.8 this method is put into practice. You may wonder what happens if an error occurs other than Ctrl-C. This error would not be intercepted by the program and will be dealt with by Python. It is possible to use multiple excepts to be able to trap other errors, like this:

```
try:
    # the normal program is here
except KeyboardInterrupt:
    # we arrive here when the user has pressed Ctrl-C
    raise
except:
    # we arrive here for all other errors
```

You have to use a raise statement after the first except, which effectively "arms" the error so that it can be trapped again by an except.

Functions

Functions are instructions that you create yourself. In practice they are snippets of program that you need to use in several places within the program. To avoid having to type in the same code every time you can create a single function, which is called each time it is required.

They are also handy when you've discovered something very useful, and you want to include it in your other programs. The best way to do this is to turn it into a function, since this is easier to copy (everything will be in one place). Experienced programmers often have their own libraries with useful functions so they don't have to write the same code over and over again.

A function begins with the word `def` (from definition). This is followed by the name, which can be anything you like as long as it doesn't have spaces or weird characters. It's recommended that you use a name that describes what the function does. After this come the variables that your function requires when it is called, enclosed between parentheses.

Inside the function are the instructions that form your snippet of code. Any new variables that you use inside the function won't be known outside it. This is what makes functions so easy to use in other programs.

It sounds more complex than it is.

```
def printme(number):
print(number)
```

You can call the function using `printme(34)`. The variable "number" in the function is assigned the value of 34, which is then displayed on the screen. You can also call the function another way:

```
value = 34
printme(value)
```

This time the variable "number" is assigned the contents of variable "value". Since the content of the variable "value" is 34, the number 34 will be shown on the screen. When this function is called without a parameter, i.e. `printme()`, you will get an error message. A neat solution is to assign a default value to the function. This default value will be used when the user hasn't entered a parameter.

```
def printme(number=0):
print(number)
```

When this function is called without a number the value 0 will be displayed, otherwise the value of the parameter will be shown.

A function can also return values back to the main program, using the `return` statement. Following the `return` should be the variable that you want to return to the main program.

```
def timesfive(number=0):
        factor = 5
        return number * factor
```

The function now has to be called slightly differently, since it returns a number, which has to end up somewhere. We can display the result straight away:

```
value = 3
print(timesfive(value))
```

The result of these instructions is the number 15 (3 times 5). The variable "factor" is used for the first time inside the function, so it is known only inside the function. Try out the following:

```
def timesfive(number=0):
        factor = 5
        return number * factor

value = 3
print(timesfive(value))
print(factor)
```

You immediately get an error stating that you're trying to display a variable that hasn't had a value assigned to it yet, so Python doesn't know what to display:

```
15

Traceback (most recent call last):
  File "C:\even\t.py", line 7, in <module>
    print(factor)
NameError: name 'factor' is not defined
```

Even when you assign a value to the variable "factor" before the function is called, Python will see this variable as a different one to the variable "factor" that is used inside the function. So this program:

```
def timesfive(number=0):
        factor = 5
        return number * factor
```

```
factor = 1
value = 3
print(timesfive(value))
print(factor)
```

will display 15 and then 1. If you don't want this to happen and want to use the same variable everywhere in the program you can declare it as a global variable. You have to use the keyword `global` wherever the variable "factor" is assigned a value for the first time, inside the function as well as outside it.

```
def timesfive(number=0):
        global factor
        factor = 5
        return number * factor

global factor
factor = 1
value = 3
print(timesfive(value))
print(factor)
```

In general, it's best not to use global variables because it makes it so much more difficult when you want to transfer the functions to other programs. It is always possible that the function depends on a global variable in a totally different area of the program for its proper operation. The following solution would be better:

```
def timesfive(number=0, factor=1):
        return number * factor


factor = 5
value = 3
print(timesfive(value, factor))
print(factor)
```

We discussed global variables here because we'll be using them later in a few graphical programs to make objects global and it's important that you understand why and how it works.

Graphics

Up to now the Python programs weren't graphical, but it is of course possible to make graphical programs with it. This will be much more complex than a standard program. For this reason we'll use a graphical template in this book. This is a minimal program that creates a window and contains everything else needed to run a graphical program. To this "empty program" we then add extra objects (such as buttons) and functions (such as what should happen when a button is clicked). This method makes graphical programming

relatively straightforward.[21] Notice that the graphics programs work in the Raspberry Pi Desktop mode.

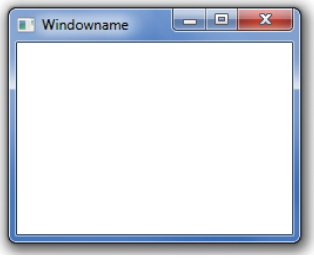


Figure 4.9. Output of the standard graphical template.

The graphical template puts an empty window on the screen. The areas in the template where you can add your own objects and functions are highlighted in bold. You can run the template as it stands from Thonny. The file is called template.pyw and it can be found in the directory book/04. [22]

Before you can create and use graphical programs, you need to have wxPython installed. On the Raspberry Pi SD card for this book this is already done for you, on the PC you should follow the installation instructions in section 3.

```
import wx


class Example(wx.Frame):

    def __init__(self, parent):
        # the window
        wx.Frame.__init__(self,parent)
        self.InitUI()

    def InitUI(self):
        self.SetBackgroundColour('white')
        self.SetSize((250, 200))
        self.SetTitle('Windowname')
        self.Centre()
        self.Show(True)

        # your own objects

    # your own functions
```

21 Unfortunately, Python doesn't have a built-in graphical editor, such as in Visual Basic, which can be used to create and position buttons onto the screen. We therefore have to use some trial and error for the placement of the windows, buttons etc.

22 It is customary to save graphical programs with an extension of .pyw instead of .py when they use the wxPython module.

```
def main():
    # the standard window
    ex = wx.App()
    Example(None)
    ex.MainLoop()

# if this is the main program then run the main function
if __name__ == '__main__':
    main()
```

In Table 4.9. we shown the template again with full explanations on how it works, so you can adapt it if you like.

`import wx`	Loads the wx module that contains the graphical instructions.
`class Example(wx.Frame):`	We create our own class, which is a sub-class of the standard wx.Frame class.

`def __init__(self, parent):`
`  # the window`
`  wx.Frame.__init__(self,parent)`
`  self.InitUI()`
	This function runs automatically when the program starts. The first thing this init function has to do is to call the init of wx.Frame, thereby creating the frame 'self'.

`def InitUI(self):`
`  self.SetBackgroundColour('white')`
`  self.SetSize((250, 200))`
`  self.SetTitle('Windowname')`
`  self.Centre()`
`  self.Show(True)`
	This function creates the window (also called a frame). The name of this window is 'self', with dimensions of 250x200, and the title 'Windowname'. The window is centered on the screen and then displayed.

`# your own objects`	This is where you should put your own objects that you want shown in the window, e.g. a button. This will happen as soon as the program starts.
`# your own functions`	This is where you should put your own functions, which can then be called from other functions or the main program. These are often functions that deal with the new objects.

`def main():`
`  # the standard window`
`  ex = wx.App()`
`  Example(None)`
`  ex.MainLoop()`
	This is the function that is run when the script is started as a program, rather than loaded as a module. wx.App is the program itself, which is given the name ex and is linked to the graphical wx part. You must first have the program (application) defined before you create a window or put any objects on it, otherwise they'll end up "in the void".
	Next, an instance of our previously created class is started. The ex.MainLoop instruction finally starts the actual program, which is event driven. This is a type of endless loop that can trap all events.

`# if this is the main program then run the main function`
`if __name__ == '__main__':`
`  main()`
	If this script is run as a program then main() is run, otherwise it isn't.

Table 4.9. The graphical template with descriptions.

There are numerous objects that can be used in this template, of which a handful are used in this book.

Button
Below is an example of a button object:

```
btn = wx.Button(self, label='Close', pos=(20, 120),size=(80, -1))
btn.Bind(wx.EVT_BUTTON, self.OnClose)
```

In the first line the button object is defined and is given the name btn. The text on the button is "Close". The position in the window is 20 from the left and 120 downwards (20,120). Its size is 80 wide, and as high as is necessary (80,-1). The button is created in the object "self", which is the window that we created with the template.

In the second line a function is linked to this object. The name of this function is `OnClose` (we've chosen this name ourselves, so we should create a function with this name later on). This function is called when a button-event takes place, in other words, when the user clicks on the button. Just as for the button, this function is created in the object "self"

The function `OnClose` can now be added to the section labeled "your own functions":

```
def OnClose(self, e):
  self.Close(True)
```

This function closes the object "self", which is the window created by the template. When this is closed the program stops as well.

We now have a button on the screen with the word "Close" on it. When the user click on this button the function `OnClose` is called, which stops the program. By all means try it out yourself!

Static Text
Static text is some text that can be put on the window. In contrast to what the name suggests, the text can be changed while the program is running. An example of its use can be found in project 5.5.

Spinbox
A spinbox is an input box where the user can select a value using the up and down arrows. An example of its use can be found in project 8.5.

Slider
A slider is a slide control that can be operated using the mouse. A different value applies to each position of the slide control. An example of its use can be found in project 8.3.

Paintbox
A paintbox is an object in which drawings can be placed, such as graphs. An example of its use can be found in project 10.2.2.

Threading

A thread is a series of instructions that are carried out sequentially by the Raspberry Pi.[23] Only when an instruction has been completed can the next one begin. If the Raspberry Pi could only execute one thread, it would mean it could only do one thing at a time. Since the Raspberry Pi is meant to do several things at once (such as dealing with the Internet, updating the screen, dealing with the mouse and keyboard, etc.) it uses a number of threads that run concurrently.

In Python programs you can also make use of threading. This means that your program can do several things at once. How this is done will be explained in project 5.9.2 (graphical) and project 12.5 (not graphical).

Debugging a Python Program

Debugging is an important part of program development and testing. By debugging a program we try to remove logical errors from the program. Thonny programming environment supports debugging and makes it very easy to check the logical operation of a program. An example is given here to show how a simple program can be debugged using Thonny.

Consider the following simple program which is basically an iteration using a for loop:

```
for j in range(10):
    k = j + 1
    m = j
    print(m)
```

Enter the above program into Thonny and save it. Then click the menu option Debug. You will notice that the program is highlighted in yellow color. Now, keep clicking button called Into and you will notice that we are single stepping through the program. The values of the variables at each programming step are displayed at the right hand side of Thonny under the tab called Variables. Therefore, by single stepping through a program we can easily identify if there are any logic errors. Click the Stop button to stop debugging. Figure 4.10 shows the debugging process.

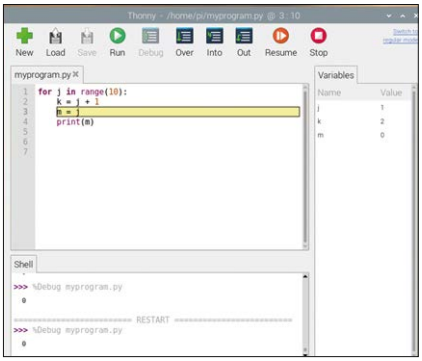


Figure 4.10 Debugging a program

23 This explanation is valid for all computers that run Linux, Windows or iOS. It applies to the Raspberry Pi as well as a Windows PC.

Conclusion

We've skipped more of Python than we've discussed, but the knowledge you've gained should be sufficient for you to read, understand and modify all programs in this book. It also forms a good base from which to discover more about Python yourself.

4.4 Programming in JavaScript

We only use JavaScript in a few projects in this book, so we'll have to limit ourselves to just the essentials. JavaScript is a language that is used in combination with HTML to enhance web pages or to make them interactive. JavaScript is therefore completely different from Java, even though they have similar names. Just like Python, JavaScript is multi-platform, which means that the same code will run on both Windows and Xwindows.

Graphical programs in JavaScript also use objects and functions, just like Python.[24] An object is a thing, a function is an action. The difference is that functions have to be at the top of the page, and objects are put according to where they should appear on the HTML page.

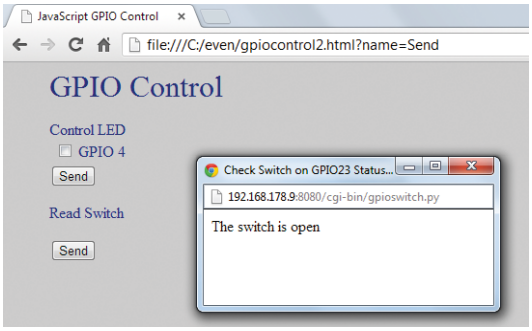


Figure 4.11. JavaScript (project 11.3.5) in web browser on the Raspberry Pi.

You need an editor to write JavaScript programs and a web browser to view your work. On the Raspberry Pi you can use the Text Editor (Accessories -> Text Editor) or nano, on a Windows PC you can use Notepad and Chrome (or Internet Explorer). [25] We start with an empty HTML template where we can put our JavaScript objects and functions.

```
<HTML>
<HEAD>
  <TITLE>Page name</TITLE>
  <SCRIPT language="JavaScript">
  </SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

24 Strictly speaking, this is not completely true. You can add individual JavaScript statements at the beginning of the page, which will be executed as soon as the page loads.

25 On a Windows PC you can open an HTML file in your browser by double-clicking the file. On the Raspberry Pi, use Applications menu->Internet->Chromium web browser

This page doesn't contain anything other than HTML tags (an example of a tag is head, which starts with <HEAD> and ends with </HEAD>). Therefore, when you load this in your browser, all you'll see will be a blank page, with just a title. [26]

On this page we'll put the objects in the body. As an example we'll add a button, which in JavaScript is of a type `submit`:

```
<INPUT TYPE="submit" VALUE="Click me" onClick="clicked()">
```

The text on the button is `Click me`. When the user clicks on the button, a function is called with the name `clicked`. This function has to be at the top of the page, in the script section:

```
function clicked()
  {
     document.write("Hello World!")
  }
```

Everything between the curly brackets belongs to the function. This only puts the text "Hello World" on the document (which is the window).

The complete page now looks as follows:

```
<HTML>
<HEAD>
  <TITLE>JavaScript Tests</TITLE>
  <SCRIPT language="JavaScript">
  function clicked()
  {
     document.write("Hello World!")
  }
  </SCRIPT>
</HEAD>
<BODY>
<INPUT TYPE="submit" VALUE="Click me" onClick="clicked()">
</BODY>
</HTML>
```

When you save this page with a suitable name that ends in .html (test.html for example) and open it with a browser you will see the button at the top-left. When you click on the button it will disappear and the text "Hello World" will be shown.

Just as with Python, we can pass variables to JavaScript functions. For example, we can create two buttons. One of which sends the text "Hello" and the other sends the text "RPi". We'll just show the function here:

26 Your browser may warn you that the page contains active components, and asks you if they can be run. This concerns the JavaScript part, which we've written, so the answer is "yes".

```
function clicked(mytext)
{
   document.write(mytext)
}
```

And the two buttons:

```
<INPUT TYPE="submit" VALUE="Click me" onClick="clicked('Hello')">
<INPUT TYPE="submit" VALUE="or me" onClick="clicked('RPi')">
```

You can add these to the JavaScript template and try it out in your browser.

Another object you'll come across in this book is the checkbox. Before you can use a checkbox, you first have to create one, of course; the JavaScript type for this is `checkbox`.

```
<FORM NAME="menu">
<INPUT TYPE="checkbox" NAME="ledbox" VALUE="checkbox">
<INPUT TYPE="submit" VALUE="Send" onClick="clicked()">
</FORM>
```

On its own, a checkbox doesn't do much. When you click on it, a tick appears and when you click on it again the tick disappears. To process it you need a function, which can be called using a button. In order to process the state of a checkbox the page needs to have a form name. In this case we've named the form `menu`. The button calls the function clicked:

```
function clicked()
{
    if (document.menu.ledbox.checked)
       {
          document.write("Ticked")
       }
    else
       {
          document.write("Not ticked")
       }
}
```

In the function we look if the checkbox called "ledbox" on the form "menu" is ticked (checked). If this is the case then we display the text "Ticked", otherwise we display "Not ticked".

When you put these objects and the function into the HTML template and open it with a browser, you'll see a checkbox with a button next to it. When you click on the button, you'll see the text according to the state of the checkbox.

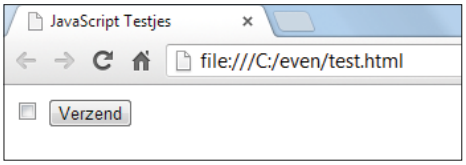


Figure 4.12. JavaScript checkbox and button in Chrome.

To recap, for a checkbox you need:

1. A formname
2. A checkbox
3. A button with a function to process the checkbox.

This is the complete program, with comments showing where your functions and objects should be placed. Any HTML code should be placed in the same part as for the objects.

```
<HTML>
<HEAD>
   <TITLE>JavaScript Tests</TITLE>
   <SCRIPT language="JavaScript">
   <!--put functions here-->
   function clicked()
   {
     if (document.menu.ledbox.checked)
     {
       document.write("Ticked")
     }
     else
     {
       document.write("Not ticked")
     }
   }
   </SCRIPT>
</HEAD>
<BODY>
   <!--put objects and HTML here-->
   <FORM NAME="menu">
   <INPUT TYPE="checkbox" NAME="ledbox" VALUE="checkbox">
   <INPUT TYPE="submit" VALUE="Send" onClick="clicked()">
   </FORM>
</BODY>
</HTML>
```

This is obviously just the tip of the iceberg of what you can do with JavaScript. However, the knowledge you've gained should be sufficient for you to read and modify the JavaScript programs in this book.

Chapter 5 • GPIO

Before you start on any of the projects, you should first read the following introduction, and in particular the safety instructions. You should therefore always make certain that the Raspberry Pi has been turned off before you build the hardware from projects. Don't just pull the plug but shut down the Raspberry Pi properly. This can be done in two ways:

1. Using the command `sudo shutdown -h now`
2. Via Menu → Shutdown → Shutdown → OK.

Usually Linux is able to recover if you have pulled the plug while the Raspberry Pi was on, but not always. If you did turn off the power unexpectedly and the Raspberry Pi won't start up again you will have to restore a backup to the SD card, which you made in section 3.2.1. Any work you did since you made the backup will be lost and you'll have to do it again.

5.1 Introduction

The GPIO header is the header at the top-left with 26 or 40 pins, depending on the model you have (Figure 5.1). Throughout this book we only use the first 26 pins, so all projects can be implemented on the earlier models as well.

The pin numbers for the first four pins are shown in the figure so you can see how the pins are numbered. All the odd numbers are on one side and all the even numbers are on the other side.

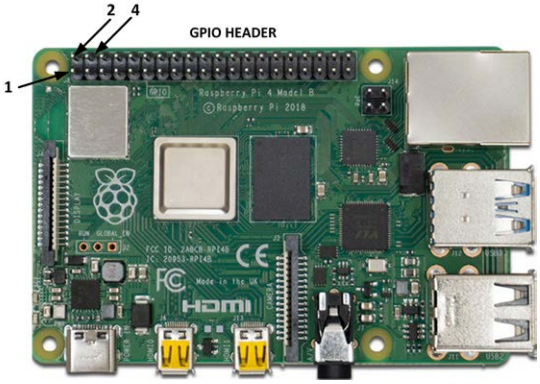


Figure 5.1. GPIO header on the Raspberry Pi 4

The GPIO header provides various functions, such as ordinary pins (in this book we call them GPIO), I^2C, SPI, PWM and serial. The way in which we can use these four types of pins will be explained in detail in later chapters. Some of the pins aren't in use on the model 1, and are labeled with a superscript 'a', for example pin 4 which supplies +5 V on models 2 and higher.

GPIO number	Function	Header pin number		Function	GPIO number
	+3.3 V	1	2	+5 V	
2	I2C SDA	3	4a	+5 V	
3	I2C SCL	5	6	GND	
4		7	8	serial TXD	14
	GND	9a	10	serial RXD	15
17		11	12	PWM0	18
27		13	14a	GND	
22		15	16		23
	+3.3 V	17a	18		24
10	SPI MOSI	19	20a	GND	
9	SPI MISO	21	22		25
11	SPI SCLK	23	24	SPI CE0	8
	GND	25	26	SPI CE1	7
	ID-SD	27	28	ID-SC	
5		29	30	GND	
6		31	32	PWM0	12
13	PWM1	33	34	GND	
19	PWM1	35	36		16
26		37	38		20
	GND	39	40		21

Table 5.1. Function overview of header pins.

You can see from the table above that the GPIO pin numbers don't correspond to the header pin numbers. For example, GPIO 2 is connected to header pin 3, whereas GPIO 7 is found all the way at header pin 26. When you're building a circuit, you must always pay attention to this! In the download package you can find the file header.bmp, which you can print out so you can always have this information available.

You can easily connect all kinds of electronic devices to the GPIO header such as switches and LEDs. One big disadvantage is that these connections have no meaningful protection. A current or voltage that's slightly too high can irreparably damage the Raspberry Pi. This will take some getting used to by those of you who are used to microcontrollers, since these can cope with a knock. And if you do damage one you can replace them very cheaply.

This all means that we have to treat the Raspberry Pi with much greater care. We therefore have the following rules of thumb:

1. Turn off the Raspberry Pi before you modify a circuit, even if you just want to disconnect a wire.[27]
2. Protect all connections from incorrect operation.
3. Only use low-power components, and use a bigger power supply if necessary.

The designed output current for the pins is 16 mA per pin. The total for all pins, including the +3.3 V connection itself, is 51 mA. When you don›t use all of the pins, you can use more than 16 mA. The total may never be more than 51 mA. It's recommended to keep below 3 mA per pin as much as possible, which we've done for the projects in this book.

With a 1 A power supply you can take about 250 mA (model B) to 300 mA (model A) from the +5 V pin. In practice this will also be dependent on the current consumption of other devices that are connected to the Raspberry Pi, such as a keyboard and suchlike.

We use the 3.3 V connection on the header for all projects in this book. The reason for this is that all of the Raspberry Pi operates at 3.3 V so all pins output (and expect) 3.3 V. There is also a +5 V connection on the header, but the other pins cannot tolerate 5 V. In other words, the designers of the Raspberry Pi haven't taken into account that somebody would put 5 V onto these pins. This doesn't mean that the Raspberry Pi will immediately break if you do put 5 V on them; it can work for a while, but eventually the Raspberry Pi will give up the ghost. [28]

5.2 LED

The maximum current that the Raspberry Pi can source from the 3.3 V connection is 51 mA, and that is for all pins! For this reason we use an LED that requires only 2 mA and still has an acceptable brightness of 8 mcd. [29]

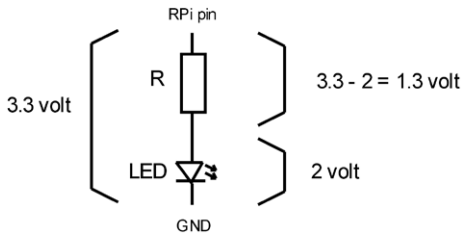


Figure 5.2. Working out the current limiting resistor.

This LED will obviously need a current limiting resistor, which can be easily calculated. According to the datasheet this LED has a forward voltage drop of 2 V. This means that there will be a voltage of 3.3 - 2.0 = 1.3 V across the current limiting resistor. Since the current should be 2 mA we can calculate the required resistor value using Ohm's Law:

27 Don't just pull the plug to turn the Raspberry Pi off, but stop it via the menu, or via LXterminal with the command "`sudo shutdown -h now`".

28 We will stick to this rule in the book, but you will find many projects on the Internet where the electronics are not connected properly. You should therefore be very careful before you build something off the Internet!

29 The Everlight 264-10UYD/S530-A3/T2. This is the LED from the hardware kit.

$V = I \times R$ or $I = V / R$ or $R = V / I$

Where: V = volts (in Volts)
I = current (in ampères)
R = resistance (in ohms)

We rewrite the formula as $R = V / I$ and fill in the details: $1.3/2\ 10^{-3} = 650\ \Omega$.

This value doesn't exist and the nearest standard values are 560 or 680 Ω. We choose 680 so that the current consumption will be slightly less than 2 mA (it will be 1.3/680 = 1.9 mA). The color code for a 680 Ω resistor is blue-grey-brown.

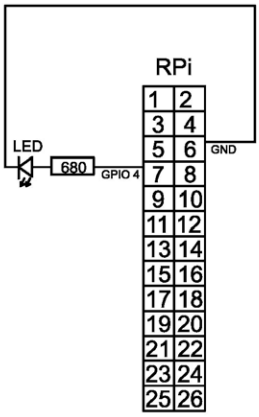


Figure 5.3. Connecting the LED to the header.

The connection takes place as follows: the strip with squares and numbers represents the header on the Raspberry Pi (because we will only use pins 1-26 we will leave out the rest for clarity). Pins 1 and 2 are at the edge of the Raspberry Pi, as can also be seen in the photo in figure 37. The resistor is connected to pin 7 and to the LED. This in turn is connected to pin 6, which is the Ground. In the appendix you can find a pin diagram with all connections and their names.

Figure 5.4. Anode and cathode of the LED.

The LED has a positive and negative connection (the positive is called the anode and the negative is called the cathode). When you look carefully inside the LED, you can see that one of the pins is much wider than the other. This is the pin that has to be connected to the negative. Often this side also has a slightly shorter connection lead. In the circuit diagram

the symbol for the LED looks like an arrow with a line at the end. A useful rule of thumb is that the arrow always points to the negative.

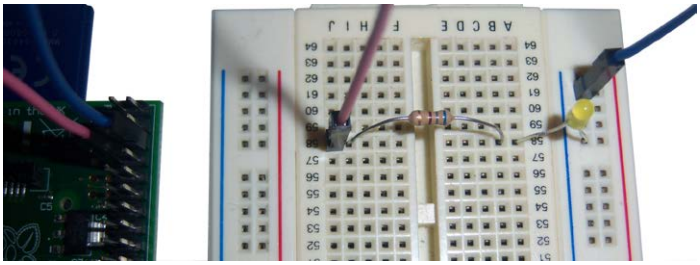


Figure 5.5. The setup on the breadboard.

Carefully copy this circuit and use the photo to confirm it is correct. All of the photos in this book were made of working setups. When the LED has been connected, you can power up the Raspberry Pi. Open a terminal or an SSH connection. [30]

The LED is connected to header pin 7, which is called GPIO 4. You are absolutely right if you find this confusing. In Appendix B is an overview of the header pins along with their names and functions. It may be a good idea to mark this page with a paperclip, as you will be needing the header overview quite often.

Let's start by turning the LED on and off to verify that we have connected it correctly. Start by opening Python using the command `python3`. The following text will appear in the terminal:

```
pi@raspberrypi:~ $ python3
Python 3.7.3 (default, Apr 3 2019, 05:39:12)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

If you are prompted with an error at this point (or when trying out any other example), please first verify that you didn't make a type error. This book is supplemented by a specially compiled SD card, to be acquired separately. Only with this SD card we can guarantee that commands and example projects work free of errors.

Now we can load the library that governs the communication between Python and the hardware of the Raspberry Pi. We don't load the full library, just the command for turning an LED on and off.

```
From gpiozero import LED
```

30 If you don't know how to do this, you should first read section 2.1 (LXterminal) or section 3.1.6 and 3.1.8 (SSH).

Next we proceed to connect the LED to GPIO4:

```
led = LED(4)
```

This command is in fact an object (refer to section 4.3 for the explanation of an object), which means it has attributes. For the LED object these are the attributes you can use:

LED object attribute	Explanation
on()	Switch LED on
off()	Switch LED off
toggle()	Switch LED on if off or off if on
blink()	Blink LED (as a thread)[1]
[1] We'll get back to this later.	

Now we can switch on the LED:

```
led.on()
```

If the LED doesn't light up, verify its polarity and make sure it is connected to the correct pin (header pin 7).

When you are done watching its beautiful light you can switch it off:

```
led.off()
```

To halt Python we type

```
exit()
```

Now the LED is switched off, Python is halted and the following should be on your terminal screen:

```
pi@raspberrypi:~ $ python3
Python 3.7.3 (default, Apr 3 2019, 05:39:12)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from gpiozero import LED
>>> led = LED(4)
>>> led.on()
>>> led.off()
>>> exit()
pi@raspberrypi:~ $
```

5.3 Flashing LED

If you input the switching commands alternately you will have created a manually operated flashing light. To save on typing we can also write a Python program to run these commands. We want flashing light to keep going forever (or in any case until we stop the program or turn off the Raspberry Pi), so we use the `while True` construct.

Type in the following program, and save it as flashled.py in a directory of your choice:

```
from gpiozero import LED
from time import sleep

led = LED(4)

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

Start the program with:

`Python3 flashled.py`, or start it from Thonny by clicking the Run button.

You can stop the program using Ctrl-C.

All programs presented in this book are available on the specially compiled SD card, so you don't need to retype them (although it is a good way to learn Python). You will find them in the folder "book".

The programs are arranged per chapter and within that per project. For example:

```
book  04
      05 5.03
         5.04
         5.05
         5.06
         5.07
        etcetera
```

Note that there are no folders for sections without projects or examples. For example in section 5.2 there is no program, so you won't find a folder "5.2".

In section 4.1 we explained how to navigate though folders in Linux. In addition, it's also possible to jump right into a specific folder with (note the space between "cd" and "~"):

```
cd ~/book/05/5.03
```

You can now start the program by typing

`Python3 flashled.py`, or start it from Thonny by clicking the Run button.

As stated in the table before the LED object also has the attribute `toggle`. If the LED was switched on it will be switched off and vice versa after giving this command. To create a blinking LED we can also make use of this command instead ("flashled1.py" in the folder):

```
from gpiozero import LED
from time import sleep

led = LED(4)
led.on()

while True:
  led.toggle()
  sleep()
```

Instead of having a flashing LED we could ask the user if the LED should be on or off. It isn't really much of a program because when the LED is on the only thing you can do is to turn it off. However, it does show you how you can ask the user a question and use the answer to drive a pin.

```
from gpiozero import LED

led = LED(4)

while True:
  value=input("Enter on or off to set the LED, or q to quit: ")
  if value=='on':
    led.on()
  if value=='off':
    led.off()
  if value=='q':
    break
```

Start the program by typing `python3 askled.py`, or start it from Thonny by clicking the Run button. This time you can stop the program by entering q instead of on or off. The program then breaks out of the While loop and since there aren't any instructions following the loop, the program stops.

5.4 Alternating Flashing LED

Now that we've managed to get a single LED to flash, it's not much of a challenge to get two LEDs to flash alternately. It's just a matter of connecting a second LED and adding it to the program. The interesting aspect of this project is that we control the LEDs in a slightly different way, using the not operator. This actually makes the program shorter, even though we've added an extra LED.

There are two types of operator. The most well known are the arithmetic operators such as addition, subtraction, etc. These operators act on numbers. The other type of operator is the logical operator. A logical operator works with only two values: true and false. A truth table is often used to show what a logical operator does. In here you can see the input and the output after the logical operator has acted on it.

input	output (not input)
true	false
false	true

Table 5.1. NOT truth table.

The way in which not, and also true and false, are written can differ a lot. The words not, true and false are often written in capitals in the text, and nearly as often in lower case. False and true have to start with a capital letter in Python (so False and True), but not doesn't require a capital letter. In Bash, nothing may start with a capital letter. In this book we write these words in lower case in the text, but in programs we follow the proper conventions for the particular programming language (the latter is necessary, otherwise the program wouldn't work).

Now back to our project. We connect the extra LED to pin 11, which is GPIO number 17. We therefore need to connect one extra cable to the Raspberry Pi, since the GND is already connected.

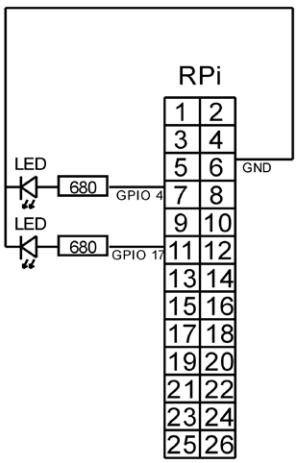


Figure 5.6. Connection of the LEDs to the header.

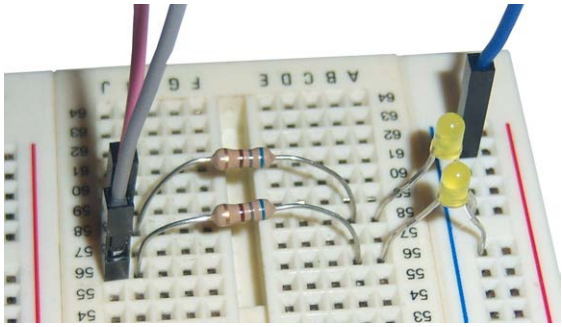


Figure 5.7. The setup on the breadboard.

There is still another issue we need to solve. Until now we have been using the LED object, which is part of the GPIOzero library. This time we would like to do something the creators of this library hadn't thought of, so we can not use the LED object. Instead we are going to use a standard output object called DigitalOutputDevice. With this we can directly access a pin with the limitation that we can only turn it on (or "high") or off ("low"). But in this case that's enough. We will see these standard devices more often in a later stage.

Type in the following program and save it as flashled2.py. Start it with `python3 flashled2.py`, or start it from Thonny by clicking the Run button.

```
from gpiozero import DigitalOutputDevice
from time import sleep

led1 = DigitalOutputDevice(4)
led2 = DigitalOutputDevice(17)

pos=False

while True:
    led1.value=pos
    led2.value=not pos
    pos = not pos
    sleep(1)
```

We've used an extra variable called pos, which is set to False. The first time round the loop, `led1` (GPIO 4) is given the value of `pos`, which at that time is False. That LED is therefore off. The LED on GPIO 17 is given the value of `not pos`, which is not False, and that is True. That LED is therefore on. The value of `pos` is then made equal to `not pos`, in other words, the opposite. `pos` was False and now becomes True. The program then waits one second and repeats the loop. Since the value of `pos` is now the opposite, the LED that was off now turns on, and the LED that was on now turns off.

You can see that this is a neat and compact way of programming, which is therefore often used by experienced programmers.

5.5 Timer LED with Window

In this project we make a timer LED: from the Raspberry Pi we turn on an LED after a preset time. This time the program is graphical, with a window and a button. We start with the standard graphical template as described in section 4.3. If you haven't read that section yet, you should do that now, otherwise it will be very difficult to follow this project.

With a non-graphical program you can use the `sleep` function. You can also use this in a graphical program, but then the program won't do anything while the function is busy, and that means it won't update the screen either. This also means the button stays pushed down. That may look neat, but we don't want that in this project. Instead of `sleep`, we therefore use a timer to keep track of how much time has passed. We make this timer execute a function every 10 milliseconds. This timer is an object, so it must be defined in the object section of the graphical template. You must also declare which function is bound to the timer:

```
# timer object
self.timer = wx.Timer(self)
self.Bind(wx.EVT_TIMER, self.OnTimer, self.timer)
```

In the next step we define the pins. We also want to define the io object here, but use it elsewhere in the program. Since io is not a standard part of wxPython, this won't work unless we declare it explicitly with the `global` statement. We first declare `led` as a global object, and then in the next line we define what exactly this object is. We can do this because we imported LED at the beginning of the program (with `from gpiozero import LED`).

```
# declare pins
global led
led = LED(4)
```

The next step is the definition of the button that's going to start the timer, and the associated function that will be called by this object.

```
# start the timer button
cbtn0 = wx.Button(self, label='Start', pos=(20, 60))
cbtn0.Bind(wx.EVT_BUTTON, self.OnStart)
```

And finally we create a text object. In here we put the instruction for the user to click on the button to start the timer. Once the timer is running we use the same object to display the time elapsed.

```
# static text
self.text = wx.StaticText(self, label='Press button to
   start',pos=(20,30))
```

The definition should really be on a single line, but there isn't enough room on the page. We have therefore continued on the next line and indented it to show that it's a continuation of the line above. In the program we obviously don't do this: it's entered neatly on a single line.

This completes the definitions of the objects. In the objects we declared that we want to use two new functions, and these have to be defined in the functions section of the graphical template. We start with the function that is called by the button. This function sets the counter to zero, starts the timer with a 10 ms interval, and turns on the LED.

```
def OnStart(self, e):
  # start the timer, event every 10 ms
  self.count=0
  self.text.SetLabel('0')
  self.timer.Start(10)
  led.on()
```

The next function is the one called by the timer every 10 ms. In this function we show the state of the timer on the screen, so you can see how much time has elapsed. We also check if the required time (in this case 2 seconds) has elapsed. If that is the case, we stop the timer and turn off the LED. Since this function is called every 10 ms, it takes 200 calls before 2 seconds have passed, so we count to 200.

```
def OnTimer(self, e):
  # increment counter every 10 ms
  self.count = self.count + 1
  # and show it
  self.text.SetLabel('Elapsed time ' + str(self.count*10) + 'ms')
  if self.count==200:
    # done stop de timer
    self.timer.Stop()
    led.off()
    self.text.SetLabel('Press button to start')
```

You'll notice that this is much more complex than a standard program, although it isn't that difficult if you fill in the graphical template one step at a time. We just change the title of the window to Timer LED, and the program is ready. The complete program now looks as follows:

```
import wx
from gpiozero import LED

class Example(wx.Frame):

  def __init__(self, parent):
    wx.Frame.__init__(self,parent)
```

```
    #self.io = wiringpi.GPIO(wiringpi.GPIO.WPI_MODE_SYS)
    self.InitUI()

  def InitUI(self):
    self.SetBackgroundColour('white')
    self.SetSize((250, 200))
    self.SetTitle('Timer LED')
    self.Centre()
    self.Show(True)

    # your own objects

    # timer object
    self.timer = wx.Timer(self)
    self.Bind(wx.EVT_TIMER, self.OnTimer, self.timer)

    # declare pins
    global led
    led = LED(4)

    # this is button 0
    cbtn0 = wx.Button(self, label='Start', pos=(20, 60))
    # function called when you click on the button
    cbtn0.Bind(wx.EVT_BUTTON, self.OnStart)

    # static text
    self.text = wx.StaticText(self, label='Press button
    to start',pos=(20,30))

  # your own functions

  def OnStart(self, e):
    # start the timer, event every 10 ms
    self.count=0
    self.text.SetLabel('0')
    self.timer.Start(10)
    led.on()

  def OnTimer(self, e):
    # increment counter every 10 ms
    self.count = self.count + 1
    # and show it
    self.text.SetLabel('Elapsed time ' + str(self.count*10) + ' ms')
    if self.count==200:
      # done stop the timer
      self.timer.Stop()
```

```
            led.off()
            self.text.SetLabel('Press button to start')

def main():
  ex = wx.App()
  Example(None)
  ex.MainLoop()

if __name__ == '__main__':
  main()
```

Start the program with `python3 timerled.pyw`, or start it from Thonny by clicking the Run button. If you want to use this program via an SSH connection from a Windows PC, you must first start VNC server and VNC Viewer or Xming before you can run this graphical application.[31]

When you click on the button, the LED turns on and the counter starts. After two seconds the LED goes off and the "Press button to start" comes back. You can stop the program by clicking on the cross at the top-right of the window.

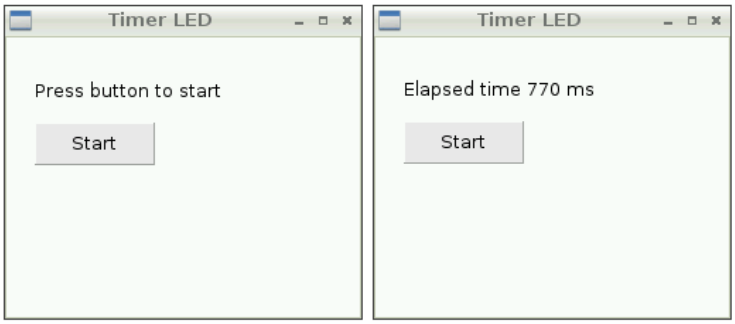


Figure 5.8. The program in action.

5.6 Switch

A pin that is used as an input may never "float", meaning it should always be connected to something, either the positive or ground. A switch would normally be connected as follows:

31 Refer to section 3.2 for more information on SSH and Xming

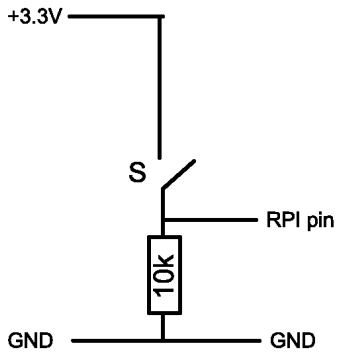


Figure 5.9. Standard switch connection.

When the switch is open, the Raspberry Pi pin is connected to ground via a 10 k resistor, so the pin is low. When the switch is closed, the pin is directly connected to +3.3 V and is therefore high. In this state a current will flow through the resistor, but this is negligible:

$$I = V / R = 3.3 / 10000 = 0.33 \text{ mA}$$

But what happens if the pin isn't defined as an input, but as an output and then set low? This is unlikely to happen in a microcontroller since the (single) program will immediately set the state of the pins to that intended by the programmer. However, on the Raspberry Pi this is definitely possible. You can have several programs running at once, which all use some of the pins. It may well be possible that your program defines the pin as an input, but that another program then defines it as an output and sets it low. It's a small mistake but it can have unpleasant consequences.

As long as the switch is open everything is fine, since the low pin is connected to ground. But as soon as you turn on the switch, the low pin will be connected to +3.3 V, causing a short. That's the end of the Raspberry Pi. We add an extra 1 k resistor to prevent this, as shown in the circuit below:

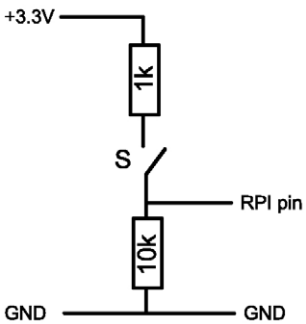


Figure 5.10. Protected switch connection for the Raspberry Pi.

During normal use this resistor has little effect. When the switch is open, it doesn't do anything; the Raspberry Pi pin is connected to ground via the 10 k resistor. With the switch closed, a potential divider is created, with the voltage at the pin:

$$V_{RPi\ pin} = 10 / (1+10) \times 3.3 = 3.0\ V.$$

This is high enough for the pin to be considered as high so this time the resistor doesn't have any effect either. The switch still operates the way we intended.

The advantage becomes obvious when the pin is accidentally defined as an output. It's no problem if the pin is high since the resistor to ground is 10 kΩ. However, when the pin is low it would normally be shorted to the positive when somebody presses on the switch. The 1 kΩ resistor now prevents any damage, since it limits the current to:

$$I_{short} = V / R = 3.3 / 1000 = 3.3\ mA$$

This is low enough for it not to cause any damage to the Raspberry Pi. Your circuit won't work (since the pin is now an output) but your Raspberry Pi will survive. This safe method of connecting a switch will be used throughout this book.

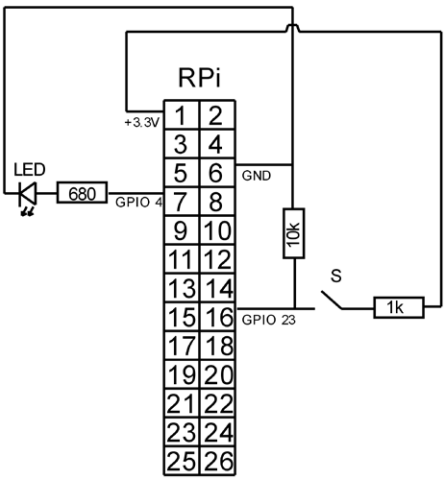


Figure 5.11. Connection of an LED and switch to the header.

The complete circuit consists of an LED and a switch. The two lines that cross with a kink aren't connected to each other. The kink indicates that one wire goes over the other. This applies to all circuit diagrams in this book.

Resistor color codes: 1 kΩ = brown-black-red
10 kΩ = brown-black-orange

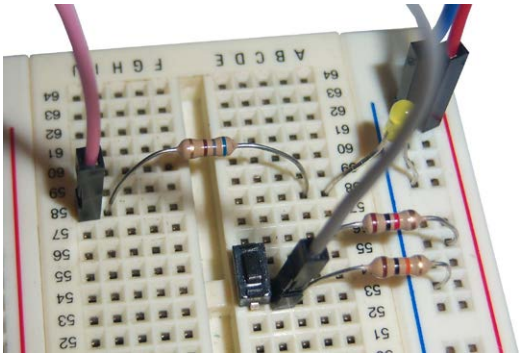


Figure 5.12. Setup on the breadboard.

The switch is connected to header pin 16, which is called GPIO 23. Just as for the LED, we have to let the Raspberry Pi know what we want to do with the pin that's connected to the switch.

There is a button object that would be ideally suited if it were not for the button reacting exactly opposite of what we want: "on" is in fact "low" (or "off"). At the end of this project we will show you how to use the button object despite this issue, but for now we will use the InputDevice object.

The first step is to start Python and import the `InputDevice`. From the command line enter:

```
Python3
from gpiozero import InputDevice
```

Next we connect our switch to GPIO 23 (pin 16 on the header):

```
switch = InputDevice(23)
```

Now we can check the state of the switch using:

```
Print(switch.value)
```

If the switch isn't pressed the result will be False, and when it is pressed the result will be True. The Raspberry Pi only looks at the time you issue the command, so if you want to see a True you have to issue the command when you're pressing the switch. Leave Python using the command

```
exit()
```

This is how your terminal should look like when the switch is not pressed at first (False) and pressed later (True):

```
pi@raspberrypi:~ $ python3
Python 3.7.3 (default, Apr 3 2019, 05:39:12)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from gpiozero import InputDevice
>>> switch = InputDevice(23)
>>> print(switch.value)
False
>>> print(switch.value)
True
>>> exit()
```

With this knowledge we now easily create a program that watches the state of the switch continuously. Create the following program and save it as `switch.py`:

```
from gpiozero import InputDevice
from time import sleep

switch = InputDevice(23)

while True:
    sleep(0.5)
    if switch.value == True;
      print("switch closed")
    else:
      print("switch open")
```

Start the program with

`Python3 switch.py`, or start it from Thonny by clicking the Run button.

You can stop the program by typing Ctrl-C. The program checks the state of the switch every 0.5 seconds and puts an appropriate message on the screen. When you run this program, you'll notice that the 0.5 seconds doesn't always seem to be right. The Raspberry Pi also runs many other programs apart from this one and in the background it will update the screen, process mouse movements, watch the Internet connection, etc. Therefore there is no guarantee that the Raspberry Pi will adhere to the 0.5 seconds!

Now that we know what the state of the switch is, we can use it to turn the LED on and off. This could also be done with a simple piece of wire, so we'll make it work the other way round: the LED will normally be on, but turns off when you press the switch.

Before you read any further, think how you would implement this yourself. It is in fact a combination of the previous program and the previous project. My solution is as follows:

```
from gpiozero import InputDevice, LED
from time import sleep
switch = InputDevice(23)
led = LED(4)

while True:
    if switch.value == True:
      led.off()
    else:
      led.on()
```

The program is called `switchled.py`, so to start it, type

`Python3 switchled.py`, or start it from Thonny by clicking the Run button.

The LED will light up instantly and will be turned off as long as you keep the button pressed. To stop the program use Ctrl-C.

We will not be using the button object in this book because it works inversely, but we've shown you now how this works so you can read programs of others [32].

Properties of button object	**Explanation**
is_pressed	True if switch is NOT pressed
wait_for_press	Wait until the switch is released
wait_for_release	Wait until the switch is pressed
blink()	Blink (as a thread)*
when_pressed	When the switch is open
when_released	When the switch is closed
* We will return to this property later on	

With this it is possible to read out a switch using the following code:

```
from gpiozero import Button
from time import sleep

switch = Button(23)

while True:
  sleep(0.5)
  if switch.is_pressed:
    print("Switch open")
```

32 Of course the button object works correctly when you design a circuit in which the pin is normally high and is turned low when you press the switch. Technically this isn't an issue, but it feels unlogical and is thus a source of mistakes and errors. In this book we will use switches the "normal" way, i.e. pressed is "on".

```
  else:
    print("Switch closed")
```

5.7 Time Switch

It is very easy to modify the previous program and turn it into a time switch. First you swap the LED on/off instructions so the LED turns on when you press the switch. Then you remove the `else` instruction and insert a `sleep` instruction below `led.on()`. With `sleep(2)` the LED will stay on for two seconds when you press the switch.

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

switch = InputDevice(23)
led = LED(4)

while True:
    if switch.value == True:
      led.on()
      sleep(2)
      led.off()
```

Start the program with

`Python3 timerled.py`, or start it from Thonny by clicking the Run button.

When you press the switch, the LED will stay on for two seconds. You can stop the program with Ctrl-C.

5.8 Toggling Switch

In this project we make a switch that turns on the LED the first time you press it and turns the LED off the second time you press it. This seems very straightforward: we start with the program from the switch project, and add a variable and the not operator to change the state of the LED. Something along these lines, for example:

```
while True:
  if switch.value == True:
    pos = not pos
    led.value = pos
```

This program does work in a sense, but you won't be able to see it. Before you get a chance to remove your finger from the switch, the LED will have turned on and off thousands of times. The state in which the LED ends up when you've actually removed your finger from the switch is a matter of pure chance. The program should therefore wait for you to remove your finger from the switch. We can use the useful while loop for this:

```
    while switch.value:
      pass
```

`Pass` is an instruction that doesn't do anything. In theory we could therefore leave it out, but Python requires that you have at least one instruction in a loop. The effect is that the program stays in this loop until you let go of the switch. The only other problem is contact bounce.

At the point that you press the switch, the internal contact moves to the other contact and slides to its final position. While it is sliding over the other contact, it will sometimes make a connection and sometimes it won't. This happens so quickly that you don't notice this, but the Raspberry Pi will certainly see this and assumes them to be a series of switch presses. A very short pause of 10 ms (0.01 seconds) gets round this problem.

Because we want to use the not operator, we can not use the LED object. Instead we will use de DigitalOutputDevice object like we did in section 5.4.

The complete program is then as follows:

```
from gpiozero import DigitalOutputDevice, InputDevice
from time import sleep

led = DigitalOutputDevice(4)
switch = InputDevice(23)
pos=False

while True:
    if switch.value == True:
      pos = not pos
led.value=pos
      while switch.value:
        pass
      sleep(0.01)
```

You start the program with `python twostate.py`, or start it from Thonny by clicking the Run button. The state of the LED will change every time that you press the switch. You can stop the program with Ctrl-C.

5.9 Switch State in a Window

It's time for another graphical project: we make a program that shows the state of the switch in a window. At first sight this seems like a simple task: we just have a loop where we look at the switch state, along these lines:

```
while True:
    if switch.value:
    # put something on the screen
```

But this won't work. A graphical program works with objects, and if we do this in one object, we'll find that the program won't deal with the other objects. This means that the text on the screen won't be updated either.[33]

To solve this we use a timer to look at the state of the switch every 10 milliseconds. This timer is an object and therefore needs to be defined in the objects section of the graphical template. The program doesn't have a start button, so we start the timer immediately .

```
# timer object
self.timer = wx.Timer(self)
self.Bind(wx.EVT_TIMER, self.OnTimer, self.timer)
# start the timer
self.timer.Start(10)
```

In the next step we define the pins. We also want to define the io object here, but use it elsewhere in the program. Since `switch` is not a standard part of wxPython, this won't work unless we declare it explicitly with the `global` statement. We first declare `switch` as a global object, and then in the next line we define what exactly this object is (we can do this because we imported gpiozero at the beginning of the program).

```
# declare pins
global switch
switch = InputDevice(23)
```

And finally we create a text object, where we show the state of the switch. Since we don't yet know what the state of the switch is, we leave the label blank.

```
# static text
self.text = wx.StaticText(self, label='',pos=(20,30))
```

The complete program now looks as follows:

```
import wx
from gpiozero import InputDevice

class Example(wx.Frame):

  def __init__(self, parent):
    wx.Frame.__init__(self,parent)
    #self.io = wiringpi.GPIO(wiringpi.GPIO.WPI_MODE_SYS)
    self.InitUI()

  def InitUI(self):
    self.SetBackgroundColour('white')
```

33 The program won't have time to look at mouse clicks in your window, so you can't even stop it by clicking on the cross at the top-right!

```
        self.SetSize((250, 200))
        self.SetTitle('Read Switch - Timer')
        self.Centre()
        self.Show(True)

        # your own objects

        # timer object
        self.timer = wx.Timer(self)
        # function called by the timer
        self.Bind(wx.EVT_TIMER, self.OnTimer, self.timer)
        # start the timer
        self.timer.Start(10)

        # declare pins
        global switch
        switch = InputDevice(23)

        # static text
        self.text = wx.StaticText(self, label='',pos=(20,30))

      # your own functions

      def OnTimer(self, e):
        if switch.value:
          self.text.SetLabel('Switch closed')
        else:
          self.text.SetLabel('Switch open')

    def main():
      ex = wx.App()
      Example(None)
      ex.MainLoop()

    if __name__ == '__main__':
      main()
```

Start this program with `python3 readswitch1.pyw`. The program will show that the switch is open. When you press the switch, the program announces that the switch is closed. Since the program doesn't check the switch continuously, but only every 10 milliseconds, there will in fact be a short delay before the display is updated.[34]

34 This is quite convenient, since the switch will certainly suffer from bouncing. This 10 ms therefore acts as a perfect debouncing time. There is nothing stopping you from calling this routine more often than every 10 ms, but that doesn't make sense with a switch. In general, it's best to be "economical" with processor time: if you don't need it, you shouldn't use it. After all, the Raspberry Pi has many more tasks to perform than running your program.

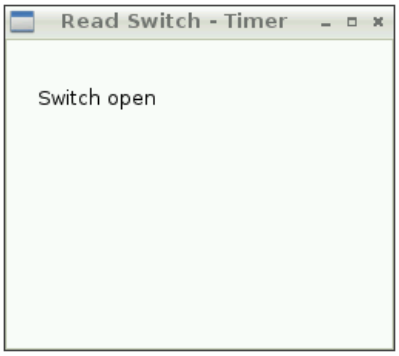


Figure 5.13. Display of the timer-driven program.

5.10 A Button with Sound

In this program we get the Raspberry Pi to play back a sound file when we press a GPIO button. This program is based on a sketch from Monty Python called "The machine that goes ping". [35] Since I don't have the rights to use "ping", but I do for "blonde", we use this sound clip.[36] You can of course use a different sound file as long as it's an mp3 file.

We use a switch on GPIO 23 (pin 16 on the header) for the button, taken from project 5.6. The SD card has a copy of the program mpg321 on it, which is used to play back mp3 files. The program has a wide range of features (you can see them with the command `man mpg321`), but we'll only use it to play back the file. Since we start the program from Python, we use the `os.system` command, just as with the exporting of pins.

In this program we let the user specify which file to play back, as a parameter on the command line. Command line parameters are items of data that the user can pass on to the program when it starts. Python puts these items into a list for you (sys.argv), so you can easily use them within your program. A simple example of this is in the following program:

```
import sys

c=0
for arg in sys.argv:
   print(c, arg)
   c+=1
```

Save the program as parmtest.py, and start it with:

```
Python3 parmtest.py a b c d
```

35 "The machine that goes ping" is in the film "The Meaning of Life". You can see the clip on YouTube: https://youtu.be/NcHdF1eHhgc
36 This goes: "Oh my god, I'm like, totally blonde", which is amusing since I'm blond.

The result will be:

```
0 parmtest.py
1 a
2 b
3 c
4 d
```

You can see that parameter 0 is the name of the program itself. The list continues with all the entered parameters. In this case the user enters one parameter, which is the name of the music file, so we only need to read index 1 of the list. We use `try/except` for this because it's always possible that the user forgot to add the parameter for the music file. In a proper program, you should offer a better error message than "No music file", but it is clear enough for personal use.

```
From gpiozero import InputDevice
from time import sleep
import os
import sys

switch = InputDevice(23)

try:
   musicfile= sys.argv[1]
except:
   print("No music file")
   exit()
while True:
   if switch.value == True:
     os.system("mpg321 "+musicfile)
```

The sound file blonde.mp3 is stored on the SD card.[37] Start the program with:

```
python3 soundswitch.py blonde.mp3
```

The sound file will be played back every time you press the button.[38] You can set the volume with the command:

```
amixer set PCM 50%
```

where the percentage is used to set the volume level (0% is no sound, 100% is maximum volume). Don't forget to include the % symbol in the command, otherwise it won't work!

37 The sound file is stored in the directory for this project on the Raspberry Pi.

38 When you use a loudspeaker or earphones that are connected directly to the Raspberry Pi, you will hear a click before and after the sound file. This is a known problem with the Raspberry Pi. You can minimize the effect by turning the volume up to maximum, since the volume of the click doesn't change. You won't hear these clicks if you're using an HDMI monitor, or the Ligawo HDMI-VGA converter.

When you don't hear any sound, you can modify the output with the command:[39]

```
sudo amixer cset numid=3 x
```

Where x is the type of output:

0 = automatic
1 = audio socket on the Raspberry Pi
2 = via HDMI

You can also set the volume level, and several other parameters, using the program alsamixer.

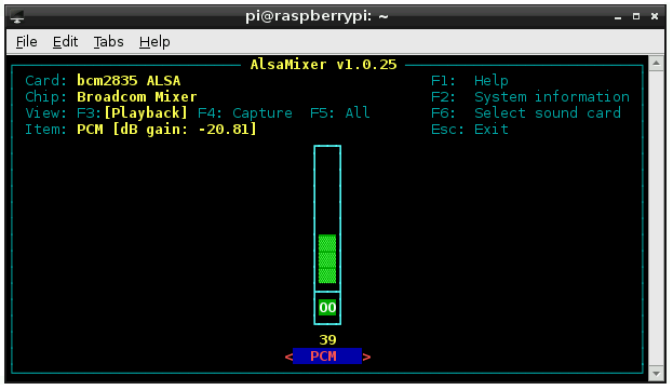


Figure 5.14. The alsamixer program.

5.11 Pin Communications

It's possible for two programs to communicate with each other via a file. An example of this can be found in project 11.3.2. However, the Raspberry Pi has another very neat way of doing this: via two GPIO pins.

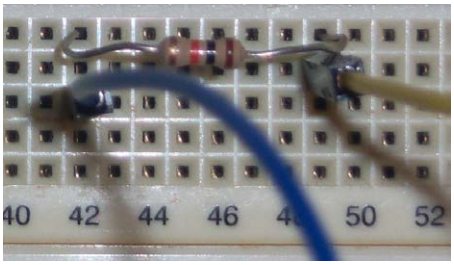


Figure 5.15. Connections on the breadboard.

39 If you're using the Ligawo HDMI-VGA converter, but you don't get any sound from it, you should follow the instructions in section 1.5.

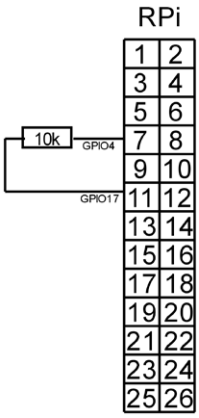


Figure 5.16. Connecting the pins on the header.

The two pins are connected together via a 10 k resistor (brown-black-orange). Strictly speaking, this resistor isn't required. As long as one pin is set as an output and the other as an input everything is fine. But if you accidentally set both pins as outputs, with one set to 1 and the other to 0, you create a short. The resistor has been added to prevent any damage to the Raspberry Pi under those circumstances. The maximum current is now:

I = V / R = 3.3 / 10,000 = 0.33 mA

The communications in this example operate as follows: One program sets the pin high and low alternately. The other program looks at the state of the pin and displays it on the screen. It's like a virtual flashing light really.

Program 1 (pinset.py)

```
from gpiozero import LED
from time import sleep

led = LED(4)

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

Program 2 (pinread.py)

```
from gpiozero import InputDevice
from time import sleep

switch = InputDevice(17)
```

```
while True:
    sleep(0.1)
    if switch.value == True:
      print("pin set")
    else:
      print("pin released")
```

The output from program 2:

```
pi@raspberrypi ~/experiments $ python3 pinread.py
pin set
pin set
pin set
pin set
pin released
pin released
pin released
pin released
pin released
pin released
pin released
pin released
pin released
pin released
pin set
pin set
pin set
```

There is not really much point to this program, but it shows you how you can get two programs to work cooperatively. They know via the pins what the other program is up to. Obviously you don't need to limit yourself to two pins.

You can also connect an LED to the output pin (via a current limiting resistor, of course) so the user can see what the state of the program is.

Chapter 6 • More Power

The amount of current that the Raspberry Pi can supply at 3.3 V is a maximum of 51 mA, and that is for all the pins together. It is recommended that a single GPIO pin should not be allowed to supply more than 16 mA. The 5 V pin can supply somewhat more. This pin can, depending on the power supply and the consumption of the Raspberry Pi, supply between 250 and 300 mA. That sounds better, but it limits you in the number of peripherals you can connect, which also require power. Some examples are a WiFi adapter or an optical mouse.

In this chapter we will have a look at several ways in which the Raspberry Pi can supply more power, as well as a higher voltage. Three projects will be considered as shown below:

Project	Implementation
TD62783	8 connections, maximum 500 mA / 50 V
ULN2003	7 connections, maximum 500 mA / 50 V (different voltages on each pin)
IRF740	1 connection, maximum 10 A / 400 V

In these projects we use an external power supply for the power devices. It is useful if you have an adjustable power supply, as it makes it easy to connect the various devices from the projects, which all need different voltages. And no doubt you'd like to experiment using other voltages. In Appendix B, you'll find a simple circuit for an adjustable power supply with a voltage from 1.24 V to 13 V at 1.5 A. This power supply has been used for all experiments in this book. It's always best to turn on the external power supply last, and to turn it off first when you want to turn off the Raspberry Pi.

6.1 TD62783 8-Channel High Source Driver [40]

As shown in Figure 6.1, the TD62783 is an 8-channel high source driver IC, which has 8 outputs rated at 500 mA, and can cope with a maximum voltage of 50 V.

Recommended Operating Conditions (Ta = −40 to 80°C)

Characteristics		Symbol	Test Condition		Min	Typ.	Max	Unit
Supply voltage		V_{CC}	—		—	—	50	V
Output current	AFG	I_{OUT}	Ta = 85°C, T_j = 120°C, T_{pw} = 25 ms	Duty = 10% 8 circuits	—	—	−260	mA/ch
				Duty = 50% 8 circuits	—	—	−59	
				Duty = 10% 8 circuits	—	—	−180	
				Duty = 50% 8 circuits	—	—	−38	

Figure 6.1. Output current according to the datasheet for the TD62783.

A single output can indeed source 500 mA, but not if several outputs are used at once. The above figure comes from the datasheet for the TD62783, and shows two values for

40 You can also use the UDN2981, which is rated at 80 V / 350 mA, but is otherwise identical. Note that the datasheet for the UDN2981 implies that this chip can also cope with 500 mA, but the graphs for the maximum current limit it to 350 mA (document Dwg. No. A-11,107B).

the maximum current when all 8 outputs are in use. At a PWM duty cycle of 10%, each channel can supply 260 mA (for a PDIP), and at a duty cycle of 50% this reduces to 59 mA per channel, which is 472 mA in total. The nice thing about this chip is that each output is provided with a clamp diode so you can connect motors directly to the chip.[41]

A positive signal at the input results in a positive signal at the output. The chip needs a TTL level of 5 V, but it considers 3.3 V as a logic 1. This means that an input on the TD62783 can be connected directly to the Raspberry Pi. The current drawn per input is about 0.13 mA, which is well below the maximum for the Raspberry Pi.

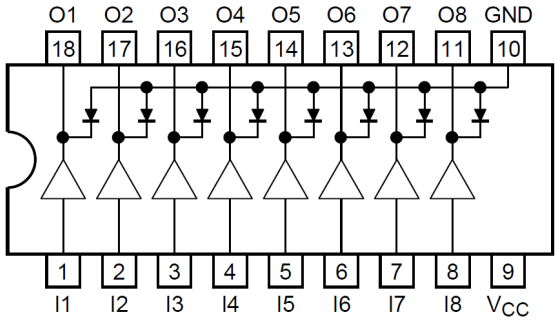


Figure 6.2. Pinout for the TD62783.

The pinout for the TD62783 is shown in Figure 6.2. All outputs have a common input voltage at pin 9 (V_{cc}). There is also a common ground at pin 10 (GND), which should be connected to the ground of the Raspberry Pi (GND, pin 6 on the header). The V_{cc} pin should obviously not be connected to the Raspberry Pi!

6.1.1 Light (6 V, 65 mA)

We start with a simple circuit where we connect a 6 V, 65 mA light. You can see that the TD62783 is connected to GPIO 23 (pin 16 on the header). The ground of the Raspberry Pi and the adjustable power supply are connected together. The +3.3 V and +6 V obviously aren't. As shown in Figure 6.3, the +3.3 V of the Raspberry Pi isn't used at all since the TD62783 is powered through the V_{cc} pin (9), which is connected to the adjustable power supply.

41 Other terms used for clamp diodes in datasheets are: flyback, snubber, suppression or catch diodes. We'll discuss later in this chapter what function a clamp diode has.

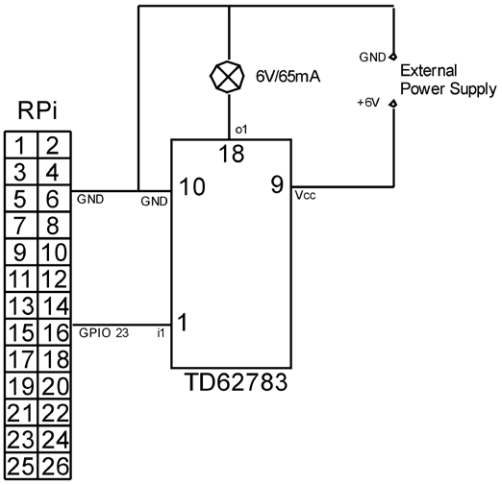


Figure 6.3. Connection of the TD62783 to the header.

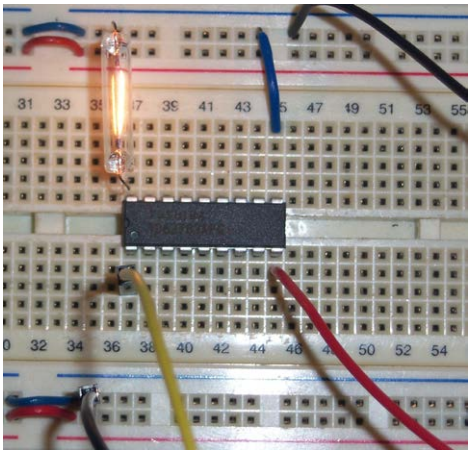


Figure 6.4. Setup on the breadboard.

The two wires shown in Figure 6.4 that leave the photo on the right go to the adjustable power supply.

We make the light flash using the following code:

```
from gpiozero import LED
from time import sleep

light = LED(23)

while True:
    light.on()
    sleep(1)
```

```
        light.off()
        sleep(1)
```

In project 7.2 we show you how to change the brightness of this light. You can start the program with `python3 td62783a.py`, and stop it with Ctrl-C.

6.1.2 Motor (5 V, 145 mA)

A motor (or a relay) contains a coil. The current through the coil persists, which means that when the voltage is removed, the current will still flow for a while. Since that current has nowhere to go because the power has been removed, the voltage will increase drastically, which could damage the Raspberry Pi or other components. To stop this happening, a diode is connected "the wrong way round" across the connections to the motor. This diode (which is called a snubber or clamp diode) normally doesn't conduct since it's connected the wrong way round, but because of this it will let the persisting current through. For a motor this problem is even greater than for a relay since the motor will continue to turn for a while after being switched off, acting like a type of dynamo.

The TD62783 already has these clamp diodes connected internally, so we can connect the motor directly to the chip.[42] In this project we use a Mitsumi 5 V motor as an example. This motor takes 15 mA when free-running and 145 mA when stalled.[43] You can use your own motor instead of this one, as long as the voltage and current are within the specifications of the TD62783. The circuit is identical to the previous project, only this time there is a motor instead of a light. Figure 6.5 shows the circuit built on a breadboard.

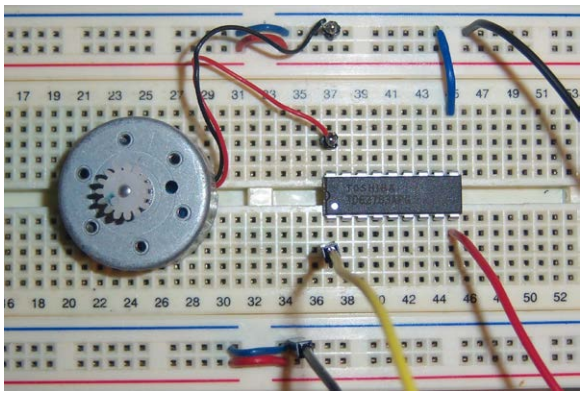


Figure 6.5. Setup with the electric motor.

We use the same program for the motor as we did for the light. In project 7.3 we show you how to change the speed of the motor.

42 This also applies to the UDN2981. If you use a different chip, you should consult its datasheet. If your chip doesn't have a clamp diode, you can add one yourself. As an example, you could use a 1N4007, which has a breakdown voltage of 1000 V and can cope with currents up to 1 A, with a surge current of 30 A. Since this diode normally shouldn't conduct, you have to connect it "the wrong way round"

43 Stalled means that the motor isn't turning while it has the full supply voltage across it. If you want to measure the stalled current yourself, you have to do this quickly because many motors only cool themselves while they're running. A stalled motor could overheat in time.

6.2 ULN2003 7 Open Darlington Arrays

The ULN2003 is a 7-channel open Darlington array that has 7 outputs and can cope with a maximum voltage of 50 V. A single output can deal with 500 mA, but not if you use more than one output at the same time. The graph in Figure 6.6, taken from the datasheet, and applies to the PDIP package (the type that fits a breadboard). The horizontal scale is the PWM duty cycle, where DC is equivalent to 100%, or full on. The vertical scale is for the maximum current consumption in mA. These are the peak ratings, so they're the absolute maximum. With one active output (an output that actually supplies a current), the maximum is 500 mA. With two active outputs, the maximum at full load (i.e. DC) is about 390 mA per output. When all seven outputs are in use, the maximum reduces to about 170 mA per output.

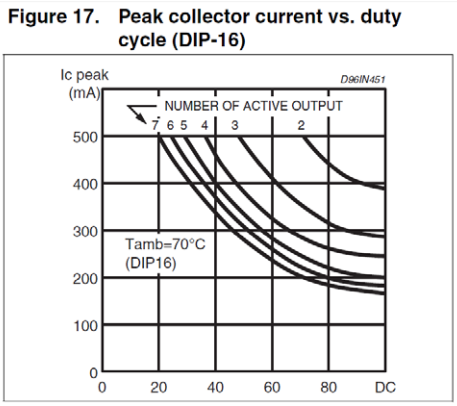


Figure 6.6. Output current versus duty cycle according to the datasheet.

The outputs of this chip can be connected in parallel to provide more power. If you connect all seven in parallel, you'll get 7 x 170 = 1190 mA available. I would have thought that under those conditions a good heatsink would be a necessity!

The chip has a common clamp diode (the datasheet calls this a common free wheeling diode). You can incorporate it into the circuit by connecting the positive of the adjustable power supply to pin 9. One limitation is that this diode can only operate when you're using a single external voltage. In all other cases you have to include a separate diode yourself. As an example, you could use a 1N4007, which has a breakdown voltage of 1000 V and can cope with currents up to 1 A, with a surge current of 30 A. Since this diode normally shouldn't conduct, you have to connect it "the wrong way round".

A positive signal at the input results in a negative signal at the output: the ULN2003 therefore switches to ground. The advantage of this is that you can connect several devices that all use different voltages, as we'll see in a moment.

The chip requires a TTL level of 5 V, but interprets 3.3 V as a logical 1. This means that an input of the ULN2003 can be connected directly to the Raspberry Pi. The current required per input is about 0.74 mA, well below the maximum for the Raspberry Pi. When this is still

too much in your application (when you draw a lot of current from the other pins), you can add a 10 kΩ resistor between the Raspberry Pi and the ULN2003. This reduces the current to 0.16 mA, but it does make the connection more sensitive to interference. Figure 6.7 shows the pinout for the ULN2003.

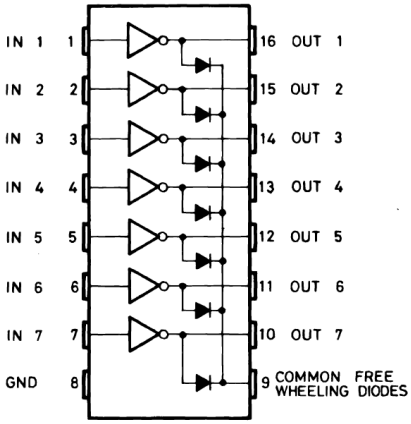


Figure 6.7. Pinout for the ULN2003.

You can see that the outputs have a common ground on pin 8 (GND), which should be connected to the ground of the Raspberry Pi (GND, pin 6 on the header). There is also a common free wheeling diode (clamp diode) connection on pin 9.

6.2.1 Fan (12 V, 150mA) and Light (6 V, 65 mA)

As a demonstration, we connect a 12 V fan with a nominal current consumption of 150 mA, and a 6 V light with a current consumption of 65 mA. Since we can only use the clamp diode connection when all outputs are connected to the same voltage, we have to use a separate clamp diode for the fan. We use a 1N4007 for this, which has a breakdown voltage of 1000 V and can cope with currents up to 1 A, with a surge current of 30 A. Since this diode normally shouldn't conduct, you have to connect it "the wrong way round". Take care when you connect it, otherwise you might short out the external power supply!. Figure 6.8 shows the circuit built on a breadboard. The circuit diagram is shown in Figure 6.9. The fan is connected to the external adjustable power supply, which is set to 12 V. We really need 6 V for the light, but it also works perfectly well at 5 V. For simplicity, we therefore use the 5 V connection from the Raspberry Pi.

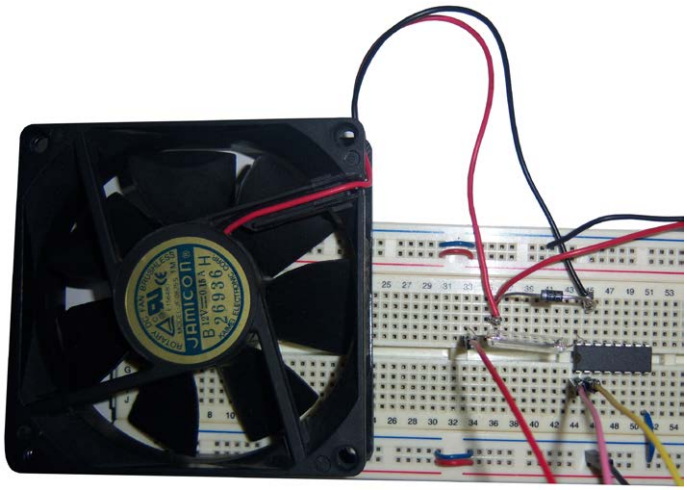


Figure 6.8. Setup with light and fan.

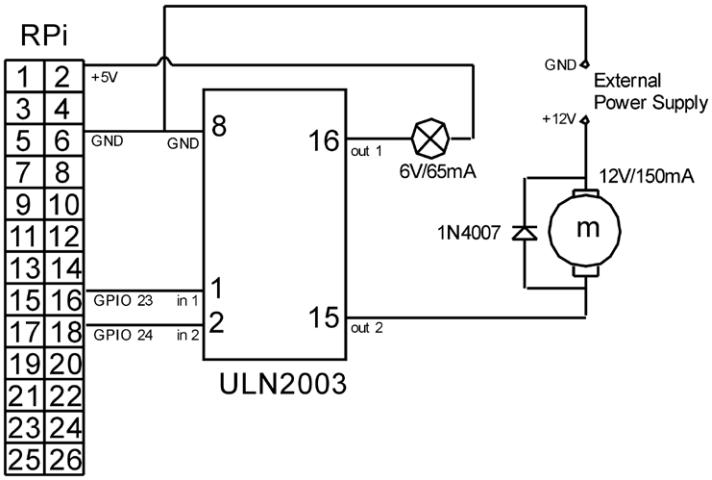


Figure 6.9. Controlling devices with different operating voltages.

We can control the light and the fan motor with the following program:

```
from gpiozero import LED
from time import sleep

out1 = LED(23)
out2 = LED(24)

while True:
value=input("Control the light with on1/off1, the fan with on2/off2, or use
q to quit: ")
    if value=='on1':
```

```
        out1.on()
      if value=='off1':
      out1.off()
      if value=='on2':
      out2.on()
      if value=='off2':
        out2.off()
    if value=='q':
      break
```

You can start the program with `python3 uln2003a.py`, and stop it by entering q.

6.3 IRF740 MOSFET

To switch really big currents, we have to use a MOSFET. A MOSFET is a Metal Oxide Semiconductor FET.[44] A MOSFET has three connections: the gate, drain and source. In a (MOS) FET the drain current is determined by the gate voltage. The source functions as a common ground for the gate and drain. The gate current is zero, which makes it very suitable for microcontrollers. Note that in the symbol for the MOSFET the arrow doesn't point to the negative; this is the only component that I know where this is the case.

6.3.1 Motor (5 V, 550 mA)

In this project we'll be using the IRF740. This MOSFET has a maximum rating of 10 A / 400 V. If you really want it to supply such a large current, you should mount it on a large heatsink. You should also bear in mind that 400 V is a deadly voltage and you should take all the necessary precautions if you work with such a high voltage. Figure 6.10 shows the drain current as a function of the gate voltage for the IRF740. [45]

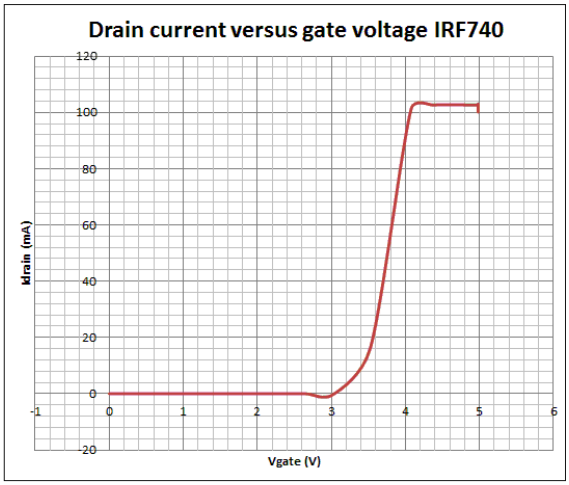


Figure 6.10. Drain current versus gate voltage for the IRF740.

44 FET is an acronym for Field Effect Transistor.

45 From the book "Super Fast PC Interfacing" which has a project for analyzing MOSFETs and displaying the results on a graph. The current in that book was set to a maximum of 100 mA, which is why the graph doesn't go any higher.

You can see that the MOSFET begins to conduct just above 3 V, and is in full conduction from about 4 V. If we include a margin of error, we would need at least 5 V, which the Raspberry Pi pin can't provide. For this reason we use a MOSFET driver, the MAX4420. This driver can make use of the supply voltage to the motor to turn on the MOSFET, and this voltage is high enough. Note that the maximum gate voltage is 20 V, and that the supply voltage for the MAX4420 may not be more than 18 V. When you connect a motor that needs more than 18 V you will have to use a lower voltage for the MAX4420, for example with the help of a 7815.

When you look at the datasheet for the MAX4420 MOSFET driver, you'll see that it can cope with a peak current of 6 A. This is more than enough to drive our motor, so why do we need a MOSFET? The reason is that this 6 A is a very short peak current. For a constant current, we have to take into account what the maximum power handling of the MAX4420 is, which is 727 mW. According to the datasheet, the maximum output resistance is 5 Ω. To calculate the power we use the following formula:

$$P = V \times I$$

And from Ohm's Law:

$$V = I \times R$$

When we combine these formulae, we can calculate the maximum current:

$$I = \sqrt{(P / R)}$$

Where I = current (in ampères)
R = resistance (in ohms)
V = voltage (in volts)
P = power (in watts)

When we fill in the values for P and R, we get $I = \sqrt{(0.727/5)} = 381$ mA.

Since this is too little for our motor, we conclude that the MOSFET really is necessary.

If you want to do the same power calculation for the IRF740, you should use the following details from the datasheet: maximum power 125 W, maximum internal resistance 0.55 Ω.

The maximum continuous current is $\sqrt{(125/0.55)} = 15$ A.

This is well above the maximum current for this MOSFET (10 A), so there won't be a problem with the power. You'd obviously need a hefty heatsink to get rid of all the heat.

The circuit is shown in Figure 6.11. Note that the ground for the Raspberry Pi, the MAX4420 and the IRF740 are connected together, and that the MAX4420 derives its power from the external power supply. If you find that the motor causes interference on the supply, you

can add a small 0.1 uF capacitor across the motor contacts (make sure that the capacitor is rated for a high enough voltage).

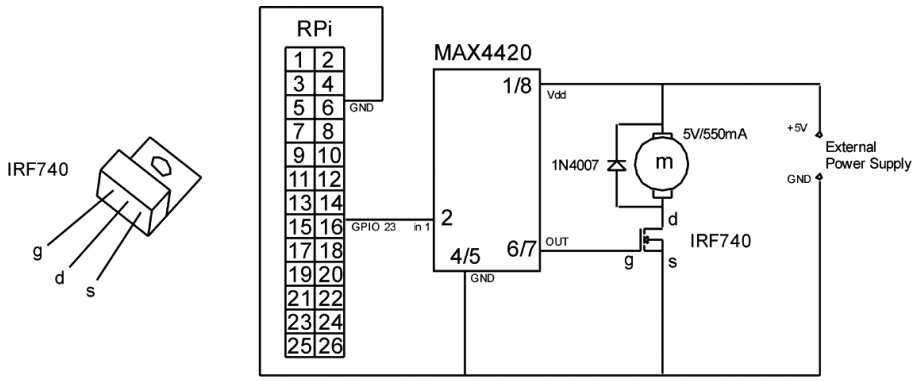


Figure 6.11. Connection of a motor and MOSFET to the header.

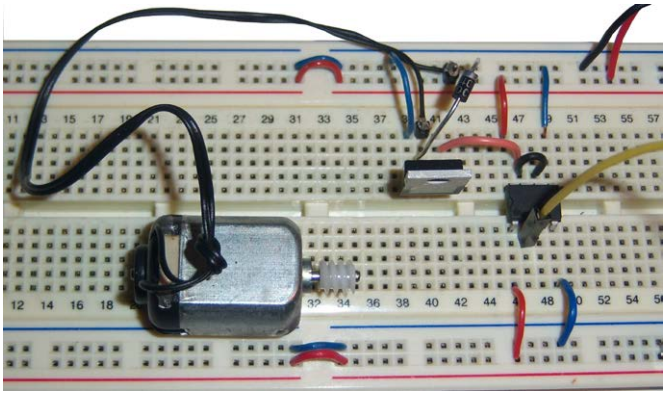


Figure 6.12. Setup on the breadboard.

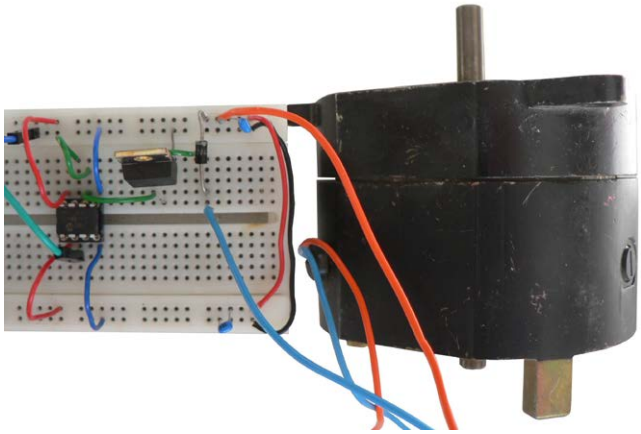


Figure 6.13. You can also connect something bigger than a toy motor...[46]

46 Setup and photo: courtesy B. Oudshoorn.

You can use the following program:

```
from gpiozero import LED

out = LED(23)

while True:
  value=input("Enter on or off to control the motor, or q to quit: ")
  if value=='on':
    out.on()
  if value=='off':
    out.off()
  if value=='q':
    break
```

Start the program with python3 `IRF740a.py`, and stop it by entering q.

Chapter 7 • PWM

When we want to change the brightness of a light, or the speed of a motor, the easiest way would be to vary the voltage. Unfortunately, this doesn't work very well. Most motors need a minimum voltage before they begin to turn, and when they finally start turning, they won't stay at a slow speed. The second problem is that a low voltage usually results in a low torque, so the motor has very little power.

The solution is to keep the voltage constant at a value equal to the maximum voltage to avoid startup problems, and then turn the voltage on and off at a very fast rate. The longer the voltage is kept on, the faster the motor will turn. When you do this fast enough, you'll find that the motor runs surprisingly smoothly. We keep the frequency at which we turn the voltage on and off equal, and only vary the length of time the pulse is on. This is better known as the pulsewidth, hence the term Pulse Width Modulation, shortened to PWM. Figure 7.1 shows PWM waveforms with different duty cycles.

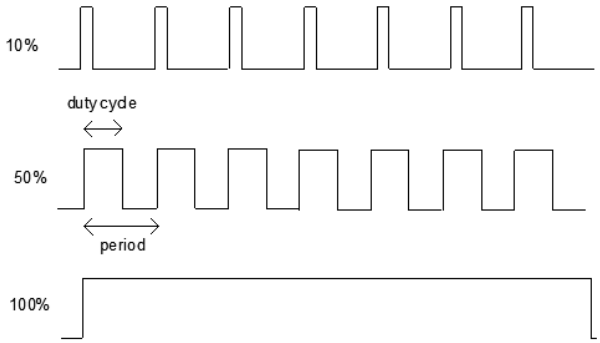


Figure 7.1. PWM signal at three different duty cycles

The relative length of the pulse is called the duty cycle. The distance between two successive beginnings of a pulse is called the period. In the figure above, you can see examples of three different duty cycles. The duty cycle in the top graph is 10%, which means that the voltage is on for 10% of the period. In the middle graph the duty cycle is 50%, and in the bottom graph it's 100%. In the last case it means that the voltage is on all the time, so the motor runs at its maximum speed.

Raspberry Pi 4 GPIO12 (PWM0), GPIO13 (PWM1), GPIO18 (PWM0) and GPIO19 (PWM1) offer PWM functionality. In this project the PWM channel at pin GPIO18 is used. We first build a small test circuit that consists of a 2 mA LED with a current limiting resistor connected to GPIO 18 (pin 12 on the header). Figure 7.2 shows the circuit built on a breadboard. The circuit diagram is shown in Figure 7.3.

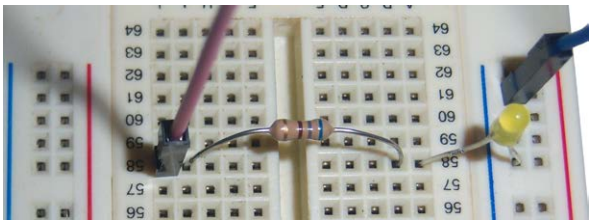


Figure 7.2. Setup on the breadboard.

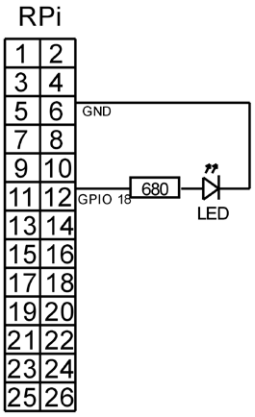


Figure 7.3. Connection to the PWM pin on the header.

Let's start by turning the LED on to check we have connected it correctly. Startup Python by typing `python3`. The following tekst will appear on your screen:

```
pi@raspberrypi:~ $ python3
Python 3.7.3 (default, Apr 3 2019, 05:39:12)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Now we will load the library that will take care of the communication between Python and the Raspberry Pi's hardware. We don't load the full library, just the part to control an LED by PWM.

```
from gpiozero import PWMLED
```

Next we connect the LED to GPIO 18:

```
led = PWMLED(18)
```

The command above is in fact an object (see section 4.3 for the explanation of an object), which means it has properties. For an LED object these are the most important properties that are available:

PWMLED object property	Explanation
PWMLED(pin, active, start, frequency)	Parameters of the object: pin = GPIO pin number active = what is seen as "on", can be True (on = high) or False = (on = low) start = start duty cycle frequency = PWM frequency (default is 100 Hz)*
value	Duty cycle from 0 (0%) to 1 (100%)
on()	Object on
off()	Object off
blink(on, off, fade_in, fade_out, number, background)	Slowly turn or and of, where on = time in seconds LED is on (default = 1) off = time in seconds LED is off (default = 1) fade_in = turn on time in seconds (default = 1) fade_out = turn off time in seconds (default = 1) number = times the event occurs (int, default = infinite) background = run as task in background (default = True)**
* The default value is used when nothing is indicated ** If True the program continues. If False the program waits until the Blink command has finished. Note that if the number of events is infinite (the defult value), the command will never be finished and the program will never continue!	

Now we can turn on the LED for 50% using the following command:

```
led.value = 0.5
```

You can now try a couple of different values (from 0 to 1). Stop Python using the `exit()`. Now the LED is off, Python has stopped and you will see the following on your screen:

```
pi@raspberrypi:~ $ python3
Python 3.7.3 (default, Apr 3 2019, 05:39:12)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from gpiozero import PWMLED
>>> led = PWMLED(18)
>>> led.value = 0.5
>>> exit()
pi@raspberrypi:~ $
```

7.1 PWM LED with Graph

In this exercise we will use the following program to drive an LED with various duty cycles. Contrary to what we did in the example above we will now use the full command for defining the PWMLED object:

```
led = PWMLED(18,True,0,100)
```

Using the table on the previous page you can find out what the parameters mean. In this chapter we want to use different frequencies for driving the LED. We start with 100 Hz (because this is the default value we actually didn't have to specify this).

```
from gpiozero import PWMLED

led = PWMLED(18,True,0,100)

while True:
  answer=input("Enter a value between 0-1, or q to quit: ")
  if answer=='q':
    break
  else:
    val=float(answer)
    if (val>=0) and (val<=1):
      led.value = val
    else:
      print("Error: value out of range.")
```

Save the program as `pwm1.py`. If you would like to produce graphs yourself you will need WinOscillo. Figure 7.4 shows the graph for the waveform at 100 Hz, 50% duty cycle.

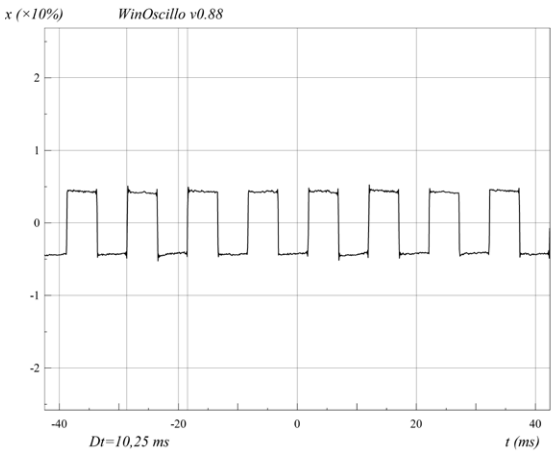


Figure 7.4. Spectrum 100 Hz, duty cycle 50%.

Having set the duty cycle frequency at 100 Hz we are now going to check the actual frequency. Start the program and enter "0.5". Hook up WinOscillo and set the slider to zero.[47] Slowly turn up the slider until you see a nice projection. Now adjust the time scale until the pulses are clearly visible. Using the curcor lines you can comfortably measure the time between two pulses. In the figure above the time between to pulses is 10.25 ms. This equals

47 Refer to section 1.12 for more information in WinOscillo, how to make the hardware and how to operate the program.

$1/10.25 \cdot 10^{-3} = 97.6$ Hz.

That is not bad at all. Now we also want to check the duty cycle (Figure 7.5).

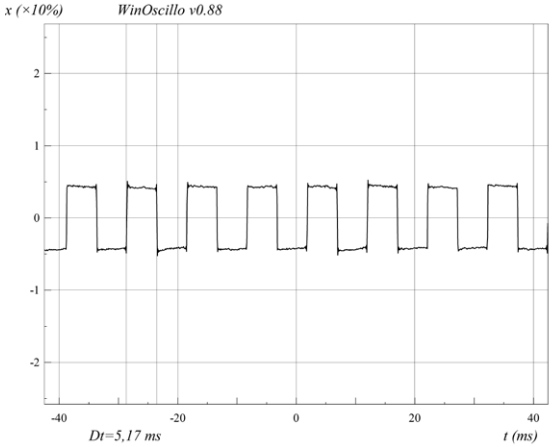


Figure 7.5. Spectrum 100 Hz, duty cycle 50%.

Here we measure 5.17 ms. This equals a duty cycle of

$5.17/10.25 \times 100\% = 50.4\%$,

which isn't bad either. Halt the program and change the PWM frequency to 1000 Hz. This line

```
led = PWMLED(18,True,0,100)
```

needs to be changed to

```
led = PWMLED(18,True,0,1000)
```

Restart the program, enter 0.5 for a 50% duty cycle and repeat the measurements. The waveform is shown in Figure 7.6.

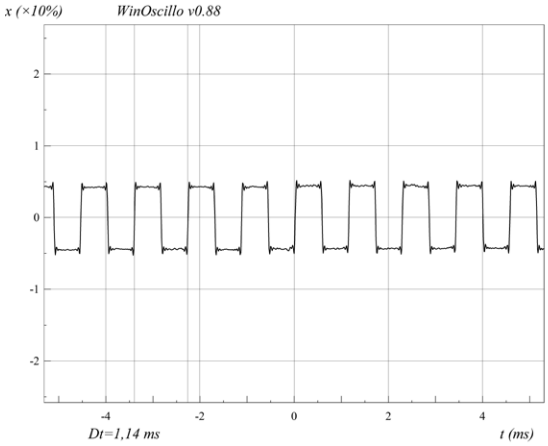


Figure 7.6. Spectrum 1000 Hz, duty cycle 50%.

The results are similar to the measurements at 100 Hz. The measured frequency is 877 Hz and the duty cycle is exactly 50%. The GPIOzero developers don't mention a maximum frequency, but with rising frequency the stability drops. At 1000 Hz a variation is clearly visible with a Raspberry Pi model 4 practically doing nothing else.

7.2 Light with Gradually Increasing Brightness

This project could be used as a wake-up lamp, which is a lamp where the light gradually increases in intensity. You could also use it next to a small aquarium to avoid startling the fish if the light suddenly went on.[48] We use three 6-volt lights, which use 65 mA each. Since both the 6 V as well as the 65 mA current are too high for the Raspberry Pi, we use a TD62783, as described in chapter 6.

You can of course use different lights as long as they stay within the specifications of the TD62783. Since we're connecting three lights, we also use three inputs, which are all connected to GPIO 18 (pin 12 on the header). [49] The ground of the Raspberry Pi and adjustable power supply have to be connected together. The +3.3 V and +6 V obviously shouldn't be. The +3.3 V from the Raspberry Pi isn't actually used since the TD62783 takes its power from the V_{cc} pin (9), which is connected to the adjustable power supply. The circuit diagram is shown in Figure 7.7.

48 If you want this to wake you up, you'll need a much more powerful light. With the knowledge you picked up in Chapter 6 it shouldn't be a problem to connect a bigger light. Note that none of the methods in this book is suitable for AC line voltages!

49 Strictly speaking this isn't necessary, since the three lights could be connected to one pin. The current consumption is below 500 mA, which one pin could supply. Since they should come on together, they could be connected to one pin, see chapter 6.

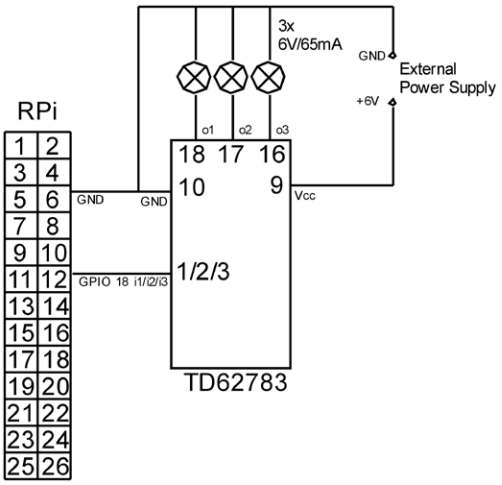


Figure 7.7. Some lights connected to the TD62783.

The circuit is built on a breadboard as shown in Figure 7.8.

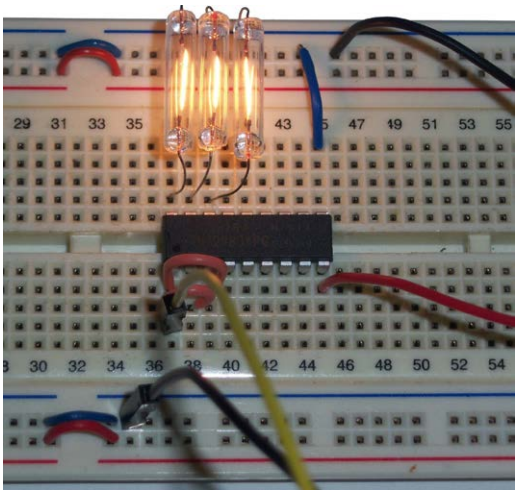


Figure 7.8. Setup on the breadboard.

The two wires that leave the photo on the right go to the adjustable power supply.

To make the brightness of the lights increase gradually, we start with a PWM of zero and slowly increase it up to 1024. The quicker we do this, the quicker the lights reach full brightness.

We create a graphical program based on our standard template. We will only discuss the sections of code that have to be added to the template.

```
# your own objects
```

We want the lights to increase in brightness slowly. We can achieve this by increasing the PWM value slightly at short intervals. For this we'll use a timer, which we'll define here. Since we need to use it elsewhere in the program, we define it as 'self'.

```
# timer object
self.timer = wx.Timer(self)
self.Bind(wx.EVT_TIMER, self.OnTimer, self.timer)
```

The PWM pin has to be turned on, with the required PWM mode selected. We will use the default frequency of 100 Hz. To be able to use the `light` object anywere in the program, we will define it as global.

```
# declare pins
global light
light = PWMLED(18,True,0,100)
```

We require two buttons, one to start the timer and to increase the brightness of the lights, and one to turn off the lights and to stop the timer in case it is still running.

```
# light on button
cbtn0 = wx.Button(self, label='Light on', pos=(20, 60))
cbtn0.Bind(wx.EVT_BUTTON, self.OnStart)

# light off button
cbtn1 = wx.Button(self, label='Light off', pos=(20, 100))
cbtn1.Bind(wx.EVT_BUTTON, self.OnStop)
```

We keep the user informed of the progress using a text field. The definition should be on a single line, but there's not enough space on this page.

```
# static text
self.text = wx.StaticText(self, label='Press Light On
  to start',pos=(20,30))
```

The first function is for the "Light On" button. This function starts the timer with an interval of 100 ms, and sets the counter for the duty cycle to zero. You can make the light increase in brightness more quickly by making this interval shorter, or slower by making the interval longer.

```
# your own functions
def OnStart(self, e):
  # start the timer, event every 100 ms
  self.count=0
  self.text.SetLabel('0')
  self.timer.Start(100)
```

This is the function for the "Light off" button. The timer is stopped and the initial text is shown again on the label.

```
def OnStop(self, e):
  # stop the timer and set pwm to 0
  self.timer.Stop()
  self.text.SetLabel('Press Light On to start')
  light.value = 0
```

The last function is for the timer itself. This is where the counter, which keeps track of the duty cycle, is incremented by 1, and where the PWM pin is controlled. It also displays the current duty cycle with 1 decimal place as a percentage, so the user knows how the program is progressing.

```
def OnTimer(self, e):
  # timer function
  self.count = self.count + 0.005
  self.text.SetLabel('Duty cycle ' +str(self.count*100)+' %')
  if self.count>=1:
    # done stop the timer
    self.timer.Stop()
    self.text.SetLabel('Good morning!')
  else:
    light.value = self.count
```

Start the program with `sudo python3 morning.pyw`, and close it with the cross. Figure 7.9 shows a typical run of the program.

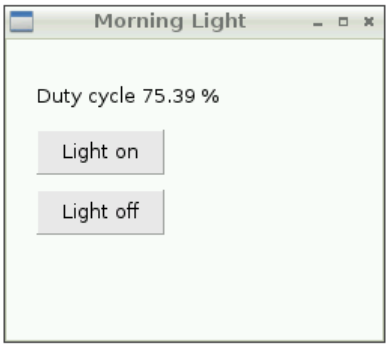


Figure 7.9. Operation of the program.

7.3 Motor with Variable Speed

Instead of the lights used in the previous project, you could also use a motor. In this example, we use a Mitsumi 5-V motor (figure 75). This motor takes 15 mA when free-running and 145 mA when stalled.[50] You can use your own motor instead, as long as its voltage and current are within the specifications of the TD62783. Figure 7.10 shows the circuit built on a breadboard.

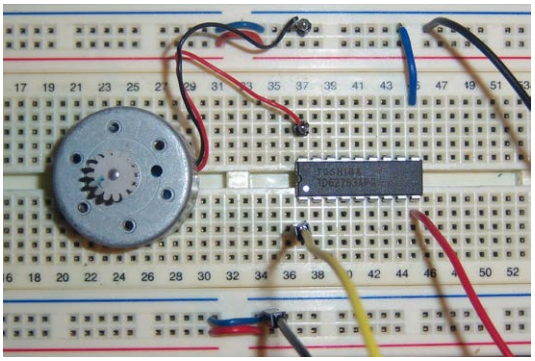


Figure 7.10. Setup with the motor on the breadboard.

You can try out the three different PWM techniques on your motor, and see which one works best. You can use the pwm1.py program for this, as used in the beginning of this chapter.

50 Stalled means that the motor isn't turning while it has the full supply voltage across it. If you want to measure the stalled current yourself, you have to do this quickly because many motors only cool themselves while they're running. A stalled motor could overheat in time.

Chapter 8 • SPI

8.1 Introduction to SPI

SPI is an acronym for Serial Peripheral Interface. There is one master and a minimum of one slave on the SPI bus. A separate connection, the CE (chip enable), is used to indicate for which slave the message is intended. The disadvantage of this is that you need an extra wire for each slave on the bus. But the advantage is that the address decoding doesn't take place on the bus. Incidentally, the CE line uses inverse logic, which means that the line is normally High, and when the master wants to select a slave, it makes its CE line Low. To show that CE uses inverse logic, it is often written as /CE. The Raspberry Pi has two CE lines, so you can add up to two SPI slaves to the bus.[51]. Figure 8.1 shows the Raspberry Pi SPI lines.

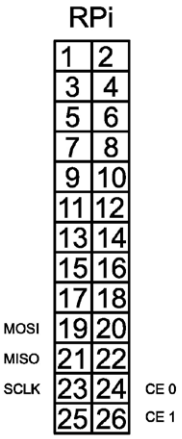


Figure 8.1. SPI pins on the header.

The SPI bus itself consists of three wires, two for data and one for the clock. The clock line (SCLK -- serial clock) is controlled by the master. In this way, all ICs on the bus will use the same frequency. One data line is reserved for traffic from the master to the slaves, the other for traffic in the opposite direction. The MOSI -- master output slave input -- pins have to be connected together, as do the MISO -- master input slave output pins.

51 This is not completely true, since you can connect more slaves by using GPIO pins, but the default is a maximum of two. An example of the use of GPIO pins can be found in project 8.4.

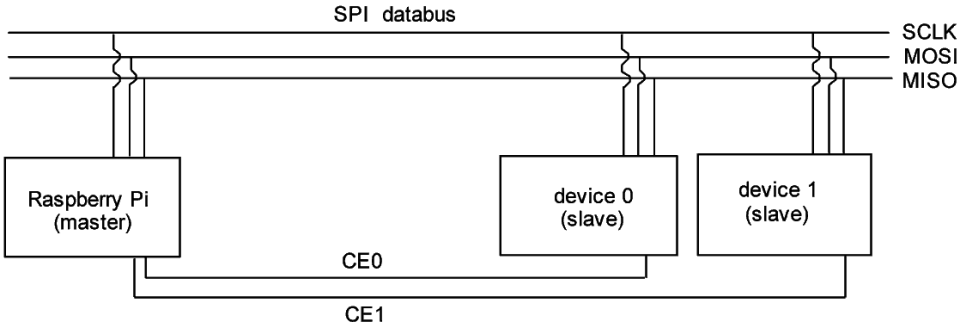


Figure 8.2. Basic SPI configuration with two slaves.

In practice you will find that the SPI connections may be given different names, depending on the manufacturer of the components:

Raspberry Pi	**Alternative names**
MOSI	Master: SDO Slave: SDI, DIN
MISO	Master: SDI Slave: SDO
SCLK	SCK, SCL
CE	CS, SS
When SDO and SDI are used, the connections should cross over: the SDO connection of one chip should be connected to SDI connection of the other chip. Data that leaves one chip (SDO) has to enter the other one (SDI).	

Table 8.1. SPI pin names.

The acronyms have the following meanings:

MOSI	Master Output Slave Input
MISO	Master Input Slave Output
SDO	Serial Data Out
SDI	Serial Data In
DIN	Data IN
SCLK, SLK, CLK	Serial Clock
CE	Chip Enable
CS	Chip Select
SS	Slave Select

The technical documentation for SPI components mentions SPI modes that are supported. The mode determines what the relationship is between the clock pulses and the data pulses. The clock pulses can be generated in two ways. The line is normally low and is made temporarily high for the clock pulse, or exactly the opposite: the line is normally high and is

made temporarily low for the clock pulse. This is known as the clock's base value or polarity (CPOL). The data pulse can be read at the start of a clock pulse (leading edge) or at the end (trailing edge) (CPHA, clock phase). There are four possible combinations, which are shown in the table below.

mode	CPOL (polarity) 0 = low 1 = high	CPHA (phase) 0 = leading edge 1 = trailing edge
0	0	0
1	0	1
2	1	0
3	1	1

Table 8.2. SPI modes.

For successful communications the master and slave must obviously support the same SPI modes and be set up accordingly. The Raspberry Pi uses mode 0,0 which is the one found most often in practice. This is something you have to keep in mind when you buy SPI components that haven't been described in this book.

The communications on the SPI bus are always initiated by the master, which starts with making the CE line low of the slave with whom it wants to communicate. Only that slave will listen to the master at that particular time. What happens next depends on the information and commands required by the slave, and will be detailed in the datasheet for that slave.

An interesting aspect of SPI communications is that the transmission and reception occur at the same time. The Raspberry Pi puts the byte to be transmitted into a buffer.[52] The byte will then be sent from this buffer one bit at a time. For every bit that's put onto the MOSI line, another is read simultaneously from the MISO line. After eight bits, the contents of the buffer will have been replaced by the received byte. The same method is used in the slave, so you will also have simultaneous reception and transmission. This particular method has two consequences:

1. The communications are very quick because the reception and transmission don't have to wait for each other.

2. The question from the master and the answer from the slave don't have a one-to-one relationship. Say that the master asks a question. During the transmission of this question an answer is received from the slave. But this is not the answer to the last question, since the slave hasn't received it yet. Instead, it's the answer to the previous question from the master.

52 All characters used in a computer (letters, figures, punctuation marks) are represented by numbers. These numbers can be found in what is called the ASCII table. The letter "a" has a value of 97, for example. These numbers are bytes, which means that they have a value from 0 to 255. A byte consists of 8 bits, which can have a value of 0 or 1. The number 97 (the letter "a") is 01100001 in binary.

It is important to remember that for every character you transmit, you must receive a character in return, and that the received character is the answer to the previous question.

We have the following commands at our disposal:

Command	Description
open(0,0)	Open a connection. The first zero is the CE pin (0=CE0 and 1=CE1) and the second zero is the bus (bus 0 is the only one available).
close()	Close the connection.
r=spi.xfer([list])	Exchange information. Variable r is the answer, list is a list of at least one element. The answer is (of course) exactly the same length as this list and is also a list. The CE line is released in between each element of the list.
r=spi.xfer2([list])	Ditto, but the CE line stays on until all elements have been sent.
spi.writebytes([list])	Send the list, but don't save the answer in a variable.

Table 8.3. SPI commands.

8.2 MAX522 Digital to Analog (DAC)

A digital signal can have two states, on (in this case +3.3 V) or off (in this case 0 V). An analog signal can have any possible value, for example 1.24 V. The Raspberry Pi can only output digital signals on the GPIO pins, as they can only be on or off.

Internally the Raspberry Pi also works with digital signals, just like any other computer. These digital signals can be turned into analog using a digital to analog converter, known as a DAC. Here we use the MAX522 (Figure 8.3), an IC made by Maxim that has two built-in DACs and an SPI connection, suitable for operation at 3.3 V.

We'll go through all the steps you would normally take to connect an unfamiliar component to the Raspberry Pi, with the help of its datasheet. This way you'll be able to connect other components yourself. For this project you don't even need to have the datasheet for the MAX522 to hand, since all relevant information has been copied to the book.

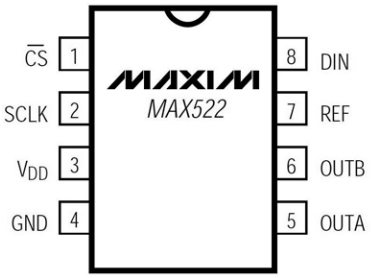


Figure 8.3. Pinout for the MAX522.

The MAX522 expects a supply voltage between 2.7 and 5.5 V, so it's suitable for use at 3.3 V. The description for the other pins is in the following table.

Pin Description

PIN	NAME	FUNCTION
1	$\overline{CS}$	Chip Select (active low). Enables data to be shifted into the 16-bit shift register. Programming commands are executed at the rising edge of $\overline{CS}$.
2	SCLK	Serial Clock Input. Data is clocked in on the rising edge of SCLK.
3	V_{DD}	Positive Power Supply (2.7V to 5.5V). Bypass with 0.22µF to GND.
4	GND	Ground
5	OUTA	DAC A Output Voltage (Buffered). **Connect 0.1µF capacitor or greater to GND.**
6	OUTB	DAC B Output Voltage (Buffered). **Connect 0.01µF capacitor or greater to GND.**
7	REF	Reference Input for DAC A and DAC B
8	DIN	Serial Data Input of the 16-bit shift register. Data is clocked into the register on the rising edge of SCLK.

Figure 8.4. Pin descriptions from the datasheet.

The manufacturers have pointed out in two places in the table that a capacitor should be added to guarantee the correct operation of the chip. Normally these comments are in the text alongside a circuit example. Apparently, this is so important that the manufacturer has repeated it here, and put it in bold for extra effect. You can see that an alternative term has been used for an SPI pin: DIN (data in). This has to be connected to the MOSI pin on the Raspberry Pi. Since the DAC doesn't return any information, no pin has been made available for this. The MISO pin on the Raspberry Pi will therefore remain unconnected. [53]

We won't be using DAC B and will turn this off via software in a moment. For this reason there is no need to add a capacitor to its output in this project. The output current that the DAC can supply is very limited. DAC A can supply a maximum of 5 mA, and for DAC B the maximum is only 0.5 mA. In this project we therefore connect a voltmeter to the output to verify that everything is working properly.[54] In the next project we'll see how we can improve on the low current output. The V_{ref} pin is connected to +3.3 V (see Figure 8.5), and we'll explain later why this is necessary. Figure 8.6 shows the circuit built on a breadboard.

53 Since something is always received with the SPI protocol, but the pin isn't connected to anything, we receive only noise.

54 If you don't have a voltmeter, you can connect a 2 mA LED and a 680 Ω resistor. The brightness of the LED will depend on the output voltage of the DAC.

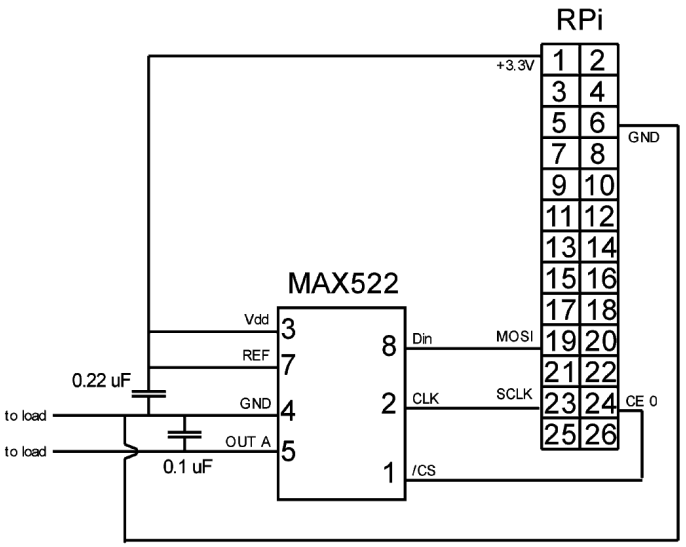


Figure 8.5. Connecting the MAX522 to the header.

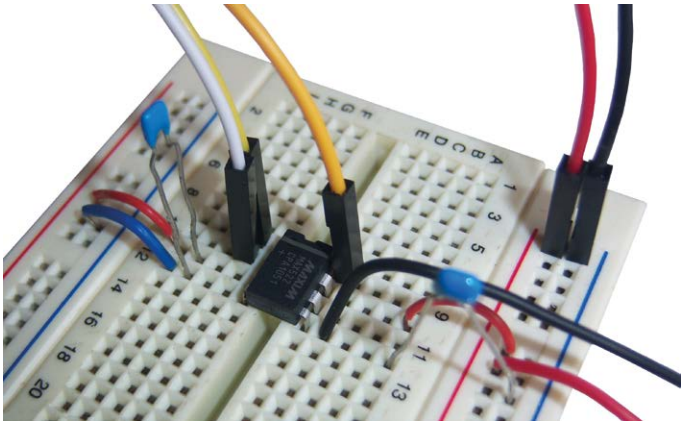


Figure 8.6. Setup on the breadboard.

The two wires leaving the breadboard at the right go to the multimeter. The code for a 0.1 µF capacitor is 104, and that for a 0.22 µF capacitor is 224. These capacitors should be mounted as close to the DAC as possible.

Now that the hardware has been connected, we have to take care of the software. The first question is whether the DAC can work in mode 0,0 (as used by the Raspberry Pi). The mode determines what the relationship is between the clock pulses and the data pulses. When the mode isn't explicitly mentioned in the datasheet it should show what combinations of CPOL and CPHA can be used. With the help of the table in the introduction, you can then find out which modes are applicable.[55]

55 If this information isn't available either, you can search through the datasheet for a timing diagram and determine the mode from that.

The datasheet for the MAX522 doesn't mention the word 'mode', so we'll look for the information on CPOL and CPHA. We found the following text:

> *"The MAX522 serial interface is compatible with Microwire, SPI, and QSPI. For SPI, clear the CPOL and CPHA bits (CPOL = 0 and CPHA = 0). CPOL = 0 sets the inactive clock state to zero and CPHA = 0 changes data at the falling edge of SCLK."*

This means that CPOL and CPHA should both be zero, and with the help of the table we see that is mode 0.

Apart from the mode, the clock speed is important as well. Most components will automatically use the clock speed provided by the master, but there is generally a maximum that the master may not exceed. According to the datasheet the maximum clock speed is 5 MHz. Since the default value of the Raspberry Pi is 500 kHz, this won't cause any problems.

The datasheet has a complex description with a table for the calculation of the DAC value, which boils down to the following formula:

$$V_{DAC} = \frac{DATA}{256 \text{ x } Vref}$$

Since we connected V_{REF} to +3.3 V it means that the DAC has a range from 0 to 3.3 V. This is quite noteworthy since many DACs have internal losses and aren't capable of achieving 3.3 V at the output. Those DACs that can do this are called "rail-to-rail" devices. The word 'rail' here applies to the supply rails, so rail-to-rail means "the full range from 0 V to V_{cc}".

The only thing left to do is to turn on the DAC and send a value to it.

Table 2. Serial-Interface Programming Commands

CONTROL								DATA								FUNCTION
UB1	UB2	UB3	SB	SA	UB4	LB	LA	B7 MSB	B6	B5	B4	B3	B2	B1	B0 LSB	
X	X	1	*	*	0	0	0	X	X	X	X	X	X	X	X	No Operation to DAC Registers
X	X	1	*	*	0	0	0									Unassigned Command
X	X	1	*	*	0	1	0	8-Bit DAC Data								Load Register to DAC B
X	X	1	*	*	0	0	1	8-Bit DAC Data								Load Register to DAC A
X	X	1	*	*	0	1	1	8-Bit DAC Data								Load Both DAC Registers
X	X	1	0	0	0	*	*	X	X	X	X	X	X	X	X	All DACs Active
X	X	1	0	0	0	*	*	X	X	X	X	X	X	X	X	Unassigned Command
X	X	1	1	0	0	*	*	X	X	X	X	X	X	X	X	Shut Down DAC B
X	X	1	0	1	0	*	*	X	X	X	X	X	X	X	X	Shut Down DAC A
X	X	1	1	1	0	*	*	X	X	X	X	X	X	X	X	Shut Down All DACs

X = *Don't care.*
* = *Not shown, for the sake of clarity. The functions of loading and shutting down the DACs and programming the logic can be combined in a single*

Figure 8.7. Command overview for the MAX522.

Figure 8.7, taken from the datasheet, shows the structure of the commands that can be sent to the MAX522. It comes to 16 bits in total of which the first eight are the command for the MAX and the second eight are the DAC value, if relevant. The SB:SA bits have stars in them on some lines, which means we may use the values from the lower half of the table. Where there is an 'X' you can put in anything you like, although we use a 0. It may not appear to matter much, but it's quite possible that in the future the MAX522 or its replacement could use the bits that are currently unused. When that happens, the manufacturer usually designs it such that any new features are turned off with a 0. If we already set the unused bits to 0, we improve our chances that we can replace a faulty MAX522 with a newer type in the future, without having to modify the program in any way.

Say that we want to use only DAC A and set its output to 50% of the V_{ref} voltage. The bits to be set are then as follows:

UB1:UB2	00	don't care, use 0
UB3	1	always a 1
SB:SA	10	turn off DAC B (shut down DAC B)
UB4	0	always a 0
LB:LA	01	use DAC A (load register to DAC A)
B7:B0	1000 0000	128 (50% van 256)

The control byte therefore becomes 0011 0001 (49 decimal) and the DAC byte is 1000 0000 (128 decimal). When we want to use a different DAC value, the control byte can stay the same, so we keep this as a fixed value in the program. The DAC value is requested of the user and sent to the MAX522.

As part of the SPI protocol, a reply is always received whenever there is a transmission, which is stored in the variable "reply". In this case it won't contain any useful information since the MISO pin isn't even connected. We therefore completely ignore the reply value in the software.

```
import spidev

spi=spidev.SpiDev()
spi.open(0,0)

while True:
  answer=input("Enter a value between 0-255, or q to quit: ")
  if answer=='q':
    break
  else:
    value=int(answer)
    if (value>=0) and (value<=255):
      reply=spi.xfer2([49,value])
    else:
      print("Error: value out of range.")
```

We start the program with `python max552a.py`, and stop it by entering `q`. The program first checks if the user entered `q`. If that is the case, the break instruction is used to exit the `while` loop, and since there aren't any other commands after the loop, the program stops. If the input wasn't a `q` then it's converted into an integer and checked if it falls within the permitted range. If it isn't, an error message is shown. If it is valid, then the value is sent to the MAX522. You will be able to see the result on the voltmeter.

Since Python is a scripting language, you could also operate the MAX522 without a program. Start Python using the command `sudo python`, and type in the following instructions, one after the other. In this example we will set the output of the DAC to 3.3 V:

```
pi@raspberrypi:~ $ sudo python3
Python 3.7.3 (default, Apr 3 2019, 05:39:12)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import spidev
>>> spi=spidev.SpiDev()
>>> spi.open(0,0)
>>> reply=spi.xfer2([49,255])
```

Then we set it to 0 V:

```
>>> reply=spi.xfer2([49,0])
```

Then stop Python using `exit()`, as follows:

```
>>> spi.close()
>>> exit()
```

This mode of operation is very useful when you want to try out a handful of commands, as there is no need to write a complete program.

8.3 DAC with Opamp

In the previous project we found out that the output current of the DAC was very limited. This is perfectly normal for a DAC, and isn't even that much different to the Raspberry Pi's outputs. We can increase the output power very easily, using an opamp.[56]

56 Opamp is short for Operational Amplifier. Opamps were originally designed to perform calculations ("operations") in analog computers. These days computers are digital and opamps are often used to amplify signals, mainly due to their attractive price.

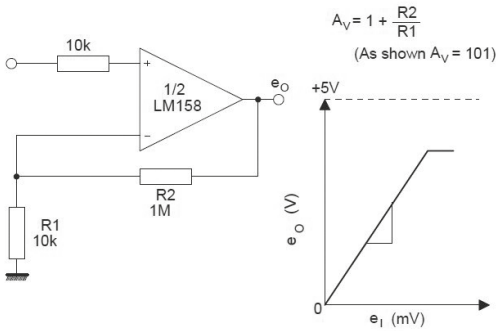


Figure 8.8. Non-inverting DC amplifier circuit.[57]

The circuit in the figure above is for a DC amplifier. The gain of the circuit is calculated using the formula:

$$A_v = 1 + \frac{R2}{R1}$$

With the resistors used in the circuit, the output voltage is therefore 101 times the input voltage. However, we don't really need any voltage amplification, since we want to follow the output voltage of the DAC exactly. Our only objective is to increase the potential amount of output current. This can be achieved by making R2 zero (in other words, a wire-link) and making R1 infinitely large (or, remove it completely). This results in the following circuit: [58]

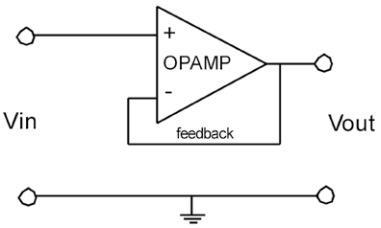


Figure 8.9. Opamp used as a non-inverting voltage follower.

In fact, an opamp amplifies the *difference* between the two inputs. The output of the DAC is connected to the non-inverting input (shown in the circuit with a +). The output is connected back to the inverting input (shown in the circuit with a -) in a feedback loop. This input will therefore always be at the same voltage as the output.

When the output voltage of the DAC rises, the voltage on the non-inverting input of the opamp will be higher than on the inverting input. The result is that the output voltage rises until both inputs are identical again. This means that the output will have to be identical

57 This circuit comes from the datasheet for the LM358, which covers several opamps, including the LM158 in the circuit. This has no effect on the way they're used.

58 You will have noticed that we've also left out the 10k resistor at the input, since it is never used in a voltage follower.

to the input connected to the DAC, which is exactly what we wanted to achieve. This type of circuit is called a "non-inverting voltage follower".[59] Since no current flows through the inputs of the opamp, this circuit is often used as a buffer.

For this project we'll use the LM358 dual opamp made by STMicroelectronics (Figure 8.10). This chip contains two opamps of which we only use one. The datasheet doesn't give a minimum value for the operating voltage, only a maximum of 32 V. The maximum output current is 40 mA, which is more than enough to drive an LED. It's assumed that the input resistance of an opamp is so large that no current flows into it, but in reality this is obviously not possible. The real input current for the LM358 is 2 µA (0.002 mA).

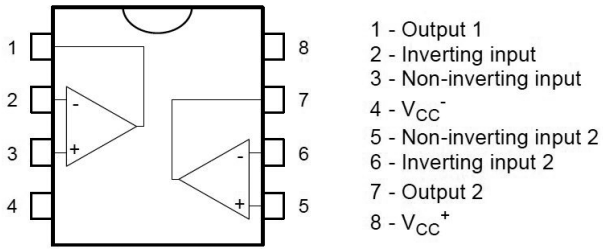


Figure 8.10. Pinout for the LM358 opamp.

The LM358 is not a rail-to-rail opamp, and according to the datasheet has an output range from 0 to V_{cc}-1.5 V. Since we're using 3.3 V, it leaves very little for the voltage follower. In this case the output of the opamp would be limited to only 3.3-1.5=1.8 V. We can improve on this by using the 5 V from the Raspberry Pi to supply the opamp. The maximum then becomes 5-1.5=3.5 V, which is enough for our application.

We could also have used a rail-to-rail opamp (for example, the TLV2372 from Texas Instruments). This means that when the opamp operates from 3.3 V, the output can also reach 3.3 V. However, it would make little sense to connect the opamp to the 3.3 V pin of the Raspberry Pi. This connection can only supply a maximum of 51 mA, which is for all pins together. Since the opamp could need over 80 mA (40 mA per output) in theory, this connection is obviously not suitable.

For this reason it is better to connect the opamp to the 5 V pin. This pin can, depending on the power supply used and the current consumption of the Raspberry Pi, supply about 250 to 300 mA. This is more than enough for the opamp and LED. Since the opamp is used as a non-inverting voltage follower, its maximum output voltage is 3.3 V (since the maximum voltage at its input on pin 3 is also 3.3 V).

Figure 8.11 shows the complete circuit. The opamp is powered directly from the +5 V pin of the Raspberry Pi. You can use a standard 20 mA LED with a 47 Ω (color code: yellow-purple-black) current limiting resistor as the load. The circuit is built on a breadboard as shown in Figure 8.12.

59 It is also known as a "Unity Buffer Amplifier".

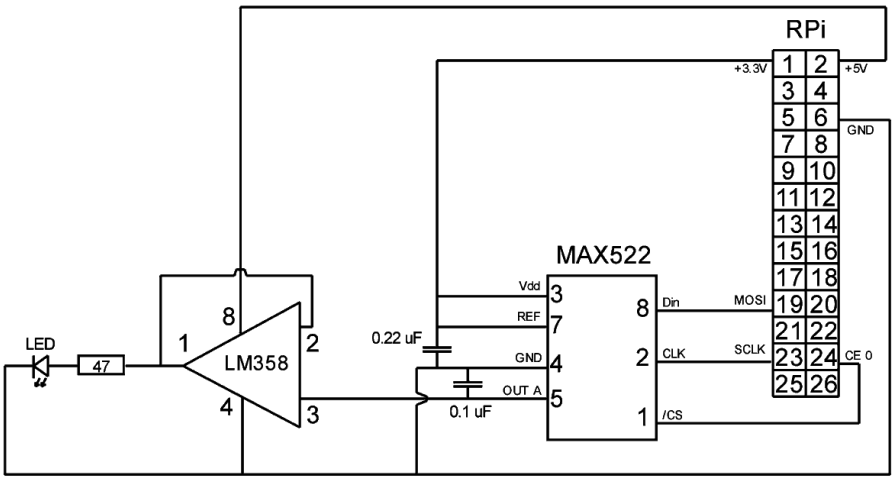


Figure 8.11. MAX522 with LM358 opamp.

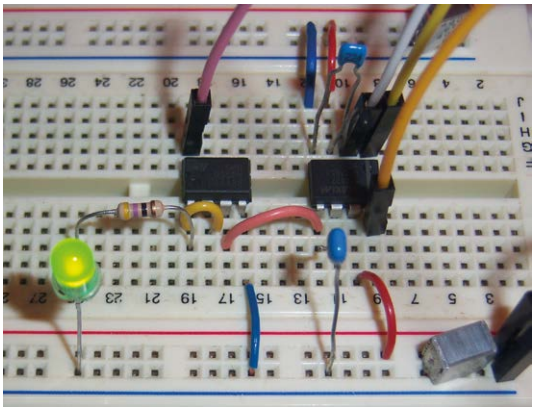


Figure 8.12. Setup on the breadboard.

The addition of an opamp doesn't change the way in which we control the MAX522, so we can use exactly the same program. This isn't very interesting, so we'll create a graphical program instead, and use a slider for setting the DAC voltage. For the slider object we specify that the function `OnSliderScroll` should be called every time the user moves the slider. In this function the slider value will be stored in the variable 'value'.

```
value=self.sld.GetValue()
```

This value has a range from 0 to 255 and is the byte that we need to send to the DAC. Next, this value is converted to its equivalent voltage, using the formula:

$$\text{voltage} = \frac{\text{value}}{255} \text{ x } 3.3\text{V}$$

Since value doesn't have any decimals (and neither does 255), Python assumes that we don't want any decimals in the answer either. The result is that `value/255` will always be zero, unless value equals 255. This is obviously not what we had in mind. We can get round this by dividing by 255.0 instead of 255. This is exactly the same division, but now Python will give the answer as a decimal number. All that's left is to round the result to two decimal places, using `round(...,2)`.

```
    value2=round((value/255.0)*3.3,2)
```

We obviously use our graphical template as discussed in section 4.3, and will only show the additional objects and functions here.

```
# your own objects

    # declare pins
    global spi
    spi=spidev.SpiDev()
    spi.open(0,0)

    # define slider
    self.sld = wx.Slider(self, value=0, minValue=0,
      maxValue=255, pos=(20,20), size=(200, -1), style=wx.SL_HORIZONTAL |
      wx.SL_LABELS )
    self.sld.SetTickFreq(10, 1)
    self.sld.Bind(wx.EVT_SCROLL, self.OnSliderScroll)

    #static text
    self.txt = wx.StaticText(self, label='0', pos=(40, 90))

# call routine to define window text
    self.OnSliderScroll(None)

# your own functions

  def OnSliderScroll(self, e):
    value=self.sld.GetValue()
    value2=round((value/255.0)*3.3,2)
    self.txt.SetLabel('DAC voltage is '+str(value2)+' volt')
    spi.xfer2([49,value])
```

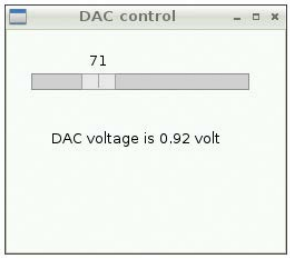


Figure 8.13. Graphical DAC control.

Start the program with `python3 max522b.pyw`. Note that `SetTickFreq`, which is used in the code to add divider lines on the slider, doesn't work in Debian Linux (you won't see any lines on the slider). However, on a Windows machine this does work properly.

8.4 More than two SPI devices

When you want to use more that two SPI devices, you'll notice that there aren't enough CE pins. Fortunately you can make your own CE pins. The SPI software always uses CE0 or CE1, which is something you can't change. For all extra devices you use CE0, but you don't connect it. Instead, you use the appropriate GPIO pin for each device in your own program. The SPI software still drives the CE0 pin for all extra devices as well (you can't change this) but that doesn't matter since nothing is connected to it.

Example 1: you have two SPI devices.

CE0	device 0 (use CE0 in the software)
CE1	device 1 (use CE1 in the software)

Example 2: you have four SPI devices.

GPIO23	device 0 (use CE0 in the software, manually select GPIO 23)
GPIO24	device 1 (use CE0 in the software, manually select GPIO 24)
GPIO25	device 2 (use CE0 in the software, manually select GPIO 25)
CE1	device 3 (use CE1 in the software)

In the example below the /CS pin (number 1) of the MAX522 is connected to GPIO 25 (pin 22 on the header). Since the CE line uses inverse logic, it is directly turned high at the beginning of the program. When the Raspberry Pi wants to transmit something, it first turns the CE line low, then turns it high after the completion of the transmission.

```
import spidev
from gpiozero import LED

spi=spidev.SpiDev()
spi.open(0,0)
# CE disable (high)
io.digitalWrite(25,1)
```

```
while True:
  answer=input("Enter a value between 0-255, or q to quit: ")
  if answer=='q':
    spi.close()
    ce.off()
    break
  else:
    value=int(answer)
    if (value>=0) and (value<=255):
      # CE disable (low)
      ce.off()
      reply=spi.xfer2([49,value])
      # CE disable (high)
      ce.on()
    else:
      print("Error: value out of range.")
```

This program is called MAX522c.py. You can connect the GPIO 25 pin directly to the MAX522; there is no need for any resistors:

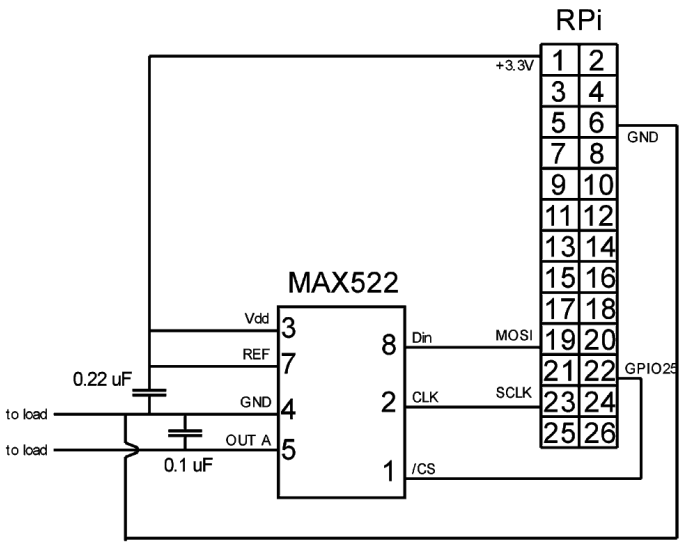


Figure 8.14. MAX522 connected to an "extra" CE pin.

8.5 MCP3008 Analog to Digital (ADC)

In this project we carry out the conversion in the opposite direction: we change an analog signal into digital. This gives us the ability to measure the position of a potentiometer with the Raspberry Pi, or to measure the light intensity (see project 12.8). For the conversion we use the MCP3008 made by Microchip (see Figure 8.15). This SPI chip has 8 ADC units (Analog to Digital Converter).

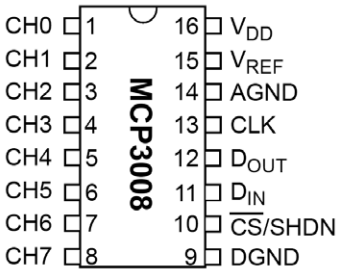


Figure 8.15. Pinout for the MCP3008.

On the right side of the chip, you'll recognize the V_{dd} (the 3.3-V-supply), and the SPI pins CLK, D_{out}, D_{in} and /CS. The AGND and DGND pins may be unfamiliar to you: these are the ground pins, with a separate GND pin for each of the digital and analog sections of the chip. It was done this way because analog signals are much more sensitive to interference than digital signals. For a digital signal it doesn't matter whether the signal is at 2.9 or 3.3 V, since the Raspberry Pi treats both of these as "high". But for an analog signal 2.9 V is very different to 3.3 V. Interference will be much reduced by having separate grounds for the digital and analog sections, which are then connected together inside the chip. We've taken that advice on board in our circuit.

Microchip also states that a 100-nF-capacitor should be added across the digital supply. We already have four on the breadboard (one at each corner), but these are all connected to the analog section, as you'll see later. For this reason we've added a capacitor to the digital section. Since the four capacitors that are always on the breadboard are never shown in the circuit, you'll only see the one capacitor in the circuit.

We start with a circuit with two 10-k-potentiometers so we can test two channels later on. We also happen to need two of them in the next project. It is very easy to make an analog signal with a potentiometer. We connect the potentiometer between GND and +3.3 V. When the wiper is against the GND contact, the voltage at the wiper will be 0 V. When it is against the +3.3-V-contact, the voltage will be 3.3 V, and you can turn the wiper to every possible value in between.

A current will obviously flow through the potentiometer since it's connected across +3.3 V and GND. This current can be calculated using Ohm's law.

$$I = V/R$$

When we put in the known values, we get:

$$I = 3.3/10k = 0.33 \text{ mA}.$$

This is not a problem for the +3.3 V connection on the Raspberry Pi, even when we connect two of them. After all, we have 51 mA available.

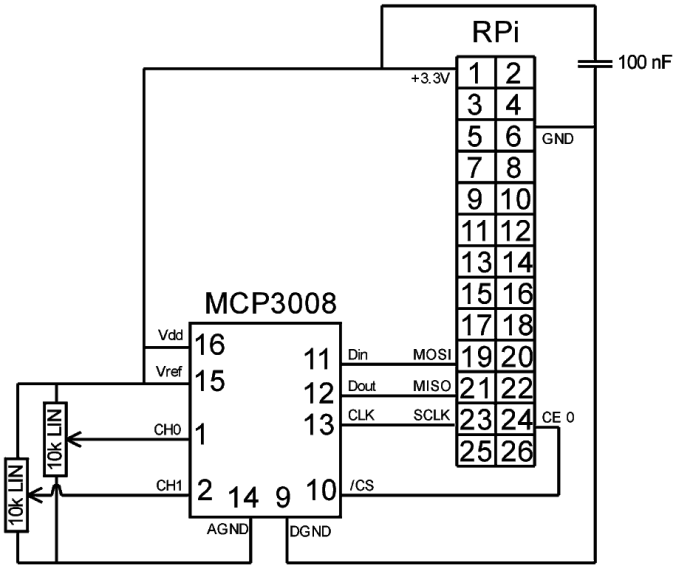


Figure 8.16. Connection of the MCP3008 to the header.

We connect the reference voltage V_{ref} to +3.3 V. V_{ref} is the voltage used to determine the maximum input voltage on the ADC channels. For example, if the analog voltage won't be greater than 1.2 V, you can connect V_{ref} to 1.2 V.[60] The full scale from 0 to 1023 would then be available across the 0-1.2 V range. In our case, 1023 corresponds to 3.3 V.

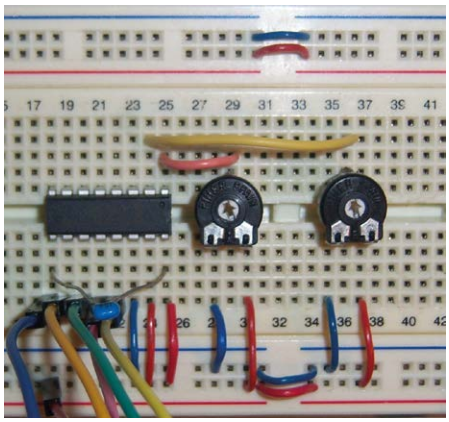


Figure 8.17. Setup on the breadboard.

The communications with the MCP3008 are via SPI, but unfortunately are quite complex.

60 With the help of a voltage reference component, such as the LM113 made by Texas Instruments.

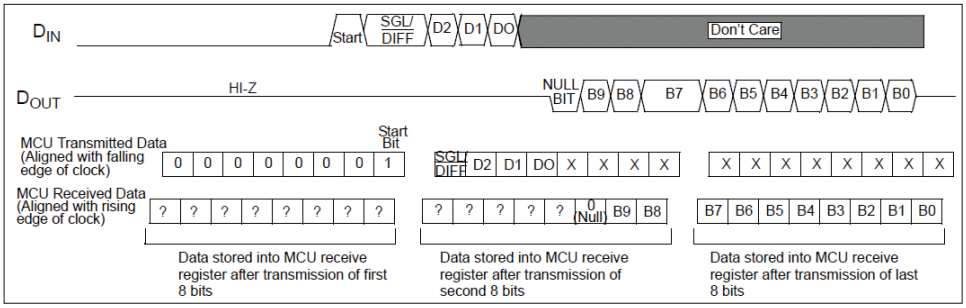


Figure 8.18. SPI communications according to the datasheet for the MCP3008.

In the previous figure from the datasheet for the MCP3008 you can see that the full command consists of three bytes. Since the transmission and reception with SPI happens simultaneously, we will also receive three bytes.

The first byte transmitted contains a 1. The second byte consists of a control bit, followed by three address bits that are used to select which ADC unit will be used. The second half of this byte and all of the third byte don't have any information in them. They are only sent to enable the result to be received. Immediately after the Raspberry Pi has sent important bits of the command, there is a "pause" for one bit, during which the MCP3008 is "thinking". This is followed by a 0, then you get the ADC value in 10 bits.

The control bit determines if we're going to measure the value of a single ADC unit (1), or the pseudo-differential between two units (0).[61] In this project we're measuring the value at one unit, so this bit is a 1. This is the left-most bit in the byte, so it has the value 128.[62]

The address bits simply contain the number of the ADC unit, where the first number is 0 and the eighth number is 7 (since computers start counting at zero). Since the address bits aren't in the right-hand side of the byte, which is usually the case, but have been put four bits to the left, we need to shift the address four bits to the left with the command <<4.

The three bytes to be sent then appear as follows:

```
[1,128+(chan<<4),0]
```

where `chan` is the channel number. Of the three bytes that we receive, we can completely ignore the first one. In the second byte we only need the two bits at the far right. Since the result is 10 bits, these bits are really the left-most bits of a 10-bit number. The values of these bits therefore aren't 1 and 2, but 256 and 512. We therefore have to multiply the second byte by 256. However, we should only multiply bits 0 and 1, and ignore the rest. We don't know what the value of the rest is (the datasheet shows question marks) so we can't just assume they're zero. To select only the two bits on the right we can use the command `&3`.

61 In the next project we'll use the pseudo-differential option.
62 See section 4.3 for bits, bytes and the << operator.

We need all of the third byte, which is just a normal number. Assuming that the result is in the variable "reqadc", we can calculate the result as follows:

```
(reqadc[1]&3)*256 + reqadc[2]
```

The complete program for reading a potentiometer then appears as follows:

```
import spidev
import time

spi = spidev.SpiDev()
spi.open(0,0)

while True:
  chan=0
  reqadc = spi.xfer2([1,128+(chan<<4),0])
  print((reqadc[1]&3)*256 + reqadc[2])
  time.sleep(1)
```

Start the program with `python3 mcp3008a.py`, and turn the potentiometer to see the value change. Stop the program with Ctrl-C.

Since there are eight channels it would be better to create a graphical program, where you could use the mouse to select a channel. A good object for this is the Spinbox. In the next figure you can see what it looks like on the Raspberry Pi.

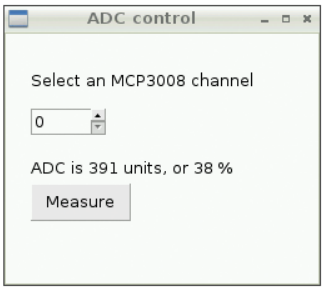


Figure 8.19. Graphical control of the ADC channels.

We make use again of our graphical template and only show you the additional code here.

```
# declare pins
global spi
spi=spidev.SpiDev()
spi.open(0,0)
```

Following the pins we add a Spinbox object. Strictly speaking, the StaticText object isn't part of this, since it's a separate object that we show next to the Spinbox. The spinbox has an initial value that applies when the program is started (value), a position, and a minimum and maximum. The channel numbers are between 0 and 7, so we use these values for min and max.

```
        # spinbox
        wx.StaticText(self, label='Select an MCP3008 channel',
           pos=(20, 30))
        self.spn=wx.SpinCtrl(self, value='0', pos=(20, 60),
          size=(60, -1), min=0, max=7)
```

Once the channel has been selected, the user can click on a button to take the measurement.

```
        # measure button
        btn = wx.Button(self, label='Measure', pos=(20, 120),
          size=(80, -1))
        btn.Bind(wx.EVT_BUTTON, self.OnSelect)
```

This is the text object used to display the result of the measurement.

```
        # static text
        self.txt = wx.StaticText(self, label=' ', pos=(20, 100))


    # your own functions
```

This function contains the same formulae that we saw in the previous program. A new addition is the calculation of the percentage of the full-scale value, which is also displayed.

```
    def OnSelect(self, e):
      chan=self.spn.GetValue()
      reqadc = spi.xfer2([1,128+(chan<<4),0])
      value = (reqadc[1]&3)*256 + reqadc[2]
      pervalue = int(value*100/1024.0)
      self.txt.SetLabel('ADC is '+str(value)+' units, or
    '+str(pervalue)+' %')
```

Start the program with `python3 mcp3008b.pyw`. [63], select a channel (only 0 and 1 have a potentiometer connected) and click on Measure to take a measurement. The channels that don't have anything connected to them will obviously return meaningless results.

63 If you're working remotely via SSH, you should use Xming (see section 3.1.9).

8.6 MCP3008 pseudo-differential measurement

When the first control bit is a 0 instead of a 1, we can use the MCP3008 to perform pseudo-differential measurements. With these we connect a certain voltage to one channel (IN-) and the other channel (IN+) would only provide a measurement when its voltage was above that of the first channel. In effect, the IN- channel acts as a threshold that the IN+ has to overcome.

Control Bit Selections				Input Configuration	Channel Selection
Single /Diff	D2	D1	D0		
0	0	0	0	differential	CH0 = IN+ CH1 = IN-
0	0	0	1	differential	CH0 = IN- CH1 = IN+
0	0	1	0	differential	CH2 = IN+ CH3 = IN-
0	0	1	1	differential	CH2 = IN- CH3 = IN+
0	1	0	0	differential	CH4 = IN+ CH5 = IN-
0	1	0	1	differential	CH4 = IN- CH5 = IN+
0	1	1	0	differential	CH6 = IN+ CH7 = IN-
0	1	1	1	differential	CH6 = IN- CH7 = IN+

Figure 8.20. MCP3008 control bits according to the datasheet.

In this project we use a potentiometer to create the voltage at the IN- channel. This isn't very useful, since we could just as easily have programmed a threshold in the software. But consider that the analog part of your project suffered from interference on the earth. This could cause the earth to be at say 0.2 V instead of the normal 0 V. This wouldn't happen continuously and you wouldn't know when it would happen next. It would be very useful if you could automatically subtract that 0.2 V from your measurement at the exact time of the interference. This is really what the pseudo-differential mode is for. You can implement this by connecting the earth to IN- and to carry out the measurement on IN+. You will only get a measurement back when the signal is above IN-, in other words above 0.2 V.

The only modification we need to make in this program is to change the control bit from normal (1) to pseudo-differential (0).

```
import spidev
import time

spi = spidev.SpiDev()
spi.open(0,0)

while True:
```

```
    mix=0
    reqadc = spi.xfer2([1,mix<<4,0])
print((reqadc[1]&3)*256 + reqadc[2])
time.sleep(1)
```

The variable `mix` is the D2:D0 combination from the previous figure. In the program we've used `mix=0`, which means that channel 0 is IN+ and channel 1 is IN-.

Start the program with `python3 mcp3008c.py`. Set the potentiometer on channel 1 to about 50%, and the one on channel 0 to 0%. Now slowly turn up the potentiometer on channel 0. The value stays at zero until the potentiometer goes over 50%. From then on, the value will increase normally, but will have a maximum of about 512. This is because you're measuring how far above the threshold you are. Since the threshold is at 50% of 1024, you only have 512 left (all depending how accurately you set the first potentiometer to 50%).

The difference between a pseudo-differential and a true differential measurement is that the range becomes smaller with a pseudo-differential one (you can only measure above the threshold), whereas the range always stays the same with a true differential measurement (in this case that would be 3.3 V).

Chapter 9 • I²C

9.1 Introduction to I²C

I²C is an abbreviation for Inter IC Communication, and is sometimes written as I2C.[64] It is a serial protocol developed by Philips in the 80's for communications between ICs in a machine. The I²C bus uses two wires, one for data and one for the clock. For this reason it's often called a 2-wire protocol. The clock line is driven by the master and can have a maximum frequency of 100 kbit per second. Interestingly, the bus has been designed such that both lines are held high by pull-up resistors. These resistors would normally be connected to the bus (often near the master), but in the case of the Raspberry Pi they've been built into the master itself.[65] You therefore don't need any external resistors.

Since the pull-up resistors are inside the master, the I²C bus operates at 3.3 V. This implies that all the slaves also have to be able to cope with 3.3 V signals. They can work at different voltages if need be, since they only pull the bus to ground and never pull it high. The voltage on the bus will therefore never rise above 3.3 V. This makes the I²C bus very flexible, since you can use a mixture of 5 V slaves and 3.3 V slaves with the Raspberry Pi. A 2-slave I²C setup is shown in Figure 9.1.

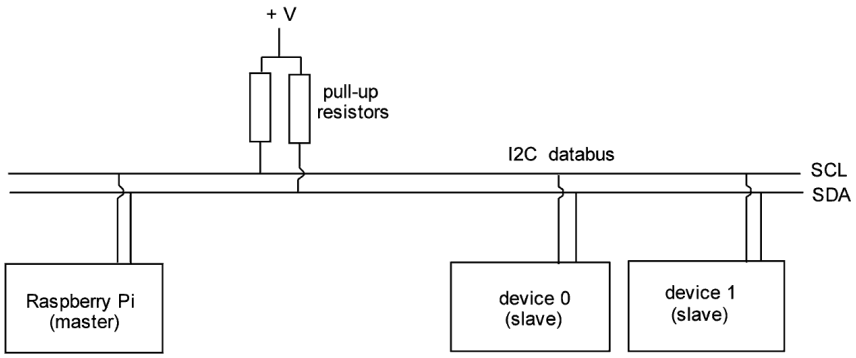


Figure 9.1. Basic I²C setup with two slaves.

Every slave on the I²C bus has its own address. The address can usually be configured by using several pins on the slave to set a few bits of the address. When a pin is connected to ground, it corresponds to 0 and a pin connected to + corresponds to 1. Some devices will have their address hard-coded by the manufacturer. In that case you can choose from a small number of addresses when you order them. The advantage of this is that it keeps the PCB simpler, which reduces its cost.

The master determines the speed of the clock and drives the clock line. However, the slave is able to influence the clock signal. Since the clock line is connected to +V via a resistor, its normal state is high. The master creates a clock pulse by pulling the line low for a short

64 I²C is pronounced as I-squared-C, I2C can be pronounced as I-two-C.
65 The value of these internal resistors is 1k8.

time and then letting it go high again. But the slave is able to keep a low clock line low for longer, so it makes no difference when the master releases the line. This way the slave can force the master to wait a while, for example when the processing of a command from the master takes longer than a single clock pulse.

Communications on the I²C bus are always initiated by the master, and starts with the sending of the address of the slave which it wants to communicate with. All the connected slaves listen for their address, but only the slave with the matching address responds. How the communications proceed from here depends on the commands and data required by the slave, which will be described in the datasheet for that slave.

The following commands can be used for I²C communications: [66]

Command	Description
`import smbus`	Import the I²C module.
`bus = smbus.SMBus(1)`	Create an I²C object and call it "bus".
`bus.write_byte_data(address, reg, data)`	Write data into a register. Where address = I²C address reg = register data = stored in the register
`bus.write_byte(address, data)`	Write data into a register without specifying the register. Where address = I²C address data = stored in the register
`data = bus.read_byte_data(address, reg)`	Read data from a register. Where address = I²C address reg = register data = read from the register
`data = bus.read_byte(address)`	Read data from a register. Where address = I²C address without specifying the register. reg = register data = read from the register
`sudo i2cdetect -y 1`	A command entered via the prompt that shows all I²C devices connected to bus 1, with their address (in hexadecimal).

Table 9.1. I²C commands.

9.2 MCP23008 I/O extender

In this project we'll be using the MCP23008 I/O expander, made by Microchip. This adds a port (8 pins) offering full control over all pins, and includes an interrupt on change. When you don't have enough pins available on the Raspberry Pi, you can use this chip to add more pins.

66 There are also 'word' commands. In that case you use _word_data instead of _byte_data. The data will then be a word (16 bits) instead of a byte (8 bits).

TABLE 1-2: REGISTER ADDRESSES

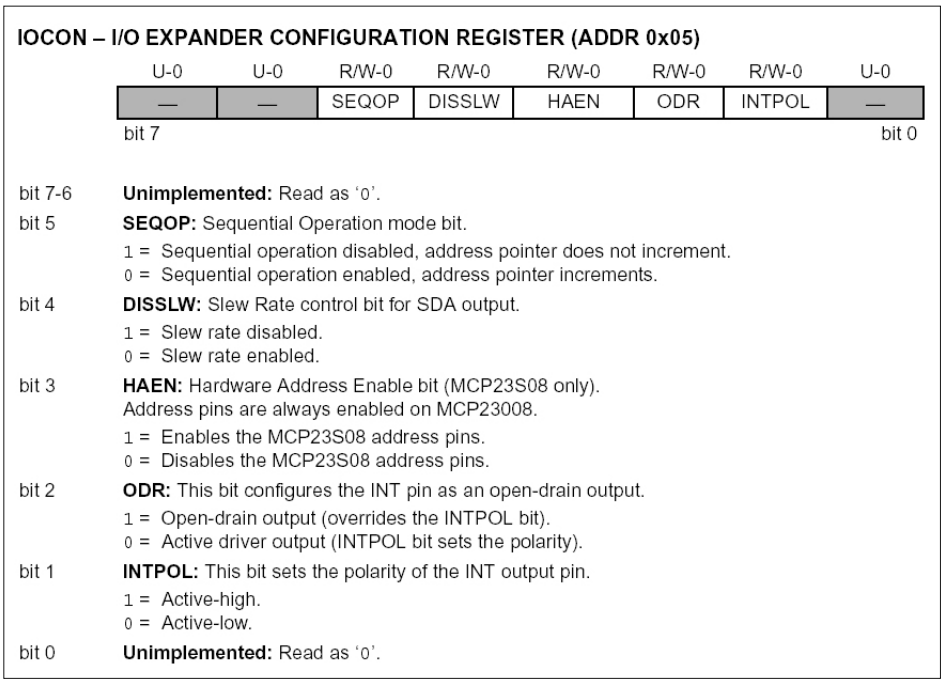
Address	Access to:
00h	IODIR
01h	IPOL
02h	GPINTEN
03h	DEFVAL
04h	INTCON
05h	IOCON
06h	GPPU
07h	INTF
08h	INTCAP (Read-only)
09h	GPIO
0Ah	OLAT

Figure 9.2. Registers in the I/O expander.

The MCP23008 is configured via a set of registers. These registers have the following purpose:

register	description
iodir	Determines whether a pin is an input (1) or an output (0). Bit 0 is for pin GP0, etc.
ipol	Sets the polarity to normal (0) or inverted (1). When inverted is selected, a high signal at the input will be seen as low.
gpinten	Sets the interrupt on change on a pin-by-pin basis (1 is on and 0 is off).
defval	Keeps a value for comparison, which is used by the intcon register.
intcon	Configures whether the interrupt on change is compared with the previous pin value or the value in defval.
iocon	Configuration register; see the figure below.
gppu	Controls if a weak pull-up is connected per pin, making a floating pin high.
inft	Interrupt flag register which shows what interrupt occurred (read only).
intcap	Stores the state of the pins at the time an interrupt occurred (read only).
gpio	Register used to set the pins high or low.
olat	A shadow register for gpio.

Table 9.2. Registers of the I/O expander.



IOCON – I/O EXPANDER CONFIGURATION REGISTER (ADDR 0x05)

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	U-0
—	—	SEQOP	DISSLW	HAEN	ODR	INTPOL	—
bit 7							bit 0

bit 7-6 **Unimplemented:** Read as '0'.

bit 5 **SEQOP:** Sequential Operation mode bit.
1 = Sequential operation disabled, address pointer does not increment.
0 = Sequential operation enabled, address pointer increments.

bit 4 **DISSLW:** Slew Rate control bit for SDA output.
1 = Slew rate disabled.
0 = Slew rate enabled.

bit 3 **HAEN:** Hardware Address Enable bit (MCP23S08 only).
Address pins are always enabled on MCP23008.
1 = Enables the MCP23S08 address pins.
0 = Disables the MCP23S08 address pins.

bit 2 **ODR:** This bit configures the INT pin as an open-drain output.
1 = Open-drain output (overrides the INTPOL bit).
0 = Active driver output (INTPOL bit sets the polarity).

bit 1 **INTPOL:** This bit sets the polarity of the INT output pin.
1 = Active-high.
0 = Active-low.

bit 0 **Unimplemented:** Read as '0'.

Figure 9.3. The iocon register of the MCP23008 I/O expander.

In this project we make an LED come on when you press a switch, where both the LED and the switch are connected to the I/O expander. This means that we have to take the following steps:

1. Set pin 0 as an output, and the rest as inputs.

```
iodir=254 (1111_1110)
```

2. The iocon register of the I/O expander has to be set up as follows:

bit	value	description
bit 7,6	0	N/A.
bit 5	0	Sequential Operation mode.
bit 4	1	Slew rate disabled.
bit 3	0	N/A.
bit 2	0	No open drain output, but normal.
bit 1	1	Interrupt pin goes high when active.
bit 0	0	N/A.

Table 9.3. Register settings.

So: `iocon=18 (0001 0010)`

3. Look at the state of the switch.

```
switch=gpio&2 (0000 0010)
```

4. If necessary, turn on the LED.

```
gpio=1 (0000 0001)
```

And turn it off.

```
gpio=0 (0000 0000)
```

A complicating factor is that these commands have to be given via the I²C protocol. In the hardware section you can see that we selected address 000 by connecting pins A0, A1 and A2 to ground. This address is obviously too short. We still have to add the general of the I/O expander to this. From the following figure taken from the datasheet, you can see that the device number is 0100, which means the complete address is as follows:

0100 000x

The x in the address is the R/W bit, which indicates whether the address is for an I²C read or write operation. This bit is set by the Raspberry Pi itself, so in our program we therefore work with an address that only has 7 bits.

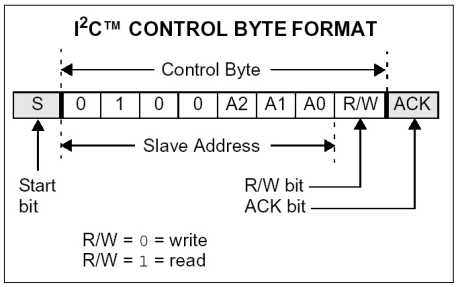


Figure 9.4. Address structure with pins A2:A0.

You could also find out what the address was without looking at the datasheet. If you enter this command:

```
sudo i2cdetect -y 1
```

You would get the following result:

```
pi@raspberrypi ~ $ sudo i2cdetect -y 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: -- -- -- -- -- -- -- --
pi@raspberrypi ~ $
```

The only reaction comes from a device at address 0x20, which is 20 hexadecimal. With the help of the Windows calculator we soon find out what this is in binary: 10 0000. This is (of course) exactly the same as we determined earlier.

We'll now go through the datasheet to see which commands we should use. The datasheet contains a comprehensive overview of the protocol, of which we'll show you the relevant parts.

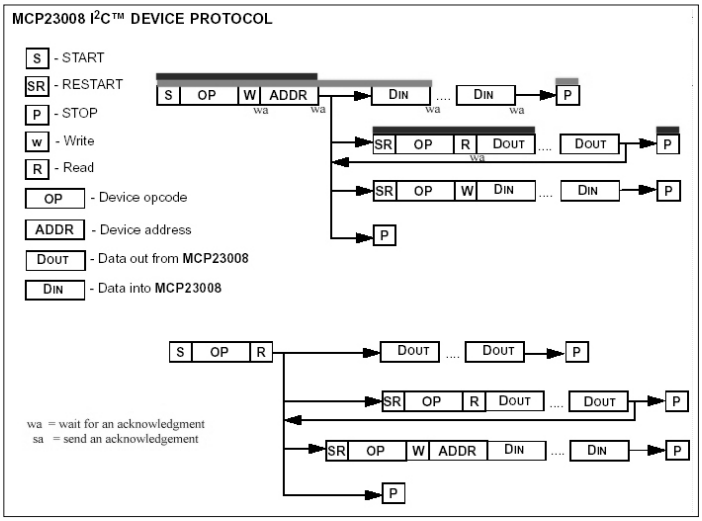


Figure 9.5. I²C device protocol for the MCP23008.

When information is written, it is highlighted with a light gray bar in the figure. It is somewhat confusing that the datasheet calls the address for the chip the "opcode", and the register is called the "address". What it comes down to is that we have to send the address of the MCP23008, followed by the register that we want to change, followed by the data that should be written into the register. The setting up of the iodir register (number 0) is as follows:

```
# set iodir register (0) to 1111_1110
bus.write_byte_data(0x20,0,254)
```

We would also like to be able to do the opposite, in other words, ask the MCP23008 for the contents of a register. As for the previous command, we have to send the address of the chip and the register. But this time we get a result back, which is the contents of the register, and which we can store in a variable. This procedure is shown in the above figure with a dark gray bar. Note that you get the state for the whole port back, which includes any output pins. Figure 9.6 shows the pinout for the MCP23008.

```
# read inputs (register 9)
tmp = bus.read_byte_data(0x20,9)
```

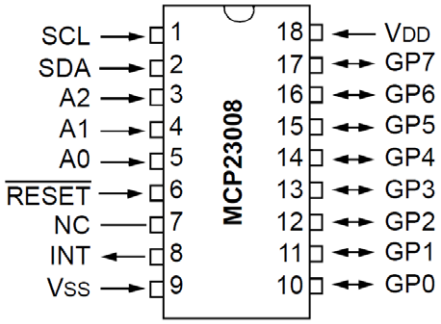


Figure 9.6. Pinout for the MCP23008.

The circuit diagram is shown in Figure 9.7. Pins A2:A0 are used to set part of the I^2C address. Since you can use any address you like, it's possible to connect several of these ICs onto the I^2C bus (up to 8). The address used in this project is 000 (A2, A1 and A0 are connected to ground). The new port added by this chip consists of GP0 to GP7. In this circuit we've made GP0 an output (LED) and GP1 an input (switch). Note that the /reset pin is connected to the positive supply.

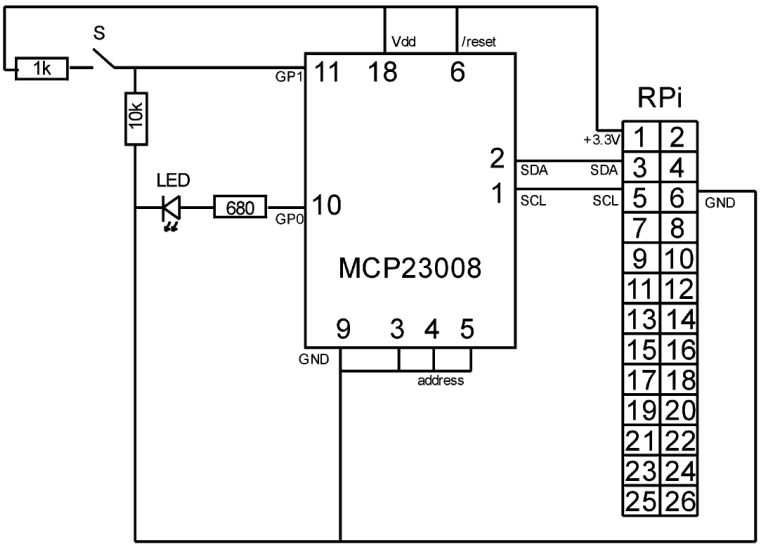


Figure 9.7. Circuit for the de I/O expander.

Again, we've used two resistors near the switch. This is to prevent any damage from occurring to the MCP23008 when you accidentally configure the pin as an output and subsequently cause a short. It is equally important that the MCP23008 is powered by the Raspberry Pi and with a bit of bad luck you may find that a short will also damage the Raspberry Pi. The extra 1k resistor is therefore a good investment. Figure 9.8 shows the circuit built on a breadboard.

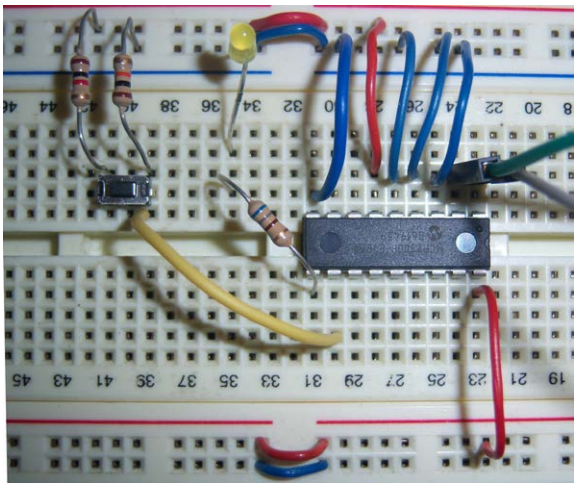


Figure 9.8. Setup on the breadboard.

The program is as follows:

```
import smbus
import time
bus = smbus.SMBus(1)
address = 0x20

# set iocon register (5) to 0001 0010
bus.write_byte_data(0x20,5,18)
#set iodir register (0) to 1111 1110
bus.write_byte_data(0x20,0,254)


while True:
  # read inputs (register 9)
  tmp = bus.read_byte_data(0x20,9)
  switch = tmp & 2
  # light LED if switch is pressed
  if switch:
    bus.write_byte_data(0x20,9,1)
  else:
    bus.write_byte_data(0x20,9,0)
    # may need time to do other stuff
```

Start the program with `python3 mcp23008a.py`. The operation is very simple: turn on the circuit, wait for the settings to be processed and press the button to turn on the LED.

9.3 More power for the extender

In this project the I/O extender took its power from the Raspberry Pi. You do get the extra inputs and outputs, but you're still limited by the maximum current that can be supplied by the 3.3 V connection, which is 51 mA in total. This includes the pins on the Raspberry Pi, the MCP23008 itself, and the pins on the MCP23008. When you need more power, you can also connect the I/O extender to its own 3.3 V power supply. You'll then have all the power from that supply available for the I/O extender and its pins.

The nice thing about I^2C is that both the master and the slaves can only pull the clock line and data line low, or release them. The lines revert to their high state via two pull-up resistors. These resistors are internal to the Raspberry Pi and are connected to 3.3 V, so the voltage on the I^2C lines can't rise above 3.3 V. This means that the I^2C slaves could work at a different voltage to that used on the Raspberry Pi, as long as they can treat 3.3 V on the I^2C lines as high. To demonstrate this, we'll use a 5 V slave in the next project.

You could easily connect the I/O extender to the 5 V supply on the Raspberry Pi. It won't have any effect on the I^2C lines, but the inputs and outputs on the I/O extender will then operate at 5 V. You should therefore take great care that you don't accidentally connect them to any Raspberry Pi pins.

9.4 TC74 digital thermometer

The TC74 is available in both a 3.3 and a 5 V version. We'll use the 5 V version to show you that the voltage doesn't matter in I²C applications.

In the previous project we could set part of the address using three pins, which were connected to 0 V or +. In the TC74 the address has been pre-programmed by the manufacturer. When you order the TC74, you can choose from 8 addresses. This address is part of the part number, as is the operating voltage. For this project we'll be using the following component: TC74 A0 5.0VAT. Figure 9.9 shows how the addresses can be configured.

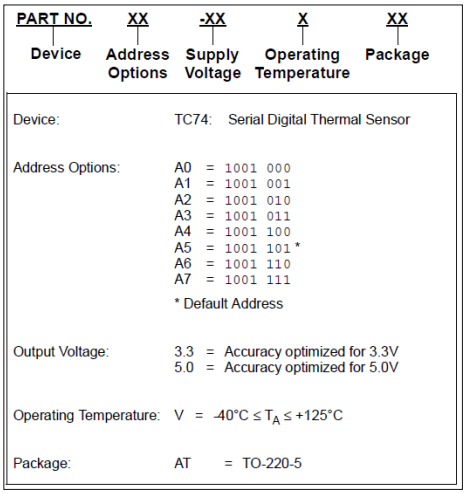


Figure 9.9. Working out the address from the part number.

The above figure comes from the datasheet for the TC74. You can see that our TC74 has address A0 (1001 000), operates at 5.0 V, can be used in temperatures between -40 to +125 °C, and comes in a TO-220-5 package.

When the TC74 has been connected, we can also find out what its address is with:

```
sudo i2cdetect -y 1.

pi@raspberrypi ~ $ sudo i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- 48 -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
pi@raspberrypi ~ $
```

Address 1001 000 is 48 hexadecimal, so the address matches that on the Raspberry Pi.

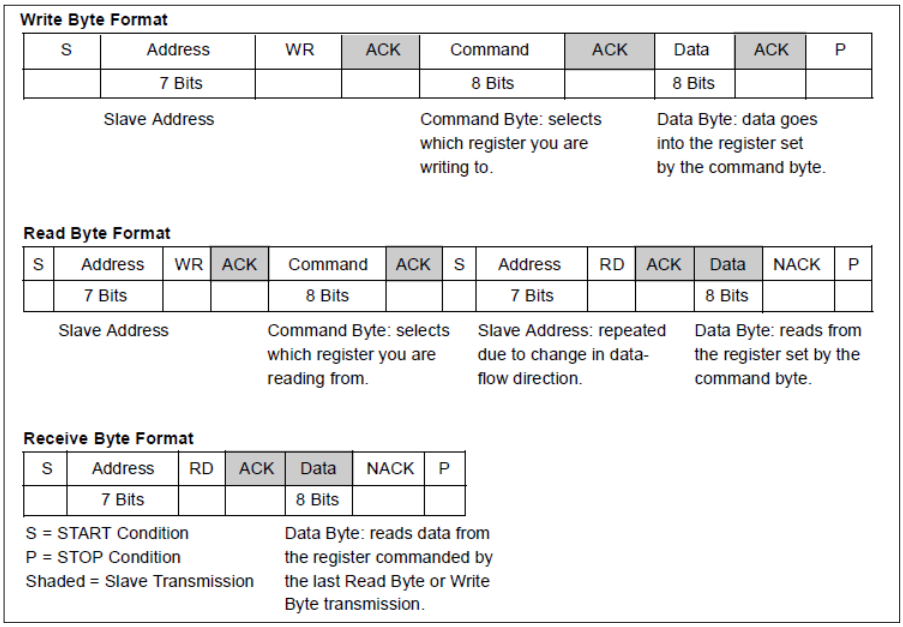


Figure 9.10. I²C communications according to the datasheet.

The communications protocols are again explained in the datasheet. You can see from the bottom command that the simplest way to receive a byte is with a read command that doesn't even need the register number from which we're reading. That sounds great, but we do have to specify the register in an earlier command, as you can see from its description: "reads data from the registers commanded by the last Read Byte or Write Byte transmission".

We cheat a little bit here and write the number 0 to the temperature register (number 0) at the start of the program. We therefore send a 0 to register 0.[67]

```
bus.write_byte_data(0x48,0,0)
```

It's obviously not possible to write anything into that register, but this way everything is set up right to read from it. We can now read the temperature with:

```
bus.read_byte(0x48)
```

The result, which is the temperature, comes in a particular format: two's complement. This means that it is a 7-bit number (rather than the usual 8-bit) where the eight bit acts as a sign bit. This bit is 0 when the number is positive, and 1 when the number is negative. For

67 You may wonder if this is allowed. It is a valid command, which sets up the register for reading later. The TC74 does receive the final zero, but can't do anything with it, and ignores it.

numbers greater than zero (in other words temperatures above zero), you don't have to do anything.

With negative numbers the sign bit will be set to 1, so it appears as a normal number that is greater than 127 (the sign bit is the leftmost bit, which is "worth" 128). Since the TC74 is rated to work at temperatures up to +125°C, you already know that 128 shouldn't be possible. To convert this into the equivalent negative number, you have to invert all bits in the number and add 1.

For example: the result from the TC74 was 201. That is more than 128, so we have to perform the two's complement conversion. We'll show the procedure in binary, since this is easiest to follow:

1100 1001 = 201

invert all bits	`0011 0110`
add 1	`0000 0001`
result	`0011 0111 = 55` (which means -55)

It's very easy to perform this conversion in a microcontroller, but in Python we have to add an extra step. We can invert all the bits with the Python "not" operator at bit level ~. For example, t=~t inverts all the bits.[68] The problem is that Python isn't working with bytes (that contain 8 bits) but with integers, which contain many more bits. And those extra bits are also inverted, which results in a totally different number. You can very easily try this out yourself by starting Python and typing in the above example:

```
pi@raspberrypi:~ $ python3
Python 3.7.3 (default, Apr 3 2019, 05:39:12)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print(~201)
-202
```

You can see that it immediately went wrong since 0011 0110 is 54, and not -202. We therefore have to add an extra step where we remove all the extra bits, leaving just the right-most 8 bits. For this we use the "&" operator. This is like the and operator, but it works at bit level. No matter how big the number is, if we perform an '& 1111 1111' on it, we will be left with just the 8 right-most bits, since the rest will be set to 0. We'll try this out straight away:

```
>>> print(~201 & 255)
54
```

68 See section 4.3 for explanations on 'and' and 'not', for normal as well as bit level operations.

With the help of this method, we can now calculate the two's complement in our program:

```
t = self.bus.read_byte(0x48)
if t>128:
  # below zero
  t = -((~t & 255)+1)
c=str(t)+' '+unichr(176)+'C'
```

We multiply the result by -1 since the temperature is below zero, so it needs a - sign. The degree symbol (°) can't be put directly into the print statement. Instead, we need to use the unicode for the symbol and add it using the `unichr` function. The code for the degree symbol is 176.

We'll write a graphical program and use the standard template for this. We make use of a timer to execute a function every second (1000 ms). We won't start the timer yet since we want to read information from the TC74 within it. However, this can't happen yet because the SMBus module hasn't been loaded. This happens in the next section, where we also set up the register so it can be used for reading. Only then do we start the timer.

Up to now we've always used the standard font and format in a text field, but this time we want large characters (font size 50), which means we have to use the wx.Font instruction.

```
wx.Font(size, font, style, weight)
```

Since this instruction has to be usable on both the Raspberry Pi and Windows, you may not recognize all the font names. You can choose from:

font
- wx.DECORATIVE
- wx.MODERN
- wx.ROMAN
- wx.SCRIPT
- wx.SWISS

style
- wx.ITALIC
- wx.NORMAL
- wx. SLANT

weight (intensity)
- wx.BOLD
- wx.LIGHT
- wx.NORMAL

The only function that we need to add to the template is the `OnTimer` function, which reads the temperature from the TC74, carries out the two's complement conversion (if needed) and shows the result on the screen.

```
    # your own objects

    # timer object
    self.timer = wx.Timer(self)
    self.Bind(wx.EVT_TIMER, self.OnTimer, self.timer)

    # declare pins
    self.bus = smbus.SMBus(1)
    self.bus.write_byte_data(0x48,0,0)
    # start the timer, once per second
    self.timer.Start(1000)

    # static text
    self.text = wx.StaticText(self, label='0',pos=(20,30))
    font = wx.Font(50, wx.DECORATIVE, wx.ITALIC, wx.NORMAL)
    self.text.SetFont(font)

# your own functions
def OnTimer(self, e):
    # read the temperature
    t = self.bus.read_byte(0x48)
    if t>128:
      # below zero
      t = -((~t & 255)+1)
    c=str(t)+' '+unichr(176)+'C'
    self.text.SetLabel(c)
```

Start the program with `python tc74b.pyw` and stop it using the 'x'.

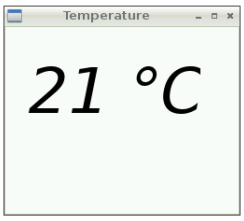


Figure 9.11. The temperature on a graphical window.

When the temperature is exactly between two whole degrees it may happen that the display will show these two values alternately. When you're measuring the temperature of something that varies quickly, and it's important that the program detects this quickly, then there's not much you can do about it. But when you're measuring the temperature of your

living room, for example, then you can avoid this effect by calculating the average value of a series of measurements. An added bonus is that you can now show the temperature with one decimal place.

We'll keep 60 measurements in a list called L. [69] We declare this list in the section where we declare the pins.

```
# declare the list
self.L=[]
```

New measurements are added to the list with the append instruction. These measurements are added to the end of the list. If this makes the list too long (if it contains more than 60 measurements), we remove the measurement that's at the beginning of the list using the `pop(0)` instruction. This way we'll have a list that always contains the last 60 measurements we made. The average temperature is now the sum of all elements in the list (calculated using the `sum` function) divided by the number of elements (calculated using the `len` function). This is called a moving average. Note that we can't just divide by 60, since the list will be shorter when we've just started the program. This means that it will take 60 measurements, or 1 minute, before the display stabilizes. The code we've added to the program is shown in **bold**. You'll notice that the sum is multiplied by 1.0 so that Python will return a result with decimals.

```
def OnTimer(self, e):
  # get temperature
  t = self.bus.read_byte(0x48)
  if t>128:
    # below zero
    t = -((~t & 255)+1)
  # append to the end of the list
  self.L.append(t)
  # remove first if more than 60 elements
  if len(self.L)>60:
    self.L.pop(0)
  # calculate average
  av=sum(self.L)*1.0/len(self.L)
  # round to 1 decimal place
  av=round(av,1)
  c=str(av)+' '+unichr(176)+'C'
  self.text.SetLabel(c)
```

Since we've added two characters to the display, the windows will be too small. For this reason we've changed the width from 250 to 300. Start the program with `python tc74c.pyw`, and stop it with the 'x'.

69 See section 4.3 for more details regarding lists.

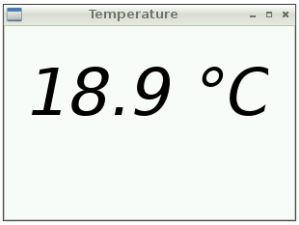


Figure 9.12. The temperature, using averaging.

All that's left is to build the circuit. As shown in Figure 9.13, It's very straightforward, but remember that we're connecting the TC74 to the +5 V pin (number 2 on the header) since we're using a 5 V type. Figure 9.14 shows the circuit built on a breadboard.

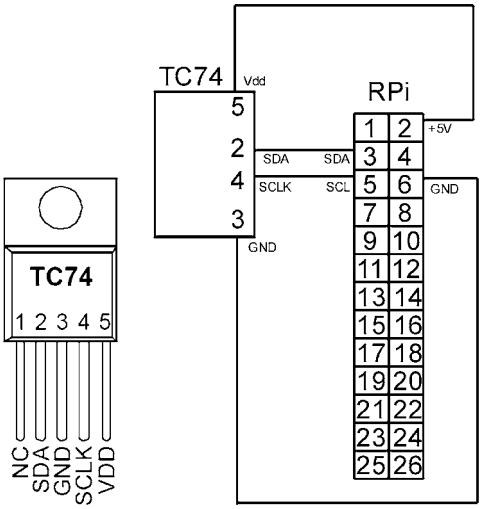


Figure 9.13. Connecting the TC74, note the +5 V pin.

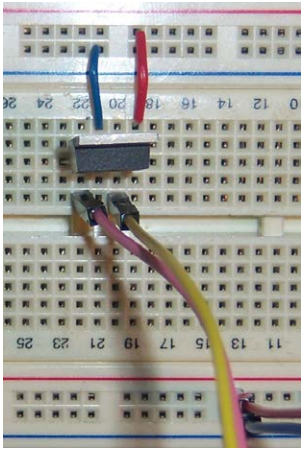


Figure 9.14. Setup on the breadboard.

Chapter 10 • Serial

The Raspberry Pi have two built-in UARTs: a PL011 and a mini UART. They are implemented using different hardware blocks, so they have slightly different characteristics. Since both are 3.3 V devices, extra care must be taken when connecting to other serial communication lines. An adapter must be used to convert the voltage levels between the two protocols so as not to damage either equipment.

On Raspberry Pi's which are equipped with the Wireless/Bluetooth modules (e.g. Raspberry Pi 3, Zero W, 4, etc), the PL011 UART is by default connected to the Bluetooth module, while the mini UART is the primary UART with the Linux console on it. In all other models, the PL011 is used as the primary UART. By default, `/dev/ttyS0` refers to the mini UART and `/dev/ttAMA0` refers to the PL011. The Linux console uses the primary UART which depends on the Raspberry Pi model used. Also, if enabled, /dev/serial0 refers to the primary UART (if enabled), and if enabled, `/dev/serial1` refers to the secondary UART

By default, the primary UART (serial0) is assigned to the Linux console. Using the serial port for other purposes requires this default configuration to be changed. On startup, systemd checks the Linux kernel command line for any console entries, and will use the console defined therein. To stop this behaviour, the serial console setting needs to be removed from command line. This is easily done by using the raspi-config utility by selecting option 5 (Interfacing options) and then P6 (Serial), and select No. Exit raspi-config and re-start your Raspberry Pi.

In Raspberry Pi 3 and 4 the serial port (/dev/ttyS0) is routed to two pins GPIO14 (TXD) and GPIO15 (RXD) on the header. This port is stable and of good quality.

Models earlier than model 3 uses this port for Bluetooth. Instead, a serial port is created in software (/dev/ttyS0). This generates a lot of disadvantages:

1. The baudrate is 1.6 times lower than the set value.
2. The baudrate is influenced by the system load.

At the moment of writing there is no solution to this. At least not one allowing the use of Bluetooth and serial port simultaneously. The projects in this book using the serial port are intended for use with Raspberry Pi 4. It is possible to make the projects usable by switching ports and correct the baudrate on the other side (in this case with the Piccolino). The issues with instability caused by system load are impossible to resolve this way. How to do this correctly has been explained with the relevant projects further down.

Te search for available serial ports use the command:

```
Dmesg | grep tty
```

The last line in the output is:

```
console  [ttyS0] enabled
```

which indicates that the console is enabled on serial port ttyS0

10.1 Introduction to RS232

RS232 was defined as a standard in 1969 for communications between computers and peripherals, usually teletypes (a combination of printer and keyboard that were used before monitors became commonplace). Given its age, it's not surprising that the specifications for the signals look a bit strange. Whereas computers work with +5 V (or +3.3 V) for 1 and 0 V for 0, RS232 uses a negative voltage between -3 and -12 V for 1 and a positive voltage between +3 and +12 V for a 0. The range between -3 and +3 V is not used in order to prevent interference due to long lines.

The connection itself uses up to eight signal lines and a common ground. In the Raspberry Pi only two of these are used, which are TX for transmitting data and RX for receiving data. What one device transmits has to be received by the other, so the signals cross over between the two devices (see Figure 10.1). The term COMMON refers to the common ground, or in the Raspberry Pi just GND.

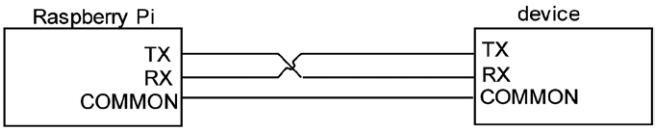


Figure 10.1. Basic serial connection.

One important parameter for these communications is the baudrate. This is a measure of the speed at which the data is transmitted. The projects in this book work at a baudrate of 9600. You can compare it to a type of clock speed, which should make it clear that this isn't a very fast protocol.

In this chapter we start with a serial loopback connection. We then set up a serial RS232 connection at TTL level with a Piccolino. The Piccolino is a rapid prototyping platform with a PIC microcontroller, which is programmed in JAL. [70]

10.1.1 Serial loopback

In this project we're going to create a serial connection with a loopback. This means that the serial output is connected to the input. This way you can quickly check that the software settings and the port work properly. In principle we could connect the TXD and RXD pins on the Raspberry Pi directly together. After all, all possible output signals on the TXD pin will inherently be at the correct voltage for the RXD pin.

70 JAL, and all other development software for the Piccolino can be freely downloaded from the Piccolino page at Elektor.

As usual, we worry what would happen if we made a mistake with the pins. It's quite possible that both pins could be configured as outputs, with one set high and the other one low. With a direct connection this would create a short, which could have disastrous consequences. To avoid that, we put a 1k8 resistor between the pins as shown in Figure 10.2. This won't affect the serial communications, but it neatly avoids damaging the Raspberry Pi when there is a programming error. The maximum current would then be:

$$I = V/R = 3.3/1k8 = 1.8 \text{ mA}$$

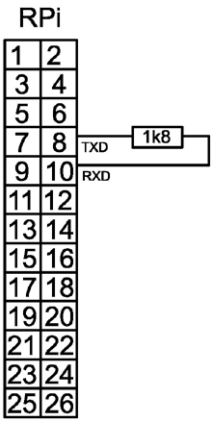


Figure 10.2. Serial loop with 1k8 resistor.

By default the Raspberry Pi's serial port is configured to be used for console input/output. To be able to use the serial port to connect and talk to other serial devices, the serial port console login needs to be disabled. After disabling the console logins, the only way to login to our Raspberry Pi will be via SSH using the Putty terminal emulator.

Disable Serial Port Login on Raspberry Pi 4
The serial port logins can be disabled by the following steps:

- Start the configuration menu:

```
sudo raspi-config
```

- Select Interfacing Options
- Select P6 Serial
- Select No to Would you like a login shell to be accessible over serial?
- Select Yes to Would you like the serial port hardware to be enabled?
- Exit and reboot your Raspberry Pi:

```
sudo reboot
```

Don't forget to go back and enable serial logins after you finish working with the serial port.

To test the serial port, enter the following command to start minicom on your Raspberry Pi 4:

The Minicom communications program is started with the command:

```
minicom -b 9600 -o -w -D /dev/ttyS0
```

Note:
If you are using earlier than model 3, Minicom is started with the command:

```
minicom -b 9600 -o -w -D /dev/ttyAMA0
```

Where the command line parameters have the following meaning: [71]

`b 9600`	The speed of the connection in baud (this is the default RS232-TTL speed for the Piccolino).
`D /dev/ttyAMA0`	The name of the serial port on the Raspberry Pi (/dev/ttyAMA0).
`o`	No initialization of the modem (there isn't one since it is a direct connection).
`w`	Turn on the line-wrap so long lines won't disappear at the right of the screen.

Minicom (Figure 10.3) is controlled fully via the keyboard. With Ctrl-A you get a status line at the bottom of the screen, showing some basic information for the connection. With the status line showing you can also enter a command by pressing a key. The most used ones are:

Ctrl-A	X	exit
	C	clear the screen
	Z	help screen

From now on, everything you type will be sent through the TXD pin and then immediately received via the resistor and RXD pin, and shown on the screen.

71 You can have a look at the manual by typing "man minicom", where man is short for manual.

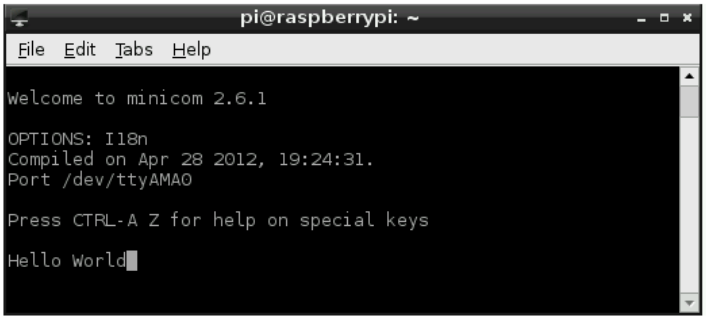


Figure 10.3. Minicom serial communications program.

10.2 Serial connection between Raspberry Pi and Piccolino

As shown in Figure 10.4, RS232 is a serial protocol that uses signals of -12 and +12 V. Neither the Piccolino nor the Raspberry Pi can produce such signals. Instead, they create signals with voltages of 0v and $+V_{dd}$. Since the protocol is still RS232, we call this RS232-TTL. [72] It wouldn't make much sense if we added conversion chips to both the Piccolino and Raspberry Pi, just so that they could produce real RS232 signals instead of TTL. We can just as easily connect them directly together.

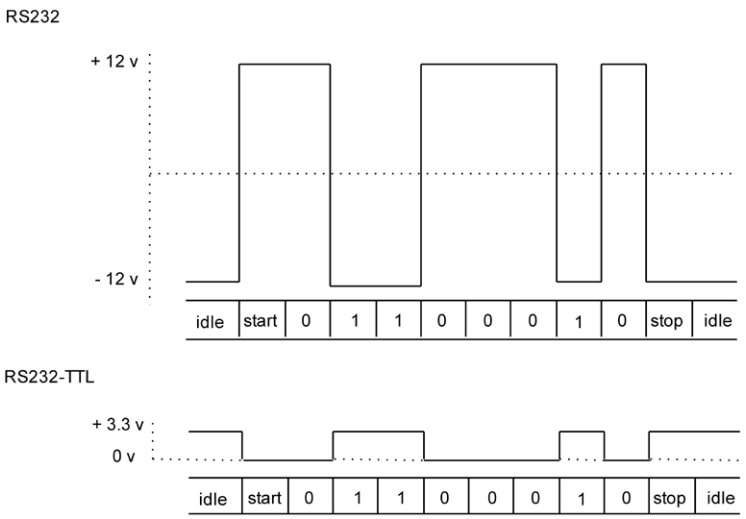


Figure 10.4. Transmitting the letter b using RS232 and RS232-TTL

The Piccolino operates at 5 V, and the Raspberry Pi at 3.3 V, so we need to use a potential divider that reduces the 5 V into 3.3 V on the connection that goes from the Piccolino to the Raspberry Pi. For the potential divider we use two resistors with values of 1k8 (brown-grey-red) and 3k3 (orange-orange-red), which are connected as follows:

72 TTL is the acronym for Transistor-Transistor Logic.

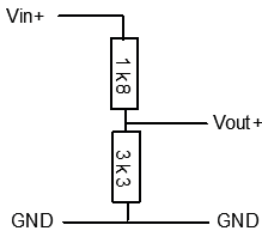


Figure 10.5. Potential divider: from 5 (V_{in+}) to 3.3 V (V_{out+}).

In the above circuit V_{in}+ comes from the Piccolino, and V_{out}+ goes to the Raspberry Pi. The GND is obviously connected to both the Piccolino and the Raspberry Pi. When V_{in} is 5 V, the output V_{out} is calculated as follows:

$$V_{out} = 5 \text{ x } \left(\frac{3k3}{3k3+1k8} \right) = 3.24 \text{ volt}$$

This is slightly less than 3.3 V, which is perfect for the Raspberry Pi. The current drawn from the Piccolino is:

$$I = \frac{5}{1k8+3k3} = 1 \text{ mA}$$

Which is low enough. The error current drawn from the Raspberry Pi (when the input is accidentily programmed as an output) is:

$$I = \frac{3.3}{3k3} = 1 \text{ mA}$$

We don't have to do anything with the connection going the other way, since 3.3 V is interpreted as "high" by the Piccolino. However, we still add a potential divider to that connection for safety's sake. It won't really affect the signal from the Raspberry Pi to the Piccolino. The resistor will reduce the current slightly but that has a no effect on the operation. The potential divider is there to protect the Raspberry Pi, since there is nothing stopping you from running a program on the Piccolino that sets the TTL_{in} pin high, which means that 5 V would go directly to the Raspberry Pi. Under those circumstances the potential divider would reduce that to a safe 3.24 V.

Another programming error you could make is to configure the Raspberry Pi pins as outputs and set them low, and have the Piccolino pins as outputs, but set high. This creates a short, but luckily the resistor of the potential divider stops the current from getting too large. The maximum current would be:

I = V/R = 5/1k8 = 2.8 mA

The Raspberry Pi can still cope with this, and Piccolino hardly notices this current.

The hardware for both projects in this chapter is identical. As shown in Figure 10.6, It consists of the Raspberry Pi, the Piccolino and two potential dividers with resistor values of 1k8 (color code: brown-grey-red) and 3k3 (color code: orange-orange-red). Figure 10.7 shows the circuit built on a breadboard.

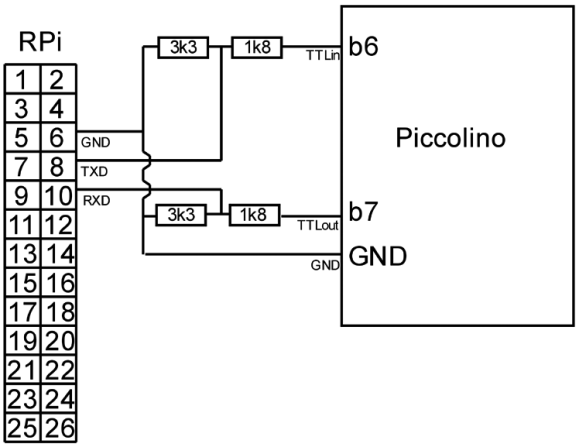


Figure 10.6. Connecting the Piccolino to the header of the Raspberry Pi.

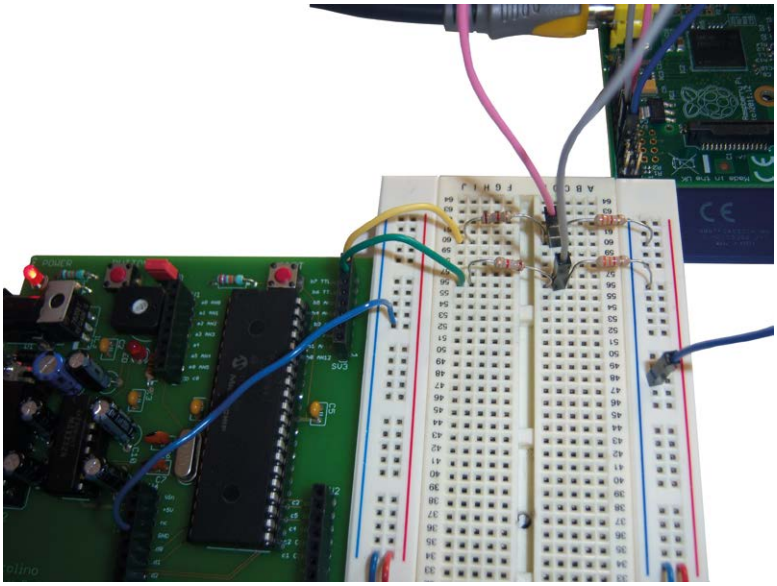


Figure 10.7. Setup on the breadboard.

10.2.1 Serial echo

In this project we get the Piccolino to send everything back that it receives from the Raspberry Pi. This is known as echoing. In the Piccolino we can't use the normal serial commands since these are for the RS232 connection between the Piccolino and your PC. We want to make use of the TTL connection, for which you can use the following commands:

command	description
`serial_sw_read(data)`	Receive information and store it in the variable "data".
`serial_sw_write(data)`	Send the contents of variable "data".
`serial_sw_byte(data)`	Send the byte "data" as (a maximum of) 3 digits instead of a single character.
`serial_sw_printf(array)`	Send a complete array (which first has to be declared as a constant) with a single command. For example: `const byte mystr[] =` "Bert van Dam" followed by `serial_sw_printf(mystr)`

Table 10.1. Serial commands for the Piccolino.

Since we're not making use of the normal RS232 facilities of the Piccolino we don't have a receive buffer either. This means that when something comes in over the RS232-TTL connection when the Piccolino is not actively looking at it, the information will be lost! You should always keep this in mind when you write Piccolino programs that use the RS232-TTL connection. There are two ways round this:

1. The Piccolino "always" waits.

2. The Raspberry Pi waits until the Piccolino asks it to send something, so we can be sure that it is waiting to receive something.

Since we want to echo everything that the Raspberry Pi sends, we have to use the first option and "always wait" for something to be received, and immediately send it back. We therefore enter the following program into the Piccolino:

rpiecho.jal

```
-- JAL 2.4o
include 16f887_bert_piccolino

var byte data
forever loop
  serial_sw_read(data)
  serial_sw_write(data)
end loop
```

Please note: if you're using a standalone microcontroller with the _bert library for this project instead of a Piccolino you have to make a few changes to the _bert library.

- The serial port on the Raspberry Pi is "Idle high", which means that the serial software has to be inverted (`const serial_sw_invert = true` in the _bert library). If you fail to do this then the Raspberry Pi and PIC Microcontroller won't understand each other, and you'll end up with an endless stream of characters that keeps the Raspberry Pi fully occupied, slowing it down to a snail's pace.

- You should also make sure that the serial communications speed has been set to 9600 baud (setting: `const serial_sw_baudrate = 9600` in the _bert library).

Note:
If you are using an Raspberry Pi model 3 or higher, use the following program:

```
const serial_sw_baudrate = 6002 -- 9600
include 16f887_bert_piccolino
var byte data
forever loop
 serial_sw_read(data)
 serial_sw_write(data)
end loop
```

Remove the command `const serial_sw_baudrate = 9600` from the _bert library.

Once you've entered the program in the Piccolino and started it, you can start the Minicom communications program on the Raspberry Pi, with the command:

```
minicom -b 9600 -o -w -D /dev/ttyS0
```

Note:
If you are using earlier than model 3, please start Minicom using the following command:

```
minicom -b 9600 -o -w -D /dev/ttyAMA0
```

When you start Minicom, you may get one or more strange characters, which is normal. From now on, everything you type in will be sent to the Piccolino, which will return it to the Raspberry Pi, where it will be displayed on the screen.

10.2.2 Serial Analog Measurement (ADC)

In this project we'll use the analog hardware of the Piccolino and measure the position of a standard potentiometer connected to the Piccolino. The measured values will be shown in a graph.

The Piccolino program waits for a request from the Raspberry Pi, then carries out an ADC measurement, and sends the result back.

```
include 16f887_bert_piccolino

-- define variable
var byte data

forever loop
  -- wait for request from RPi
  serial_sw_read(data)

  -- measure ADC on pAN and send to RPi
  data=adc_read_low_res(pAN)
  serial_sw_write(data)
end loop
```

Note:
If you are using an Raspberry Pi model 3 or higher, use the following program:

```
const serial_sw_baudrate = 6002 -- 9600
include 16f887_bert_piccolino

-- define variable
var byte data

forever loop
  -- wait for request from RPi
  serial_sw_read(data)

  -- measure ADC on pAN and send to RPi
  data=adc_read_low_res(pAN)
```

```
        serial_sw_write(data)
      end loop
```

Remove the command `const serial_sw_baudrate = 9600` from the _bert library.

The ADC measurement made using `adc_read_low_res` returns a byte, which therefore has a value from 0 to 255. We send this byte directly to the Raspberry Pi. According to the RS232 protocol, we're therefore not really sending a number, but an ASCII character. If the ADC measurement returns 97, for example, and we send that as a single byte, it's really a character with an ASCII value of 97.

Therefore, we're not sticking to the conventions, because this method is much quicker than having to send the individual digits. Since we're writing the program for the Raspberry Pi ourselves, it will be an easy matter to convert the received character into a number, using the Python `ord()` function.

The communications therefore proceed as follows (in this example we still assume that the result of the ADC measurement was 97):

Piccolino	RS232	Raspberry Pi
Take an ADC measurement (with a result of 97), and send it as a byte.	Send the character 'a' (which has an ASCII value of 97).	Receive an 'a'. We know that this is really a byte, so use ord('a') to convert it back to 97.

Table 10.2. Serial communication.

There is a dedicated Python module for serial communications called pyserial. This module adds the following commands to Python:

Command	Description
`open(x, 9600)`	Open port x with a speed of 9600. 73 On the PC x could be 'COM5' for example. On the Raspberry Pi x is always '/dev/ttyAMA0'. You can add an optional time-out value.
`close()`	Close the open port.
`write(x)`	Send x, where x can be a byte (a number from 0 to255) or a string (e.g. "Hello")
`x=read()`	Read one byte.
`x=read(5)`	Read five 5 bytes (unless a time-out occurs before completion)

73 If you don't add anything else, the other parameters default to: Parity: none, Bits:8, Stop bits:1, Flow control:none. These settings are correct for use with the Raspberry Pi, and for the Piccolino. If you want to set up a connection to another device, then this device needs to use the same settings, or you will need to change the settings here.

`a = ser.readline()`	Read a whole line, which should be terminated with a '\n'. You have to be careful when you use this command. If the terminating character isn't sent the program will appear to hang, unless you've used a time-out.

Table 10.3. Serial commands.

We open the port with a time-out of 3 seconds. Imagine what would happen if you didn't turn on the Piccolino, but still sent a character from the Raspberry Pi. The program would now wait "forever" for a reply from the Piccolino, and you couldn't even stop it with the q. The time-out ensures that the program will continue after 3 seconds, even when nothing has been received.

```
import serial
import time

ser=serial.Serial('/dev/ttyAMA0',9600,timeout=3)

while True:
   a=input()
   ser.write('x')
   print('got:', ord(ser.read(1)))
   if a=='q':
      print('Done')
      ser.close()
      break
```

Note:
If you are using an Raspberry Pi model 3 or higher, use the following program:

```
import serial
import time

ser=serial.Serial('/dev/ttyS0',9600,timeout=3)

while True:
   a=input()
   ser.write('x')
   print('got:', ord(ser.read(1)))
   if a=='q':
      print('Done')
      ser.close()
      break
```

Start the program with `sudo python3 serialadc.py`, and press the Enter key. The program now sends the letter x. It doesn't matter which character is sent, as long as something is sent, since that is the signal for the Piccolino to jump into action. The Piccolino replies with the number representing the position of the potentiometer (0 is turned fully anti-clockwise, 255 is turned fully clockwise). You can stop the program by entering q. [74]

Now we know that the communications are working properly we'll write a graphical program that displays the measurements in a graph. [75] We'll use the standard template and only discuss what we're adding to it.

```
        # your own objects
```

We're going to put gridlines on the background, since that looks better. However, this will only work when the window has a fixed format, otherwise the gridlines may be too small, or worse, too large when they'd obstruct the buttons. For this reason we'll create the window with a fixed size, using `SetSizeHints`.

```
        # fixed window size
        self.SetSizeHints(500,320,500,320)
```

Since we want to create a graph, it's to be expected that we need a few variables. The two lists, graphx and graphy, may come as something of a surprise. It turns out that the graph we're drawing isn't permanent. This means that when the user minimizes the window and then restores it, the graph will have disappeared. We're therefore forced to redraw the graph when that happens, which we can only do if all previous measurements have been saved.

```
        global x,y,index,graphx,graphy
        x=0
        y=0
        index=0
        graphx=[]
        graphy=[]
```

This is for the serial connection, again with time-out of three seconds.

```
        # open serial connection
        self.ser=serial.Serial('/dev/ttyAMA0',9600,timeout=3)
```

74 The Piccolino doesn't have a buffer on its TTL connection, but the Raspberry Pi does have one. After you've experimented with the Piccolino it may happen that a character remains in the Raspberry Pi's buffer from a previous experiment. The information will then appear to be delayed. You turn the potentiometer, press Enter, and you get the previous value. When this happens you can clear the buffer by starting Minicom and closing it again. The same also applies to the graphical version with the graph.

75 It's always sensible to try out a new programming technique using a very simple program. This makes it easier to spot (and correct) any errors, before you use the technique in a more complex program.

There are two buttons: one for taking the next measurement, the other for stopping the program.

```
        # button for next measurement
        btn = wx.Button(self, label='Next', pos=(200, 270))
        btn.Bind(wx.EVT_BUTTON, self.OnClick)

        # button to stop program
        btn0 = wx.Button(self, label='Close', pos=(300, 270))
        btn0.Bind(wx.EVT_BUTTON, self.OnClose)
```

Next, we create a background object in which we'll put an image of the gridlines later on. Python keeps track of when windows need to be redrawn, for example if they've been minimized or if another window was in front of it. This happens using the paint event (EVT_PAINT). We also need to use this event to redraw our graph, so we link a function called OnPaint to this event.

```
        # prepare background
        self.dc = wx.ClientDC(self)

        # bind paint event
        self.Bind(wx.EVT_PAINT, self.OnPaint)

        # some text
        self.txt = wx.StaticText(self, label=' ', pos=(20, 280))

    # your own functions
```

We make room on the background for an image with the name a, and draw this against the top-left corner (0,0) of the background. When we call this function, we must therefore pass it an image. This function is called when the program starts (by the `OnPaint` function) and when the screen becomes full up and needs to be cleared again (by the `OnClick` function).

```
    def BackGnd(self,a):
        # load gridlines
        bmp = wx.Bitmap(a)
        self.dc.DrawBitmap(bmp, 0, 0)
```

The following function is linked to the "Next" button. As soon as the user clicks this, the old values for x and y are stored in `oldx` and `oldy`. A new ADC measurement is then requested from the Piccolino, and stored in y. The origin of the y-axis in a graph is at the bottom, but it's at the top in a window. We therefore turn the y-value round by subtracting it from 255 and storing the result. The x-value is increased by 5, so we move a bit to the right. A line from the last point to the new point is then added to the graph. The values for x and y are then added to the lists. The ADC value is then displayed on the screen, followed by the percentage of its full-scale value.

With the x-value incremented by 5 in each step, the window will be full after 100 measurements (after all, it's 500 pixels wide). When that happens, x and the index are reset, the `graphx` and `graphy` lists are cleared, and the background is redrawn with just the gridlines, which clears the graph at the same time.

```
def OnClick(self,e):
  global x,y,index,graphx,graphy
  # previous point
  oldx=x
  oldy=y
  # get new measurement
  self.ser.write('a')
  y = 255-ord(self.ser.read(1))
  # increase counter and draw line
  x += 5
  self.dc.DrawLine(oldx,oldy,x,y)
  index+=1
  graphx.append(x)
  graphy.append(y)
  pery = int((255-y)*100/255.0)
  self.txt.SetLabel('ADC is '+str(255-y)+' units, or
     '+str(pery)+' %')

  # restart when screen is full
  if index>100:
    # reset x, clear and restore background
    x=0
    index=0
    graphx=[]
    graphy=[]
    self.BackGnd("graph.png")
```

When we stop the program, we also close the serial port.

```
def OnClose(self, e):
  # function for the stop button
  self.ser.close()
  self.Close(True)
```

The function `OnPaint` is automatically called by Python when the window has to be redrawn, for example when it was previously obscured by another window. In that case, we should also redraw our graph, since that doesn't happen automatically. We restore the background, put the gridlines on it and redraw the full graph according to the information stored in the graphx and graphy lists. When the program is started, this function is also called, since we need to draw the whole screen from scratch. The lists will still be empty, but the background will be loaded.

```
def OnPaint(self, event):
  # repaint everything
  dc = wx.PaintDC(self)
  # background from file
  self.BackGnd("graph.png")
  # datapoints from the lists
  global counter,index,graph
  for t in range(1,index):
    self.dc.DrawLine(graphx[t-1],graphy[t-
      1],graphx[t],graphy[t])
```

You can see that it is more complex to use graphs (and other images) on the screen, compared to normal programs. However, with the explanations given you should be able to modify the program to your own specifications.

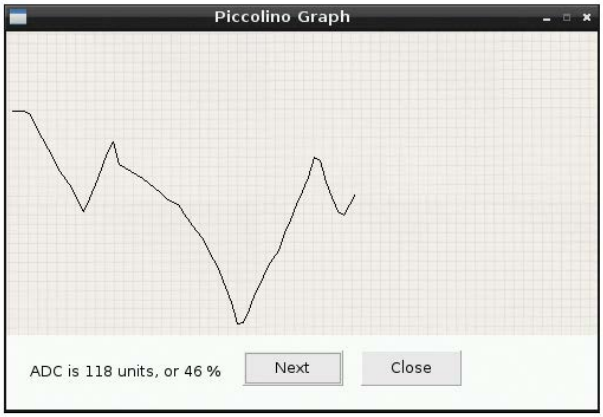


Figure 10.8. Graphical representation of the received measurements.

Note:
If you are using a model earlier than 3, please change

```
/dev/ttyS0
```

into

```
/dev/ttyAMA0
```

Because this is a graphical program that uses some resources of the Raspberry Pi there is a fair chance you will run into issues when you are using the Raspberry Pi 3. Data may be corrupted because of possible variations in baudrate.

Start the program with `python3 serialadc.pyw`. Click on the Next button to get the next measurement. When the screen becomes full, up it will be cleared and the graph starts again. Click on the Close button to stop the program.

10.3 Bluetooth

The Raspberry Pi 4 features a built-in Bluetooth module. Bluetooth on two devices must be paired before they can exchange data with each other.

There are two ways you can enable Bluetooth on the Raspberry Pi 4: using graphical desktop (GUI mode), or using the command mode.

Using the Graphical Desktop

The steps for enabling Bluetooth on the Raspberry Pi 4 using the graphical desktop are given below (assuming that you are in the desktop mode):

- Click on the blue Bluetooth icon on your Raspberry Pi 4 screen at the top right hand side, and turn Bluetooth ON, if it is not already ON. Then, select **Make Discoverable.** You should see the Bluetooth icon flashing.

- Select **raspberrypi** in the Bluetooth menu (**raspberrypi** is the default Bluetooth name of your Raspberry Pi 4) on your mobile device (you may have to scan on your mobile device). You should see the **Connecting** message on your mobile device.

- Accept the pairing request on your Raspberry Pi 4 as shown in Figure 10.9

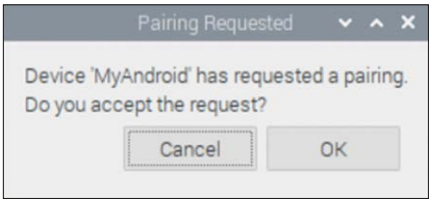


Figure 10.9 Bluetooth pairing request on Raspberry Pi

You should now see the message **Connected Successfully** on your Raspberry Pi 4

Using Command Mode

You can enable Bluetooth on your Raspberry Pi 4 using the command mode. Additionally, you can make Bluetooth discoverable, scan for nearby Bluetooth devices and then connect to a Bluetooth device. The steps are given below (characters types by the user are in bold for clarity):

- Make your Bluetooth discoverable with the following command:

```
pi@raspberrypi: ~ $ sudo hciconfig hci0 piscan
```

- Start the Bluetooth tool on your Raspberry Pi 4 from the command mode:

```
pi@raspberrypi:~ $ bluetoothctl
```

- Turn Bluetooth ON:

```
[bluetooth]# power on
```

- Configure Bluetooth to run:

```
[bluetooth]# agent on
[bluetooth]# default-agent
```

- Make device discoverable:

```
[bluetooth]# discoverable on
```

- Scan for nearby Bluetooth devices, you may have to wait several minutes:

```
[bluetooth]# scan on
```

- Enter command `devices` to see the nearby Bluetooth devices (see Figure 10.10). Make a note of the MAC address of the device you wish to connect to (Android mobile phone in this example) as we will be using this address to connect to the device. An example is shown in Figure 10.10:

```
[Bluetooth]# devices
```

```
[bluetooth]# scan on
Failed to start discovery: org.bluez.Error.InProgress
[CHG] Device B4:CD:27:15:68:7B RSSI: -81
[CHG] Device 7A:DC:6F:D2:AF:62 RSSI: -76
[CHG] Device 7A:DC:6F:D2:AF:62 RSSI: -90
[CHG] Device 7A:DC:6F:D2:AF:62 RSSI: -76
[CHG] Device 7A:DC:6F:D2:AF:62 RSSI: -89
[bluetooth]# devices
Device B4:CD:27:15:68:7B MyAndroid
Device 5D:9B:58:54:77:C4 Our bedroom TV
Device 5F:F0:A5:55:A8:A7 5F-F0-A5-55-A8-A7
Device 7A:DC:6F:D2:AF:62 Our bedroom TV
```

Figure 10.10 Nearby Bluetooth devices

In this example our mobile phone is MyAndoid and the Bluetooth MAC address is: B4:CD:27:15:68:7B

- Pair the device:

```
[bluetooth]# pair B4:CD:27:15:68:7B
```

- Connect to our mobile phone:

```
[bluetooth]# connect B4:CD:27:15:68:7B
```

- Exit from the Bluetooth tool by entering Cntrl+Z

  You can find the Bluetooth MAC address of your Raspberry Pi 4 by entering the following command:

```
pi@raspberrypi:~ $ hciconfig | grep "BD Address"
```

  You can change the Bluetooth broadcast name by the following command:

```
pi@raspberrypi:~ $ sudo hciconfig hci0 name "new name"
```

  To see your Bluetooth broadcast name, enter:

```
pi@raspberrypi:~ $ sudo hciconfig hci0 name
```

  Some other useful Raspberry Pi 4 Bluetooth commands are:

- To reset Bluetooth adapter:

```
sudo hciconfig hci0 reset
```

- To restart Bluetooth:

```
sudo invoke-rc.d bluetooth restart
```

- To list Bluetooth adapters:

```
hciconfig
```

You can find the Bluetooth MAC address of your Android phone as follows:

- Go to Settings menu
- Tap About Phone
- Tap Status
- Scroll down to see your Bluetooth address

Chapter 11 • Web Server (WiFi or Wired)

11.1 Introduction

In this chapter we'll make an intranet web server on the Raspberry Pi so you can use another computer to view HTML pages stored on the Raspberry Pi, run programs or control electronic devices. The web server is mainly meant for use on your own network. It would be perfectly possible to set up the server so it can be accessed via the Internet, so anybody in the world would be able to access your Raspberry Pi. The server on the Raspberry Pi only has minimal security though, so just about any hacker would be able to get into your Raspberry Pi, and that's best avoided!

To try out the projects in this chapter you need at least an Raspberry Pi and a PC, both connected to your Internet router. You will also need to know what the IP address is for your Raspberry Pi. You can refer back to section 3.1.4 to find out how to do this.

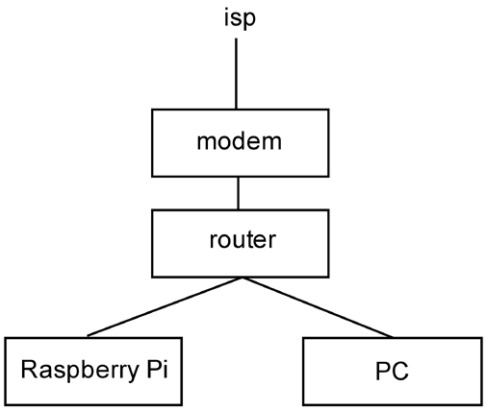


Figure 11.1. Hardware requirements for this chapter.

11.2 HTML server

The first project is a web server that can let you view HTML pages. We'll make a very simple HTML page that consists of just a head with a title and a body with the text "Hello Folks!", which we'll save as index.html. That name is very important: when you access a server without specifying a page name it will automatically serve the file called index.html. In this way, any visitors can always access a website, even if they don't know any page names. For example, when you go to Google you don't need to specify the page name, but it always works.

```
<HTML>
<HEAD>
  <TITLE>Mini Python Server Test Page</TITLE>
</HEAD>
<BODY>

Hello Folks!
```

```
</BODY>
</HTML>
```

We save this file in a dedicated sub-directory of your home directory, with the appropriate name of "server". You can create this directory with the command `mkdir server`. This will keep all server files in one place. Then enter the following command, but note that this should be done from within the server directory!

```
Python –m CGIHTTPServer 8080
```

This command starts the web server, using port 8080. The normal port for a server is generally 80, but on the Raspberry Pi you would need a root account to access this. To get round this we use an alternative, which is often port 8000 or 8080. Here we use 8080, and you can see that the server is now active and is listening on this port:

```
pi@raspberrypi ~/server $ python –m CGIHTTPServer 8080
Serving HTTP on 0.0.0.0 port 8080 ...
```

Now start your Internet browser on your PC (e.g. Internet Explorer or Google Chrome) and go to the website with the following address (you should obviously put in the IP address of your Raspberry Pi PI instead of 192.168.178.9):

```
http://192.168.178.9:8080/
```

The result for this address is the HTML page that we've just created. [76] The title is shown in the tab at the top, and the text is on the page itself.

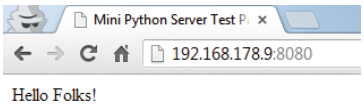


Figure 11.2. HTML page from the web server.

That all looks fine, but when you look back at the Raspberry Pi you'll see that the server gave an error message:

```
pi@raspberrypi ~/server $ python –m CGIHTTPServer 8080
Serving HTTP on 0.0.0.0 port 8080 ...
192.168.178.10 - - [26/Jun/2017 22:28:54] "GET / HTTP/1.1" 200 -
192.168.178.10 - - [26/Jun/2017 22:28:54] code 404, message File not found
192.168.178.10 - - [26/Jun/2017 22:28:54] "GET /favicon.ico HTTP/1.1" 404 -
```

The first line tells us that somebody with an IP address of 192.168.178.10 (my PC) requested a page. But then something else was requested, which didn't exist, resulting in error message 404. The last line shows us which file couldn't be found: favicon.ico. This probably surprises you, since that name didn't occur in our HTML file.

76 The screen shots were produced with Google Chrome running in Incognito mode.

This relates to the icon that's shown to the left of the page title in the browser. In the previous figure this is a piece of paper with a folded corner: this is the default icon that Google Chrome displays when the favicon isn't available from the Internet. Therefore, it was really the browser that asked for this file, independently from you.

You could just ignore this error message and everything will work as it should, but it would be nicer to put an icon file on the server. If you happen to have a program that lets you make such an icon, you could you that, of course. A good alternative is this website, where you can create your own favicon free of charge:

```
http://www.favicon.cc/
```

You can convert an existing image into an icon, or you can create something yourself. I decided to make an orange star with my name above it in blue letters:

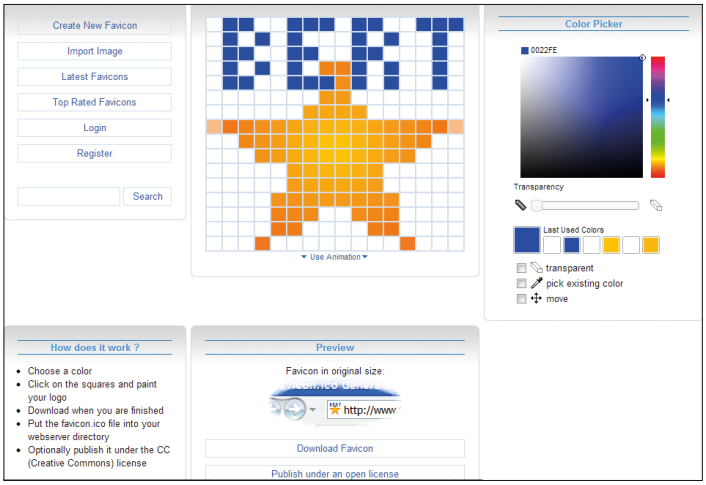


Figure 11.3. Make your own personal favicon.

When you're finished, you can download the file and put it into the server directory, in the same location where you stored your HTML page. You don't have to do anything else: for every HTML page requested, this file will be automatically included. When you refresh the page in your browser you'll see the icon appear at the top.

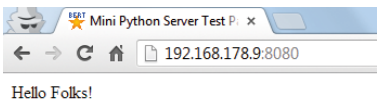


Figure 11.4. HTML page with a favicon on the web server.

You can now add other HTML files to this directory, or add images to your pages. In fact, you can do everything that you could do on a normal Internet server. [77]

77 We carry out these experiments on the Raspberry Pi, but you could just as easily do them on your PC, assuming you installed Python on it. If you have a Python server running on your Windows PC and you want to connect to it from the same PC, you should use

11.3 CGI

CGI is the acronym for Common Gateway Interface, a method that enables a page stored on the server to start a program. This program can in turn create a web page and send it to the browser. The result is a dynamic web page. For example, when you do a search in Google then a program will be started on the Google server that creates a page with your search results. Since this page is specifically for you, with the contents determined by your search terms, it is a dynamic page.

11.3.1 Hello User

We create a new directory in the server directory, called "cgi-bin". Note that you can't choose any other name: the directory must have this name, and it has to be in the server directory! The reason for this is that the files stored in this directory (and only this directory) are treated as programs. The server will therefore try to run the files from this directory as if they were programs. Files stored in other directories aren't treated as programs and will be displayed as html files.

The directory structure on the Raspberry Pi therefore has to be as follows:

server	the directory with HTML files
server\cgi-bin	the directory with Python programs

You should now save the following file in the cgi-bin directory, with the name hello.py.

```
#!/usr/bin/env python3
print("Content-Type: text/html")
print("""
<HEAD>
<TITLE>Mini Python Server CGI Test Page</TITLE>
</HEAD>
<html>
<body>
<h2>Hello World!</h2>
</body>
</html>
""")
```

You can see that this is a Python program. The first line may well come as a surprise to you. When we manually start a Python program such as this one, we type in `python3 hello.py`, so the Raspberry Pi knows beyond any doubt that this is a Python program.

In contrast to a Windows PC, extensions such as ".py" don't mean anything to the Raspberry Pi. When the program is started via CGI we therefore have to let the CGI know in which programming language the file was written. This is the purpose of the first line. It states that the file is a Python program and that Python itself can be found in the directory `/usr/`

127.0.0.1 as the IP address. The requests for the server will then go directly to your server instead of via the router. This is because 127.0.0.1 is the standard IP address for the device itself.

`bin/env`. You should always make sure that you include this line in every Python program that you want to run via CGI.

You can also run the program normally and see what happens (go to the cgi-bin directory and type `python3 hello.py`):

```
pi@raspberrypi ~/server/cgi-bin $ python3 hello.py

Content-Type: text/html

<HEAD>
<TITLE>Mini Python Server CGI Test Page</TITLE>
</HEAD>
<html>
<body>
<h2>Hello World!</h2>
</body>
</html>
```

You can see that the program displays some text, and that this text is in fact an HTML page. Everything that's displayed by a program running in CGI is sent directly to the browser that started the program, rather than to the screen of the Raspberry Pi.

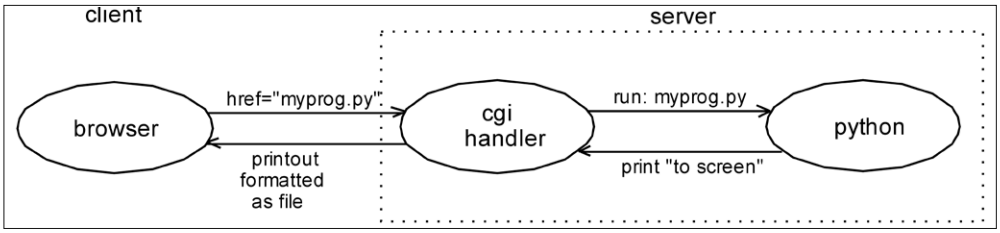


Figure 11.5. Handling of a (Python) CGI request to a web server.

Before we can try this out in practice there is one other thing left to do. Every file that's run via CGI has to have its permissions set accordingly. We need to set the execute bit for the owner. [78] From the cgi-bin directory you should issue the following command:

```
chmod u+x hello.py
```

Return to the server directory and start the server with:

```
python -m CGIHTTPServer 8080
```

78 If you don't know what permissions, execute and owner mean, you should first read section 4.1. If you forget to set the execute permissions for the owner you will see the following error message on the Raspberry Pi: "OSError: [Errno 13] Permission denied". In that case you won't see anything in the browser.

Then open your web browser on the PC and go to:

```
http://192.168.178.9:8080/cgi-bin/hello.py
```

The program hello.py will now start, and everything that the program would have displayed is sent to your browser:

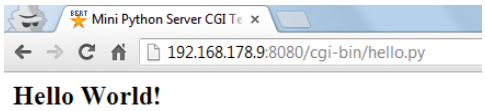


Figure 11.6. A Python program via CGI on the web server.

If you created the hello.py program on your PC, and used WinSCP to transfer it to the Raspberry Pi, it may happen that you get a blank page (and an error message on the Raspberry Pi) when you try to open the file in your browser.

```
192.168.178.10 - - [15/Jun/2017 09:04:27] "GET /cgi-bin/hello.py HTTP/1.1"
200 -
: No such file or directory
192.168.178.10 - - [15/Jun/2017 09:04:32] CGI script exit status 0x7f00
```

This error message tells us that a file or directory doesn't exist, and it appears that it concerns the file hello.py. Assuming that you have created this file and stored it in the correct place, it follows that it must concern a different file. The first line of hello.py could be relevant:

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

This tells the CGI handler where it can find Python. One problem is that Windows and Linux deal with text files in slightly different ways. When you type in a line of text in Windows and press the Enter key, Windows will add two (invisible) characters to the end of the line. <LF> (line feed) to move the cursor to the next line, and <CR> (carriage return) to move the cursor to the beginning of the line. This is the convention for DOS, on which Windows is based.

However, Linux only uses a single character, which is the <LF>. This is the convention for Unix, on which Linux is based. The extra character added by Windows is usually ignored by Linux, so you won't notice the difference when you create a program on your Windows PC and transfer it to a Linux machine using an FTP program such as winSCP. Unfortunately, Linux does look at it when it concerns a file name. As far as Linux is concerned, it doesn't see

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

but

```
#!/usr/bin/env python3<CR>
```

And that program (`python3<CR>`) doesn't exist. You need to convert the program from the DOS convention to the Unix convention before you copy it to the Linux machine with winSCP. The simplest way is to create the file using WinSCP, and then copy the contents of the file on your Windows PC to that file. When you now save the file, the line endings will be automatically correct.

Since you're using WinSCP anyway, you can also use it to set the file permissions. Right-click on the file (when it's been copied to the Raspberry Pi), select properties and put a tick in the X after owner.

An alternative way to correct the line endings is with the program Notepad2 (part of the free download package). Open hello.py in Notepad2. Select File, Line Endings, Unix, and save the file. You can now copy it to your Raspberry Pi.

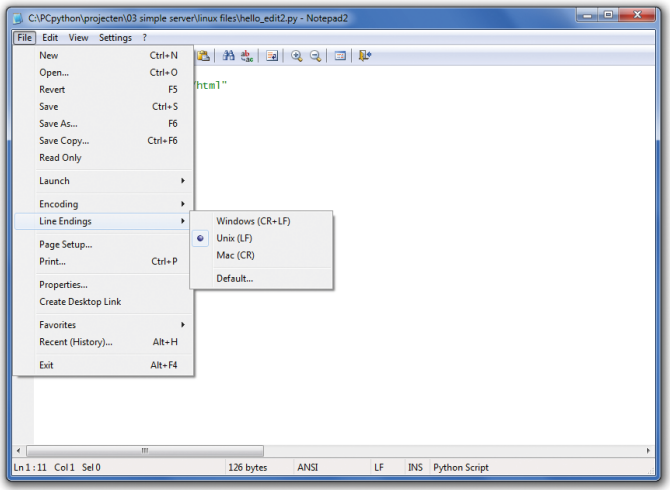


Figure 11.7. Conversion of line endings in Notepad2.

11.3.2 Visitor counter in a file

The program in the previous project didn't really do much, apart from showing some text. We could easily have achieved the same result with a standard HTML file. In this project we make the Python program keep track of a counter. As soon as the program stops the counter value would be lost, so it would always show 1, unless we save the counter value in a file.

We decided to call the file counter.dat and store it in the cgi-bin directory. In the Python program the path for this file is `cgi-bin/counter.dat`. [79]

This seems easy enough, but the file has to exist, otherwise Python will issue an error message and stop the program. As a user we can't see any of that; the only thing we notice is that the page remains blank when an error occurs. We must therefore make sure that the

79 You can also use these programs on a Windows PC. You should bear in mind that Windows uses a backslash \ in paths instead a slash /.

program doesn't stop with an error message when the file doesn't exist. Instead, it should create the file itself.

We'll be using the try/except construct for this. After `try` we try to open the file for reading ("r"). If we succeed, we read the counter value and close the file again. Note that when we read from a file, the result is always a string. Since we want to use it in calculations, we should first convert the string into a number using the `int()` function.

If we couldn't open the file, we would know that this is the first time that we've run the program and we set the counter to zero.

We then increment the counter by one and open the file for writing. If the file doesn't exist yet, this command will create it. If it did exist, the previous contents will be overwritten with the new counter value. Since the file can only contain string, we first convert the number into a string using the `str()` function.

We want to display the value of the counter in the text, and use %s to indicate where it should be displayed. The variable itself is at the end of the print command, which in this case is at the very end of the program, shown as `% number`.[80]

```
#!/usr/bin/env python3
try:
    # if it exists get the number
    myfile = open('cgi-bin/counter.dat','r')
    number=int(myfile.readline())
    myfile.close()
except:
    # else the number is zero
    number=0

# increment the counter and overwrite the file
myfile = open('cgi-bin/counter.dat','w')
number+=1
myfile.write(str(number))
myfile.close()

# show the HTML page
Print("Content-Type: text/html")
Print("""
<HEAD>
<TITLE>Server Counter</TITLE>
</HEAD>
<html>
<body>
```

80 If you don't know what try/except is, or how files work, or how to display variables using %, you should read section 4.3.

```
We have counted %s visits.
</body>
</html>
""" % number)
```

Save this program in the cgi-bin directory, with the name `counter.py`. Remember to change the file permission for execution by the owner, using the command:

```
chmod u+x counter.py
```

Start the server from inside the server directory as you did in the previous projects, and type the following into your browser, remembering to use the IP address for your Raspberry Pi:

```
http://192.168.178.9:8080/cgi-bin/counter.py
```

Every time you click on refresh, it causes the page to be requested again, with the counter increasing by one.

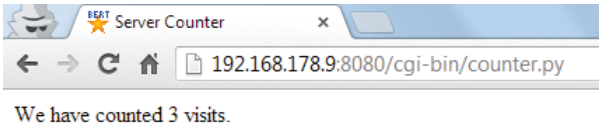


Figure 11.8. Visitor counter.

11.3.3 Passing variables from the browser to the server

It is not possible in the previous program to reset the counter to zero again. If you want to start from scratch, you would have to delete the `counter.dat` file from the Raspberry Pi. This isn't much of a problem, but it would be much better if you could do this remotely. Since we're going to make a change anyway, let's have the ability to set the counter to any value, rather than just zero. We're going to use command-line parameters for this, which you've come across before in project 5.10.

In this case the user has to add one parameter, which is the new counter value. When the user has actually added this, i.e. when it has been read and converted successfully into a number, it will be stored in the file. Otherwise the user will get an error message. Save the program in cgi-bin with the name `setcounter.py`.

```
#!/usr/bin/env python3
import sys

try:
    number = int(sys.argv[1])
    myfile = open('cgi-bin/counter.dat','w')
    myfile.write(str(number))
    myfile.close()
```

```
    comment="Counter reset to %s" % number
except:
    comment="Error: no value received"

# show the HTML page
print("Content-Type: text/html")
print("""
<HEAD>
<TITLE>Set Server Counter</TITLE>
</HEAD>
<html>
<body>
%s
</body>
</html>
""" % comment)
```

You can use `len(sys.argv)` to check how many parameters the user has entered. Remember that parameter 0 also counts, so when the user enters 1 parameter the length of `sys.argv` will be 2. We could also have written the program like this:

```
if len(sys.argv)==2:
    number = int(sys.argv[1])
    myfile = open('cgi-bin/counter.dat','w')
    myfile.write(str(number))
    myfile.close()
    comment="Counter reset to %s" % number
else:
  comment="Error: no value received"
```

The advantage of using try/except is that it will trap all possible errors. For example, if the user entered some text such as "me" instead of a number, it would result in the error:

```
valueError: invalid literal for int() with base 10: 'me' .
```

The question now is how we can pass this value in the browser, since we're not permitted to use a space in the address. Instead of a space, you can use a question mark (?) in front of the first parameter. If you wanted to set the counter to 34 you would type the following:

```
http://192.168.178.9:8080/cgi-bin/setcounter.py?34
```

The figures below show the results for a correct as well as an incorrect (or missing) parameter.

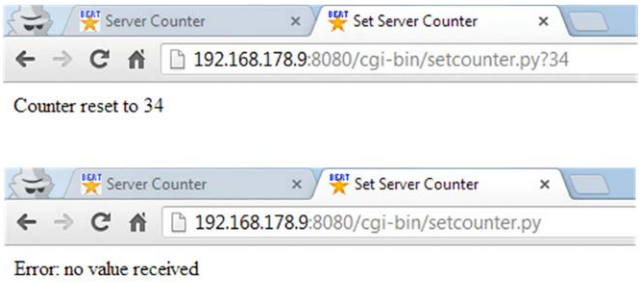


Figure 11.9. Reset the counter and an error message.

Unfortunately you can only use the question mark once in the address line, which means you can only pass one parameter. A simple solution is to put all parameters in one big parameter. It is customary to separate the parameters with the & character. The resulting parameter is then split into its component parts with the `split()` command.

```
#!/usr/bin/env python3
import sys

# get arguments
s=sys.argv[1]

# split them into a list and add them as integers
x=0
for t in s.split('&'):
    x=x+int(t)

print("Content-Type: text/html")
print("""
<HEAD>
<TITLE>Arguments Sum</TITLE>
</HEAD>
<html>
<body>
The sum of the arguments = %s
</body>
</html>
""" % x)
```

Save this program in the cgi-bin directory with the name `sum.py`. Remember to give the program 'execute' permissions. Type the following command into a browser:

```
http://192.168.178.9:8080/cgi-bin/sum.py?1&20&8
```

The parameters will now be added up, with the result shown on the page.

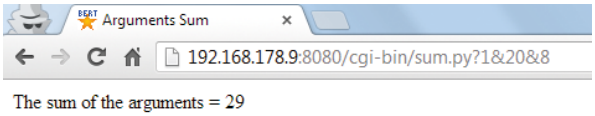


Figure 11.10. Several parameters have been added up correctly.

11.3.4 GPIO

We can use the same method to control any hardware connected to the Raspberry Pi. In this project we use the setup with an LED and a switch from Figures 11.11 and 11.12.

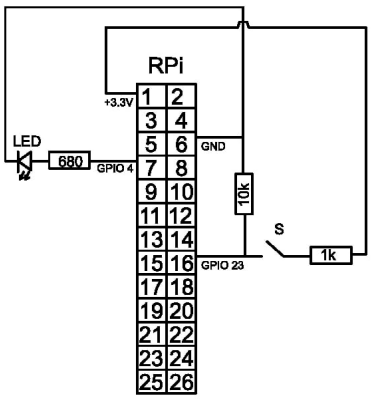


Figure 11.11. Connection of a switch and LED to the header.

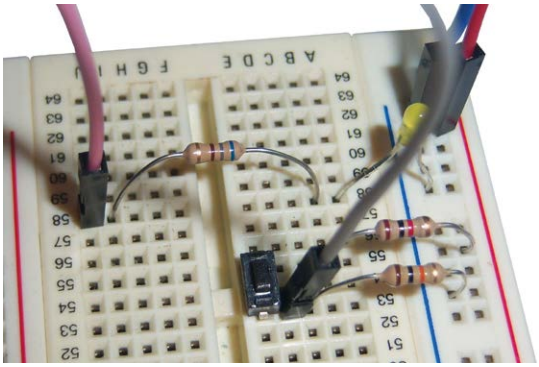


Figure 11.12. Setup on the breadboard.

The first part of the Python program is identical to that in project 5.6. The second part of the program is our 'standard' HTML page program.

```
#!/usr/bin/env python3
import sys
from gpiozero import InputDevice

# set pin direction
```

```
switch = InputDevice(23)

if switch.value == True:
    comment= "The switch is closed"
else:
    comment= "The switch is open"

# show the HTML page
print("Content-Type: text/html")
print("""
<head>
<title>Check Switch on GPIO23 Status</title>
</head>
<html>
<body>
%s
</body>
</html>
""" % comment)
```

Save the program in the cgi-bin directory with the name `gpioswitch.py`, and remember to set the 'execute' permissions with:

```
chmod u+x gpioswitch.py
```

If the server isn't running, you should go back to the server directory and restart it:

```
Python3 -m CGIHTTPServer 8080
```

Using your browser you can now view the switch status by going to the page:

```
http://192.168.178.9:8080/cgi-bin/gpioswitch.py
```

You should get a similar image as shown below.

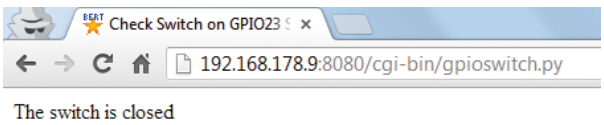


Figure 11.13. Viewing the switch state in the browser.

We can go about the same way to control an LED. We will write a program that allows us to switch an LED on and off via the webbrowser by using the argument "1" for switching it on and "0" for switching it off. The first part of the Python program is identical to that of section 5.2 where we control an LED without a browser.

The second part of the program is based on the program from the previous project. First we check whether an argument has been given. If this is the case we execute the requested instruction by switching the LED on or off. The action taken is then presented as text in the variable `comment`. If there is no argument we will display an error in this variable. This method allows us to use the same HTML page in multiple situations. The only difference would be the content of the variable `comment`.

```
#!/usr/bin/env python3
import sys
from gpiozero import LED
from time import sleep

# set pin direction
led = LED(4)

try:
  number = int(sys.argv[1])
  if number==1:
    led.on()
    comment= "The LED is now on"
  else:
    led.off()
    comment= "The LED is now off"
except:
  comment="Error: no command received"

# show the HTML page
print("Content-Type: text/html")
print("""
<head>
<title>Control GPIO4 LED</title>
</head>
<html>
<body>
%s
</body>
</html>
""" % comment)
```

In fact, the program only looks for a parameter of 1, or another number. When it's equal to 1 it will turn on the LED, with any other number it will turn the LED off. If you add a parameter that isn't a number, or nothing at all, the program responds with "Error: no command received".

Unfortunately this program doesn't work. Well, it does work, but you don't see it working. When the program halts the LED will turn off. You can test this by adding the command

`sleep(10)` at the bottom. Save the program as `gpioledtest.py` in the cgi-bin folder[81]. You can turn on theLED by navigating to the page using the argument "1":

```
http://192.168.178.9:8080/cgi-bin/gpioledtest.py?1
```

Your webbrowser will show the LED status and the LED that you connected to the Raspberry Pi will be switched on.

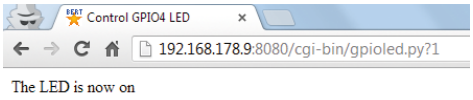


Figure 11.14. Browser menu with hyperlinks.

The LED will switch off automatically after 10 seconds.

If this is not how it's supposed to be done, what is the correct way? We need a program that runs on the Raspberry Pi and controls its GPIO pins. The program should never stop, so the outputs remain in their "on state" indefinitely. The program checks a file on the SD card to determine which outputs should be activated.

The user can start a CGI program from anywhere on the internet via a web page, just like we did in the example above. This program doesn't control GPIO pins directly, but puts information in the file which is then read out by the other program. The programs communicate via this file: the first puts in information, the second reads out the information.

Of course we have to decide on what will be in the file and what it implicates. To keep it simple we will use a short one character long code. This gives us 36 possible entries, 26 letters and 10 numbers. If this is too little for you it is of course possible to use more characters.

Code	Meaning
0	LED on gpio 4 off
1	LED on gpio 4 on

We start with the program that receives command from the user and puts them in a file. This is in fact a combination of the two programs of the previous projects. Save the program as `gpioled.py` in the server/cgi-bin folder and ensure the correct authorisations.

```
#!/usr/bin/env python
import sys
from gpiozero import LED

try:
  number = int(sys.argv[1])
  myfile = open('cgi-bin/gpiocommand.dat','w')
```

81 Because this program isn't functional, it is not included on the SD card. In case you would like to try it, you will have to type it yourself. Don't forget to edit the execution rights using `chmod u+x gpioledtest.py`.

```
  myfile.write(str(number))
  myfile.close()
  comment="Error: no valid command received"

  if number==1:
    comment= "The GPIO4 LED is now on"
  if number==0:
    comment= "The GPIO4 LED is now off"
except:
  comment="Error: no command received"

# show the HTML page
print("Content-Type: text/html")
print("""
<head>
<title>Control GPIO4 LED</title>
</head>
<html>
<body>
%s
</body>
</html>
""" % comment)
```

The server program reads the file and executes the command that is in it. We use the `try/except` construction because the file may possibly not even exist yet (because the user hasn't issued a command yet, for example). It is also possible that the file is unaccessible because it is being written. The `try/except` construction makes sure the program keeps running. There is a short sleep command to enable the Raspberry Pi to run other programs. Save the following program as `gpioserver.py` in the server/cgi-bin folder.

```
#!/usr/bin/env python
from gpiozero import LED
from time import sleep

# set pin direction
ledGPIO4 = LED(4)

while(1):
  try:
    # if it exists, get the command
    myfile = open('gpiocommand.dat','r')
    number=int(myfile.readline())
    myfile.close()

    # execute the command
```

```
    if number==1:
      # LED GPIO4 on
      ledgpio4.on()
    if number==0:
      # LED GPIO4 off
      ledgpio4.off()

  except:
    # command file doesn't exist
    pass
  sleep(0.1)
```

Note that the path to the command file is not mentioned in the `gpioserver.py` program.

```
myfile = open('gpiocommand.dat','r')
```

In the `gpioled.py` program the path is mentioned:

```
myfile = open('cgi-bin/gpiocommand.dat','w')
```

It is only logical that a path description is not used in the server program. The program is located in the same folder as the file and so it is not necessary to enter the full path. So it is a bit surprising that the `gpioled.py` program does use a path description, although it is in the same folder. But it is necessary because the program is started from within the server folder and by consequence runs from this folder. Seen from the server folder the file is located in a different folder, namely the cgi-bin folder. We have also seen this with the visitor counter program in project 11.3.2.

Starting requires an additional step:

1. CD to the server folder and start the server:

   ```
   cd ~/book/11/server
   python -m CGIHTTPServer 8080
   ```

2. Open a second terminal (or a second Putty session if you're working through SSH), CD to the CGI-bin folder and start the gpioserver program:

   ```
   cd ~/book/11/server/cgi-bin
   python gpioserver.py
   ```

3. You can now turn on the LED by browsing to the following address:

   `http://192.168.178.26:8080/cgi-bin/gpioled.py?1`

You can disable the LED by sending a "0" instead of a "1". Please note that your browser replies the the LED is on, but that in fact it only means that the correct command is written into the file. If you forgot to start the `gpioserver` program, the LED will not be switched on.

In case you want to be sure that the LED is switched on, you can use a second file to have the `gpioserver` program report to `gpioled.py` and thus the user about the LED status. This isn't difficult, but it makes the situation a bit more complex. Because this exercise is not intended to be a professional solution, we don't care about it and leave this addition to you.

11.3.5 GPIO with JavaScript

The GPIO programs in the previous projects worked very well, but it is inconvenient that you have to remember exactly how they work and what, if any, parameters they need. One way round this would be to have a page with a few links, so all you needed to do was to click on the link.

The page would look like this in HTML. Save this page on your Windows PC in a directory of your choice, with the name `gpiocontrol1.html`. Remember to replace `192.168.178.9` with the IP address of your own Raspberry Pi.

```
<HTML>
<HEAD>
<TITLE>GPIO Control</TITLE>
</HEAD>
<BODY BGCOLOR="#ffffff">

<P><B>RPi actions</B></P>

<P><A HREF="http://192.168.178.9:8080/cgi-bin/gpioled.py?1">
LED on GPIO4 on</A></P>

<P><A HREF="http://192.168.178.9:8080/cgi-bin/gpioled.py?0">
LED on GPIO4 off</A></P>

<P><A HREF="http://192.168.178.9:8080/cgi-bin/gpioswitch.py">
Check the status of the switch on GPIO23</A>

</BODY>
</HTML>
```

Open this page in your browser by double-clicking it. Your browser should now display the following:

Figure 11.15. Browser menu with hyperlinks.

This works perfectly and makes life a lot easier. Now imagine what would happen if you connected an LED to every GPIO pin on the Raspberry Pi. This page would soon become quite a mess. A better solution is to use JavaScript.[82] Save this page on your Windows PC in a directory of your choice, with the name `gpiocontrol2.html`. Remember to replace `192.168.178.9` with the IP address of your own Raspberry Pi.

```
<HTML>
<HEAD>
  <TITLE>JavaScript GPIO Control </TITLE>
  <SCRIPT language="JavaScript">
  function setled()
  {
         if (document.javamenu.ledbox.checked)
            {
            pagename='http://192.168.178.9:8080/cgi-
     bin/gpioled.py?1'
            }
         else
            {
            pagename='http://192.168.178.9:8080/cgi-
     bin/gpioled.py?0'
            }
         window.open(pagename, 'Window1',
    'resizable,height=40,width=300,top=150,left=150');
  }
  function getswitch()
  {

         pagename='http://192.168.178.9:8080/cgi-bin/gpioswitch.py'
         window.open(pagename, 'Window1',
resizable,height=40,width=300,top=150,left=150');
  }

  </SCRIPT>
</HEAD>
```

82 You can find a short introduction to JavaScript in section 4.4, which will help you understand how this program works.

```
<BODY BGCOLOR="#cccccc" TEXT="#000099">

<FORM NAME="javamenu">
<BLOCKQUOTE>
  <P><FONT SIZE="+3">GPIO Control</FONT></P>
  <P>Control LED<!--SELECTION--><!--/SELECTION-->
   <TABLE WIDTH="168" BORDER="0" CELLSPACING="2"
   CELLPADDING="0">

    <TR>
      <TD COLSPAN="3">
       <INPUT TYPE="checkbox" NAME="ledbox" >
      GPIO 23</TD>

    </TR>
  </TABLE><INPUT NAME="name" TYPE="submit" VALUE="Send"
onClick="setled()"></P>
  <P>Read Switch</P>
  <P><INPUT NAME="name" TYPE="submit" VALUE="Send" onClick="getswitch()">
</BLOCKQUOTE>
</FORM>

</BODY>
</HTML>
```

You can recognize the standard JavaScript structure as described in section 4.4. The page starts with two functions, which are `setled()` and `getswitch()`. The `setled()` function checks if the ledbox (a checkbox) object is ticked. The `gpioled.py` program is then called with a parameter that depends on the previous result. What is interesting is that a small window now opens that shows the reply from the Raspberry Pi. Since this small window was opened as 'window1' it was created as a new window on top of the old one. This keeps the main page visible, and it isn't replaced either. This makes everything look pretty neat.[83]

The second function is `getswitch()`. All this does is to call the `gpioswitch.py` program and display the result in a small window.

The three objects in the second part of the program are a checkbox and two buttons in a minimal implementation. They're all within a blockquote and table to make sure that we have a tidy layout, as you can see from the next image, where we used Chrome on a Windows machine.

83 If you view this page on the Raspberry Pi using web browser, you will get a new tab instead of a small window.

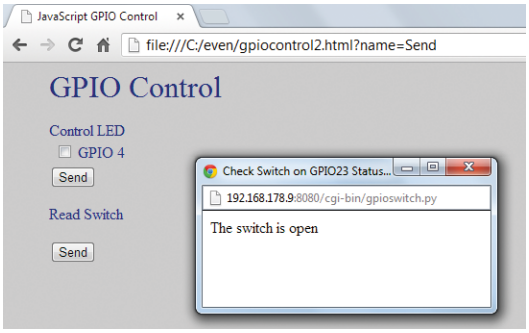


Figure 11.16. Browser menu using JavaScript (showing the switch status window).[84]

11.3.6 I²C Fridge alarm with automatic web page

In this project we'll make a fridge alarm. This is an alarm that is triggered when the temperature inside the fridge becomes too high. We want the program to run on the Raspberry Pi (which will be near the fridge), and have the ability to check the state of the fridge via a web page. We have to consider the following aspects:

1. We need a standard program that keeps an eye on the temperature via an I²C sensor, and turns on an LED if necessary. This program runs continuously, since you also want the alarm to go off when nobody watches the web page. The program also writes the last measured temperature to a file.

2. A second program is started via a web browser. This program reads the file created by the first program, and displays the retrieved temperature on an HTML page. This program only runs when somebody is looking at the web page.

3. Since the temperature is liable to change, it's not enough to view the temperature once only. The page on the web browser will therefore be automatically refreshed every couple of seconds.

The first program is an adaptation of the program used in project 9.4. We use the non-graphical version and add the temperature averaging to it. Since we're taking a sample every 0.3 seconds, we've reduced the number of samples used for calculating the average from 60 to 10. The temperature in the fridge will vary more quickly than that of the room, and if we would use a long sequence of samples this wouldn't be very visible. The average temperature is written to a file.

It can happen that the program tries to write to the file, just at the same time as another program is using the file, which results in an error. In this case, the `try/except` construct will fail and the program won't update the temperature in the file. En error message is then shown on the Raspberry Pi so you know that this problem occurred. Since both programs access the file at different rates, this problem should rarely happen, statistically speaking.

84 With Midori on the Raspberry Pi, the status appears in a new tab instead of a small window.

We've also added an LED to GPIO 4 (pin 7 on the header), which turns on when the temperature rises too much. This is rather feeble as an alarm, but you can always apply what you've learnt in chapter 6 and add a noisy alarm bell.

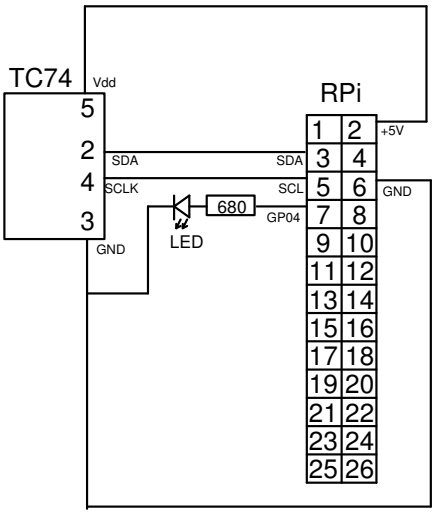


Figure 11.17. Connecting the TC74 and a local alarm LED.

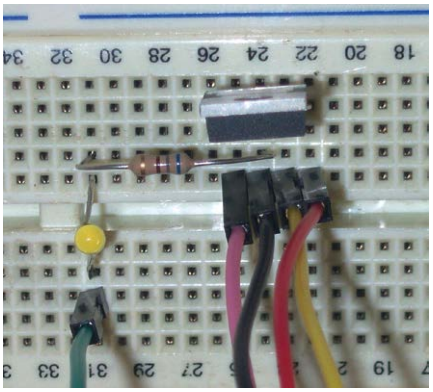


Figure 11.18. Setup on the breadboard.

The ideal temperature inside a fridge is between 1 and 4 degrees, depending what you keep in it, and where you measure the temperature. The alarm goes off at 4 degrees. If you want to use a different temperature, you will have to change its value in both programs.

```
#!/usr/bin/env python3
import smbus
import time
```

```
import os
import sys
import wiringpi
from gpiozero import LED

# set pin
led = LED(4)

# set the I2C bus
bus = smbus.SMBus(1)
bus.write_byte_data(0x48,0,0)
L=[]

while True:
    # get temperature
    t = bus.read_byte(0x48)
    if t>128:
      # below zero
      t = -((~t & 255)+1)
    # calculate average over 10 measurements
    L.append(t)
    # remove first if too long
    if len(L)>10;
      L.pop(0)
    # calculate
    av=sum(L)*1.0/len(L)
    # round to 1 decimal
    av=round(av,1)
    # write to file if possible
    try:
      # file available
      myfile = open('temperature.dat','w')
      myfile.write(str(av))
      myfile.close()
    except:
      # file not available
      print("File busy")
    # set alarm LED if needed
    if av>4:
      led.on()
    else:
      led.off()
    # wait a bit
    time.sleep(0.3)
```

Save this program in the cgi-bin directory with the name `tc74server.py`. There is no need to change any permissions, since you'll be starting this program yourself with `python tc74server.py`. From now on, the program will monitor the temperature inside the fridge and turns on the alarm if necessary.

The second program is started via the web browser in the same way as the others in this chapter. This program reads the temperature from the file written by the previous program and puts it into an HTML page. If it can't read from the file because another program is already using it, the program will display a message to that effect. Note that you can also get this error if you start the program on the PC before you started the program on the Raspberry Pi. In that case the data file wouldn't exist yet, and couldn't be opened, of course.

In the header of the HTML page created by this program you can see the line:

```
<meta http-equiv="refresh" content="5;url=http://192.168.178.9:8080/cgi-bin/
tc74remote.py">
```

The meta keyword means that the information following it won't be displayed on the page, instead it is about the page. The `refresh` informs the browser that it should refresh the page. After `content` you have the number of seconds that the browser should wait, in this case 5. Then there is a semi-colon, followed by the page that should be refreshed. In this case it is the page that starts the cgi-bin program. That program creates this page again, which will start the cgi-bin program, etc. This repeats itself until you close the browser. This way you only need to open the page once, after which it will refresh itself automatically.

You should of course replace my IP address in the meta command with your own.

```
#!/usr/bin/env python3
comment=""
try:
    # if it exists, get the temperature
    myfile = open('cgi-bin/temperature.dat','r')
    t=float(myfile.readline())
    myfile.close()
    if t>4:
      comment= "High temperature alert!"
except:
    # temperature not available
    t="--"
    comment="Temperature not available."

# show the HTML page
print("Content-Type: text/html")
print("""
<HEAD>
<TITLE>Fridge Alert</TITLE>
```

```
<meta http-equiv="refresh" content="5;url=http://192.168.178.9:8080/cgi-bin/
tc74remote.py">
</HEAD>
<html>
<body>
Current temperature is %s degrees. %s
</body>
</html>
""") % (t, comment)
```

Save the program in the cgi-bin directory with the name `tc74remote.py` and remember to set the 'execute' permissions:

```
chmod u+x tc74remote.py
```

If the server isn't running anymore you should go back to the server directory (by starting another LXterminal or Putty), and restart the server from within the server directory with:

```
Python -m CGIHTTPServer 8080
```

Make sure that the first program is running, and then type the following address in your browser:

```
http://192.168.178.9:8080/cgi-bin/tc74remote.py
```

You only have to enter this address once, since the page will be automatically refreshed every 5 seconds.

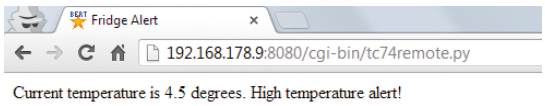


Figure 11.19. High temperature alarm in the browser.

Chapter 12 • Client Server (WiFi or Wired)

12.1 Introduction

We'll assume that your Raspberry Pi and PC are connected to a network as described in the previous chapter, and that you know what IP address your Raspberry Pi uses. You will also have installed Python on your PC.[85]

In the Web server projects, a program on the server is started by going to a certain web address. That program runs once, sends a reply back to the browser and stops. In a client-server application, we use a dedicated program on the client side, instead of a web browser. This program communicates via the network with another program that runs on the server. Both programs therefore run continuously and can communicate both ways. That opens up a lot more possibilities. Furthermore, you can now use Python on the client side as well (instead of HTML and JavaScript).

Two common protocols on the Internet are TCP and UDP. TCP (Transmission Control Protocol) is used more often, since it has error correction built in. If there is no error message, you know that the packet was correctly sent and received.

UDP (User Datagram Protocol) is never used for important applications such as web pages. Instead, it's used for the streaming of sound or video. UDP doesn't have any type of error correction, so it's very fast.

12.2 TCP multiplication

This project demonstrates some of the features of a client-server application. The client makes contact with the server, checks that it is the correct server, and then sends a number to the server. The server doubles this number and sends the result back. These programs will form the basis of the other projects in this chapter. Due to their simplicity you should be able to adapt them to your own requirements.

The server

The server program begins with the loading of several libraries.

```
# TCP server
import socket
import time
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

A socket is an object that makes communications possible. You could compare this to a phone. Both parties need to have a phone in order to talk to each other, and those phones need to be connected to each other. The same applies to sockets. The server opens a socket and listens. When a client makes a connection, the socket of the client is connected to the socket of the server, and communications can proceed. A server can have multiple sockets, but the communications always take place on a one to one basis.

85 If that isn't the case, you should first read section 3.3.2 (installing Python and wxPython on your Windows PC) and section 3.2 (determine your IP address, and optionally set it as a fixed address).

```
server.bind(("192.168.178.9", 5000))
server.listen(5)
```

Remember to use the IP address of your Raspberry Pi in the first line. A port ensures that a standard protocol is used between sockets. In the previous project we used port 8080 for HTTP. In this project we use port 5000 for TCP. We bind this port to the IP address of the Raspberry Pi (the server program runs on the Raspberry Pi). This server can only deal with one connection at a time, but it can listen to five. A subsequent connection will therefore be put in a queue, which means that the client concerned won't get an error, but it won't have a connection (yet) either.

```
print "TCP server listening on port 5000"

while 1:
    myclient, address = server.accept()
    print("Connected to ", address)
```

As soon as a client requests a connection from the server it will be accepted, and then created as an object. Everything that now happens to the "myclient" object applies to this connection.

```
# confirm connection
myclient.send(b'0')
while 1:
```

The server sends a 0 to the client to make itself known. This is used as a confirmation for the client that it has made a connection to the correct server. The server now waits for the client, using a 512-byte buffer.

```
# wait for client data
data = myclient.recv(512)
```

When the data received is either 'q' or 'qq' the connection to the client should be closed. In all other cases the server displays the data that it received. This data is a string, but we want to perform a calculation on it, so we need to convert it into number first. The server converts it into a float, multiplies it by two and converts the result back into a string. This string can then be sent back to the client.

```
if chr(data[0]) == "q" or data.decode('utf-8') == "qq":
  myclient.close()
  break
else:
  print("Received:" , data.decode('utf-8'))
  newdata=str(float(data)*2)
  print(" Send:" , newdata)
  myclient.send(newdata.encode('utf-8'))
```

When 'q' or 'qq' was received, the connection to the client was closed. In the case of 'qq' the server itself is also stopped.

```
if data.decode('utf-8') == "qq":
  server.close()
  print("TCP server is down")
  break
else:
  print("TCP server listening on port 5000")
```

The program shown above runs on the Raspberry Pi. You can start it with `python tcpserver1.py`, and stop it by sending qq from the client. Don't stop the program with Ctrl-C! If you do this then you won't be able to start the program again because the socket hasn't been closed. You would then get an error message like this: :

```
Traceback (most recent call last):
  File "tcpserver1.py", line 4, in <module>
    server.bind(('192.168.178.9', 5000))
  File "/usr/lib/python2.7/socket.py", line 224, in meth
    return getattr(self._sock,name)(*args)
socket.error: [Errno 98] Address already in use
```

The command `netstat -n | grep 5000` will return a list of open sockets using port 5000. If you stopped the server incorrectly, you could get the following result:

```
pi@raspberrypi ~/ $ netstat -n | grep 5000
tcp        0      0 192.168.178.9:5000      192.168.178.11:50039    ESTABLISHED
tcp        0      0 192.168.178.9:5000      192.168.178.11:50045    ESTABLISHED
tcp        0      0 192.168.178.9:5000      192.168.178.11:50108    ESTABLISHED
tcp        0      0 192.168.178.9:5000      192.168.178.11:50035    ESTABLISHED
```

The sockets are shown as `ESTABLISHED` which means they'll stay open. If you stop the program properly, it may happen that you still get the socket error message. Using the command `netstat -n | grep 5000` you'd find out that the socket is still waiting (TIME_WAIT) to see if anything will happen. After about 30 seconds the socket will close automatically.

```
pi@raspberrypi ~/ $ netstat -n | grep 5000
tcp        0      0 192.168.178.9:5000      192.168.178.11:50368    TIME_WAIT
```

It's therefore safe to start the server program again after 30 seconds. The complete program is as follows:

```
# TCP server
import socket
import time
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
server.bind(("192.168.178.9", 5000))
server.listen(5)

print("TCP server listening on port 5000")

while 1:
 myclient, address = server.accept()
 print("Connected to ", address)

 # confirm connection
 myclient.send(b'0')
 while 1:
   # wait for client data
   data = myclient.recv(512)
   if chr(data[0]) == "q" or data.decode('utf-8') == "qq":
     myclient.close()
     break
   else:
     print("Received:" , data.decode('utf-8'))
     newdata=str(float(data)*2)
     print(" Send:" , newdata)
     myclient.send(newdata.encode('utf-8'))

 if data.decode('utf-8') == "qq":
   server.close()
   print("TCP server is down")
   break
 else:
   print("TCP server listening on port 5000")
```

The data displayed in the Thonny shell for the server side when numbers 2 and 22 were sent followed by qq was as follows:

```
TCP server listening on port 5000
Connected to  ('192.168.1.178', 59310)
Received: 2
    Send: 4.0
Received: 22
  Send: 44.0
TCP server is down
```

The client

The client also loads the libraries at the beginning, then creates a socket and tries to connect to the server at the IP address of the Raspberry Pi.

```
# TCP client
import socket
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(("192.168.178.9",5000))
```

When the connection has been established the client expects to receive a 0 from the server. Once that has been received, we can be certain that we connected to the correct server. If the reply wasn't a zero we send a 'q' to close the connection and then stop the client. The server will continue normally.

```
# connect to server
data = client.recv(512)
if chr(data[0]) != "0":
  print("Unexpected reply from the server, aborting")
  print(data.decode('utf-8'))
  client.send(b'q')
  client.close()
  exit()
```

The client asks the user for some input and sends it to the server. If the input was 'q' or 'qq' the client stops, otherwise it waits for a reply from the server (using a 512-byte buffer). When the reply comes, the client displays it after which it's the user's turn again.

```
# communications loop
while 1:
  data = input("Send (q to quit, qq to stop server):")
  client.send(data.encode('utf-8'))
  if data == "q" or data == "qq":
    client.close()
    break;
  data = client.recv(512)
  print("Received:" , data.decode('utf-8'))
```

The complete client program is as follows:

```
# TCP client
import socket
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(("192.168.178.9",5000))

# connect to server
data = client.recv(512)
```

```
if chr(data[0]) != "0":
     print("Unexpected reply from the server, aborting")
   print(data.decode('utf-8'))
     client.send(b'q')
     client.close()
   exit()

# communications loop
while 1:
    data = input("Send (q to quit, qq to stop server):")
    client.send(data.encode('utf-8'))
    if data == "q" or data == "qq":
    client.close()
    break;
    data = client.recv(512)
  print("Received:" , data.decode('utf-8'))
```

The data displayed in the Thonny shell for the client side when numbers 2 and 22 were sent followed by q was as follows:

```
Send (q to quit, qq to stop server):2
Received: 4

Send (q to quit, qq to stop server):22
Received: 44

Send (q to quit, qq to stop server):q
```

The client runs on your PC (which must have Python installed). First start the server on the Raspberry Pi, and then the client on your Windows PC by double-clicking on `tcpclient1.py`. Next, input some number, which will be doubled by the server. Remember to replace my IP address with yours.

As an example, we'll show you how the communications proceed when the client connects to the server, sends 3.5 and then closes the connection.

Client	Server
	Listen
Request connection	Accept and send 0
Receive 0 and send 3.5	Receive 3.5 and send 7
Receive 7 and send q	Receive q and close connection
	Listen

Table 12.1. Communications session.

12.3 TCP LED control

We use the basic structure from the last project to create a client-server combination that can be used to control an LED. The server takes care of actually driving the LED, whereas the client sends commands to the server.

As shown in Figure 12.1, the hardware consists of an LED with a 680 Ω resistor connected to GPIO4 (pin 7 on the header).

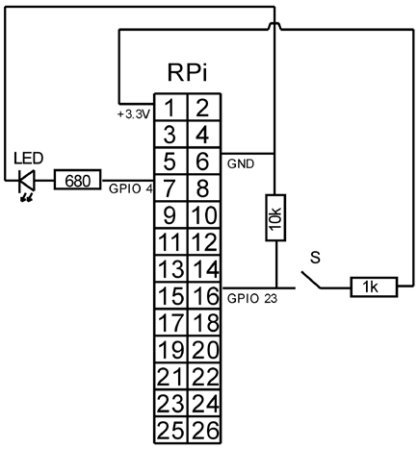


Figure 12.1. Connecting the LED to the header.

In the server program we add a section to load the required libraries and set up GPIO4 (pin 7 on the header).

```
import os
from gpiozero import LED

# prepare pin
led = LED(4)
```

The only other part we have to change is where we deal with the information received from the client. Instead of the multiplication, we turn the LED on or off, depending on the user's instructions.

```
if data == '1':
  led.on()
  newdata = 'LED on'
else:
  led.off()
  newdata='LED off'
print(" Send:" , newdata)
myclient.send(newdata.encode('utf-8'))
```

The rest of the program stays the same. The client stays identical to that used in the previous project. On the Raspberry Pi you should start the server program with `python3 tcpsever2.py`, and then you can run the program `tcpclient1.py` on your PC.

On the client you can turn the LED on with a 1 and turn it off with a 0.

Maybe you are wondering why we are now able to switch the LED on and off while with the web server we needed a complicated construction. The reason behind this is that the server program keeps running, while the cgi program stops after switching on the LED (which caused the LED to turn off by itself).

12.4 TCP DAC

We'll use the basic structure of the first project again, and adapt the libraries and pin settings:

```
import spidev

spi=spidev.SpiDev()
spi.open(0,0)
```

And the part that processes the data:

```
 value=int(data)
 if (value>=0) and (value<=255):
   reply=spi.xfer2([49,value])
   newdata="Command executed"
 else:
   newdata="Error: value out of range."
 Print(" Send:" , newdata)
 myclient.send(newdata.encode('utf-8'))
```

The client program can remain the same again. You can now start tcpserver3.py on the server (with `sudo python3 tcpserver3.py`) and run `tcpclient1.py` on your PC. Remember to replace my IP address with your own.

On the client you should input a number from 0 to 255, to get a voltage output of 0 to 3.3 V from the MAX522.

The hardware consists of a MAX522 DAC, connected to the SPI pins (Figure 12.3). You can connect a voltmeter to the output of the DAC. [86]

86 If you don't have a voltmeter, you can connect a 2-mA LED and a 680 Ω resistor. The brightness of the LED will depend on the output voltage of the DAC.

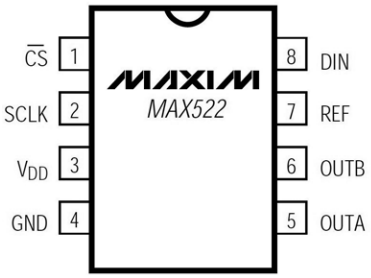


Figure 12.2. Pinout for the MAX522.

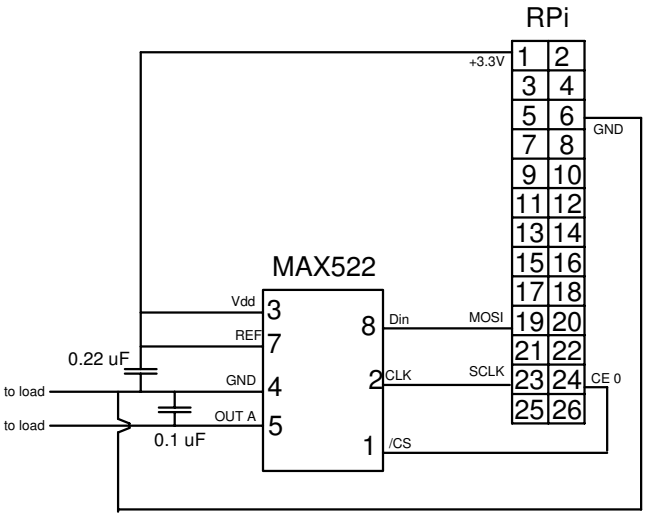


Figure 12.3. Connecting the MAX522 to the header.

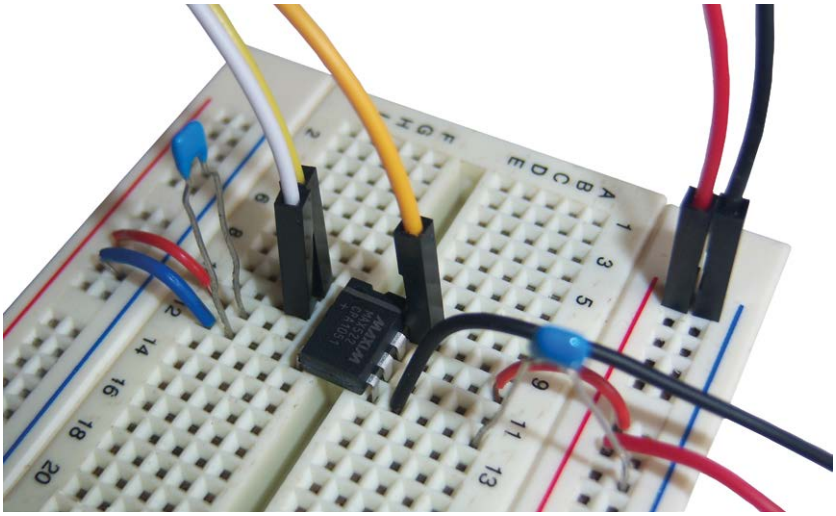


Figure 12.4. Setup on the breadboard.

12.5 TCP sawtooth and square wave generator

In this project we make a waveform generator that can create a sawtooth and square wave. The generator is part of the server and can be controlled via a client. A complicating factor is that the server has to deal with two tasks at the same time:

1. Listen on the socket, create a connection to the client, then keep the connection active and wait for commands from the client.

2. Calculate the correct voltage for the waveform and send it to the MAX522.

Our solution is to use a thread: the first task is taken care of by the main program, and the second task is performed in a thread. We have already seen how to use threading in project 5.9.2, but that was in a graphical application. Although it would be possible to use this here as well, a server program is generally not controlled directly by the user, so a graphical shell would be pointless.

In principle it's fairly easy to create a thread. We first need to load the threading module, and then define a global variable that can be accessed in both the main program and the thread.

We then define a thread that uses this global variable (a), and adds one to it every 0.1 seconds. The thread is started immediately. We haven't set a limit for the value of a. This is somewhat careless since the value of a could take on some pretty high values if we let the program run for a long period. Eventually the memory of the Raspberry Pi overflows, with dire consequences. If you're planning on going shopping after starting the program it would be better to add something like `if a==1000: a=0` to the thread.

In the main program we ask the user to input a value, which we store in the variable a. The thread, which is already running, now has this new value for a, and keeps adding 1 to it. After one second the main program displays the value of a. Since the thread has added 1 to a every 0.1 seconds, the value of a has increased by 10.

```
import threading
import time

# set interthread variable
global a
a=0

# define thread
class ThreadClass(threading.Thread):
    def run(self):
        global a
        while True:
            a=a+1
            time.sleep(0.1)
t = ThreadClass()
t.setDaemon(True)
t.start()

# main program
while True:
    b=input("Enter number or q to quit ")
    if b=='q':
        break
    else:
        a=int(b)
    print(a)
    time.sleep(1)
    print(a)
print("Done")
```

Start the program with `python examplethread.py` and enter a few values:

```
pi@raspberrypi ~/experiments $ python3 examplethread.py
Enter number or q to quit 3
3
13
Enter number or q to quit 8
8
18
Enter number or q to quit q
Done
```

We can make a thread in two ways. The usual way assumes that the thread runs for a while, and then stops by itself. This isn't the case here, since the thread continues adding indefinitely. This means that when the program stops, this thread would continue! You won't be able to see this, nor can you access it, but it's still there, using resources from the Raspberry Pi to keep the thread running. Your Raspberry Pi will therefore become a little slower every time you start and stop this program.

You can fix this in two ways:

1. Use an extra global variable that can be set in the main program to signal to the thread that it should stop. This is easy to program, as long as you can be sure that the main program is able to set this variable in time. You have to rely on the user not to press Ctrl-C to stop the program (or to trap it), and that the program won't crash unexpectedly. This method was used in project 5.9.2.

2. Start the thread as a daemon.[87] This method doesn't suffer from the disadvantages of the first method, but you do have to ensure that you don't stop the daemon at an inopportune moment. This is the method used in this project, so you can familiarize yourself with both methods.

We start the thread as a daemon with the command `setDaemon(True)`. This binds the thread to the main program. When the main program stops, the daemon thread will be stopped as well. This could have some nasty side effects if the daemon thread wasn't expecting this, but we'll come back to that later.

We'll now use the daemon method to make a thread in our standard server program. The part of the program that creates the square wave and sawtooth has to bear in mind that the waveform could change completely unexpectedly. The thread cannot "know" when the user will change the value of a. We deal with this in the following way:

```
t=0
square=False
while True:
  # sawtooth
  if a==1:
    reply=spi.xfer2([49,t])
  # square wave
  if a==2:
    if t==0 and square==False:
      reply=spi.xfer2([49,255])
      square=True
    else:
      if t==0 and square==True:
        reply=spi.xfer2([49,0])
```

87 Daemon has its origins in Greek mythology and concerns a being that performs tasks that the gods don't want to be bothered by (Description by priestess Diotima to Plato in Symposium, 380 BC).

```
            square=False
        time.sleep(0.001)
        t+=10
        if t==250:
          t=0
```

For every loop (so every 0.001 second), the thread looks which waveform should be produced, and sets the output voltage accordingly. The square wave has to change from low to high, or vice versa, after 25 steps. The current state of the square wave is stored in the variable `square`. When this variable is `True` the square wave is high, when the value is `False` the square wave is low. You may think that the double if-then construct could be done differently, leaving out the 'else':

```
    if t==0 and square==False:
      reply=spi.xfer2([49,255])
      square=True
    if t==0 and square==True:
      reply=spi.xfer2([49,0])
      square=False
```

However, this would mean that the square wave would never be high. If 't' was zero and `square` was `False`, the first condition would be satisfied. The square wave becomes high and square is set to `True`. But this then satisfies the second if-then condition, which immediately makes the square wave low again, and sets `square` back to `False`. The `else` instruction stops this from happening.

When the program stops, the daemon thread is forced to stop as well. If it happened to be in the middle of a `sleep` instruction it would cause problems. Halfway through the `sleep` instruction it suddenly wouldn't exist any more, which would came as a total surprise to the daemon thread and it wouldn't know what to do next. The result will be an exception error. An exception error is such a serious error that the program can't continue and will stop abruptly. We may have succeeded in stopping our thread, but it wasn't done very elegantly:

```
Exception in thread Thread-1 (most likely raised during interpreter
shutdown):
Traceback (most recent call last):
  File "/usr/lib/python2.7/threading.py", line 551, in __bootstrap_inner
  File "tcpserverthread2.py", line 40, in run
<type 'exceptions.AttributeError'>: 'NoneType' object has no attribute
'sleep'
```

Python tells us that the exception was most likely caused when the main program stopped (`interpreter shutdown`). We must therefore ensure that the daemon thread won't be stopped in the middle of a sleep instruction.

It's best to first turn off the waveform generator, so the daemon thread can be stopped without any problems. This is done by setting the value for a to 0, so no waveform will be produced. We then turn the MAX522 off. Since the `sleep` in the daemon thread is very short, we assume that it will have finished in the meantime. When you use a longer `sleep` it is better to wait a bit longer at this point in the main program as well. When you're certain that the daemon `sleep` is over, the program can be stopped, since there are no remaining instructions.

```
# stop thread voltage control
a=0
# kill output voltage
reply=spi.xfer2([49,0])
```

When the main program stops unexpectedly, when the user presses Ctrl-C, for example, or when the program stops due to a severe error, you could still get a daemon error message. In the latter case it doesn't really matter: you already have an exception error from the main program, so you might as well have another one. It would be better if we could trap Ctrl-C, and we'll show you in section 12.6 how this can be done. In this project we assume that you know you shouldn't stop your threading programs with Ctrl-C.

The complete program is now as follows:

```
# TCP server
import socket
import time
import spidev
import threading

# setup SPI
spi=spidev.SpiDev()
spi.open(0,0)

# setup server
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(("192.168.178.9", 5000))
server.listen(5)
print("TCP server listening on port 5000")

# global inter-thread variable
global a
a=0

# set up thread as daemonthread
class ThreadClass(threading.Thread):
 def run(self):
   global a
```

```
    t=0
    square=False
    while True:
      # sawtooth
      if a==1:
        reply=spi.xfer2([49,t])
      # square wave
      if a==2:
        if t==0 and square==False:
          reply=spi.xfer2([49,255])
          square=True
        else:
          if t==0 and square==True:
            reply=spi.xfer2([49,0])
            square=False
      time.sleep(0.001)
      t+=10
      if t==250:
        t=0


t = ThreadClass()
t.setDaemon(True)
t.start()

# main program loop
while 1:
  myclient, address = server.accept()
  print("Connected to ", address)

  # confirm connection
  myclient.send(b'0')
  while 1:
    # wait for client data
    data = myclient.recv(512)
    if chr(data[0]) == "q" or data.decode('utf-8') == "qq":
      myclient.close()
      break
    else:
      print("Received:" , data.decode('utf-8'))
      value=int(data)
      if (value>=0) and (value<=255):
        a=value
        newdata="Command executed"
      else:
        newdata="Error: value out of range."
```

```
            print(" Send:" , newdata)
            myclient.send(newdata.encode('utf-8'))
        if data.decode('utf-8') == "qq":
          server.close()
          print("TCP server is down")
          break
        else:
          print("TCP server listening on port 5000")

    # stop thread voltage control
    a=0
    # kill output voltage
    reply=spi.xfer2([49,0])
```

The nice thing about this program, and hence the waveform generator, is that it continues running after the client has disconnected the connection. You can therefore make a connection, select the waveform you want, disconnect, and even turn off your PC.

We use the same hardware as in the previous project. Remember to replace my IP address with your own, and start the server with the command:

```
Python3 tcpserverthread2.py
```

Then start the client by double-clicking on `tcpclient1.py`. You can input any of the following options:

`1`	sawtooth
`2`	square wave
`q`	stop client
`qq`	stop client and server

Below is an example of the communications.

Client:

```
Send (q to quit, qq to stop server):1
Received: Command executed
Send (q to quit, qq to stop server):2
Received: Command executed
Send (q to quit, qq to stop server):qq
```

Server:

```
pi@raspberrypi ~/experiments $  sudo python tcpserverthread2.py
TCP server listening on port 5000
Connected to  ('192.168.178.10', 49511)
Received: 1  Send: Command executed
Received: 2  Send: Command executed
TCP server is down
```

With the help of WinOscillo you can also see the waveforms on your PC.

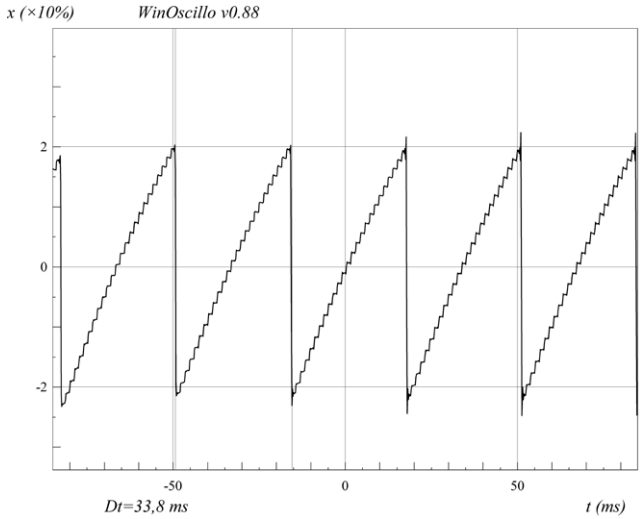


Figure 12.5. Sawtooth.

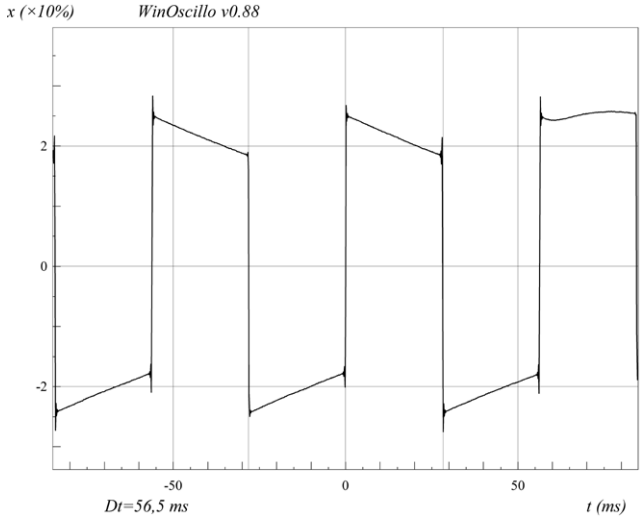


Figure 12.6. Square wave.

According to WinOscillo the sawtooth has a period of 33.8 ms, which at 25 steps corresponds to 1.35 ms per step. This is a long way out from the time set in the `sleep` instruction, which was 1 ms. This is because the daemon thread does more than just create the waveform. For each step it also has to communicate via the SPI bus. Furthermore, the Raspberry Pi has to perform many other tasks as well, so the daemon thread doesn't get all of the processing power. That would be impossible, since the main program has to run as well. Then there are the background tasks, such as dealing with the Internet connection, updating the screen, dealing with the mouse, etc.

The period for the square wave is 56.5 ms, but that consists of two sets, so take up 50 steps. This results in 1.13 ms per step. This is quite a lot quicker, which is because this part of the program only sends a message to the MAX522 when the square wave changes state. This means it only needs to communicate via the SPI bus once every 25 steps, whereas for the sawtooth this has to happen every step.

12.6 TCP Voltmeter with 8 channels

In this project we're creating an 8-channel voltmeter. The measurements take place on the Raspberry Pi in a server program. The control and display output are done on a PC via a graphical program. We use a MCP3008, an 8-channel SPI ADC chip made by Microchip, which we've already seen in project 8.5.

For the server we'll use the basic structure of the first project in this chapter, and adapt the libraries and pin settings:

```
import spidev

spi=spidev.SpiDev()
spi.open(0,0)
```

We also need to change the part that processes the data. The incoming data will be the channel number of the MCP3008 on which the measurement should take place, or qq when the server should be stopped.

```
    # wait for client data
    data = myclient.recv(512)
    if chr(data[0]) == "q" or data.decode('utf-8') == "qq":
      myclient.close()
      break
      else:
      print("Received:" , data.decode('utf-8'))
      chan=int(data)
      reqadc = spi.xfer2([1,128+(chan<<4),0])
      newdata= (reqadc[1]&3)*256 + reqadc[2]
      print(" Send:" , newdata)
      myclient.send(newdata.encode('utf-8'))
```

```
    if data.decode('utf-8') == "qq":
    server.close()
    print("TCP server is down")
    break
  else:
    print("TCP server listening on port 5000")
```

This program is called `tcpserver4.py`. We're creating a graphical program for the client. The values returned for each channel will be shown as vertical bars, like the bar graph displays on audio equipment. [88]

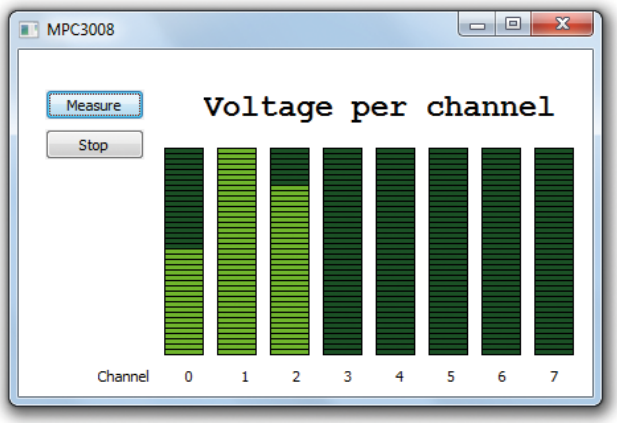


Figure 12.7. Voltmeter with 8 channels.

We use the standard template for graphical programs, where we've changed the window size to 450 x 300. The extra libraries required have been added to the start of the program.

```
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(("192.168.178.9",5000))
```

The next step is the connection to the server. If the server isn't running the program stops after displaying an error message.

```
# connect to server
data = client.recv(512)
if char(data[0]) != "0":
        print("Unexpected reply from the server, aborting")
        print(data)
        client.send(b'q')
        client.close()
        exit()
```

88 If you want to, you can add a maximum to each channel, and change the program so that the bar turns red above that value

We'll be storing the ADC measurements in a list called "mdata", which should be globally accessible. It is filled with eight zeroes, so it immediately has the correct length.

```
global mdata
mdata=[0,0,0,0,0,0,0,0]
```

We need two buttons for the operation, a background for the bar graph, and a binding from the paint event to the OnPaint function.[89] There will also be some text displayed above and below the bar graph.

```
# your own objects

 # the measurement button
 cbtn0 = wx.Button(self, label='Measure', pos=(20, 30))
 cbtn0.Bind(wx.EVT_BUTTON, self.OnMeasure)

 # the stop button
 cbtn1 = wx.Button(self, label='Stop', pos=(20, 60))
 cbtn1.Bind(wx.EVT_BUTTON, self.OnStop)

 # prepare background
 self.dc = wx.ClientDC(self)

 # bind the paint event
 self.Bind(wx.EVT_PAINT, self.OnPaint)

 # static texts
 self.txt = wx.StaticText(self, label='Channel 0
 1           2           3           4
 5           6           7', pos=(60, 240))
 self.txt2 = wx.StaticText(self, label='Voltage per channel
 ', pos=(140, 30))
 font = wx.Font(18, wx.MODERN, wx.NORMAL, wx.BOLD)
 self.txt2.SetFont(font)
```

The measurements are carried out in the OnMeasure function. Requests are sent one after the other to the server for the ADC values of each channel, with the results stored in the global list mdata. When all measurements have taken place, the self.Refresh() command is issued, which redraws the whole window again. This results in the function OnPaint being called. This then uses the information in mdata to draw the new bar graph. It's purely a matter of personal taste how big the bars are and what their color is.[90]

89 Backgrounds and paint events were also used in project 10.2.2

90 You can use a "color picker" to choose your own color, and put its HEX code in the program. An example of such a program can be found at: www.2createawebsite.com/build/hex-colors.html

```
# your own functions

def OnPaint(self, event):
        global mdata
        dc = wx.PaintDC(self)
        dc.SetDeviceOrigin(100, 70)
        for c in range (0,8):
                for i in range(1, 40):
                        if i > mdata[c]/26:
                                dc.SetBrush(wx.Brush('#075100'))
                                dc.DrawRectangle(10+(40*c), 160-i*4,
                                30, 5)
                        else:
                                dc.SetBrush(wx.Brush('#36ff27'))
                                dc.DrawRectangle(10+(40*c), 160-i*4,
                                30, 5)


def OnMeasure(self, e):
        # read all channels using the server
        global mdata
        for c in range (0,8):
                chan = str(c)
                client.send(chan)
                mdata[c] = int(client.recv(512))
        self.Refresh()

def OnStop(self, e):
        # stop server, socket and client
        chan = 'qq'
        client.send(chan.encode('utf-8'))
        client.close()
        self.Close(True)
```

This program is called `tcpclient2.pyw`. As an example of what this project can do, we've connected potentiometers to two channels and an LDR to the third channel. This is connected in series with a 4k7 resistor, with the center tap connected to the ADC. An LDR (Light Dependent Resistor) is a resistor whose value depends on the amount of light falling on it. The LDR and the 4k7 resistor are a potential divider, where the output is connected to channel 2. When the amount of light striking the LDR changes, its resistance changes as well, which results in a different voltage from the potential divider. The voltage therefore provides a measure of the light intensity.

Nothing is connected to the other channels. Since we still measure these channels, they've been connected to GND so the measurements always return zero.

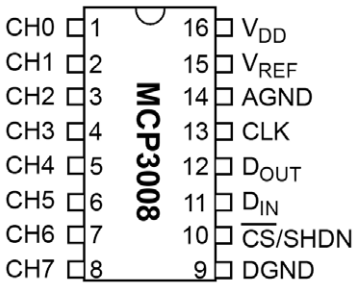


Figure 12.8. Pinout for the MCP3008.

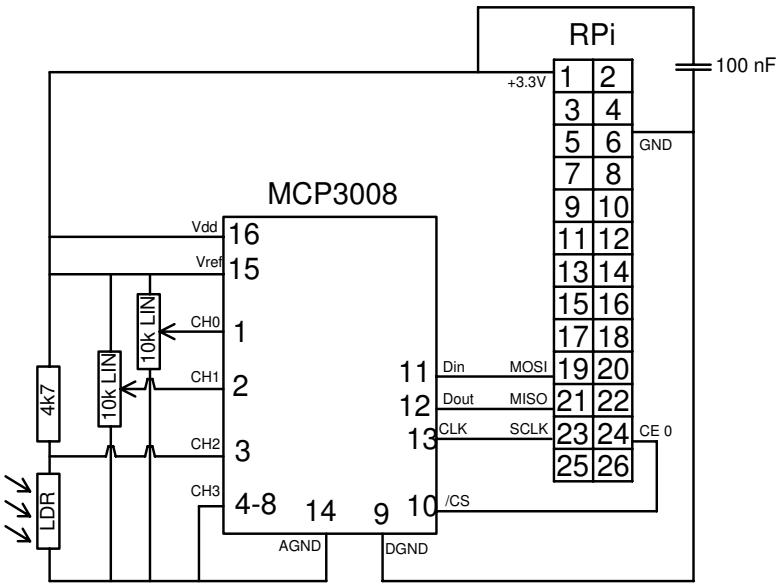


Figure 12.9. Connecting an LDR and two potentiometers.

The breadboard for this circuit looks as follows. Note the 100 nF capacitor that's connected to the digital ground of the MCP3008, and the separation of the digital ground (DGND) and analog ground (AGND). The color code for the 4k7 resistor is yellow-purple-red.

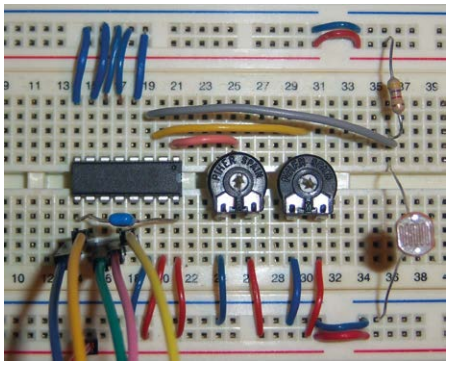


Figure 12.10. Setup on the breadboard.

12.7 UDP echo

In contrast to TCP, the UDP protocol doesn't check if a packet has been received successfully. It is therefore used mainly for streaming movies and the like, since it doesn't matter so much if the occasional packet gets lost, as long as the streaming is as fast as possible.

In this project we'll use the UDP protocol for letting several clients communicate with a single server. Since we're using UDP we can turn the clients on and off at will, without the need to let the server know. The beginning of the client program is almost the same as the TCP programs, except that the socket is now configured as a DGRAM, an abbreviation for datagram, the UDP protocol.

```
# UDP client

# prepare connection
import socket
client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
client_socket.settimeout(3)
client_socket.bind(("", 6000))
```

The client opens a socket and sends the information to the IP address of the server using port 6000. At this point we don't know if the server is running, or if it even exists, but this doesn't matter with this protocol. When the user hasn't requested to stop, the program will wait for a reply from the server. This is a risky situation, since the program could hang when the server isn't available (which is always a possibility with UDP). For this reason we've added a time-out of three seconds at the beginning of the program. If nothing has been received after this time-out, the program gives a `socket.timeout` error. We trap this error using a `try/except` construct.

When a time-out occurs we stay in the `while` loop, since the program only stops when the user inputs a q (stop program) or qq (stop program and server).

```
# main program
while 1:
    data = input("Enter data (q to exit qq to stop server):
       ")
    client_socket.sendto(data.encode('utf-8'), ("192.168.178.9",6000))
    if data=="q" or data=="qq":
      # stop program (and server)
      break
    else:
      # see if server is still available to reply
      try:
        data, address = client_socket.recvfrom(256)
        print("Server reply : ", data.decode('utf-8'))
      except socket.timeout:
        print("Timeout, cannot reach server")


# close connection
client_socket.close()
```

Remember to replace my IP address with your own. The way in which you start the program depends on which computer you're using. On the Raspberry Pi it is `python udpclient1.py`, and on a Windows PC you can just double-click on `udpclient1.py`.

The server obviously listens on the same port, and does nothing other than display the received information along with its associated IP address and socket number, so you can see where it originated. The information is then echoed back to the client, to confirm that the information has been received.

```
# UDP server

# prepare connection
import socket
server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server_socket.bind(("192.168.178.9", 6000))

# wait for transmission from client
print("UDPServer listening on port 6000")
while 1:
    # transmission received
    data, address = server_socket.recvfrom(256)
    print("IP" ,address[0], "socket" , address[1] , "said : ",
       data.decode('utf-8'))
    if data.decode('utf-8')=="qq":
      # stop the server
      break
```

```
        else:
          # or echo the received information
          server_socket.sendto(data,("192.168.178,10",6000))

    # close all open connections
    server_socket.close()
```

Remember to replace my IP address with your own, and start the program with `python3 udpserver1.py`. Notice that in this example, "192.168.178.10" is the address of the client where the data is echoed back, and "192.168.178.9" is the IP address of our Raspberry Pi.

When you run the server on a PC with a firewall (e.g. Windows 10) you will immediately see a warning. You should then select the option "allow access". A nice feature of UDP is that there are no checks to see if the devices are still on the network. You get a request and send a reply, without worrying if the intended recipient has received it, or is still there. We'll make use of this feature in the following project where we get several clients to communicate with the server. If you have two PCs in addition to your Raspberry Pi, you could use the following setup. However, you can also run all the clients on one PC or on the Raspberry Pi if that is easier for you.

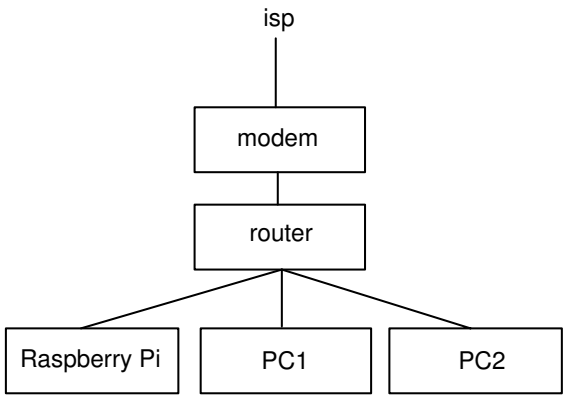


Figure 12.11. Optional network configuration for this project.

This is a transcript of the Raspberry Pi in the previous setup where it's acting as the server. There are four clients in all: one on each of the PCS and two on the Raspberry Pi.

```
pi@raspberrypi ~/experiments $ python udpserver1.py
UDPServer listening on port 6000
IP 192.168.178.11 socket 50190 said :  PC1
IP 192.168.178.10 socket 65498 said :  PC2
IP 192.168.178.9 socket 52228 said :  RPia
IP 192.168.178.9 socket 40842 said :  RPib
IP 192.168.178.9 socket 40842 said :  qq
```

The UDP server runs on the Raspberry Pi, which has an IP address of 9 (for simplicity, we leave out the "192.168.178." since that is the same for all addresses). The first contact with the server comes from a PC with IP address 11. Next is a PC with IP address 10, and finally there are the two clients on the Raspberry Pi itself. The IP address for these clients is obviously 9 as well, but they have differing socket numbers (52228 and 40842) so we can tell them apart. The last client then sent the qq, which stopped the server.

You can see that all the clients can communicate with the same server whenever they like.

12.8 UDP light meter

In this project we create a light meter that can be accessed from several computers at the same time. It's possible to use different clients on each computer, depending on the user's requirements, while the server remains the same for everybody. This is because the server has the simple task of transmitting the value of the light intensity to anybody who asks for it, regardless of what they do with the information.

We use the same hardware in this project as in project 12.6. Since we're only using the LDR here, you can remove the two potentiometers if you like.

The server is the same as in the previous project, except that the MCP3008 only reads from channel 2, which has the LDR connected to it. Since several clients can be connected at the same time, it wouldn't be very sensible if any client could stop the server. At that point the other clients would suddenly lose contact with the server, since it's been turned off. We therefore need to be able to stop the server on the Raspberry Pi itself. Since most people stop Python programs with Ctrl-C, which would leave the port open, we're going to trap this key combination. We use the `try/except` construct, where the entire main program is put in the try section. When an error occurs, the program jumps to the except section, where the port is closed properly.

```
# UDP server

# prepare connection
import socket
server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server_socket.bind(("192.168.178.9", 6000))

# prepare spi
import spidev
spi=spidev.SpiDev()
spi.open(0,0)

# trap keyboard interrupt for Ctrl-C
try:
    # listen for transmission from client
    print("UDPServer listening on port 6000")
    while 1:
```

```
    # transmission received
    data, address = server_socket.recvfrom(256)
print("IP" ,address[0], "socket" ,
address[1] , "said : ", data.decode('utf-8'))
    if data.decode('utf-8')== "qq":
      # stop the server
      break
    else:
      # read channel 2 for the LDR,
      # and return the result
      chan=2
      reqadc = spi.xfer2([1,128+(chan<<4),0])
      newdata= (reqadc[1]&3)*256 + reqadc[2]
      server_socket.sendto(newdata.encode('utf-8'),("192.168.178.10",6000))
except KeyboardInterrupt:
  print("Server aborted using Ctrl-C")
# close all open connections
server_socket.close()
```

Remember to replace my IP address with your own, and start the program with `python udpserver2.py`.

We'll use a graphical program for the client. Just as in project 9.4, we take the average over a number of measurements, which are stored in a list called `L`. Since light measurements are more variable than temperature measurements, we're taking the average over 120 measurements (at 1 measurement per second this is 2 minutes). We use the standard graphical template, with the addition at the beginning to import the socket module.

```
     # your own objects

     # timer object
     self.timer = wx.Timer(self)
     self.Bind(wx.EVT_TIMER, self.OnTimer, self.timer)

     # prepare socket
     self.client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
     self.client_socket.settimeout(3)

     # declare the list
     self.L=[]

     # start timer, every second
     self.timer.Start(1000)

     # text object
               self.text = wx.StaticText(self, label='0',pos=(20,30))
```

```
        font = wx.Font(50, wx.DECORATIVE, wx.ITALIC, wx.NORMAL)
        self.text.SetFont(font)

    # your own functions
    def OnTimer(self, e):
        # get light level
        self.client_socket.sendto(b'r', ("192.168.178.9",6000))
        # see if the server is still available to reply
        try:
                data, address = self.client_socket.recvfrom(256)
                # calculate moving average
                self.L.append(int(data))
                if len(self.L)>120:
                        self.L.pop(0)
                av=sum(self.L)*1.0/len(self.L)
                reply=str(int(av))
        except socket.timeout:
                reply= "Timeout"
                self.text.SetLabel(reply)
```

The light measurement is requested from the server, but it's possible that it is no longer on line, which is why a `try/except` construct was added. When there is a time-out, you can (re) start the server and the program will work again. If you want to close the client using the red cross, you have to turn the timer off for a while, otherwise there is no time left for the mouse click to be recognized. Since we're adding a `sleep` instruction we also have to add `import time` at the beginning of the program, so we can use this construct:

```
        except socket.timeout:
                reply= "Timeout"
                self.timer.Stop()
                time.sleep(0.1)
                self.timer.Start(1000)
```

This is what the output of the program `udpclient2.pyw` looks like on a Windows PC and on the Raspberry Pi. When you're using several clients that haven't been started at the same time it will take a while before they display the same result. Initially, very little averaging takes place, so the individual measurements have a greater influence. After two minutes all clients will show the same value.[91]

91 You can open several clients on the Rasberry Pi by opening several LXterminals, or starting several Putty sessions. They are all different clients as far as the server is concerned.

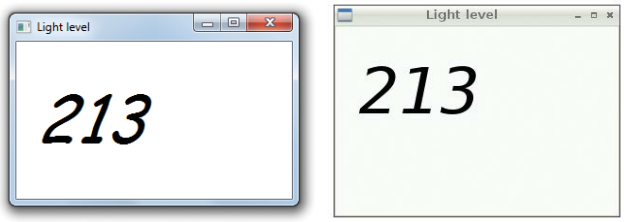


Figure 12.12. Light meter, Windows PC on the left, Xwindows/Raspberry Pi on the right.

You can use different clients if you like, which process the data in different ways. One client could take the average over 15 minutes and use the result to turn on the lights in the garden. Another client could take the average over a shorter period and show the result as a graph. You're only limited by your own imagination!

Chapter 13 • Bluetooth Project

Bluetooth is commonly used nowadays to transfer data, pictures, and messages between mobile devices. All current mobile phones, laptop computers, and tablets are equipped with the Bluetooth communications modules. The advantage of the Bluetooth compard to other communication technologies is that it can be used anywhere and at any time since it dows not operate through a router like a WiFi device. Additionally, Bluetooth is an easy means of transferring data between different devices. The range of present day Bluetooth devices is comparable and even longer than the range of WiFi.

In this chapter we shall be looking at the design of a Bluetooth based project. In this an LED is connected to one of the Raspberry Pi GPIO ports. The LED is controlled by sending commands from an Android mobile phone.

The connection diagram of the project is as shown in Figure 5.3, where the LED is connected to GPIO 4 through a 680 Ohm resistor.

The LED is controlled by sending the following commands from the mobile phone:

L1	Turn the LED ON
L0	Turn the LED OFF

In this project we are assuming that the Bluetooth is enabled on your Raspberry Pi and it is already paired with your Android mobile phone. Before developig your program, you will need to install the Bluetooth library on your Raspberry Pi. This is done by entering the following command on your Raspberry Pi while in command mode:

```
sudo apt-get install bluetooth libbluetooth-dev
sudo python3 -m pip install pybluez
```

In order to be able to access the Raspberry Pi from a mobile phone apps, make the following changes to your Raspberry Pi from the command line:

- Start the nano text editor and edit the following file:

```
sudo nano /etc/systemd/system/dbus-org.bluez.service
```

- Add –C at the end of the ExecStart= line. Also add another line after the ExecStart line. The final two lines should look like:

```
ExecStart=/usr/lib/bluetooth/bluetoothd -C
ExecStartPost=/usr/bin/sdptool add SP
```

- Exit and save the file by entering Ctrl+X followed by Y

- Reboot your Raspberry Pi:

```
pi@raspberrypi:~ $ sudo reboot
```

13.1 Android Bluetooth Apps

In this project we will be sending commands to the Raspberry Pi from an Android based mobile phone. We therefore need an application on our mobile phone where we can send data through Bluetooth to our Raspberry Pi. There are many such applications free of charge in the Android Play Store. In this book we use an apps called "Arduino Bluetooth Controller – All in One by Apps valley" (see Figure 13.1). You should install this application (or a similar one) on your Android phone so that you can send commands to the Raspberry Pi.

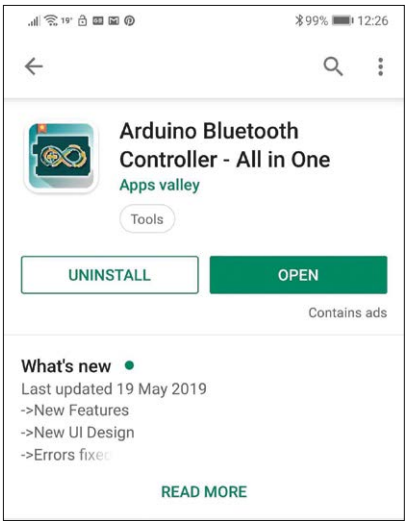


Figure 13.1 Android Bluetooth apps

The program (rpibluetooth.py) was developed under Thonny and the program listing is shown in Figure 13.2. At the beginning of the program, the LED at GPIO port 4 is defined as led. The program then creates a Bluetooth socket, binds and listens on this socket, and then waits to accept a connection. The remainder of the program is executed in a loop where the program issues statement `client.recv` and waits to receive commands from the mobile phone. The received command is decoded and the LED is turned ON or OFF as requested by the Android mobile phone.

Notice that the program makes use of the MAC address of the Raspberry Pi. This can be obtained by entering the following command in terminal mode:

```
hciconfig | grep "BD Address"
```

In this example project, the MAC address of the Raspberry Pi was found to be: `DC:A6:32:00:E4:2A`.

```
import socket                              # Import socket
import RPi.GPIO as GPIO                    # Import GPIO
GPIO.setwarnings(False)
GPIO.setmode(GPIO.BCM)

#
# LED is on GPIO 17
#
LED = 17                                   # LED on GPIO 17

#
# Configure LEDs as outputs
#
GPIO.setup(LED, GPIO.OUT)

#
# Turn OFF the LED to start with
#
GPIO.output(LED, 0)

#
# Start of main program loop. Read commands from the Android
# mobile phone, decode them, and control the LED
#
#
Port = 1
MAC = 'DC:A6:32:00:E4:2A'
s=socket.socket(socket.AF_BLUETOOTH,socket.SOCK_STREAM,socket.
BTPROTO_RFCOMM)
s.bind((MAC, Port))
s.listen(1)
client, addr = s.accept()

#
# Turn ON the LED if the command is:       L1
# Turn OFF the LED if the command is:      L0
#
try:

  while True:
    data = client.recv(1024)
    if chr(data[0]) == 'L' and chr(data[1]) == '1':     # L received
        GPIO.output(LED, 1)                             # LED ON
    elif chr(data[0]) == 'L' and chr(data[1]) == '0':   # 0 received
        GPIO.output(LED, 0)                             # LED OFF
```

```
except KeyboardInterrupt:
    client.close()                                    # Close
    s.close()
```

Figure 13.2 Program listing

Notice that what is returned into variable Data is byte objects which is actually arrays of integers. Here, we are using the built-in chr function to convert them into their equivalent ASCII characters. We could also have used the following statements instead (`utf-8` is the default encoding and decoding parameter in Python 3):

```
try:

  while True:
      data = client.recv(1024)              ## Receive data bytes
      if data.decode('utf-8') == "L1":      # L1 received
          led.on()                          # LED ON
      elif data.decode('utf-8') == 'L0':    # L0 received
          led.off()                         # LED OFF

except KeyboardInterrupt:
    client.close()                          # Close
    s.close()
```

Figure 13.3 shows the mobile application sending a command to turn the LED ON and then OFF.

The steps to use the project is as follows:

- Make sure that Bluetooth is enabled on your Android mobile phone
- Make Bluetooth Discoverable by clicking the Bluetooth icon at top right hand side of your Raspberry Pi.
- You may have to accept to pair the two Bluetooth devices
- Run your program either inside Thonny or from the command line as:

```
python3 rpibluetooth.py
```

- Start the mobile apps and click the button at the top right hand side to refresh the Bluetooth scan list
- Click on raspberrypi to connect to the Raspberry Pi
- Click on TERMINAL icon (icon of a keyboard) to get into terminal mode
- Enter the required command. e.g. L1 to turn the LED ON, L0 to turn the LED OFF.

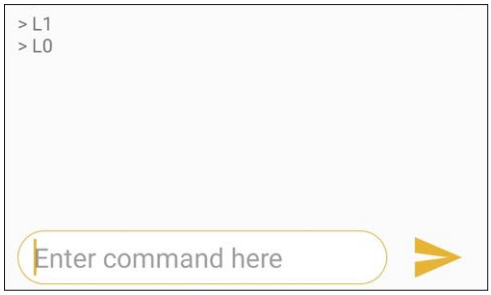


Figure 13.3 Mobile application

Chapter 14 • LEGO Board

14.1 Introduction

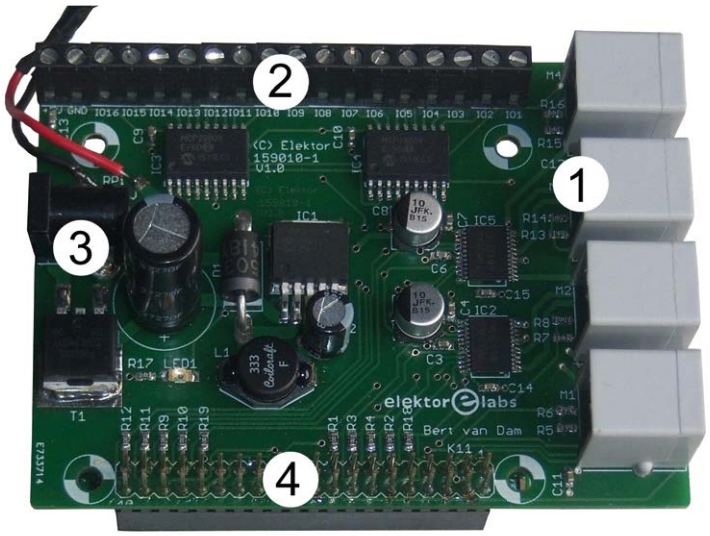


Figure 14.1. LEGO Raspberry Pi board.

This board combines the flexibility of the mechanical construction of LEGO with the flexibility of the Raspberry Pi with regards to its software, Internet communications and wide range of sensors. Unfortunately the LEGO board library is not supported by Python 3 and you will have to use Python 2 for the applications programs given in this chapter. i.e. start your programs by entering python followed by the filename (with .py extension). The numbers below refer to those shown in the figure above.

1. When you plug this board into the Raspberry Pi you can drive up to four EV3 (or compatible) motors, which you can use to make your models move.

2. The board also has 16 buffered digital inputs and outputs, which can be used to connect your own sensors, LEDs, etc. You can't connect LEGO sensors (in that case you would be better off with the actual EV3), but you can connect virtually any other sensor, such as infrared, ultrasonic, tilt sensors and switches. You're only limited by your imagination.

3. There is a voltage regulator on board for the Raspberry Pi, so you only need to connect a 9-volt supply to the board (the positive is on the centre pin, using either batteries or an AC mains supply, with a recommended rating of 5 Amp). This is used to power the motors, the board itself and the Raspberry Pi. You can use batteries for mobile use, since the power supply for the Raspberry Pi board has been designed with an efficient Buck converter.

4. Since the Raspberry Pi header is fed through the LEGO board, you can still connect SPI and I2C components and sensors. You can therefore combine any of the projects and sensors from this book with your LEGO models!

The Raspberry Pi can be used for communications via the Internet, either by cable or via WiFi. When you connect a USB/4G dongle to the Raspberry Pi you can also maintain contact with your model away from home.

In conclusion, you can build models that combine LEGO motors with the power of the Raspberry Pi, and all the possible sensors that can be connected to the Raspberry Pi!

14.2 Design

The LEGO Raspberry Pi board is not a copy of the EV3, but combines the strong points of both the Raspberry Pi and LEGO.

LEGO strong points	**LEGO weak points**
1. The mechanical construction that lets you build whatever you want. 2. The motors, although you need servos and hydraulics for more serious applications.	1. The small number of connections for sensors. A slightly ambitious robot needs four ultrasonic sensors, and that's just for starters. 2. Video processing, and all other things that LEGO hasn't provided a sensor for.
Raspberry Pi strong points	**Raspberry Pi weak points**
1. It uses an open system. You can create numerous applications using the easy to learn language Python. For the enthusiast, the C language provides a more powerful alternative. 2. It has many connectors, and it offers "limitless" expansion via I/O extenders.	1. The pins can't take much punishment and they're not protected. 2. The power consumption is comparatively high.

Table 14.1. Comparison of LEGO and Raspberry Pi.

The conclusion is that this board should meet the following criteria (which are all included, of course):

- 4 LEGO EV3 motor connectors (for LEGO 45502,or 45503, or compatible), where:
  - 4 have tacho signals
  - 2 have PWM
- 16 buffered I/O pins (expandable via the I^2C connector)
- efficient power supply for the Raspberry Pi (using a Buck converter)

- free Python 2 software library
- feed through of the Raspberry Pi header (so you can add extra connections, such as buffered I/O or analog I/O.)

This is the circuit diagram for the Raspberry Pi LEGO board. You may only use this circuit, any other information and the library for non-commercial applications.

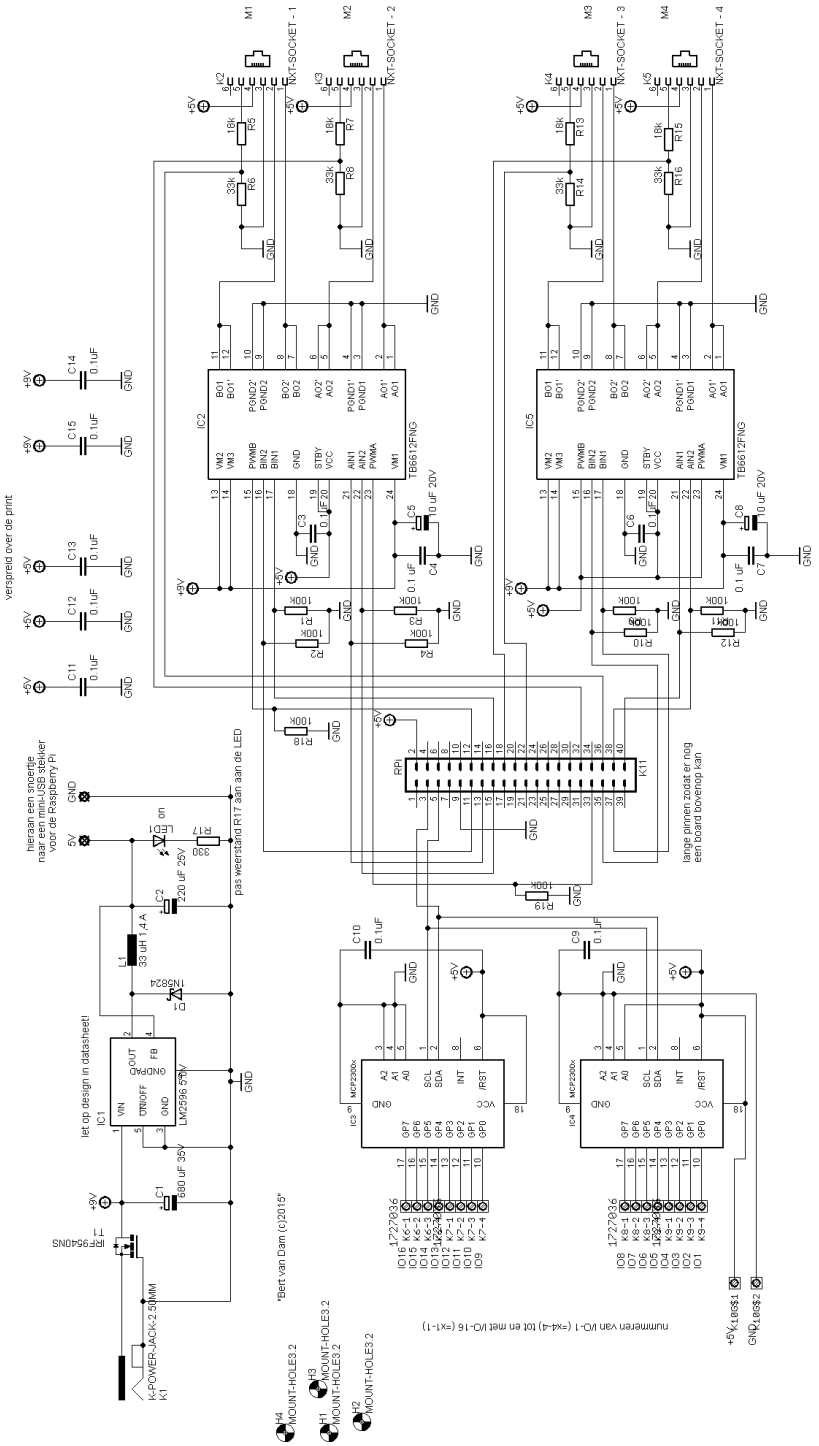


Figure 14.2. Circuit diagram for the LEGO Raspberry Pi board.

A few interesting details:

1. Because it is preferable to have a single power supply, the Raspberry Pi is powered via this board. The board itself is powered by a 9 V DC power supply. This can be made up of batteries, or a heavy-duty mains supply (5 Amp is recommended).

   The conversion of 9 V into 5 V (the required voltage for the Raspberry Pi) has to be done as efficiently as possible to prevent the batteries from draining too quickly. We've chosen a Buck converter for this, based on the LM2596 - a 5.0 V Simple Switcher® Power Converter 150 kHz 3A Step-Down Voltage Regulator.

   A Buck converter operates as follows: the ring made up of the diode (D1), the inductor (L1) and the capacitor (C2) forms a type of electrical flywheel. The LM2596 starts with turning its output fully on. This sets the flywheel in motion. This doesn't happen instantly, since the capacitor needs to be charged up via the inductor. The inductor itself creates a magnetic field. The diode won't conduct at this point, since the current flows round via the LM2596.

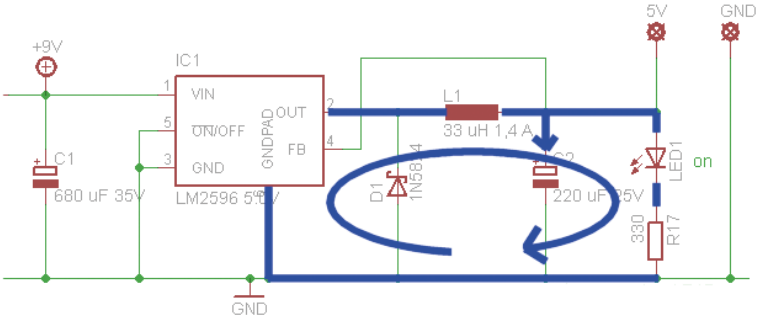


Figure 14.3. Buck converter: starting the flywheel.

   When the voltage at the output is about to become too high, the LM2596 switches its output off. The magnetic field of the inductor collapses, which causes a reversal of the voltage, and the diode can now conduct. This creates a current, which flows via the load and the diode (the path through the LM2596 no longer exists, since its output has been turned off). The capacitor discharges during this process. When the voltage at the output is about to become too low, the LM2596 turns on again.

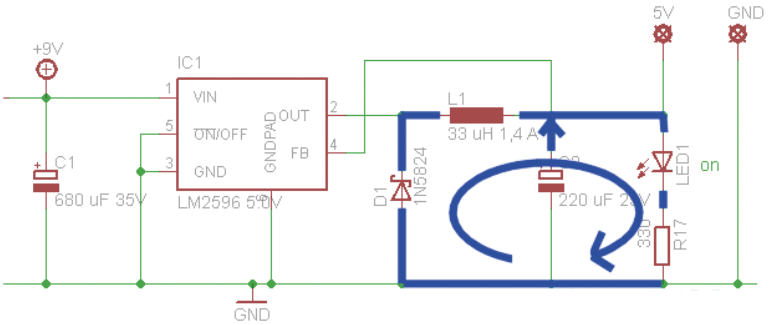


Figure 14.4. Buck converter: flywheel winding down.

The LM2596 therefore only turns fully on or off. The effect of this is that there are very few losses and the LM2596 hardly becomes warm. Due to the continuous switch on and off, the value of C1 is of importance, as is the current consumption.

2. A couple of MCP23008 I/O extenders have been used to provide protection of the Raspberry Pi I/O. They're controlled via the I^2C lines. We've already come across these chips in project 9.3. They have the following advantages:

    a. The I/O operates at 5 volt.

    b. The I/O can cope with larger currents (compared with the Raspberry Pi outputs)

    c. Should the worst happen, and something goes wrong, the damage should be limited to the I/O extender chip rather than the Raspberry Pi.

The maximum output current of an I/O pin of the MCP23008 is 25 mA, with the maximum per chip being 125 mA. Note that the maximum power that can be dissipated by a chip is 700 mW. This is for each MCP23008, so when you require "larger" currents, it's best to spread the load between the two MCP23008 chips, rather than connect everything to a single chip.

Since the Raspberry Pi pins have been fed through, you can connect extra I^2C components, as long as they have different addresses. The LEGO Raspberry Pi board uses addresses 0x20 and 0x21. You could add more I/O extenders for example, or A/D converters so that you can generate analog signals, or D/A converters for measuring analog signals.

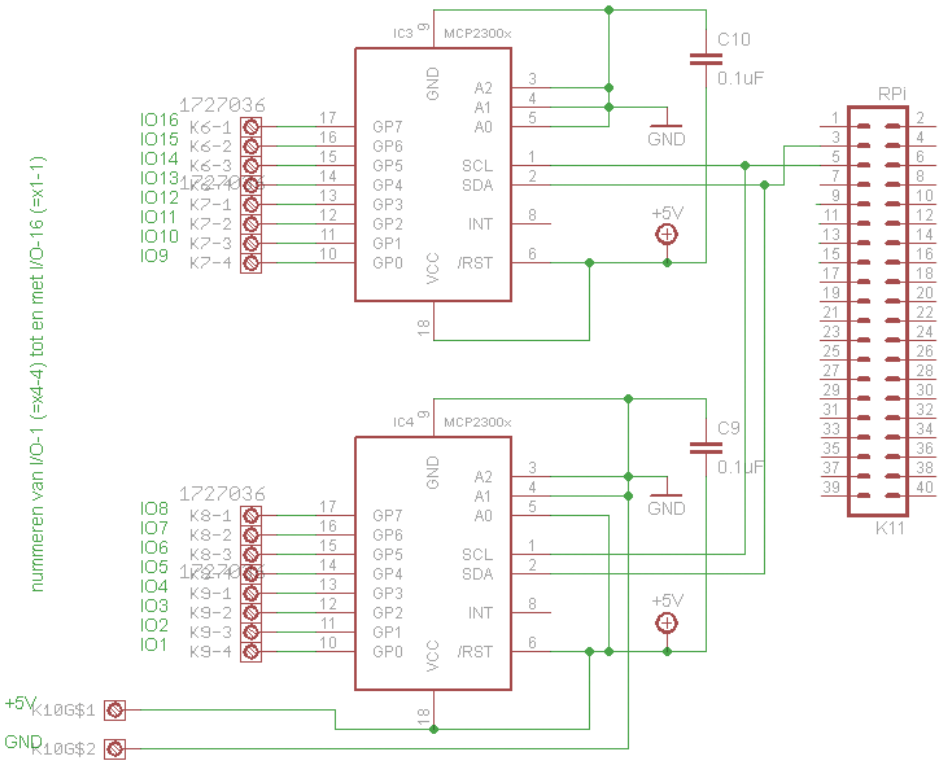


Figure 14.5. LEGO Raspberry Pi board I²C I/O.

1. The motors are driven by TB6621FNG Dual H-bridge chips, each of which takes care of two motors. The approximate current consumption of the motors is:

motor state	small EV3	large EV3
free running	85 mA	80 mA
barely turning	500 mA	1000 mA
stalled	780 mA	1800 mA

Table 14.2. Approximate current consumption of LEGO motors.

An EV3 motor can only be stalled for a short time, because the safety cutout of the motor will then turn it off. However, it is always wise to avoid stalling the motor because it could burn out if the protection fails to operate.

The TB6621FNG can normally supply 1.2 A (per channel, hence per motor) with a peak current of 3 A, which is more than enough.

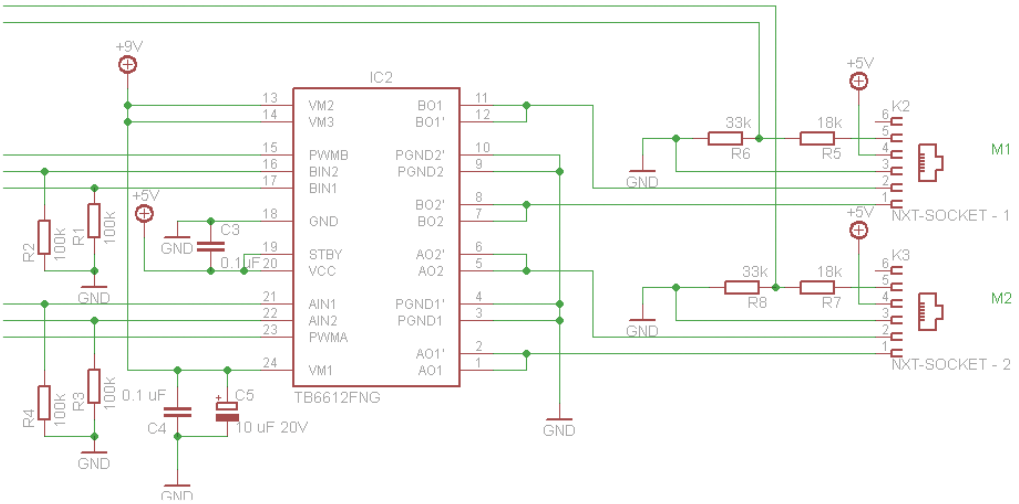


Figure 14.6. LEGO Raspberry Pi board motor driver (M1 and M2).

If you want to create your own library, bear in mind that the PWM pins are active-high on the TB6621FNG. A PWM signal fed to these pins will always work, irrespective of the direction of the motor.

14.3 Commands

The library uses threading for the implementation of PWM. This means that the library cannot be loaded in the usual way, but should be loaded like a program, using the command `execfile('rpirobot.lib')`. Apart from this, the library works like any other normal Python library.

The library contains the following commands:

General	Description
`execfile('rpirobot.lib')`	Loads the library and starts the threads.
`time.sleep(0.000001)`	A short pause, to ensure that the threads also multi-task on multi-core Raspberry Pi model 2 and higher.

Table 14.3. General library commands.

Inside the loops you have to ensure that the Raspberry Pi has sufficient time to deal with the PWM threads that have been started by the library. This is perticularly the case with Raspberry Pi models 2 and higher, since these have a multi-core processor. At the time of writing, the supplied operating system still has problems with inter-core threading for these models.

Protected I/O	Description	Example [92]
`iodir (pin, in/out)`	pin: 1-16 in/out: 0 (output) 1 (input) Set a pin on the I/O strip as input or output.	pin 9 output: `iodir(9,0)`
`value= ioread(pin)`	pin: 1-16 value: 0 (off) of 1 (on) Read the value of a pin on the I/O strip.	Determine the value of pin 6: `value=ioread(6)`
`iowrite(pin, value)`	pin: 1-16 value: 0 (off) or 1 (on) Set a pin on the I/O strip on or off.	Turn pin 16 on: `iowrite(16,1)`

Table 14.4. I/O library commands.

"Forwards" and "reverse" is a relative concept for motors. When two motors receive the same command via different outputs, they will turn in the same direction.

Motors	Description	Example
`forward(motor)`	motor: 1-4 Turn the motor forward (motors 1 and 2 turn at a speed set by their respective RPM).[93]	Make motor 3 turn forward: `forward(3)`
`reverse(motor)`	motor: 1-4 Turn the motor in reverse (motors 1 and 2 turn at a speed set by their respective RPM).	Make motor 2 turn in reverse: `reverse(2)`
`brake(motor)`	motor: 1-4 Brake and stop the motor.	Brake and stop motor 1: `brake(1)`
`float(motor)`	motor: 1-4 Let the motor freewheel until it stops by itself. The brake is off, and the motor can be turned manually.	Let motor 4 freewheel: `float(4)`

`pwm(motor, value)`	motor: 1-2
	value:0-100
	Set the power level for the motor (note that this command doesn't start the motor)
		Make motor 1 turn forward at 30% power.
		`forward(1)`
		`pwm(1,30)`

92 The examples assume that `iodir()` has been used to set the pins to the required mode.
93 When the PWM for motor 1 or 2 has not yet been turned on, the motor won't start!

`value=tacho(motor)`	motor=1-4 value=0 (low) or 1 (high) Determines the tacho state of a motor.	Look at the tacho state for motor 4: `value=tacho(4)`

Table 14.5. Motor library commands.

We'll immediately put this knowledge into practice. Connect a LEGO motor to connection 1. Ensure that you're in the directory for this project (since this is where the library is) and start Python with the command:

```
python
```

Next, start the library. If you get an error, you're probably not in the lego directory:

```
execfile('rpirobot.lib')
```

Set the speed of the motor and make it turn forward. The motor now turns forward with a PWM speed of 30%.

```
pwm(1,30)
forward(1)
```

When you're finished you should stop the motor and close Python.

```
float(1)
exit()
```

The full sequence of events looks as follows:

```
pi@raspberrypi:~/lego $ python
Python 2.7.9 (default, Jul  8 2017, 00:52:26)
[GCC 4.9.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> execfile('rpirobot.lib')
>>> pwm(1,30)
1
>>> forward(1)
1
>>> float(1)
1
>>> exit()
pi@raspberrypi:~/lego $
```

You can see that the LEGO library responds to the commands with a 1. This means that the command was recognized and executed successfully. You will get a response of -1 when this isn't the case. You can easily check this by trying to set the PWM for motor 3 to 20%, since motor 3 doesn't have any PWM:

```
>>> pwm(3,20)
-1
```

14.4 The Useless Box

The concept for the useless box is a box with a switch on top. When you turn the switch on, a finger comes out of the box that turns the switch off again. We'll only build the internal mechanism in this project; you will have to build the box yourself. A cardboard box is easiest, although wood is stronger and looks better. There is a short video included with the download for this project, which shows the mechanism in operation. Both the photos in this chapter and the video only show the mechanical side of the project, without any box. This way you can see much better how it all works (and I'm not quite good enough to make a neat box).

We'll be using the "large Lego motor", type LEGO 45502, which can be found in the Lego Mindstorms EV3 box, and other parts as shown in Figure 14.7. There is a short video in the download that shows how to use these parts to construct the motorized finger shown at the bottom of the photo. You then have to carefully fix a toggle switch to the top platform in such a way that the end of the finger can switch it off.

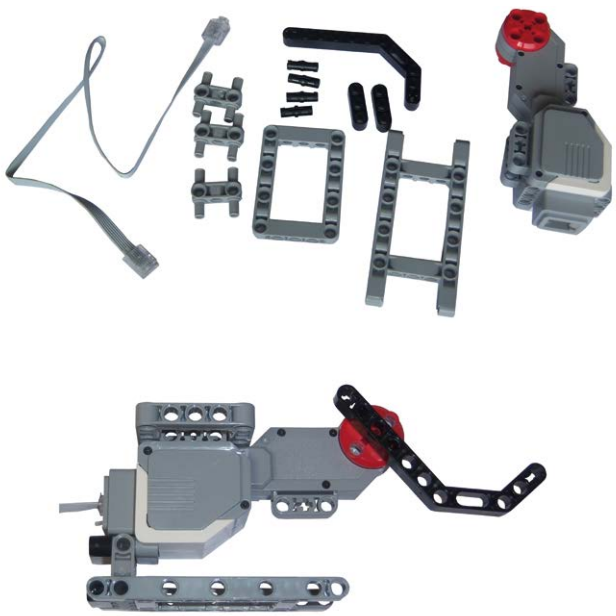


Figure 14.7. Parts for the motorized finger, and a completed model.

This switch is connected via two resistors (see Figure 14.9) to I/O 16. It could be done with a single resistor (the 10k one), but the second resistor is there for protection. If you accidentally set the input as an output and make it low, nothing untoward happens. Without the extra 1k resistor it's possible that you could damage the I/O.

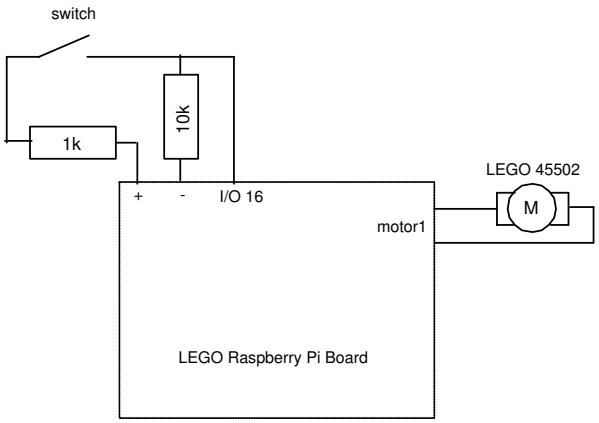


Figure 14.8. Connection diagram.

The software is surprisingly straightforward.[94] We first load the `rpirobot` library, and immediately start it with the command `execfile`. The motor outputs on the Raspberry Pi LEGO board will be configured automatically, and the PWM threads will be running. Next we configure pin 16 of the Raspberry Pi LEGO board I/O as an input, since that's connected to the switch.

```
execfile('rpirobot.lib')

# Switch on io 16, input
iodir(16,1)

print "Useless Box Program running "
```

We make use of a try/except construct so that the user can stop the program with a Ctrl-C. We then wait in a loop for the user to turn the switch on. When this happens, the program waits half a second for the user to pull their finger out of the way, then starts turning the LEGO finger with PWM 60 (60 % of the maximum speed). We wait in a loop because of the threads. It doesn't help that nothing happens inside this loop, so we add a very short `sleep` command to it. This command takes 0.000001 second, which is one millionth of a second, which won't have any noticeable effect to the speed of the program. As soon as the switch is off, the program puts the brakes on the motor.

94 Note that the library expects Python 2 programs! More information about the library and the commands can be found in the manual that is part of the free download for this board.

```
try:
  while 1:
    # wait for the user
    # to push switch on
    if ioread(16):
      # push switch off
      time.sleep(0.5)
      reverse(1)
      pwm(1,60)
      while ioread(16):
        time.sleep(0.000001)
      brake(1)
```

The next step is to make the LEGO finger go back at a slower speed (PWM 30, 30 % of the maximum speed). This time there isn't a switch at the end of the movement, so the program uses the tacho. The program counts the number of tacho changes and when it gets to 100 (28 % of a full circle) the motor is stopped using the `float` command. The LEGO finger then ends up about a centimeter above the base.

```
    # retract lever
    forward(1)
    pwm(1,30)
    counter1=0
    lastpin1=tacho(1)
    while counter1<100:
      time.sleep(0.000001)
      if tacho(1)!=lastpin1:
        counter1+=1
        lastpin1=tacho(1)
    # float motor
    float(1)
```

The `except` construct catches errors, which includes the pressing of the Ctrl-C keys by the user. The program then gives the `abort` command to the Raspberry Pi library and finally stops itself. It would be possible to stop the program via Ctrl-C without this construct, but in that case the library would still be running, which is something that we don't want.

```
except:
  # user aborts program with Ctrl-C
  print "Program aborting"
  abort()
  print "Done"
```

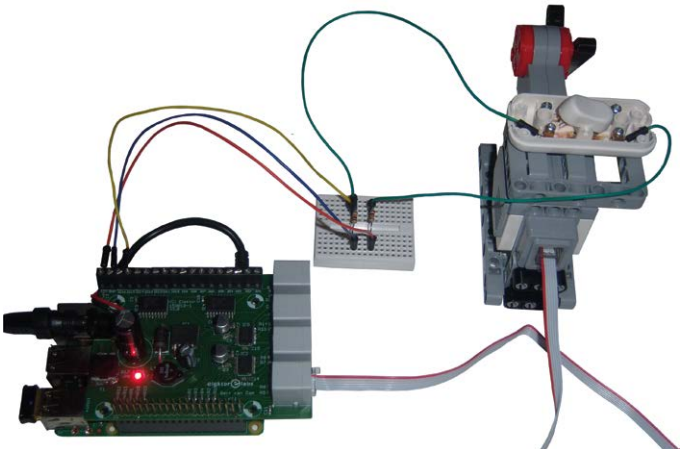


Figure 14.9. The complete model.

Operating procedures:

1. Move the LEGO finger to its rest position, which is as far away as possible from the switch (on the base, for example).

2. Start Raspberry Pi and wait for the start-up procedure to finish.

3. Go to the directory for this project.

4. Enter the command: `python uselessbox.py`

5. Wait for the text `Useless Box Program running` to appear on the Raspberry Pi screen, then turn the switch on (remember to pull your finger away!). The LEGO finger will now turn the switch off again.

6. You can repeat step 5 as often as you like, then stop the program using Ctrl-C.

A short video has been included with the download for this book, which shows the Useless Box in operation.

Appendix A

1 Adjustable power supply

The adjustable power supply described in this appendix has an output covering 1.2 to 13 V. This power supply is perfect for all the projects in this book that require an external supply, since this was the supply that was used during the writing of the book. When a heatsink with an area of about 10 cm^2 is used, the power supply can supply a maximum of 1.5 A, which is more than enough.

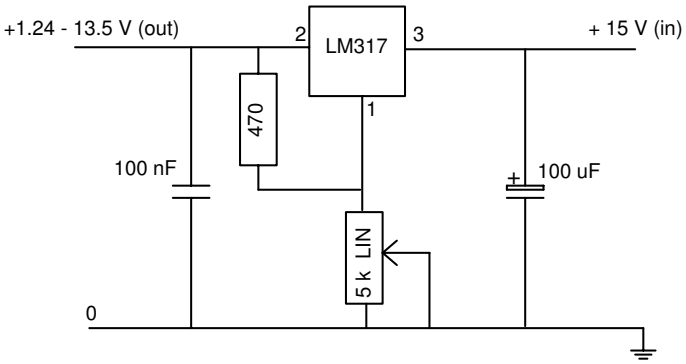


Figure A.1. Circuit for the adjustable power supply.

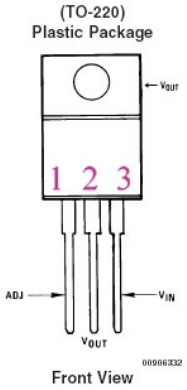


Figure A.2. Pinout for the LM317.

You will need a multimeter to determine the position of the potentiometer required for the different voltages. When you build this power supply into a box with a scale for the potentiometer you only need to carry out the calibration once. This project is powered by a 15 V DC "wall wart" (a plug with a built-in power supply). If you want to use a different voltage, you should read the following notes and change the component values as appropriate. If you want to use an AC supply you should add a bridge rectifier with a smoothing capacitor to turn it into DC.

Figure A.3. The power supply in a box.

The LM317 is a 1.2 V voltage regulator. This means that the LM317 will try to maintain a voltage difference of 1.2 V between pins 1 and 2. The maximum output voltage we want is 13 V, which should be possible with a 15-V transformer. We use a 5k LIN potentiometer to adjust the voltage. At the maximum output voltage the voltage across the potentiometer is 11.8 V (13 - 1.2) and the current through it is 2.36 mA.

V_{max} = 13
V_{var} = 13 - 1.2 = 11.8 V
I_{var} = V/R = 11.8 / 5 k = 2.36 mA

This current doesn't flow through the LM317 so we use a bypass resistor. Since the voltage drop across the LM317 is 1.2 V and we've just calculated the current, it's easy to determine the resistor value:

R_{fix} = V/I = 1.2 / 2.36 10^{-3} = 508 Ω.

This value doesn't exist, and the nearest values we can use are 470 and 560 Ω. We'd rather have a slightly higher maximum voltage so we choose the 470 Ω resistor (yellow-purple-brown). If you want to add an LED as power indicator you should connect it to the 15 V input along with a 1k current limiting resistor (brown-black-red).

2 GPIO header circuit
The even numbered pins are all on one side of the header, with the odd-numbered pins on the other side. The even numbers are closest to the edge of the Raspberry Pi.

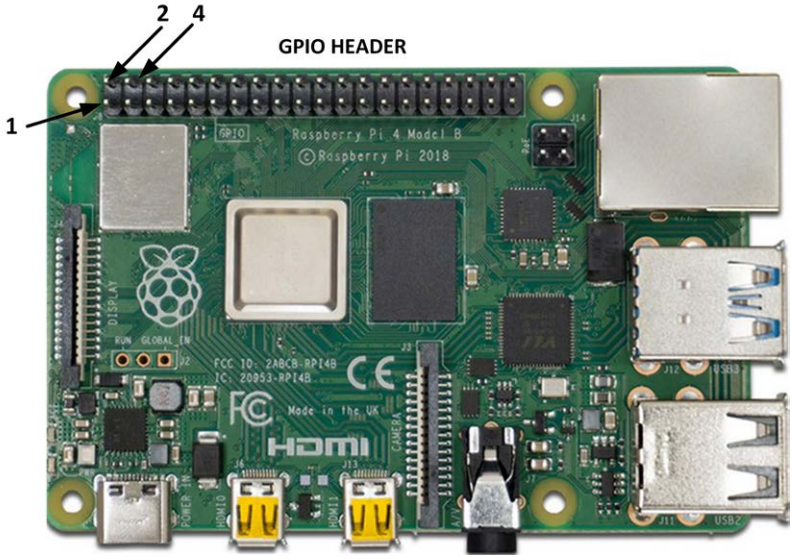


Figure A.4. The header on the Raspberry Pi.

The GPIO header provides various functions, such as ordinary pins (in this book we call them GPIO), I^2C, SPI, PWM and serial. Some of the pins are not used on model 1; they are marked 'a'. For example pin 4, which is not used on model 1 but supplies +5 V on model 2 and up.

GPIO number	function	header pin number		function	GPIO number
	+3.3 v	1	2	+5 V	
2	I2C SDA	3	4a		
3	I2C SCL	5	6	GND	
4		7	8	serial TXD	14
	n/a	9a	10	serial RXD	15
17		11	12	PWM0	18
27		13	14a	n/a	
22		15	16		23
	n/a	17a	18		24
10	SPI MOSI	19	202	n/a	
9	SPI MISO	21	22		25
11	SPI SCLK	23	24	SPI CE0	8
	GND	25	26	SPI CE1	7
	ID-SD	27	28	ID-SC	

5		29	30	GND	
6		31	32	PWM0	12
13	PWM1	33	34	GND	
19	PWM1	35	36		16
26		37	38		20
	GND	39	40		21

Table A.1. Function overview of header pins.

From Table A.1 you can see that the GPIO numbers do not correspond to the header pin numbers. For example, GPIO 2 is assigned to pin 3 and we find GPIO 7 at pin 26. Make sure to keep this in mind when you are assembling the circuit! In the download the file "header.bmp" is included, which you can print so you can have the pin layout at hand.

The designed output current for the pins is 3 mA per pin. The total for all pins, including the +3.3 V connection itself, is 51 mA. When you don't use all of the pins, you can use more than 3 mA, up to a maximum of 16 mA per pin. However, the total may never be more than 51 mA. It's recommended to keep below 3 mA per pin as much as possible, which we've done for the projects in this book.

With a 1 A power supply you can take about 250 mA (model B) to 300 mA (model A) from the +5 V pin. In practice this will also be dependent on the current consumption of other devices that are connected to the Raspberry Pi, such as a keyboard and suchlike.

Appendix B

1 Contents of the download package

Raspberry Pi

All the software required for the Raspberry Pi is pre-installed on the SD card that you can buy for this book. This means that all the software and drivers for SSH, SPI, I^2C, GPIO, PWM, sound, Python, wxPython, Thonny and the serial connection are guaranteed to work. All of the programs used in the projects in this book have also been included on the card.

The programs on the SD card are under the folder called book. The sub-folders ae organized based on the chapter numbers. The structure of this folder and the files in this folder are as shows below:

```
book
├── 04
│   └── template.pyw
├── 05
│   ├── 5.03
│   │   ├── askled.py
│   │   ├── flashled1.py
│   │   └── flashled.py
│   ├── 5.04
│   │   └── flashled2.py
│   ├── 5.05
│   │   └── timerled.pyw
│   ├── 5.06
│   │   ├── switchled.py
│   │   └── switch.py
│   ├── 5.07
│   │   └── timerled.py
│   ├── 5.08
│   │   └── twostate.py
│   ├── 5.09
│   │   └── readswitch1.pyw
│   ├── 5.10
│   │   ├── blonde.mp3
│   │   └── soundswitch.py
│   └── 5.11
│       ├── pinread.py
│       └── pinset.py
├── 06
│   ├── 6.1
│   │   └── td62783a.py
│   ├── 6.2
│   │   └── uln2003a.py
```

```
│   └── 6.3
│       └── irf740a.py
├── 07
│   ├── 7.1
│   │   └── pwm1.py
│   └── 7.2
│       └── morning.pyw
├── 08
│   ├── 8.2
│   │   └── max522a.py
│   ├── 8.3
│   │   └── max522b.pyw
│   ├── 8.4
│   │   └── max522c.py
│   ├── 8.5
│   │   ├── mcp3008a.py
│   │   └── mcp3008b.pyw
│   └── 8.6
│       └── mcp3008c.py
├── 09
│   ├── 9.2
│   │   └── mcp23008a.py
│   └── 9.4
│       ├── tc74b.pyw
│       └── tc74c.pyw
├── 10
│   └── 10.2
│       ├── graph.png
│       ├── serialadc.py
│       └── serialadc.pyw
├── 11
│   └── server
│       ├── cgi-bin
│       │   ├── counter.dat
│       │   ├── counter.py
│       │   ├── gpiocommand.dat
│       │   ├── gpioled.py
│       │   ├── gpioserver.py
│       │   ├── gpioswitch.py
│       │   ├── hello.py
│       │   ├── setcounter.py
│       │   ├── sum.py
│       │   ├── tc74remote.py
│       │   ├── tc74server.py
│       │   └── temperature.dat
│       ├── favicon.ico
```

```
│       └── index.html
├── 12
│   ├── 12.2
│   │   ├── tcpclient1.py
│   │   └── tcpserver1.py
│   ├── 12.3
│   │   ├── tcpclient1.py
│   │   └── tcpserver2.py
│   ├── 12.4
│   │   ├── tcpclient1.py
│   │   └── tcpserver3.py
│   ├── 12.5
│   │   ├── examplethread.py
│   │   └── tcpserverthread2.py
│   ├── 12.6
│   │   ├── tcpclient2.pyw
│   │   └── tcpserver4.py
│   ├── 12.7
│   │   ├── udpclient1.py
│   │   └── udpserver1.py
│   └── 12.8
│       ├── udpclient2.pyw
│       └── udpserver2.py
└── 13
    ├── 13.3
    │   └── rpirobot.lib
    └── 13.4
        ├── rpirobot.lib
        └── uselessbox.py
```

Using the SD Card
The steps for using the SD card of this book are as follows:

If you wish to use remotely from a PC:

- Insert the SD card into your Raspberry Pi 4
- Connect your Raspberry Pi directly to your router hub
- Power up the Raspberry Pi
- Find the IP address of your Raspberry Pi from the router
- Login to your Raspberry Pi using Putty. The default username and password are pi and raspberry respectively
- Configure the WiFi on your Raspberry Pi to match your wireless router name and password
- Disconnect from Ethernet
- Login using Putty
- Access the programs in folder called book

If you wish to use a directly connected monitor:

- Insert the SD card into your Raspberry Pi 4
- Connect the monitor and keyboard to your Raspberry Pi
- Connect your Raspberry Pi to your router hub (if you wish to use the Ethernet)
- Power up the Raspberry Pi
- Login to your Raspberry Pi
- Configure the WiFi on your Raspberry Pi (if you wish to use the WiFi)

Windows PC
The free download package for this book contains all the software used in this book for Windows PCs, neatly packaged into a single ZIP file. All of the software is free, so there is nothing to pay. You don't even need to own this book to download the package. The package is intended for use on Microsoft Windows XP or later versions of Microsoft operating systems.

A video showing the Useless Box from project 14.4 is also included.

Software
WinOscillo
Putty
Xming
WinSCP
DiskImager
Notepad2
Useless Box video

Installation instructions:

1. Go to the elektor webpage of this book.
2. Download the software package to your PC.
3. Unzip the file to a folder of your choice, keeping the folder structure intact. You should follow the instructions in this book for the individual programs; you don't have to do anything else right now.

2 Parts list
This is the parts list with the components you'll need to carry out all the projects in this book. You should ask for components that fit on a breadboard or stripboard. Some components will have pins that are too thick for a breadboard. You should solder thinner leads to these pins. Never force them into the breadboard as you could bend the internal contacts and cause permanent damage to the breadboard.

You can buy a starter pack that has a selection of the components needed to build the projects in this book, including a set of useful extension cables to connect the header on the Raspberry Pi to a breadboard. Visit the support page for more information and ordering details.

Component	Requirement
Breadboard	1x
Resistors (1/4 W)	1x 47, 2x 680, 1x 1k, 2x 1k8, 2x 3k3, 1x 4k7, 1x 10kΩ
Switch	1x single pole push to make
Presets	2x 10k LIN
LEDs	2x 2 mA (yellow)95, 1x 20 mA (red)
Male-female cables	10x
Capacitors	1x 0.22uF, 5x 100 nF
Miscellaneous	TD62783 of UDN2981 ULN2003 1N4007 MAX4420 IRF740 LM358 MCP3008 MCP23008 LDR TC74 A0 5.0VAT MAX522
These components were used in the book as examples to illustrate a point. You should use your own comparable "power" components instead.	Light 6 V/65 mA Motor 5 V/145mA Motor 5 V/500mA Fan 12 V/125mA Lego motor type 45502 Toggle switch

Table B.1.Components used in this book.

This overview doesn't include the interface you need for WinOscillo and the Spectrum Analyzer, nor the components required to make the variable external power supply.

95 Everlight 264-10UYD/S530-A3/T2 or similar two mA type.

Appendix C

Python 2.x vs Python 3.x

In this book we have been using Python 3.x in all the projects. The readers should be aware that Python 2.x is still very popular an is used by many people around the world. Although Python 3.x is very similar to Python 2.x, there are some important differences between the two versions and readers who wish to move from Python 2.x to Python 3.x shold know these differences before migrating to the new version.

A list of all the differences between the two versions is available at the following web site:

https://docs.python.org/3/whatsnew

A list of some of the important changes between the two versions is summarized below.

Python 2.x	Python 3.x
`unequal sign <>`	`!=`
`print "hello"`	`print("hello")`
`raw_input()`	`input()`
5/2 = 2	5/2 = 2.5
Text and (binary) data mixed	Text and (binary) data are different. All text is Unicode, but encoded Unicode is represented as binary data. Binary data can be converted to strings using the decode function. Similarly, strings can be converted to binary data using encode function. For example: `use str.encode()` to go from str to bytes, and use: `bytes.decode()` to go from bytes to str. The default is to use `utf-8` in encode/decode.
`u` used for Unicode	`u` cannot be used for Unicode. Use `b' '` for binary data True, False and None are reserved words.
`exec()`	`exec(f.read())` From in import name is only allowed at the module level, no longer inside functions. Many modules removed from the library.
`round(2.5) = 3`	`rount(2.5) = 2`

Index

Explore the Raspberry Pi
in 45 Electronics Projects

Bert van Dam is a freelance author of project books, courseware and articles on microcontrollers including ARM and PIC devices. In his publications he also covers Raspberry Pi, artificial intelligence and several programming languages like JAL, C, assembler, Python, and Flowcode.

The Raspberry Pi is a very cheap but complete computer system that allows all sorts of electronics parts and extensions to be connected. This book addresses one of the strongest aspects of the Raspberry Pi: the ability to combine hands-on electronics and programming.

Combine hands-on electronics and programming

After a short introduction to the Raspberry Pi you proceed with installing the required software. The SD card that can be purchased in conjunction with this book contains everything to get started with the Raspberry Pi. At the side of the (optional) Windows PC, software is used which is free for downloading.
The book continues with a concise introduction to the Linux operating system, after which you start programming in Bash, Python 3 and Javascript. Although the emphasis is on Python, the coverage is brief and to the point in all cases - just enabling you to grasp the essence of all projects and start adapting them to your requirements. All set, you can carry on with fun projects.

The book is ideal for self-study

No fewer than 45 exciting and compelling projects are discussed and elaborated in detail. From a flashing lights to driving an electromotor; from processing and generating analog signals to a lux meter and a temperature control. We also move to more complex projects like a motor speed controller, a web server with CGI, client-server applications and Xwindows programs.

Each project has details of the way it got designed that way

The process of reading, building, and programming not only provides insight into the Raspberry Pi, Python, and the electronic parts used, but also enables you to modify or extend the projects any way you like. Also, feel free to combine several projects into a larger design.

ISBN 978-1-907920-82-0

9 781907 920820

Elektor International Media

www.elektor.com

elektor

LEARN DESIGN SHARE