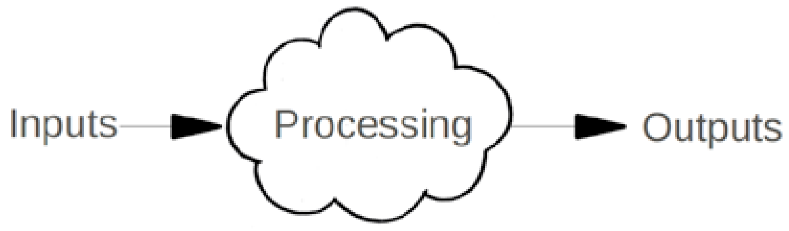
Notes:

Hear the word "**computer**" and you probably think of a box with a monitor attached, like a desktop PC. But these days, almost every electronic device, from your TV or your parents' satnav, right up to the auto-pilot in a jumbo jet, is controlled by a computer. Computers are everywhere.

In order to do the job they're programmed for, computers need to receive information from the real world telling them what to do; these are called "**inputs**". An input could be a user typing a command or clicking on something with the mouse, or it could be readings from the sensors on an aeroplane wing, telling the auto-pilot the wind speed, air pressure and compass location of the plane. It all depends on the type of computer and the type of program.

To function, computers need inputs – information from an external source. They process this information and produce a result, called an "output".



Fun with ports

Inputs and outputs could also come in the form of a data connection, such as a network link to the internet. Think of the internet, and you probably think of web pages. In fact, the World Wide Web is only one of the many internet applications.

There are lots of other ways to use the internet, such as email, instant messaging, text-based newsgroups or logging on to another computer using the Secure Shell (SSH) network protocol. Each of these different ways of using the internet has a port number associated with it.

By "**port**", I don't mean an actual physical connection on your computer, such as a USB or FireWire port. In this context, a port is a software connection. A port can either be open or closed. If it's open, then that simply means that the program will accept connections over it. If the port is closed, then the program won't accept connections and the service in question won't accept any inputs over the network.

For instance, the HTTP protocol used by the World Wide Web accepts inputs using the TCP port 80. If port 80 is open, a web server will respond to a request from a client, such as your web browser, by displaying the requested web page. If one of that server's administrators has closed port 80, then you won't be able to access the website and your browser will display an error message.

Notes:

Other common ports include TCP port 25, which is used for the SMTP email protocol; ports 20 and 21, which are used for the file transfer protocol (FTP); and port 161, for the SNMP network management protocol. There are thousands of different ports available to use and hence thousands of different ways to communicate on the internet!

In this chapter, we are going to use your Raspberry Pi to interface with a range of internet applications, including Twitter and email. We'll communicate with each program via the appropriate port, providing input that causes that program to run a specific function. And we're going to do it all using the Python programming language. You won't touch a browser once.

All the examples in this section are designed to be as simple as possible. Each exercise demonstrates just one type of communication. Think of these examples as ingredients in a recipe that, when mixed together, produce something much better than the individual parts. The extent of what you make is only limited by your imagination! You have your Raspberry Pi – now get cooking!

What tools will I need?

To complete the exercises in this chapter you will need the "**IDLE**" program that you used in the last chapter. Some of the exercises also require you to install extra Python modules. I'll tell you when this is the case. Wherever possible, I have made these extras available on the SD card that came with your Raspberry Pi, so look out for the SD card logo!

These resources are also available on Google Drive, here: ***http://goo.gl/vK3VP***

Lesson 4.1: Twitter

To complete this exercise you will need to install the Python Twitter Tools. You can either download these from ***http://pypi.python.org/pypi/twitter/*** or install them from the SD card that came with your Raspberry Pi.

The following Python script will tweet the message "Hello, World!" from your Raspberry Pi. It then displays a list of all your personal tweets. No web browser in sight! This example shows how simple this type of application can be, using Python.

```
import os
from twitter import *

# go to https://dev.twitter.com/apps/new to create your own
# CONSUMER_KEY and CONSUMER_SECRET
# Note that you need to set the access level to read and write
# for this script to work (Found in the settings tab once you
# have created a new application)
CONSUMER_KEY = '9HvofZMpvHo0KGZyg9ckg'
CONSUMER_SECRET = 'S75MwXN2H0h2qIYszc51WtHTbpbouhDkr6CCTBPzA'
# get full pathname of .twitterdemo_oauth file in the
# home directory of the current user
oauth_filename = os.path.join(os.path.expanduser('~'),
'.twitterdemo_oauth')

# get twitter account login info
if not os.path.exists(oauth_filename):
    oauth_dance('Raspberry Pi Twitter Demo', CONSUMER_KEY,
CONSUMER_SECRET, oauth_filename)
(oauth_token, oauth_token_secret) = read_token_file(oauth_filename)

# log in to Twitter
auth = OAuth(oauth_token, oauth_token_secret, CONSUMER_KEY,
CONSUMER_SECRET)
twitter = Twitter(auth=auth)

# Tweet a new status update
twitter.statuses.update(status="Hello, World!")

# Display all my tweets
for tweet in twitter.statuses.user_timeline():
    print('Created at',tweet['created_at'])
    print(tweet['text'])
    print('-'*80)
```

Over to you

Try to think of other ways in which you could use this mechanism. For instance, take an input from somewhere, process it and tweet the result. You could detect the weather outside and tweet the result. You could even make a tweet from your mobile phone and have your Raspberry Pi read and react to it (by playing some music, for example)!

Notes:

Tip...

As in the previous chapter, we sometimes have more code to write than fits on one line. When you see a line of code that goes onto a new line, don't press Return to start a new line, just let it word wrap and Python will work it out for you.

Lesson 4.2: Email application

Notes:

Email is transferred across the internet using something called SMTP – Simple Mail Transfer Protocol. SMTP uses port 25. Don't worry; you don't need to understand how SMTP works to be able to send an email. There is a Python module that does all the hard work for you.

Use the Python script below to send a single email.

email-example.py:

```
import smtplib
from email.mime.text import MIMEText

# create the email
message = """This is a test of using Python on Raspberry Pi
to send an email. Have fun!"""
msg = MIMEText(message)
msg['Subject'] = 'RPi Python test'
msg['From'] = 'My RPi <my_rpi@example.com>'
msg['To'] = 'you@yourdomain.com'

# send the email
s = smtplib.SMTP('smtpserver')
s.login('username', 'password')
s.sendmail(msg['From'], msg['To'], msg.as_string())
s.quit()
```

You can modify the program to send email attachments as well as simple text. The example on the next page sends an image (such as a JPG file) as an attachment to the message.

email-attachment.py:

```
import smtplib
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText
from email.mime.image import MIMEImage

# create multipart email
msg = MIMEMultipart()
msg['Subject'] = 'Lily'
msg['From'] = 'My RPi <my_rpi@example.com>'
msg['To'] = 'you@yourdomain.com'
msg.preamble = 'This is a multi-part message in MIME format.'

# attach email text
message = """This is a test of using Python on Raspberry Pi
to send an email. This email also includes a picture as an
email attachment. Have fun!"""
msg.attach(MIMEText(message))

# attach a JPG file
filename = 'picture.jpg'
with open(filename, 'rb') as f:
    img = MIMEImage(f.read())
img.add_header('Content-Disposition', 'attachment', filename=filename)
msg.attach(img)

# send email
s = smtplib.SMTP('smtpserver')
s.login('username', 'password')
s.send_message(msg)
s.quit()
```

Notes:

Over to you

Again, try and think of other ways you could use this script. For instance, you could generate the content of the email (the output) based on an input. For example, checking the room temperature and emailing a warning if it is getting too hot. This is regularly used in server rooms, for instance, to detect when the air conditioning has failed.

NOTE: You will have to change the text highlighted in yellow to your own email address and email server details for this program to work.

There is more information about the email modules in Python available at ***http://docs.python.org/py3k/library/email.html***

Lesson 4.3: Remote Procedure Call

Notes:

A **Remote Procedure Call** (**RPC**) is bit of code in one program that causes another program to do something, for instance run a subroutine (a bit of a program that performs a specific task). The two programs don't necessarily have to be on the same computer, although often they are.

When using RPC, calling a subroutine looks the same in your code as if it was being executed as part of your program. The communication between programs remains largely hidden and the programmer does not have to worry about the details of how it is done.

The language the programs are written in and type of computers do not have to be the same either – they can be on completely different hardware, operating systems and programming languages. For example, a mobile phone app could be programmed to call a Python program running on a Raspberry Pi to perform a task and return a result.

The type of Remote Procedure Call we are using for this example is called **XMLRPC**. Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding data in a format that is both human-readable and machine-readable.

There are two programs here – a "**server**" and a "**client**". The server sits waiting for one or more clients to ask it to do something.

rpc_server.py:

```
from xmlrpc.server import SimpleXMLRPCServer

def hello(name='World'):
    """Return a string saying hello to the name given"""
    return 'Hello, %s!' % name

def add(x, y):
    """Add two variables together and return the result"""
    return x + y

# set up the server
server = SimpleXMLRPCServer(("localhost", 8000))

# register our functions
server.register_function(hello)
server.register_function(add)

# Run the server's main loop
server.serve_forever()
```

rpc_client.py:

```
import xmlrpc.client

s = xmlrpc.client.ServerProxy('http://localhost:8000')

# call the 'hello' function with no parameters
print(s.hello())

# call the 'hello' function with a name
print(s.hello('XMLRPC'))

# add two numbers together
print('2+3=%s' % s.add(2,3))
```

Notes:

To run this example, you need two separate IDLE windows – one for the server and one for the client. You need to run the server first so that it is waiting for requests from the client. To stop the server, press Ctrl-C.

How does it work? The server program sits and waits for requests from clients. The client program, when run, calls the "hello" and "add" subroutines. The calls are passed from the client to the server program, where the functions are actually run. The result is then returned to the client program, which displays the result on screen.

Where could this XMLRPC mechanism be used? Well, imagine that you have several Raspberry Pis, each one connected to a screen located in a different room. You also have a central computer running a control application with a Graphical User Interface (GUI) designed to talk to every room over the network.

This central application could monitor and control every room by communicating with individual Raspberry Pis – telling them what to display on the screen, control devices via GPIO, and return data recorded from the room (such as light levels or temperature). You could even take pictures with a camera! Each Raspberry Pi would run as an XMLRPC server; the central control program would act as the client.

Over to you

You can read more about Python's XMLRPC modules here:
http://docs.python.org/py3k/library/xmlrpc.server.html
http://docs.python.org/py3k/library/xmlrpc.client.html

There are, however, XMLRPC libraries available in lots of other programming languages.

Lesson 4.4: Web applications

Notes:

A **web application** is a program that communicates on the World Wide Web. Like any other program, it has inputs, processes them and produces some output. Not every input and output from a web application has to be through a web browser, though!

The World Wide Web uses something called **HTTP** (Hyper Text Transfer Protocol). This is what web servers and web browsers use to communicate with each other at a low level. The port number used for HTTP is 80. The detail of how HTTP works does not need to concern us at this stage – all the work is done for us by web browsers and libraries in Python.

A **URL** (Uniform Resource Locator), otherwise known as a web address, is what you would type into the address bar of a web browser. It can be made up of a few parts:

protocol :// hostname : port / address

Let's look at those parts in order:

Protocol
For a website, this is normally "HTTP:" but doesn't have to be. Secure websites use the encrypted "HTTPS:" protocol, and most browsers can also work with the file transfer protocol ("FTP:").

Hostname
This is the name of the computer, the web server, on which the website resides. In these examples, we use the hostname "localhost", which means the computer on which a program resides.

Port
This is a number between 1 and 65,535, specifying the TCP port to be used for the communication. If not specified, the default port, 80, is used.

Address
This is the address of the specific webpage that you want to view. For instance, in the URL ***www.raspberrypi.org/faqs*** the suffix "faqs" is the address.

HTML (Hypertext Markup Language) is a language used to describe the layout of a web page. In both of these examples here, you will notice that the web page created is fairly basic. Imagine how you can improve this just by modifying the HTML!

Tip...

*If you want to find out the **IP address** of your Raspberry Pi, open the Terminal, type the command "ifconfig" and press Return. This command does the same job as "ipconfig" on a MS Windows computer.*

Inputs

Notes:

The main input to a web application is in the form of a "**page request**" from a web browser. As part of this request, variables are passed from the web browser to the web server (in this case, your program). These variables might include things such as the specific page you want to see, login details, details about how you would like the site to look and so on.

Normally, these variables exist only for the lifetime of one request. Once you close your browser, they're gone, and if you visit the site again, you'll have to specify the details of your request to the server all over again. If you want variables that persist between requests, you can use a special type of variable called a "**cookie**". Cookies are stored by the web browser and remembered between requests.

Your application's inputs can be more than just page requests from a web browser though – a database is often used to store data, for example. You could even use the temperature of the room from a thermometer connected to your Raspberry Pi!

Outputs

Outputs from a web application are called a "**response**". The output is normally in the form of the HTML of a web page, although it could be any data type, such as a JPG file (a picture). The content of this output is generated by the web application. You could even make your application produce a completely different type of output as well as a web page – for example, changing the message on a large electronic sign connected to your Raspberry Pi.

Processing

On the following pages are two simple Python programs that run as web servers. This code will also run without modification on a commercial web server running Apache web server software and the mod_wsgi Python module.

To complete this exercise, you will need to download and install the "WebOb 1.2" Python module. You'll find this either on the Raspberry Pi SD card or at the web address ***http://pypi.python.org/pypi/WebOb/***

hello-web.py:

Notes:

```
from webob import Request, Response

class WebApp(object):
    def __call__(self, environ, start_response):
        # capture the request (input)
        req = Request(environ)

        # get the name variable,
        # default value of 'World' if it is not set
        name = req.params.get('name', 'World')

        # generate the HTML for the response (output)
        html = """
<p>Hello %s!</p>
<form method="post">
Enter your name: <input type="text" name="name">
<input type="submit" value="Submit Form">
</form>
"""

        # create and return the response
        resp = Response(html % name)
        return resp(environ, start_response)

application = WebApp()

# main program - the web server
if __name__ == '__main__':
    from wsgiref.simple_server import make_server
    port = 8080

    httpd = make_server('', port, application)
    print('Serving HTTP on port %s...'%port)
    httpd.serve_forever()
```

Run this application in IDLE, then launch a web browser. Go to the URL "http://localhost:8080". If you're working from another computer available on the same network, change "localhost" to the IP address of your Raspberry Pi. Remember, you can find your IP address by typing "ifconfig" into the Terminal. You can stop the server by pressing Ctrl-C.

Note that most web pages are normally served on port 80. In order to run on a port below 1024, your program must be run with "superuser" (root) privileges. We have used port 8080 with this program – this is commonly used by web applications that are run as a normal user.

How does it work? The "name" variable is passed through to your program in the form of a "POST" request. This is generated using a form in HTML. The first time your page is viewed, the "name" variable is not set, so a default value of "World" is used.

Cookies

Notes:

We have just seen an example using a simple form. You may have noticed a drawback – the “name” variable from that form only lasts for the lifetime of one request, then it is lost. If we want a variable to remain between requests, we have to use another type of variable called a “**cookie**”. The program below is a modified version of the program above that demonstrates how to use cookies.

cookies.py:

```
from webob import Request, Response

class WebApp(object):
    def __call__(self, environ, start_response):
        # capture the request (input)
        req = Request(environ)

        # get the cookiechange variable,
        # default value is an empty string
        cookiechange = req.params.get('cookiechange', '')

        if len(cookiechange.strip()) > 0:
            # change the value of cookie if cookiechange box
            # has been completed
            cookie = cookiechange
        else:
            # get the cookie, default value is an empty string
            cookie = req.cookies.get('cookie', '')

        # generate the HTML for the response (output)
        html = """
<p>The cookie is set to '%s'</p>
<form method="post">
Change cookie value: <input type="text" name="cookiechange">
<input type="submit">
</form>
"""

        # create the response variable
        resp = Response(html % cookie)

        # store the cookie as part of the response,
        # max age 30 days (=60*60*24*30 seconds)
        resp.set_cookie('cookie', cookie, max_age=60*60*24*30)

        return resp(environ, start_response)

application = WebApp()

# main program - the web server
if __name__ == '__main__':
    from wsgiref.simple_server import make_server
    port = 8080

    httpd = make_server('', port, application)
    print('Serving HTTP on port %s...'%port)
    httpd.serve_forever()
```

When using this program, you will notice that the value of the cookie is remembered even if you close the web browser.

How does it work? First, we display the current value of the cookie. If the "cookiechange" variable is set, the value of the cookie is changed and this is sent back to the web browser to store. The cookie has a maximum age though – we have used 30 days in this example. The cookie may be lost even sooner if it is deleted by the user in the web browser. The web browser may even be set up to ignore cookies completely and never store them, although this is not usually the case.

Notes:

Further reading

More information about the "WebOb" library can be found at
http://docs.webob.org/en/latest/

There are also several Python web frameworks available that do a lot of the work for you. A framework is recommended if you are thinking about writing an entire website. A popular example is the Django framework; see
http://www.djangoproject.com/

We have only given a very small taster of what is possible here – there is a huge amount that can be learned about writing web applications.

Lesson 4.5: General Purpose Input/Output (GPIO)

Notes:

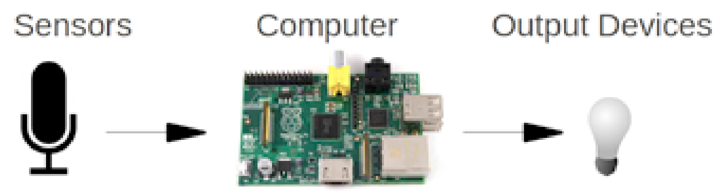
GPIO is short for "**General Purpose Input/Output**". Your keyboard, mouse and monitor are examples of input and output devices on a computer, but they are for specialised and well-defined tasks. The "General" part of GPIO indicates that you can design your own device and connect it up to the Raspberry Pi.

This section explains the technicalities of how inputs and outputs are connected and processed by computers. Following that, there is a technical reference of the GPIO capabilities of a Raspberry Pi. I'll finish off with some simple electronic circuits and software that you can build and use on your Raspberry Pi. In this guide, we are going to be concentrating on the digital GPIO interface, which is the simplest to use and understand as a beginner.

So, let's begin by looking at how computers receive and use information.

Sensors and output devices

To produce useful results, computers need data to work with. Data comes from inputs, and inputs arrive via sensors, of one kind or another.



Inputs are pieces of information sent into a computer, much like a human can feel, smell, taste, hear and see things. Outputs are the pieces of information that are produced and sent out by a computer, much as you can speak or gesture. As computers are electrical, inputs and outputs must be converted to and from an electrical form so that the computer can work with them. A sensor is a piece of electrical hardware that detects and converts something in the real world (such as the speed of a wheel) into an electrical signal. An output device is something that converts an electrical signal to another form (such as a light, a buzzer or a motor).

Digital and analogue input/ouput

A digital signal is one that can exist in one of only two states, such as a light switch that is either on or off. While one digital channel (digit) can be either on ("1") or off ("0"), we can combine several digits to make a number. This is the binary system covered in the Python chapter – each binary digit represents one digital channel.

With two digital bits we can create:

Binary	Decimal
0	0
1	1
10	2
11	3

Notes:

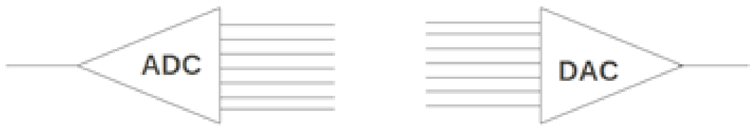
This gives us 4 (2^2) combinations that we can use. The first “2” is the number of states in a digital channel – on and off. The second “2” is the number of digital channels we are using.

An analogue signal is one that can have a range of values, whereas a digital signal only has two. One way to think about digital versus analogue is comparing a flight of stairs with a ramp. When climbing the stairs you can be quite certain that you are on the seventh step, for example. You have no option to stand halfway between the sixth and seventh stair. This is much like a digital signal. In contrast, when climbing the ramp, you can reach any height above the ground, but with less certainty about what that height actually is. This is similar to an analogue signal.

Just remember that a digital signal comes in steps (possibly very small steps, but still steps), while an analogue signal is smooth, allowing any value, but harder to gauge precisely.

A computer can only understand data in a digital form, so all analogue inputs have to be converted into a digital format before they can be used by the computer. Likewise, analogue outputs have to be created by converting the digital values coming out from a computer. On the Raspberry Pi, this conversion has to be done using external electronics.

Analogue-to-digital converters (ADCs) and digital-to-analogue converters (DACs) are a key part in most computer input/output systems.



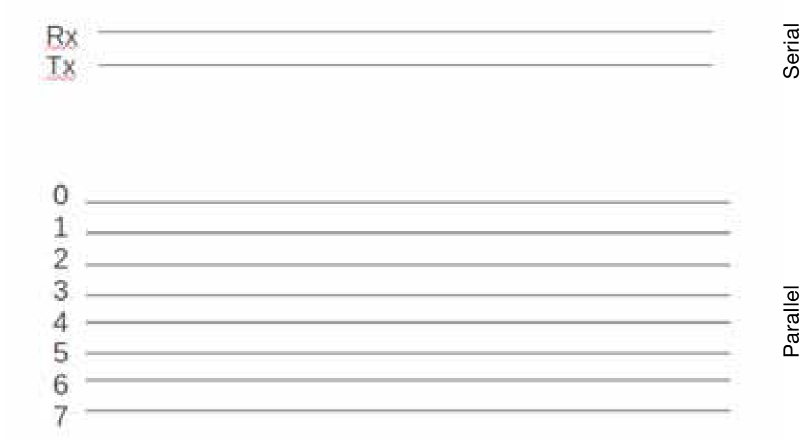
Digital-to-analogue converters (DACs) and analogue-to-digital converters (ADCs) are widely used in the electronics that connect to computers. The more digital bits that the converter handles, the more resolution (more accurate) the conversion will be. For example, if a temperature sensor produces a linear analogue signal in the range 0 °C to 100 °C and you wanted to be able to measure to at least the nearest degree on your computer, you would need at least seven digital bits ($2^7 = 128$).

Serial and parallel data

Notes:

In serial data signals, there are two information streams: transmit (Tx) and receive (Rx).

In a parallel signal, many bits of data can be sent or received simultaneously.



Digital data flows in and out of a computer in one of two ways – in serial or in parallel. On a parallel interface, there is one data channel for each bit of data that is input. Think of it as having lots of lanes on a motorway. On a serial interface, there is just one data channel for each direction (Transmit and Receive), which every single data bit travels along, one after the other. Think of it like a single road. Parallel interfaces are simpler to design and use, but require more hardware and data cable to construct (much like the motorway taking up more space than the road).

Interrupts, polling and multitasking

Now we know the basics of how information (data) flows in and out of a computer, we have to know how to handle the data that is coming in when it arrives. Consider this situation: You are working on your computer but, at the same time, you are also expecting one or more phone calls. Think of the phone call as a source of input data. There are three ways to be able to deal with both tasks at the same time:

1. Work on the computer until you are interrupted by the phone ringing. Answer the phone then continue working on your computer.
2. Work on the computer for a minute. Pick up the phone and check to see if there is someone on the other end. Put the phone down then do another minute of work on the computer. Check the phone again. Repeat this all day!
3. Work on the computer yourself and get another person to take phone calls for you.

The first method is an example of using "**interrupts**". When on the phone, you could write down a note for yourself so that you can prioritise any work that occurs as a result of the phone call in a timely manner. Using interrupts is the most efficient use of time with infrequent events. There is, however, a disadvantage to using interrupts: it requires low-level hardware or software (ringtone on the phone) to be available.

The second method is an example of "**polling**". It tends to waste a lot of your time by frequently checking the phone when there is nobody there. It also takes time to swap from one task to the other. This is what you would have to do if the phone had no ringtone. Polling is the method that is used when events occur frequently and must be handled in a timely manner.

The third method is an example of "**multitasking**". The first person can concentrate on their work on the computer. The second person could be checking (polling) the phone all the time and pass messages to you as and when needed. The problem is that there is not always a second person around. Don't worry – operating systems such as Linux on the Raspberry Pi do make it possible for you to do multitasking, even though there is only one processor.

Notes:

Embedded applications

An embedded application is where a computer is built into another device. For example, a Freeview box for a TV or a satnav in a car. If you compare a desktop PC with a Raspberry Pi, you can see that a desktop PC is not suitable as a component to build into small devices, unlike a Raspberry Pi. A standard PC does not normally have any GPIO interfaces fitted either! A Raspberry Pi is much more versatile in this regard than a desktop PC.

Real-time applications

A lot of real-world control applications are said to function in "real time". For real-time applications, it is often necessary to be able to read inputs, process them and produce outputs thousands of times a second. The rate of this processing has to be predictable – one calculation in a fixed timeframe. For example, a CD has a sample rate of 44.1 kHz – that means that for each data sample point, you only have 1/44100 seconds (22.6 µS) to do all the processing! In a multitasking operating system, such as Linux on the Raspberry Pi, you cannot guarantee that your program will have full control of the CPU during those few microseconds – the operating system may be busy, communicating on the network port, for example.

When there is an irregular sample rate on input data, this is called "**jitter**". The predictable timing accuracy that is required needs either dedicated hardware or special real-time operating systems and low-level programming languages, such as C or Assembler. The operating systems currently available on the Raspberry Pi are not really suitable for real-time applications. Fortunately, the Raspberry Pi does include a C compiler (called GCC) if you want to learn how to write a lower-level program.

Notes:

Don't worry, though, if you had your heart set on creating some real-time applications. There is another small electronics prototyping platform, called an **Arduino**, which contains a programmable microcontroller suitable for real-time applications. It is quite easy to use a Raspberry Pi and an Arduino board together. An Arduino can be used for the high-speed, real-time parts of the design and a Raspberry Pi can run a higher-level GUI or web application that controls the Arduino.

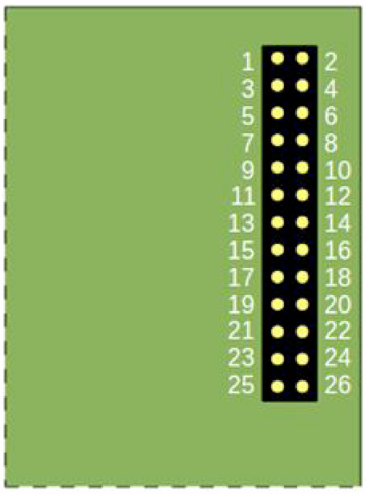
Although the Raspberry Pi can run fast I/O applications and the Arduino can use web applications, this is not what they are best suited for. Remember: always use the right tools for the job! There is an example of communicating between an Arduino and a Raspberry Pi later in this guide.

For more information about Arduino boards, see ***http://www.aurduino.cc***

GPIO hardware interfaces on the Raspberry Pi

There are several types of interface pins on the Raspberry Pi. They can be configured and used for lots of applications. Note that because the pins on the Raspberry Pi board are connected straight into the **system on a chip** (SOC), it is quite easy to damage your Raspberry Pi or SD card if you are not careful. Make sure you only use **3.3V** on the pins, **not 5V**. For this reason, it is recommended that you use an interface board, such as the **Gertboard**, between the Raspberry Pi and any circuits you build.

DNC = Do Not Connect. These pins are reserved for future use.



Pin	Function
1	3.3V
2	5V
4	DNC
6	0V
9	DNC
14	DNC
17	DNC
20	DNC
25	DNC

The maximum permitted current draw from the 3.3V pin is 50mA. The maximum current draw on the 5V pin depends on your power supply – you must leave enough for the Raspberry Pi to run! Pins not listed in the table above are described by type in the sections below. Note that some pins can be configured for more than one type of interface.

GPIO board pins

Notes:

There are 17 pins available to operate in GPIO mode, configurable as either inputs or outputs. They carry just one bit of digital data.

High = 3.3V

Low = 0V

Board pin	BCM GPIO number
3*	0
5*	1
7	4
8	14
10	15
11	17
12	18
13	21
15	22

Board pin	BCM GPIO number
16	23
18	24
19	10
21	9
22	25
23	11
24	8
26	7

* Note that these pins have a 1.8k pull-up resistor on the Raspberry Pi board.

Inter-Integrated Circuit (I²C)

I²C is an interface on which you can connect multiple I²C slave devices. The Raspberry Pi acts as the master on the bus.

Board pin	BCM GPIO number	Function	Description
3*	0	SDA	Data
5*	1	SCL	Clock

Serial Peripheral Interface (SPI)

SPI is an interface on which you can connect multiple SPI slave devices. The Raspberry Pi can only act as the master on the bus.

There are five pins available to connect devices to the Raspberry Pi using SPI:

Board pin	BCM GPIO number	Function	Description
19	10	MOSI	Master Out, Slave In
21	9	MISO	Master In, Slave Out
23	11	SCLK	Serial Clock
24	8	CE0	Channel Enable 0. Also known as Slave Select (SS)
26	7	CE1	Channel Enable 1. Also known as Slave Select (SS)

Universal Asynchronous Receiver/Transmitter (UART)

The UART is a serial bus connection. Note that these pins run at 3.3V and the RS232 specification is for 12V. If you connect this to a RS232 serial device, you could potentially damage your Raspberry Pi. Please be careful!

Board pin	BCM GPIO number	Function	Description
8	14	TX	Transmit
10	15	RX	Receive

Tip...

Please be careful with the UART connection!
If you use too much current then you could easily break something!

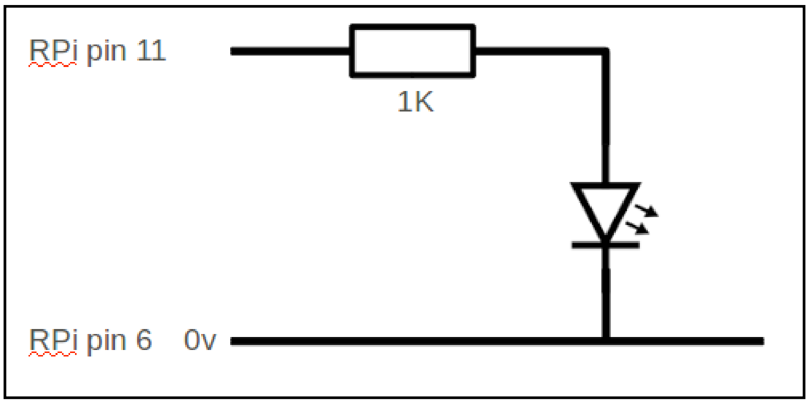
LED circuit

Notes:

Now we know all about data and the Raspberry Pi's input/output options, let's get on with building something!

This is about the simplest circuit you can build to test the GPIO outputs of a Raspberry Pi. This circuit contains just two components: a 1k resistor and an LED (Light Emitting Diode). The resistor is used to limit the current that flows out of the Raspberry Pi and into the LED. If there is too much current, you could break something!

LED circuit experiment wiring diagram.



Tip...

Please be careful:
If you use too much current then you could easily break something!

Note that you need to connect the LED up the correct way round. The flat side of the LED denotes the negative side; the longer leg denotes the positive side of the LED. The Gertboard already has LEDs wired up exactly like this on some channels.

The following Python program will let you switch the LED on and off. To complete this exercise you will need the Raspberry Pi GPIO modules. You can either install these from the Raspberry Pi SD card or download them from
http://pypi.python.org/pypi/RPi.GPIO/

```
import RPi.GPIO as GPIO

# set up pin 11 to output
GPIO.setup(11, GPIO.OUT)

state = False
while 1:
    GPIO.output(11, state)
    command = input("Press return to switch the LED on/off or
'Q' to quit: ")
    if command.strip().upper().startswith('Q'):
        break
    state = not state
```

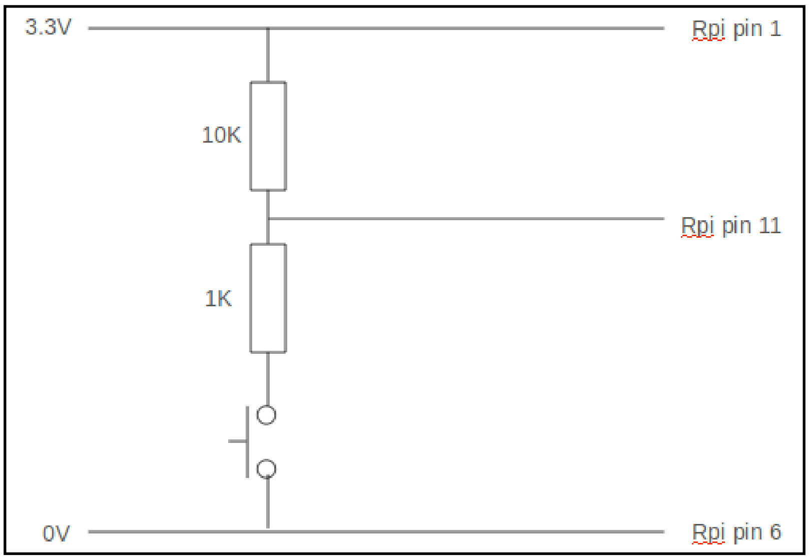
Note that this Python script must be run with superuser privileges (as root). You can do this by running your program from the command line and putting "sudo" in front of the command you are typing. For example: "sudo python led.py".

Push-button circuit

Notes:

This is the about the simplest circuit you can use to test GPIO inputs with your Raspberry Pi. The 10k resistor is what is known as a "pull-up" resistor – that means that the input will be pulled high (to 3.3V) when the button is not pressed. When you press the button, it connects the input to 0V via the 1k resistor, sending the input low. The 1k resistor is present to protect your Raspberry Pi in case you accidentally set it up as an output instead of an input. The Gertboard has some channels wired up like this circuit.

Push-button circuit experiment wiring diagram.



On the next page is an example of some Python code that monitors a push button. We use the tasking features of Python to create a class that monitors a push button to demonstrate how we might use multitasking. This is so that we do not miss the button press while the program is busy doing other things. You will probably notice that this is similar to checking for events when using PyGame.

When using the Python RPi.GPIO module, LOW = False and HIGH = True. As in the previous example, this program must be run as root by putting "sudo" in front of the Python command.

```
import threading
import time
import RPi.GPIO as GPIO

class Button(threading.Thread):
    """A Thread that monitors a GPIO button"""

    def __init__(self, channel):
        threading.Thread.__init__(self)
        self._pressed = False
        self.channel = channel

        # set up pin as input
        GPIO.setup(self.channel, GPIO.IN)

        # terminate this thread when main program finishes
        self.daemon = True
        # start thread running
        self.start()

    def pressed(self):
        if self._pressed:
            # clear the pressed flag now we have detected it
            self._pressed = False
            return True
        else:
            return False

    def run(self):
        previous = None
        while 1:
            # read gpio channel
            current = GPIO.input(self.channel)
            time.sleep(0.01) # wait 10 ms

            # detect change from 1 to 0 (a button press)
            if current == False and previous == True:
                self._pressed = True

                # wait for flag to be cleared
                while self._pressed:
                    time.sleep(0.05) # wait 50 ms

            previous = current

def onButtonPress():
    print('Button has been pressed!')

# create a button thread for a button on pin 11
button = Button(11)

while True:
    # ask for a name and say hello
    name = input('Enter a name (or Q to quit): ')
    if name.upper() == ('Q'):
        break
    print('Hello', name)

    # check if button has been pressed
    if button.pressed():
        onButtonPress()
```

Notes:

Arduino interface

Notes:

To complete this exercise you will need the Python Serial Port Extension. You can either install this from the Raspberry Pi SD card or download them from ***http://pypi.python.org/pypi/pyserial/***. You will also need the Debian package called "arduino", in order to install the Arduino development IDE. You can download this from the Debian website.

Obviously, you will also need an Arduino board.

Arduino is an open-source electronics prototyping platform based on flexible, easy-to-use hardware and software. An Arduino board connected to a Raspberry Pi is a very useful and powerful combination. You can learn more about the Arduino platform at the website ***http://www.arduino.cc/***

This example uses an Arduino board connected to a Raspberry Pi using a USB cable. You do not have to build any circuits to make this program work – that is left up to your imagination. This program is very simple; it asks for a character on your Raspberry Pi, and then sends it to the Arduino. The Arduino responds by returning the character and its ASCII code. Finally, the response is printed on the screen of the Raspberry Pi.

Code for the Arduino:

```
// set up the serial connection speed
void setup()
{
  Serial.begin(9600);
}

void loop()
{
  int inByte;

  if (Serial.available() > 0)
  {
    // read data from the Raspberry Pi
    inByte = Serial.read();

    // send data to the Raspberry Pi
    Serial.write(inByte);
    Serial.print(" = ");
    Serial.println(inByte);
  }
}
```

Python code for the Raspberry Pi:

```
import serial

# set up the serial connection speed
ser = serial.Serial('/dev/ttyACM0', 9600)

# main loop
while 1:
    c = input('Enter a char: ')
    if len(c) == 1:
        # send data to the Arduino
        ser.write(c.encode())

        # receive data from the Arduino
        response = ser.readline()

        print(response.decode().strip())
```

Notes: