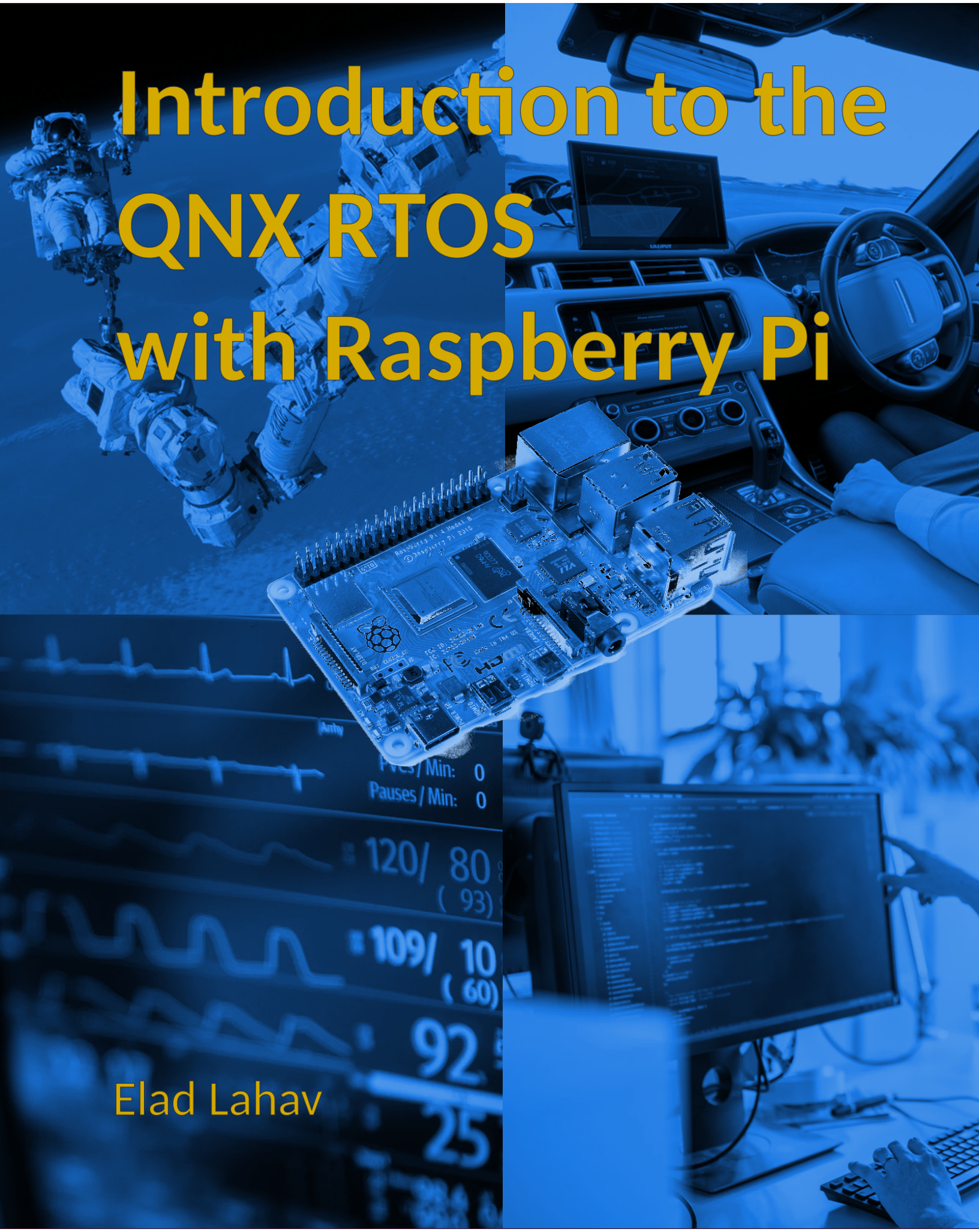
Introduction to the QNX RTOS with Raspberry Pi

Elad Lahav

Introduction to the QNX RTOS with Raspberry Pi



Elad Lahav

Contents

Preface

In a talk about microkernels given in February 2023 the QNX operating system was described as "historical".[1] While the operating system has existed in various incarnations for over 40 years (and thus represents one of the longest surviving OS lines), and while many people have unknowingly interacted with it by using computer systems based on QNX, it is still unknown to most people, even those who are otherwise proficient in the field. Those who are familiar with QNX often remember only the 1.44MB demo disk from the 90s, or its use in the commercially-unsuccessful BB10 devices from BlackBerry.

And yet the operating system is still very much in use in various fields. In recent years it has been very popular in the automotive market, first to power infotainment systems and then transitioning more to ADAS, instrumentation clusters and other safety-critical subsystems of the car. It is also used as the operating system of various medical devices, robots and industrial systems.

These use cases may suggest that the QNX operating system is by nature a deeply embedded one, requiring specialized knowledge and tools. In fact, QNX retains its origins as a general-purpose operating system that can be used anywhere and by anyone familiar with UNIX-like operating systems such as Linux and FreeBSD. Using a shell with standard tools, a C/C++ compiler or a Python interpreter, anyone familiar with those other systems can write and run programs on a QNX system. Code written to standard interfaces such as C11, C++17 or POSIX will, in most cases, require just recompilation for QNX. The API for processes, threads, file systems, sockets and more, while implemented very differently from other systems, still looks the same for those used to system-level programming on other UNIX-like operating systems. Perhaps ironically, it is embedded programmers used to small, restricted, environments who may find things unfamiliar, with processes running in their own virtual, 512GB address spaces (randomized with ASLR), 32,000 threads per process, dynamically-loaded libraries, the use of a standard compiler and linker (optionally on the target system), etc.

The emergence of the Raspberry Pi as a (very) cheap, yet capable, single-board computer, makes it much easier than before for anyone to run QNX on the type of hardware that is used by its customers. Unlike a PC, the Raspberry Pi is built to control

[1] https://archive.fosdem.org/2023/schedule/event/microkernel2023/

various devices in a simple, easy to to follow manner. A plethora of accessories, examples, tutorials, books and forums exist to help both the novice and the expert in making the most out of this tiny computer.

This book makes use of the Raspberry Pi 4 as a platform for introducing people to the QNX real-time operating system. It is primarily directed at people with some programming experience both in Python and C. The book assumes the reader is at least familiar with opening a programming text editor, writing code, building and running programs (be it on the command line or via an integrated development environment). The examples presented in this book have been purposefully kept very simple. The Python examples in Chapter 4 will look very familiar to anyone who has followed tutorials or books on using Python with the Raspberry Pi. The goal of this chapter is to highlight the similarities in user (or, rather programmer) experience with other systems. For people not familiar with the Raspberry Pi it serves as a brief introduction to the way external devices can be controlled using simple Python scripts. Chapter 5 takes the opposite approach ofhighlighting QNX-specific interfaces, allowing programmers to make the most out of the operating system.

The book is accompanied by a QNX system image that can be copied to a micro-SD card and used with a Raspberry Pi 4. The image contains the tools necessary to follow the examples in the book, and experiment further with programming for QNX. The image is free for non-commercial use only. It is designed for simplicity, lowering the barriers for people familiar with other operating systems to get started with QNX. However, it does not represent a QNX product. You may notice that the system has not been hardened for security in any way: a root account, password-based logins, a writable system partition, the lack of a security policy and a debug service open to the world are all examples of very poor design for real systems, that are nevertheless useful for an introductory image.

Chapter 1

Introduction

1.1 What is a Real-Time Operating System?

That is an excellent question, and the answer is not clear. The term Real-Time Operating System (abbreviated as RTOS) has been applied to a wide variety of foundational software that is intended to support the execution of applications with strict timing requirements (for example industrial controllers or audio streaming). On the one extreme the term has been used for simple executives into which the application code is added to create a single program running on a basic micro-controller. On the other extreme are complete UNIX and UNIX-like operating systems with real-time capabilities, executing on complex multi-processors with large amounts of memory and many peripherals. The QNX RTOS belongs to the latter class.

An operating system is software designed to allow multiple programs to share a single computer. The operating system governs the access of these programs to the shared resources of the computer: the processor, memory, storage devices, display, input devices, etc. A computer that only runs one program does not require an operating system, as the program can monopolize all resources without any adverse consequences. This is the case for many micro-controllers, even those that perform multiple, complicated tasks. Such a control program may be built together with some additional code that provides routines for accessing the hardware and scheduling the various tasks, but that does not make that code a true operating system.

Even among those real-time operating systems that are true to their name, there is considerable variability in terms of capabilities and guarantees. Some real-time operating systems are designed to operate on MCUs and MPUs that are more restricted and usually less powerful than general-purpose CPUs, but often provide better guarantees about execution time than the latter. In fact, the advent of multi-core, multi-cache and aggressively speculating processors raises the question of what guarantees an operating system can even provide with respect to timing. The notion of a fixed execution time must be replaced with that of adherence to deadlines, and even that sometimes

needs to be relaxed from a full guarantee to achieving a high level of confidence.

The design of the QNX RTOS, which is intended to run on modern, full-feature CPUs, is often pulled in different directions in response to customer demands. Balancing safety, security, scalability and latency is a difficult task. The choice of whether to excel at one or two of these areas at the expense of the rest, or to aim for a less than perfect mix of all, is one of the dilemmas of modern RTOS design.

1.2 A Brief History of QNX

Dan Dodge and Gordon Bell founded Quantum Software Systems in 1980. The company released an operating system for the Intel 8088 called Quantum UNIX, or QUNIX. However, due to legal issues, the name of the system was changed to QNX, and then the company itself became QNX Software Systems. QNX 2 followed in the 80s. At the time, QNX was marketed as a real-time operating system for PCs and was considerably more advanced than DOS, though obviously lost the commercial race.

In 1991 the company released QNX 4 with a focus on POSIX compatibility and on cleaner separation between client and server processes. QNX 4 has enjoyed a long life, with the company still receiving the occasional request for support to this day (which prompts a search for that ancient computer under somebody's desk that is still capable of running it).

A few years later, work began on the next version of the operating system. One of the main goals for this version, which became QNX Neutrino 1.0, was to create a multi-platform operating system, whereas all versions before that were only able to run on x86 computers. At the behest of Cisco, who wanted to use QNX for some of its routers, the new version was implemented for MIPS, with PPC, ARM and SH versions following over the years (in addition to x86). SMP support was also introduced.

In the early 2000s QNX started gaining popularity with companies building infotainment systems for car manufacturers. Eventually QNX Software Systems was sold to one of these companies, Harman International. At this point Gordon Bell left QNX.

In 2010 Research In Motion, then a leading smartphone company of BlackBerry fame, was looking to replace the home-grown operating system on its devices. After meeting Dan Dodge, RIM bosses Mike Lazaridis and Jim Balsillie decided to purchase QNX from Harman International. The first RIM product to use QNX was the PlayBook tablet, released that year. Soon afterwards, work began on a new generation of smartphones running QNX, which became the basis for the BB10 operating system. Neither the tablet nor the smartphones were commercially successful, and by 2015 BlackBerry (as Research In Motion was now called) switched to Android-based phones, and then shut down its smartphone business completely. A year later Dan Dodge, along with several QNX veterans, left for Apple.

Despite this failure, the QNX operating system remained popular in certain markets, primarily automotive but also medical and industrial. While the smartphone business

was grinding to a halt, the infotainment business, and then a focus on more safety-oriented systems in these fields, kept the company going. It does so to this day.

The version of the system that accompanies this book represents the first major redesign of key components of the system since Neutrino 1.0. The operating system is now 64-bit only (supporting the AArch64 and x86_64 architectures) and can run on systems with up to 64 processors and 16TB of RAM.

1.3 QNX RTOS FAQ

Is QNX a variant of Linux? No, QNX is an operating system built around the Neutrino microkernel, which shares no code with the monolithic Linux kernel. Also, the original QNX operating system predates Linux by quite a few years.

Is QNX a type of UNIX? No, UNIX is the name for a family of operating systems from various vendors that adhere to a specific standard. However, the QNX operating system is POSIX-compatible, which means that code written for UNIX and UNIX-like operating systems is likely to run on the QNX operating system with little or no modification. Also, the QNX software development platform includes many open-source components that are available in other operating systems, resulting in a familiar environment.

Is QNX Neutrino a true microkernel? Yes, QNX Neutrino provides a limited set of functionality, such as scheduling, inter-process communication, synchronization primitives and timers. Much of the functionality of a monolithic kernel is performed by user-mode processes that are completely separated from the kernel, i.e., run in a non-privileged mode, each with its own address space. That said, the kernel is bundled with a few services that do run in privileged mode and share the same address space as the kernel - the memory manager, path manager and process manager. Microkernel purists may object to such a design choice, but it was felt that running these specific services as stand-alone processes would complicate the system and incur a significant performance penalty.

Is QNX an automotive operating system? No, QNX is a general-purpose operating system. It can be used anywhere such a system is required. In recent years the automotive market has been the largest adopter of the QNX RTOS, and the system abides by various automotive standards. Nevertheless, QNX is used in other fields, and, as you will discover in this book, is versatile enough for any task.

1.4 Conventions

The following conventions are used in this book:

- File names and paths are written in a **bold typeface**.
- Shell commands and code snippets are written in a `teletype face`.

- In shell examples, a stand-alone `$` sign indicates that the shell is running on your host system (PC), while a `qnxuser@qnxpi:~\$` prompt indicates that it is running on the Raspberry Pi.
- This book uses Canadian English for spelling, as it is clearly the right way to spell.

Chapter 2

Getting Started

2.1 Shopping List

In order to follow the exercises in this book you will need a few items.

1. **Raspberry Pi 4 board** Version 4 of this small-board computer is the only one officially supported by QNX at the time the book was written. You can use either the 2GB, 4GB or 8GB variants of this board. The 1GB variant is not supported. Please purchase the board from an authorized dealer.
2. **Micro SD card** This should be at least 8GB in size. Note that not all SD cards work, and you may need to try different ones (see Section 2.7 below). I have not had problems with SanDisk cards.
3. **HDMI display, micro-HDMI connector and a USB keyboard** Only required if you want to connect to the system with a keyboard and a display.
4. **USB-TTL converter** Only required for troubleshooting the system over a serial console.
5. **Breadboard, jumper wires, LEDs, resistors, motor, servos** See Chapter 4 for a full list. Note that many sellers of Raspberry Pi boards also offer cheap kits with these components.

It is possible to interact with the Raspberry Pi using a network connection only, in which case there is no need to have a display, a keyboard or a USB-TTL converter. Nevertheless, you may wish to have these available in case the system fails to boot, or does not connect to the network. See Section 2.5 below.

2.2 Installing the QNX SDP

The QNX Software Development Platform (SDP) provides the necessary files for building a QNX-based system. These files include pre-built binaries (programs and libraries), as well the tools needed for writing and building new software (compilers,

linkers, header files, etc.). The SDP is freely available for non-commercial use.

Obtaining the SDP involves the following steps:

1. Register for a myQNX account.
2. Request a free licence.
3. Download QNX Software Centre.
4. Install the SDP on your computer.

To get started, visit qnx.com/getqnx. You will be prompted for your myQNX account credentials. If you do not have an account, create one first. Once you have logged in, you can request a free QNX SDP licence to be associated with your account. Follow the steps on the web page for getting the licence, activating it, and associating it with your account.

The next step is to download the QNX Software Centre (QSC). Pick the version that matches your host operating system. Windows users can run the installer directly, while Linux users will have to make the file executable first. Refer to the installation instructions for further information.

With QSC installed, we can now get the SDP. Run QSC, choose **Add Installation**, and then select the SDP version to install. As this book was written for QNX 8.0, use this version of the SDP. Follow the prompts to complete the installation. We will assume that the SDP was installed in a folder called **qnx800** under your home directory, but you are free to choose any location.

Take a look at the installation directory.

```
$ ls ~/qnx800
host  qnxsdp-env.bat  qnxsdp-env.sh  target
```

The two scripts can be used to set up the environment for development (one for Windows and one for Linux). The **host** folder is where you will find the tools necessary for building programs on your computer, while the **target** folder contains all of the files that can go on your QNX system (though you will only ever likely need a small subset of these).

The base installation contains only a small portion of the content that is available as part of the SDP. Additional packages can be installed with QSC, some of which are free and some require a commercial licence. To develop for a particular board, you will need to add the relevant *board support package* (BSP), which contains the source code and binaries for the specific hardware.

The SDP provides everything you need in order to build your own QNX system. However, to get you up and running quickly with Raspberry Pi, we will use a pre-defined image that you can just copy to an SD card. Open QNX Software Centre, and install the "QNX® SDP 8.0 Quick Start image for Raspberry Pi 4" package. The image file is now located in the **images** folder under SDP installation path:

```
$ ls ~/qnx/sdp/8.0/images/
qnx_sdp8.0_rpi4_quickstart_20240920.img
```

(The name of the image will likely be different for you, as it includes the version number and date.)

2.3 Creating The SD Card Image

The Raspberry Pi boots from a micro SD card, which is inserted to a slot under the board. While it is possible to use other storage devices once the system is running, we will use the SD card both for the boot image and for holding the primary file systems. SD cards are neither fast nor resilient when compared with some other storage devices, but they will do fine for the activities described in this book.

2.3.1 Generate the Image

Note that we cannot just copy the image file to the card. The image file holds the partition and file system information, along with all of the file data that the system requires. Copying the image file to the SD card, which is typically formatted with a FAT file system, will not have the desired effect, as the disk image is not bootable. Instead, we are going to perform a raw copy of the image file to the card.

2.3.2 Copy the Image: Raspberry Pi Imager

The Raspberry Pi Foundation publishes a utility called **Raspberry Pi Imager**, which provides an easy way to transfer any OS image to a removable medium, such as an SD card. The utility is available for Windows, Linux and macOS.[1]

Once you have downloaded, installed, and run the Imager, you will be prompted for the following information:

1. Device: Choose the Raspberry Pi 4.
2. OS: Choose **Use custom**, and then navigate to pick the QNX image.
3. Storage: Choose the SD card to use (assuming it is already connected via a card reader).

Click **Next** and wait for the Imager to complete before removing the device.

2.3.3 Copy the Image: Linux Command Line

The `dd` command can be used to write raw data from one file or device to another. The source is the image file, while the destination is the device name for the SD card.

[1] https://www.raspberrypi.com/software/

Figure 2.1: Raspberry Pi Imager

> **Warning**
>
> It is crucial that you determine the correct device name! Copying the image to the wrong device can cause permanent damage to other storage devices, including the one used for your computer's file system.

First, insert the card to a card reader connected to your computer. The operating system may mount the card automatically, presenting an existing file system under some path. We are not interested in this file system or path. Instead, we want to establish the name of the device.

The `lsblk` command can be used to list the available block devices. Find the device that matches the size of the card you have just inserted. In the example below, this is **/dev/sdb**. Note that the numbered devices appearing below it are partitions on the device, but we want the primary device name in order to write the image to the entire card. If you have doubt, remove the card, run `lsblk` again to confirm that device is no longer listed (or is listed with a 0 size), and repeat the exercise after re-inserting the card. Alternatively, use the `dmesg` command to see which device was detected when the SD card was inserted.

```
$ lsblk
NAME        MAJ:MIN RM   SIZE RO TYPE MOUNTPOINTS
...
sda           8:0    1     0B  0 disk
sdb           8:16   1  14.5G  0 disk
|-sdb1        8:17   1     1G  0 part /media/elahav/1945-029C
|-sdb2        8:18   1     1G  0 part
|-sdb3        8:19   1  12.5G  0 part
nvme0n1     259:0    0   1.9T  0 disk
|-nvme0n1p1 259:1    0   260M  0 part /boot/efi
...
```

You are now ready to copy the image. Run the following command from the directory that contains the image file:

```
$ dd if=qnx_rpi_sdcard.img of=<device name> bs=4M status=progress
```

Make sure to replace `<device name>` with the name of the card block device discovered above (e.g., **/dev/sdb**).

Your system may require root privileges for this command, in which case you will need to run the command as root. On most modern Linux systems this is done by prefixing the `sudo` command:

```
$ sudo dd if=qnx_rpi_sdcard.img of=<device name> bs=4M status=progress
```

2.4 Booting The Raspberry Pi

Once the image has been copied to the SD card, and while it is still plugged in to your computer, you should be able to see the files written to the first partition (a FAT partition, visible to any operating system). In that partition there are two files that need to be edited to configure the system. The first is **qnx_config.txt**, which can be left alone unless you want to change the default host name of **qnxpi**, or control where the console output goes.

The second file, which you must update, is **wpa_supplicant.conf**. At the very bottom of this file (after much documentation) is the network settings configuration structure. Update the `ssid` entry to the name of your WiFi network, and the `psk` entry to the password. These settings assume a standard home network WPA authentication scheme. On a different kind of network you may need to change these settings.[2]

After updating these files, eject the SD card and insert it into the Raspberry Pi's slot. Connect the Raspberry Pi to the power supply to boot the system. This image is not set up to provide a graphical desktop, but it does show information on the screen. Assuming that all goes well and the network configuration is correct, the system should be ready within a few seconds.

[2]You may consult documentation such as https://wiki.netbsd.org/tutorials/how_to_use_wpa_supplicant/, or your organization's IT professional.

2.5 Connecting to The System

There are different ways to interact with the system. The most convenient and powerful is over a secure network connection (SSH). However, it may be required, at least at first, to have a way to interact with the system in case the network connection fails (see "Troubleshooting" below). Whenever you need to log in, use **qnxuser** as the user name and password, or, if root access is required, **root** as the user name and password.

> **Note**
>
> Make sure you update these passwords the first time you log in. Passwords can be changed using the `passwd` shell command. Note that root access over SSH is disabled by default.

2.5.1 Display and Keyboard

While the system does not support a graphical desktop, it does provide a simple terminal program. Attach a display to the first micro-HDMI connector (the one closest to the power connector), and a keyboard and a mouse to the USB ports. When the system boots you should see a welcome screen. Pressing the button in the middle starts a terminal instance. You can log in as either **qnxuser** or **root** and execute shell commands.

2.5.2 Serial Connection

The most resilient way to connect to the system is with the aid of a USB-to-TTL converter, which provides a basic serial connection between the Raspberry Pi and another computer. There are many models of such converters available for purchase, and many of those have instructions on how to connect to the Raspberry Pi.

> **Note**
>
> In order to get a login prompt on the serial console you must edit the **qnx_config.txt** file in the boot partition to remove (or comment out) the line that says `CONSOLE=/dev/con1`. This can be done either from the Raspberry Pi itself, or when the SD card is plugged into your computer.

The connection involves three pins on the Raspberry Pi 40-pin GPIO header (see Chapter 6). Pin 8 is the transmit pin, pin 10 the receive pin and the third is any of the ground pins (though usually either 6 or 14 is used due to their proximity to the transmit/receive pins). Which of the converter connections goes to which pin on the Raspberry Pi depends on the converter itself. If you need to buy one, make sure it has instructions for the Raspberry Pi.

> **Warning**
>
> Never connect a USB-TTL converter to any of the power pins on the Raspberry Pi. Some converters have a fourth connector that should be left unattached. Such converters tend to be very susceptible to current surges and once damaged are beyond repair.

An example connection of a USB-TTL converter to a computer is shown in Figure 2.2.

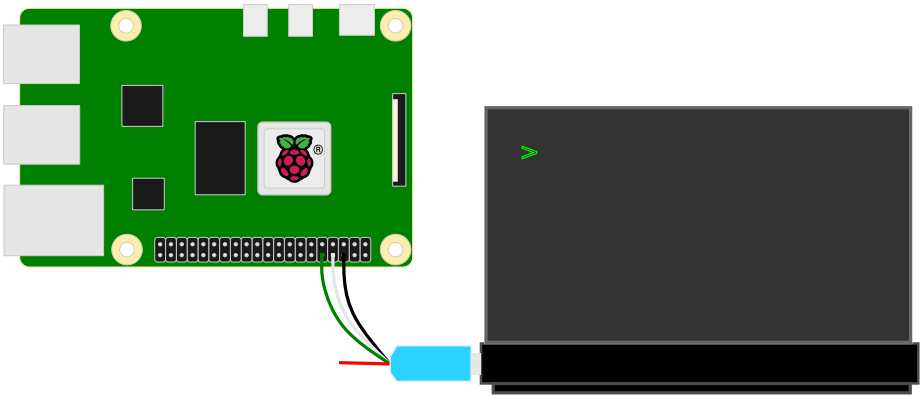


Figure 2.2: USB-TTL connection example

Once the USB-TTL converter is connected to the computer you can use a terminal program to connect to the Raspberry Pi. Terminal programs include **minicom**, **GNU Screen** and **c-kermit** for UNIX-like systems, or **PuTTY** for Windows. There are many tutorials on the Internet on how to use such a program with the Raspberry Pi and you should follow one of those. Next, boot the Raspberry Pi and look for output in the terminal.

2.5.3 Secure Shell

The system is configured to use mDNS to advertise its IP address on the local network. Assuming your computer is set up for mDNS you should be able to log in using the host name specified in the **qnx_config.txt** file. By default that name is **qnxpi**, and the advertised name is **qnxpi.local**.

```
$ ssh qnxuser@qnxpi.local
```

You should see the system's shell prompt:

```
qnxuser@qnxpi:~$
```

2.6 Writing Code

In Chapter 4 we will be writing code in Python for controlling various external devices using the Raspberry Pi. In Chapter 5 we will be writing code in the C language for better control and real-time responsiveness. In both cases you will need to create and edit source files.

There are a few options for writing code for use with the QNX Raspberry Pi image. The first is to use the **Vim** editor included with the image. For that you log in via SSH and run the `vim` command. While **Vim** is a very popular editor, people who first encounter it can be baffled by the interface, especially by the split between a command mode and an insert (or edit) mode. There are many guides on the Internet for getting started with **Vim** and you can follow these. If you have already started **Vim** and would really like to quit, press the `ESC` key to go back to command mode, and then `:q` (colon followed by the letter q) and `ENTER` to quit.

A second option is to write the code on your computer, using your preferred text editor[3]. In the case of Python such code can then be run on the Raspberry Pi using the included Python interpreter (see Section 4.1). Code written in C needs to be compiled first into an executable using the QNX C compiler. The resulting executable can then be copied over to the Raspberry Pi.

When using an editor on your computer the source files are typically saved on the computer's own file system. In the case of Python code these files will have to be copied over to the Raspberry Pi before they can be run with the Python interpreter. An alternative to copying is to mount a directory from the Raspberry Pi file system into your computer, using SSHFS, which is available both for Linux and Windows. The following example mounts the **/data/home/qnxuser/python** directory on a Linux user's **qnxrpi** directory (assuming both directories exist):

```
$ mount qnxuser@qnxpi.local:python ./qnxrpi
```

Once this command is executed any file saved on **qnxrpi** from the computer will be seen under **/data/home/qnxuser/python**, and updates will be kept in sync. This way you can write Python code on your computer and run it on the Raspberry Pi without constantly copying the files after every update.

Finally, it is possible to use VSCode[4] with the QNX plugin to edit, build, deploy and debug code. The plugin also allows you to inspect the system, e.g., by listing running processes, analyzing memory usage and collecting trace logs for system activity. Install the QNX Toolkit extension, which is available on the VSCode Extension Marketplace.

[3] Just remember that Microsoft Word is NOT a text editor!
[4] https://code.visualstudio.com

2.7 Troubleshooting

What if the system doesn't boot, or it does but you cannot connect to it? Before we start diagnosing any issues with the QNX image for Raspberry Pi, ensure that the board itself is properly connected and powered.

Does the red LED on the board turn on? If not, the board is not powered. Check that the board is connected to a proper power supply. The official power supply for Raspberry Pi is 5.1V and 3A.

Does the firmware detect the SD card? Connect the Raspberry Pi to a display, using one of the micro-HDMI connectors on the board. If the SD card is not properly inserted you will see a message on the display that no system image was found.

Does the firmware detect the image? A sign of the firmware starting the system image is a multi-coloured rectangle displayed on the screen.

Any of the above issues are indicative of a problem with the setup of the board (or, if you are very unlucky, with the board itself), and not with the QNX image. Assuming the firmware appears to boot fine, we need to examine what is not working with the QNX image.

Is there any output? If not, try to use the official Raspbian image instead, enabling its UART output option. Again, follow tutorials on how to do that, ensuring that you have connected the USB-TTL converter correctly, that the terminal program is connected to the right device (e.g., **/dev/ttyUSB0** on Linux) and with the correct settings: a baud rate of 115200, 8 bits, no parity, 1 stop bit and no control flow. On Windows, make sure that the driver for the converter is installed by checking the device manager.

Do you see errors from the QNX image about a failure to mount file systems? Some SD cards are not recognized by the SD driver. Try a different card, preferably from a different manufacturer.

Does the system boot to a command prompt, but there is no network connection? Run `ifconfig` and check that the **bcm0** interface shows up and is associated with an IP address. If not, ensure that the network configuration in **wpa_supplicant.conf** are correct.

> **Note**
>
> Most logs on a QNX system are accessible by running the `slog2info` command. This command will dump all logs from all processes. You can refine it with the `-b <component>` command, where `<component>` should be replaced by the name of the process. For example, `devb_sdmmc_bcm2711` is the SD card driver.

Chapter 3

Exploring The System

3.1 The Shell

Once you have logged in with SSH you are presented with the shell prompt:

```
qnxuser@qnxpi:~$
```

The shell is an interactive command interpreter: you type in various commands and the shell executes these, typically by creating other processes. Anyone who has ever used a *NIX system is familiar with the basic shell commands, such as **echo**, **cat** and **ls**.

The default shell on QNX systems is a variant of the Korn Shell, called **pdksh**, but our Raspberry Pi image uses the more familiar **bash** (the Bourne Again Shell), which is the default shell on many modern *NIX systems.

Many of the simple (and some not so simple) shell utilties are now provided by a single program call **toybox**. This program uses the name by which it is invoked as indication of which utility it should run. The system uses symbolic links from various utility names to **toybox**, which allows the user to use standard shell commands. For example, we can use **ls** to list files:

```
qnxuser@qnxpi:~$ ls /tmp
encoder.py  robot.py
qnxuser@qnxpi:~$ which ls
/proc/boot/ls
qnxuser@qnxpi:~$ ls -l /proc/boot/ls
lrwxrwxrwx  1 root root 6 2023-10-14 15:18 /proc/boot/ls -> toybox
```

The sequence above shows that **ls** is a symbolic link under **/proc/boot** to **toybox**.

3.2 The File System

3.2.1 Background

Before exploring the file system on the image there are a few concepts that need to be understood. A *file system* is a collection of files, typically organized hierarchically into directories, that can be stored on some medium (such as a hard drive, an SD card, network storage or even RAM). There are different formats for file systems, some of the more common ones being FAT and NTFS from Microsoft and ext4 from Linux. QNX supports a few file system formats, primarily the native QNX6 power-safe file system.

Every file and every directory in a file system has a name, and a file is identified uniquely within a file system by its *path*, which is the chain of directory names leading to that file, separated by slashes, followed by the file's name. For example, the file **foo**, contained within the directory **bar**, which is itself contained within the directory **goo**, is identified by the path **goo/bar/foo** within that file system.

Each file system can be *mounted* at some path in the system, with all paths starting at the root directory, identified by a single slash. If the file system in the example above is mounted at **/some**, then the full path of the aforementioned file is **/some/goo/bar/foo**. The collection of all paths from all file systems is referred to as the *path space*, and is handled by the path manager in the QNX kernel.

With the exception of the most trivial ones, all QNX-based systems employ more than one file system. These different file systems are provided by different server processes. Any request to access a file queries the path manager for the server that provides the path to be opened, and then asks that server for the file.

3.2.2 File Systems on the Image

The **df** command allows us to see which file systems are included in the image, and under which path each of those is mounted:

```
qnxuser@qnxpi:~$ df
ifs                      26344     26344         0     100%  /
/dev/hd0t17           26212320   1288784  24923536       5%  /data/
/dev/hd0t17            2097120    600088   1497032      29%  /system/
/dev/hd0t11            2093016     31256   2061760       2%  /boot/
/dev/hd0              30408704  30408704         0     100%
/dev/shmem                   0         0         0     100%  (/dev/shmem)
```

The first entry shows the Image File System (IFS). This is a read-only file system that is loaded into memory early during boot, and remains resident in memory. The IFS contains the startup executable, the kernel and various essential drivers, libraries and utilities. At a minimum, the IFS must contain enough of these drivers and libraries to allow for the other file systems to be mounted. Some QNX-based systems stick to this minimum, while other use the IFS to host many more binaries, both executables and libraries. There are advantages and disadvantages to each approach. The IFS in

this image is mounted at the root directory, with most of its files under **/proc/boot**. It is possible to change this mount point.

Next are two QNX6 file systems, mounted at **/system** and **/data**, respectively. This separation allows for the system partition to be mounted read-only, potentially with added verification and encryption, while the data partition remains writable. In this image both partitions are writable, to allow for executables and libraries to be added later and for configuration files to be edited.

Under the **/system** mount point you will find directories for executables (**/system/bin**), libraries (**/system/lib**), configuration files (**/system/etc**) and more. The data partition holds the users' home directories (**/data/home**), the temporary directory (**/data/tmp**, also linked via **/tmp**) and a run-time directory for various services (**/data/var**).

The FAT file system mounted at **/boot** is used by the Raspberry Pi firmware to boot the system. It also holds a few configuration files to allow for easy setup before the system is booted for the first time (see Section 2.4).

The next entry shows the entire SD card and does not represent a file system.

Finally, **/dev/shmem** is the path under which shared memory objects appear. The reason it is listed here is that it is possible to write to files under **/dev/shmem**, which are kept in memory until the system is shut down (or the file removed). It is not, however, a full file system, and certain operations (e.g., `mkdir`) on it will fail. Do not treat it as a file system.

At this point people familiar with previous versions of QNX, with Linux or with other UNIX-like operating systems, may be wondering what happened to the traditional paths, such as **/bin** or **/usr/lib**. The answer is that you can still create a file system on QNX with these paths and mount it at the root directory. There are two reasons this image employs a different strategy:

1. The separation of **/system** and **/data** makes it easier to protect the former, as mentioned above.
2. This layout avoids union paths, which have both performance and security concerns.

Union paths are an interesting feature of the operating system that allows multiple file systems to provide the same path. For example, both the IFS and one (or more) of the QNX6 file systems can provide a folder called **etc**. If both are mounted under the root path then the directory **/etc** is populated as a mix of the files in the folders of both file systems. Moreover, if both file systems have a file called **etc/passwd** then the file visible to anyone trying to access **/etc/passwd** is the one provided by the file system mounted at the front. This feature can be quite useful in some circumstances, such as providing updates to a read-only file system by mounting a patch file system in front of it, but it complicates path resolution, can lead to confusion and, under malicious circumstances, can fool a program into opening the wrong file.

3.3 Processes

A *process* is a running instance of a program. Each process contains one or more *threads*, each representing a stream of execution within that program. Any QNX-based system has multiple processes running at any given point. The first process in such a system is the process that hosts the kernel, which, mainly for historical reasons, is called `procnto-smp-instr`. The program for this process comprises the Neutrino microkernel, as well as a set of basic services, including the process manager (for creating new processes), the memory manager for managing virtual memory and the path manager for resolving path names to the servers that host them.

The microkernel architecture means that most of the functionality provided by a monolithic kernel in other operating systems is provided by stand-alone user processes in the QNX RTOS. Such services include file systems, the network stack, USB, graphics drivers, audio and more. A QNX system does not require all of these to run. A headless system does not need a graphics process, while a simple controller may do away with a permanent file system.

Finally, a system will have application processes to implement the functionality that the end user requires from it. An interactive system can have processes for the shell and its utilities, editors, compilers, web browsers, media players, etc. A non-interactive system can have processes for controlling and monitoring various devices, such as those found in automotive, industrial or medical systems.

The `pidin` command can be used to list all processes and threads in the system. The following is an example of the output of this command. Note, however, that the output was trimmed to remove many of the threads in each process for the purpose of this example (indicated by "…"):

```
qnxuser@qnxpi:~$ pidin
     pid tid name                         prio STATE       Blocked
       1   1 /proc/boot/procnto-smp-instr   0f READY
       1   2 /proc/boot/procnto-smp-instr   0f READY
       1   3 /proc/boot/procnto-smp-instr   0f RUNNING
       1   4 /proc/boot/procnto-smp-instr   0f RUNNING
       1   5 /proc/boot/procnto-smp-instr 255i INTR
       1   6 /proc/boot/procnto-smp-instr 255i INTR
       ...
       1  20 /proc/boot/procnto-smp-instr  10r RECEIVE     1
   12291   1 proc/boot/pipe                 10r SIGWAITINFO
   12291   2 proc/boot/pipe                 10r RECEIVE     1
   12291   3 proc/boot/pipe                 10r RECEIVE     1
   12291   4 proc/boot/pipe                 10r RECEIVE     1
   12291   5 proc/boot/pipe                 10r RECEIVE     1
   12292   1 proc/boot/dumper               10r RECEIVE     1
   12292   2 proc/boot/dumper               10r RECEIVE     2
   12293   1 proc/boot/devc-pty             10r RECEIVE     1
   12294   1 proc/boot/random               10r SIGWAITINFO
   12294   2 proc/boot/random               10r NANOSLEEP
   12294   3 proc/boot/random               10r RECEIVE     1
   12294   4 proc/boot/random               10r RECEIVE     2
   12295   1 proc/boot/rpi_mbox             10r RECEIVE     1
   12296   1 proc/boot/devc-serminiuart     10r RECEIVE     1
   12296   2 proc/boot/devc-serminiuart    254i INTR
   12297   1 proc/boot/i2c-bcm2711          10r RECEIVE     1
   12298   1 proc/boot/devb-sdmmc-bcm2711   10r SIGWAITINFO
   12298   2 proc/boot/devb-sdmmc-bcm2711   21r RECEIVE     1
   12298   3 proc/boot/devb-sdmmc-bcm2711   21r RECEIVE     2
   ...
   12298  16 proc/boot/devb-sdmmc-bcm2711   21r RECEIVE     4
   36875   1 proc/boot/pci-server           10r RECEIVE     1
   36875   2 proc/boot/pci-server           10r RECEIVE     1
   36875   3 proc/boot/pci-server           10r RECEIVE     1
   49166   1 proc/boot/rpi_frame_buffer     10r RECEIVE     1
   61455   1 proc/boot/devc-bootcon         10r RECEIVE     1
   61455   2 proc/boot/devc-bootcon         10r CONDVAR     (0x50d562a3cc)
   73744   1 proc/boot/rpi_thermal          10r RECEIVE     1
   86033   1 proc/boot/rpi_gpio             10r RECEIVE     1
   86033   2 proc/boot/rpi_gpio            200r INTR
  118802   1 proc/boot/io-usb-otg           10r SIGWAITINFO
  118802   2 proc/boot/io-usb-otg           10r CONDVAR     (0x5970918880)
  118802   3 proc/boot/io-usb-otg           10r CONDVAR     (0x59709193a0)
  ...
  118802  13 proc/boot/io-usb-otg           10r RECEIVE     2
  139283   1 system/bin/io-pkt-v6-hc        21r SIGWAITINFO
  139283   2 system/bin/io-pkt-v6-hc        21r RECEIVE     1
  139283   3 system/bin/io-pkt-v6-hc        22r RECEIVE     2
  ...
  139283   9 system/bin/io-pkt-v6-hc        21r RECEIVE     7
  163860   1 ystem/bin/wpa_supplicant-2.9   10r SIGWAITINFO
  163861   1 system/bin/dhclient            10r SIGWAITINFO
  192535   1 system/bin/sshd                10r SIGWAITINFO
  196630   1 system/bin/qconn               10r SIGWAITINFO
  196630   2 system/bin/qconn               10r RECEIVE     1
  208908   1 proc/boot/ksh                  10r SIGSUSPEND
  270349   1 proc/boot/pidin                10r REPLY       1
```

Each process has its own process ID, while each thread within a process has its own thread ID. The output of pidin lists all threads, grouped by process, along with the path to the executable for that process, the priority and scheduling policy for the thread, its state and some information relevant to that state. For example,

```
118802 3 proc/boot/io-usb-otg  10r CONDVAR  (0x59709193a0)
```

shows that thread 3 in process 118802 belongs to a process that runs the `io-usb-otg` executable (the USB service). Its priority is 10 while the scheduling policy is round-robin. It is currently blocked waiting on a condition variable whose virtual address within the process is 0x59709193a0.

It is possible to show other types of information for each process with the `pidin` command: `pidin fds` shows the file descriptors open for each process, `pidin arguments` shows the command-line arguments used when the process was executed, `pidin env` shows the environment variables for each process and `pidin mem` provides memory information. It is also possible to restrict the output to just a single process ID (e.g., `pidin -p 208908`) or to all processes matching a given name (e.g., `pidin -p ksh`).

3.4 Memory

RAM, like the CPU, is a shared resource that every process requires, no matter how simple. As such, all processes compete with each other for use of memory, and it is up to the system to divide up memory and provide it to processes. In a QNX system, control of memory is in the hands of the virtual memory manager, which is a part of the `procnto-smp-instr` process (recall that this is a special process that bundles the microkernel with a few services).

A common problem with the use of memory is that the system can run out of it, either due to misbehaving processes, or to poor design that does not account for the requirements of the system. Such situations naturally lead to two frequently-asked questions:

1. How much memory does my system use?
2. How much does each process contribute to total system use?

The first question is easy to answer. The second, perhaps surprisingly, is not.

A quick answer to the first question can be obtained with `pidin`:

```
qnxuser@qnxpi:~$ pidin info
CPU:AARCH64 Release:8.0.0  FreeMem:3689MB/4032MB BootTime: ...
Processes: 28, Threads: 105
Processor1: 1091555459 Cortex-A72 1500MHz FPU
Processor2: 1091555459 Cortex-A72 1500MHz FPU
Processor3: 1091555459 Cortex-A72 1500MHz FPU
Processor4: 1091555459 Cortex-A72 1500MHz FPU
```

The first line shows that the system has 4GB of RAM (depending on the model, yours may have 1GB, 2GB, 4GB or 8GB), with the system managing 4032MB, out of which 3689MB are still available for use. The QNX memory manager may not have access to all of RAM on a board if some of it is excluded by various boot-time services, hypervisors, etc.

A more detailed view is available by looking at the **/proc/vm/stats** pseudo-file. You will need to access this file as the root user, e.g.:

```
qnxuser@qnxpi:~$ su -c cat /proc/vm/stats
```

The output provides quite a bit of information, some of it only makes sense for people familiar with the internal workings of the memory manager. However, a few of the lines are of interest:

- `page_count=0xfc000` shows the number of 4K RAM pages available to the memory manager. The total amount of memory in the `pidin info` output should match this value.
- `vmem_avail=0xe67f6` is the number of pages that have not been reserved by allocations, and are thus available for new allocations. This value corresponds to the free memory number provided by `pidin info`.
- `vm_aspace=30` is the number of address spaces in the system, which corresponds to the number of active processes.[1]
- `pages_kernel=0x4347` is the number of pages used by the kernel for its own purposes. Note that the value includes the page-table pages allocated for processes.
- `pages_reserved=0x9012` is the number of pages that the memory manager knows about, but cannot allocate to processes as regular memory. Such ranges are typically the result of the startup program, which runs before the kernel, reserving memory for special-purpose pools, or as a way to reduce boot time (in which case the memory can be added to the system later).

The remaining `pages_*` lines represent allocated pages in different internal states.

The breakup of physical memory into various areas can be observed with the following command

```
qnxuser@qnxpi:~$ pidin syspage=asinfo
```

All entries that end with `sysram` are available for the memory manager to use when servicing normal allocation requests from processes. Entries that are part of RAM but outside the sysram ranges can be allocated using a special interface known as *typed memory*.

We will now attempt to answer the question of how much does a process contribute to total memory use in the system. To answer this question, we will use the **mmap_demo** program that is included with the Raspberry Pi image. Start this program in the background, so that it remains alive while we examine its memory usage.

[1] The word *active* here is not a redundancy, as a process in the middle of its creation, or after termination but before being claimed by its parent (i.e., a zombie) does not have an address space.

```
qnxuser@qnxpi:~$ mmap_demo &
[1] 1617946
qnxuser@qnxpi:~$ Anonymous private mapping 3de2cf0000-3de2cf9fff
Anonymous shared mapping 3de2cfa000-3de2cfbfff
File-backed private mapping 209b3a6000-209b3a9fff
File-backed shared mapping 209b3aa000-209b3adfff
Virtual address range only mapping 3de2cfc000-3ed6f3bfff
```

This program creates different types of mappings (though it doesn't come close to exhausting all the different combinations of options that can be passed to `mmap()`). The first two mappings are anonymous, which means that the resulting memory is filled with zeros. The next two are file-backed, which means that the memory reflects the contents of a file (the program uses a temporary file for the purpose of the demonstration). Finally, the last mapping just carves a large portion of the process' virtual address space, without backing it with memory. The output shows the virtual addresses assigned to each of these mappings. The start address is chosen by the *address space layout randomization* (ASLR) algorithm, while the end address of each range reflects the requested size. The output you see will therefore be different than the example above for the start addresses, but the sizes will be the same.

To get a view of the memory usage by this process we can examine all the mappings it has. This is done by looking at the **/proc/<pid>/pmap** file, where **<pid>** should be replaced by the process ID (1617946 in the example above):[2]

```
qnxuser@qnxpi:~$ cat /proc/1617946/pmap
vaddr,size,flags,prot,maxprot,dev,ino,offset,rsv,guardsize,refcnt,mapc...
0x0000001985914000,0x0000000000080000,0x00081002,0x03,0x0f,0x00000001,...
0x000000209b3a6000,0x0000000000004000,0x00000002,0x03,0x0f,0x0000040c,...
0x000000209b3aa000,0x0000000000004000,0x00000001,0x03,0x0f,0x0000040c,...
0x0000003de2ced000,0x000000000000d000,0x00080002,0x03,0x0f,0x00000001,...
0x0000003de2cfa000,0x0000000000002000,0x00080001,0x03,0x0f,0x00000001,...
0x0000003de2cfc000,0x00000000f4240000,0x00080002,0x00,0x0f,0x00000001,...
0x0000004843966000,0x0000000000038000,0x00010031,0x05,0x0d,0x00000802,...
0x000000484399e000,0x0000000000002000,0x00010032,0x01,0x0f,0x00000802,...
0x00000048439a0000,0x0000000000001000,0x00010032,0x03,0x0f,0x00000802,...
0x00000048439a1000,0x0000000000001000,0x00080032,0x03,0x0f,0x00000001,...
0x00000048439a2000,0x000000000a6000,0x00010031,0x05,0x0d,0x00000802,...
0x0000004843a48000,0x0000000000004000,0x00080032,0x01,0x0f,0x00000001,...
0x0000004843a4c000,0x0000000000002000,0x00010032,0x03,0x0f,0x00000802,...
0x0000004843a4e000,0x0000000000006000,0x00080032,0x03,0x0f,0x00000001,...
0x0000004843a54000,0x0000000000013000,0x00010031,0x05,0x0d,0x00000802,...
0x0000004843a67000,0x0000000000001000,0x00080032,0x01,0x0f,0x00000001,...
0x0000004843a68000,0x0000000000001000,0x00010032,0x03,0x0f,0x00000802,...
0x0000005b7124b000,0x0000000000001000,0x00000031,0x05,0x0d,0x0000040b,...
0x0000005b7124c000,0x0000000000001000,0x00000032,0x01,0x0f,0x0000040b,...
0x0000005b7124d000,0x0000000000001000,0x00000032,0x03,0x0f,0x0000040b,...
```

Note that the output was truncated to fit the page. Here is a complete line that will be examined in detail:

[2] Confusingly, every process also has a **/proc/<pid>/mappings** file, which provides the state of every page assigned to this process (both from the virtual and physical point of view). This file can be huge and is rarely of interest.

```
0x000000209b3a6000,0x0000000000004000,0x00000002,0x03,0x0f,0x0000040c,
0x000000000000001c,0x0000000000000000,0x0000000000004000,0x00000000,
0x00000004,0x00000002,/data/home/qnxuser/mmap_demo.9ubgxv,{sysram}
```

The first two columns show the virtual address (0x209b3a6000) and size (0x4000, or 4 4K pages) of the mapping. You should find a matching entry in the output of **mmap_demo**. The next two columns show the flags and protection bits passed to the `mmap()` call.[3] This line corresponds to a private mapping with read and write permissions. The next 4 fields are not important for this discussion.

Next comes the reservation field, which is crucial. This field shows how many pages were billed to this process for the mapping. In this case the value is 0x4000, which is the same as the size of the mapping. The reason is that this is a private mapping of a shared object, and such a mapping requires that the system allocate new pages for it with a *copy* of the contents of the object (in this case the temporary file). By contrast, a read-only shared mapping of the same file will show a value of 0.

Of the last two fields, the first shows the mapped object. This can be a file, a shared-memory object, or a single anonymous object that is used to map the process' stacks and heaps. The second field (new in version 8.0 of the OS) shows the typed memory object from which memory was taken (recall that "sysram" refers to generic memory that the memory manager can use to satisfy most allocations).

Given the information provided here, why is it hard to answer the question about per-process memory consumption? The complexity comes from shared objects. Every process maps multiple such objects, including its own executable and the C library. As mentioned above, shared mapping of such objects (that is, mappings that directly reflect the contents of the object, rather than creating a private copy) show up in the **pmap** list as though they consume no memory. And yet the underlying object may[4] require memory allocation that does affect the amount of memory available in the system. Additionally, there may be shared memory objects that are not mapped by any process: one process can create such an object with a call to `shm_open()` and never map it. That process may even exit, leaving the shared memory object to linger (which is required for POSIX semantics). A full system analysis of memory consumption thus requires a careful consideration of all shared objects, along with which processes, if any, map them.

3.5 Resource Managers

A resource manager is a process that registers one or more paths in the path space, and then services requests that open files under these paths. The best example of a resource manager is a file system, which registers its mount path, and then handles all requests to open directories and files under that path. Once a file is opened by a client process, the client can send messages to the resource manager, which acts as

[3] The protection bit values are shifted right by 8 bits, due to internal representation concerns.

[4] Only *may*, because some objects do not reduce the amount of allocatable memory, such as files in the IFS, typed-memory pools or memory-mapped devices.

the server. Resource managers typically handle some standard message types, such as read, write, stat and close, but can also handle ad-hoc messages that are relevant to the specific service they provide.

An example of a resource manager on the QNX Raspberry Pi image is the GPIO server, **rpi_gpio**. You can see it running on the system by using the **pidin** command, as described above:

```
qnxuser@qnxpi:~$ pidin -p rpi_gpio
     pid tid name                         prio STATE       Blocked
   86033   1 proc/boot/rpi_gpio            10r RECEIVE     1
   86033   2 proc/boot/rpi_gpio           200r INTR
```

The resource managed by this process is the 40-pin GPIO header on the Raspberry Pi (See Chapter 6} for more information). It does so by memory-mapping the hardware registers that control the header, and then handling requests by various client programs to control GPIOs, e.g., to turn an output GPIO pin on or off. The advantage of this design is that it allows only one process to have access to the hardware registers, preventing GPIO users from interfering with each other, either by accident or maliciously.

The **rpi_gpio** resource manager registers the path **/dev/gpio** (though it can be configured to register a different path, if desired, via a command-line argument). Under this path it creates a file for each GPIO pin, with a name matching that of the GPIO number, as well as a single **msg** file.

```
qnxuser@qnxpi:~$ ls /dev/gpio
0   12  16  2   23  27  30  34  38  41  45  49  52  8
1   13  17  20  24  28  31  35  39  42  46  5   53  9
10  14  18  21  25  29  32  36  4   43  47  50  6   msg
11  15  19  22  26  3   33  37  40  44  48  51  7
```

The numbered files can be read and written, which means that the resource manager handles messages of type `_IO_READ` and `_IO_WRITE` on these files. Consequently, we can use shell utilities such as **echo** and **cat** on these files. To see this at work, follow the instructions for building a simple LED circuit, as described in Section 4.2. Do not proceed to write Python code at this point. Once the circuit is built, log in to the Raspberry Pi and run the following shell commands:

```
qnxuser@qnxpi:~$ echo out > /dev/gpio/16
qnxuser@qnxpi:~$ echo on > /dev/gpio/16
qnxuser@qnxpi:~$ echo off > /dev/gpio/16
```

The first command configures GPIO 16 as an output. The second turns the GPIO on (sets it to "HIGH") and the third turns the GPIO off (sets it to "LOW").

Let us consider how these shell commands end up changing the state of the GPIO pin.

1. The command `echo on > /dev/gpio/16` causes the shell to open the file **/dev/gpio/16**, which establishes a connection (via a file descriptor) to the

rpi_gpio resource manager.
2. The shell executes the **echo** command, with its standard output replaced by the connection to the resource manager.
3. **echo** invokes the `write()` function on its standard output with a buffer containing the string "on". The C library implements that call by sending a `_IO_WRITE` message to the resource manager.
4. The resource manager handles the `_IO_WRITE` message sent by **echo** as the client. It knows (from the original request to open the file **16** under its path) that the message pertains to GPIO 16, and it knows from the buffer passed in the message payload that the requested command is "on". It then proceeds to change the state of the GPIO pin by writing to the GPIO header's registers.

The **msg** file can be used for sending ad-hoc messages, which provide better control of GPIOs without the need for the resource manager to parse text. Such messages are easier to handle and are less error prone. The structure of each message is defined in a header file that client programs can incorporate into their code. We will see in Section 4.7 how a Python library uses such messages to implement a GPIO interface.

For information on how to write resource managers, see the QNX documentation.

Chapter 4

Controlling I/O with Python

In this chapter we will get the Raspberry Pi to do some useful (and fun!) work, by controlling various external devices. Such control will be achieved by writing code in the Python programming language and running it on the Raspberry Pi. To complete the exercises you will need a few components, including:

- breadboard
- jumper wires
- various resistors
- LEDs
- breadboard push buttons
- DC motor
- two or more servos
- infrared LED and a photodiode
- PCF8591 Analog-to-digital converter
- 10KΩ potentiometer
- PCA9685-based 16-channel PWM board
- L293 or L298 H-bridge
- external DC power supply (a battery pack will do)

You may already have some or all of these components. If not, there are many startup kits available for the Raspberry Pi that include all of these. Such kits often come with their own Python tutorials, and these tutorials can be much more comprehensive than what is available in this book. There are also books dedicated to maker projects and robotics with the Raspberry Pi, and many of those also use Python. Feel free to use any of these resources on top of, or in lieu, of the information in this chapter. The Python libraries for GPIO and I2C included with the QNX image for Raspberry Pi have been designed to be as compatible as possible with those available for other operating systems. You can then come back to Section 4.7 to learn how the Python code you write allows the Raspberry Pi to control external devices.

⚠ Warning

Never power a high-current device (such as a motor or a servo) directly from the Raspberry Pi's GPIO header. Such devices should get their power from an external source that matches their voltage and current requirements.

4.1 Running Python Programs

Section 2.6 describes the various methods in which you can write code for programs to run on the image. Since Python is an interpreted language it needs a special program, the Python interpreter, which runs the code directly from the source file (or files). There is no need first to compile the code into an executable. As with writing code, there are a few options for running the resulting program on the Raspberry Pi. For the following examples we will use a trivial Python program to print "Hello World!":

hello.py

```
1 print("Hello World!")
```

4.1.1 Using the command line

Before we can run the program we need to save the code in a file, e.g., **hello.py**. Henceforth we will assume that the files for all of the Python examples in this book are stored in a folder called **python** under the home directory of the **qnxuser** user. The full path for a file called **hello.py** is thus **/data/home/qnxuser/python/hello.py** or, using an abbreviated notation, **~qnxuser/python/hello.py**. If the directory does not yet exist, log into the Raspberry Pi and create it with the `mkdir` command:

```
qnxuser@qnxpi:~$ mkdir ~/python
```

If you write the code directly on the Raspberry Pi, or if you use SSHFS to mount a directory from the Raspberry Pi into your image, then the file is now stored in the target directory. On the other hand, if you saved it on your local computer then it needs to be copied to the Raspberry Pi:

```
$ scp hello.py qnxuser@qnxpi:python/
```

We can now invoke the python interpreter to run the program. If you haven't done so already, log into the Raspberry Pi with SSH and type the following commands:

```
qnxuser@qnxpi:~$ cd ~/python
qnxuser@qnxpi:~/python$ python hello.py
Hello World!
```

4.1.2 Remote Execution

While SSH can be used to start a shell on the Raspberry Pi, it can also be used to execute a command remotely. The following commands first copy the file from the computer to the Raspberry Pi, and then run the program:

```
$ scp hello.py qnxuser@qnxpi:python/
$ ssh -t qnxuser@qnxpi python python/hello.py
```

Note

Make sure to specify `-t` as part of the SSH command. Without it, the program may continue running on the Raspberry Pi even after the SSH command exits.

4.2 Basic Output (LED)

Our first project will blink a light-emitting diode (LED). You will need a breadboard, an LED (any colour will do), a small resistor and two jumper wires. The value of the resistor depends on the LED but for the purpose of this simple experiment any value in the range of a 30Ω to 330Ω will do. The purpose of the resistor is to limit the current and protect the LED from burning out.

Build the circuit by following these steps. See Chapter 6 for pin number assignment. Note that GPIO numbers and pin numbers are not the same.

1. Connect the long leg of the LED (the positive leg, or *anode*) to any hole in the breadboard.
2. Connect the short leg of the LED (the negative leg, or *cathode*) to a different hole. Make sure that different hole is also in a different row, as all holes within the same row are connected.
3. Connect one leg of the resistor to a hole in the same row as the LED long leg.
4. Connect the other leg of the resistor to a hole in a row different from either the long or short leg of the LED.
5. Connect a jumper wire such that one end plugs into a hole in the same row as the resistor leg in step 4, and the other end is connected to pin 36 in the Raspberry Pi GPIO header. This is GPIO 16.
6. Connect another jumper wire such that one end plugs into a hole in the same row as the short leg of the LED and the other end is connected to pin 14 in the Raspberry Pi GPIO header. This is a ground pin.

Note

The jumper wires can be connected to any other pin in the GPIO header, as long as the short leg of the LED is connected to a ground pin and the resistor is connected to a GPIO pin. If you decide to use a different GPIO pin make sure to adjust the code such that it uses the correct number.

The following diagram illustrates the circuit:

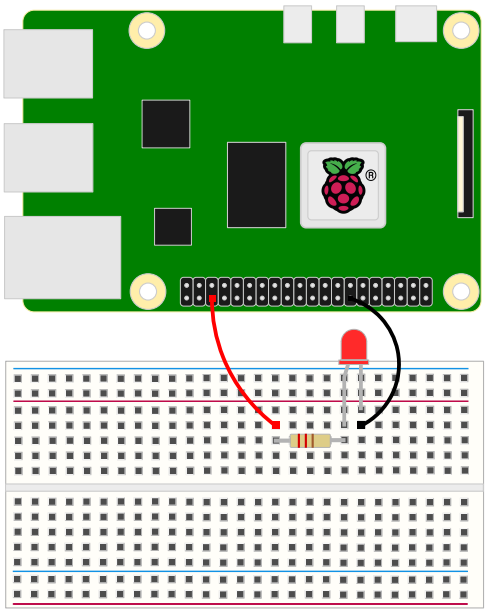


Figure 4.1: A basic LED circuit

Next, we will write the code for making the LED blink:

led.py

```
1  import rpi_gpio as GPIO
2  import time
3
4  GPIO.setup(16, GPIO.OUT)
5  GPIO.output(16, GPIO.LOW)
6
7  while True:
8      GPIO.output(16, GPIO.HIGH)
9      time.sleep(.5)
10     GPIO.output(16, GPIO.LOW)
11     time.sleep(.5)
```

Save this file as **led.py**, and run the program as described in the previous section. For example, if using the command line while connected via SSH, run the command:

```
qnxuser@qnxpi:~/python$ python led.py
```

If all goes well the LED should turn on and off at half second intervals. You can press Ctrl-C to stop it.

If the program doesn't work, try the following steps to determine what has gone wrong:

1. Double-check all connections.
2. Ensure the jumper wires are connected to the correct GPIO header pins.
3. Connect the positive jumper wire to a 3v pin instead of a GPIO pin (e.g., pin number 1). The LED should turn on. If it doesn't, try a different LED (in case this one is burnt out).
4. Check your code to ensure that it is exactly the same as in the listing. If you connected the positive wire to a different GPIO pin, make sure that all references to 16 have been replace by the right number.

We will now take a closer look at the code. The first two lines make the necessary libraries available to the program. The `rpi_gpio` library provides the functionality for working with the GPIO pins. It is made visible using an alias `GPIO` so that the code matches examples found on the Internet for similar libraries written for other operating systems, such as the Raspbian. The time library is used to add the necessary delays via the `time.sleep` call.

Next, the program prepares the GPIO to be used as an output. Every GPIO can function as either an output or an input. Many GPIO pins also have other functions, some of which will be explored in the next sections. The program then enters an infinite loop in which the LED's state is toggled between high (on) and low (off), using a delay of 500 milliseconds between each state change.

> **Note**
>
> The code in this program was stripped to the bare minimum, eschewing functionality that may be considered as best practice. For example, it does not ensure that the GPIO state is set back as low when the program is terminated.

4.3 Basic Input (Push Button)

The next exercise adds a push button to the LED circuit. A push button usually has 4 legs, such that each pair protruding from opposite sides is permanently connected. When the button is pressed all four legs become connected, allowing current to flow. We will use the Raspberry Pi to detect when the button is pressed and turn on the LED.

Modify the circuit from the previous exercise to match the following diagram:

Note that the Raspberry Pi ground pin is now connected to the breadboard's ground

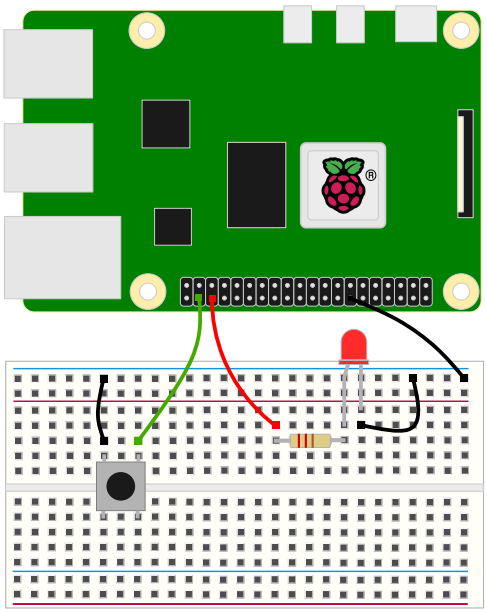


Figure 4.2: A circuit with a push button and an LED

bus. This allows us to have a common ground for all components, without using more ground pins from the Raspberry Pi. The LED's anode is connected to GPIO 16 as before. The push button is connected to ground on one leg and to GPIO 20 on another. The permanently connected legs straddle the trough in the middle of the breadboard.

Save the following program as **button.py**:

button.py

```
1  import rpi_gpio as GPIO
2  import time
3
4  GPIO.setup(16, GPIO.OUT)
5  GPIO.output(16, GPIO.LOW)
6
7  GPIO.setup(20, GPIO.IN, GPIO.PUD_UP)
8
9  while True:
10     if GPIO.input(20) == GPIO.LOW:
11         GPIO.output(16, GPIO.HIGH)
12     else:
13         GPIO.output(16, GPIO.LOW)
14
15     time.sleep(.01)
```

The first lines are identical to those in the previous exercise. Line 7 defines GPIO 20 as input. The `GPIO.PUD_UP` argument tells the Raspberry Pi to use its internal pull-up resistor on this pin. Without a pull-up resistor the state of the pin is undetermined (or "floating"), which means that the connection to ground established by pushing

the button may not be detected. Pulling up means that the pin detects a high state by default, and then a low state (from the connection to ground) when the button is pressed. We could have replaced the connection of the button to ground with a connection to a 3V pin and used `GPIO.PUD_DOWN`, in which case the default state of GPIO 20 would have been low, and would change to high when the button is pressed. The internal pull resistors in the Raspberry Pi avoid the need for using discrete resistors in the circuit to achieve the same effect.

The loop on lines 9-15 checks the state of GPIO 20 every 10 milliseconds. If the state is high (the default, due to the pull-up resistor) then the LED is turned off. If the state is low that means that the button is pressed and the LED is turned on.

Having a check run every 10 milliseconds can waste processor cycles, while at the same time miss some events (especially if the button is replaced by some other detector for events that happen at a higher frequency). Instead of polling for button state changes, we can ask the system to detect such changes and notify our program when they happen:

button_event.py

```
1  import rpi_gpio as GPIO
2  import time
3
4  def buttonPressed(pin):
5      if GPIO.input(20) == GPIO.LOW:
6          GPIO.output(16, GPIO.HIGH)
7      else:
8          GPIO.output(16, GPIO.LOW)
9
10 GPIO.setup(16, GPIO.OUT)
11 GPIO.output(16, GPIO.LOW)
12
13 GPIO.setup(20, GPIO.IN, GPIO.PUD_UP)
14
15 GPIO.add_event_detect(20, GPIO.BOTH, callback = buttonPressed)
16
17 while True:
18     time.sleep(1)
```

The call to `GPIO.add_event_detect()` installs a callback function which is executed whenever GPIO 20 detects a change from high to low ("falling edge") or from low to high ("rising edge"). The callback can be restricted to just one of these changes, by changing `GPIO.BOTH` to `GPIO.FALLING` or `GPIO.RISING`, respectively. The loop on lines 17-18 just sleeps, but can be replaced with some other code to perform tasks while still servicing button presses.

One problem that may arise when dealing with inputs is that the value can change rapidly, or "bounce", before it stabilizes. This is not an issue in this exercise, as the state of the LED follows that of the button. If, however, you change the program such that the state of the LED changes with every button press (i.e., the LED is turned on when the button is pressed, and then turned off when the button is pressed a second time), you may find that on some presses the state of the LED is not the desired one. Fixing such problems is referred to as "debouncing", which can be done either electronically

(e.g., by adding a capacitor), or in code (e.g., by requiring the same value to be read some number of times consecutively before deciding that the value has changed).

It may seem redundant, and somewhat excessive, to use a 4-core, multi-gigabyte computer to achieve the same effect as connecting the button directly to the LED. In a real project the button can be used to initiate a much more interesting activity. Also, the input may not be a push button at all. Other common components that provide discrete input (i.e., input that is either "on" or "off") include magnetic reed switches, photo-resistors detecting infrared light and various sensors.

4.4 PWM

4.4.1 Background

The GPIO pins on the Raspberry Pi are only capable of discrete output - each pin, when functioning as an output, can be either in a high state or a low state. Sometimes, however, it is necessary to have intermediate values, e.g., to vary the brightness of an LED or to generate sound at different pitch levels. One option for accomplishing such tasks is to use a digital to analog converter, which is a separate device that can be connected to the Raspberry Pi over a bus such as I2C or SPI (see below). Alternatively, we can use Pulse Width Modulation (PWM) to approximate an intermediate state.

PWM works by changing the state of a GPIO output repeatedly at regular intervals. Within a period of time, the state is set to high for a certain fraction of that time, and to low for the remainder. If the period is long enough the result is a noticeable pulse. With a short period the effect is perceived as an intermediate value. What constitutes "short" and "long" depends on the device and human perception.

PWM is specified using two values: the period, which is the repeating interval, and the duty cycle, which is the fraction of the period in which the output is high. The period can be specified in units of time (typically milliseconds or microseconds), or in terms of frequency (specified in Hz). Thus a period of 20ms is equivalent to a frequency of 50Hz. The duty cycle can be specified as a percentage of the period, or in time units. A duty cycle of 1ms for a period of 10ms is thus 10%. Period and duty cycle are demonstrated in Figure 4.3.

It is possible to implement PWM in software, by configuring any GPIO pin as an output and using a repeating timer to toggle its state. However, the Raspberry Pi provides a hardware implementation, albeit on just four GPIO pins: 12, 13, 18 and 19. Note, however, that only two PWM channels are available: channel 1 for GPIOs 12 and 18, and channel 2 for GPIOs 13 and 19. This means that if both GPIO 13 and 19 are configured for hardware PWM then they will have the same period and duty cycle. Moreover, both channels share the same clock, further limiting their independence. Expansion boards are available that allow for more PWM channels with finer control, and these are recommended for applications that require multiple PWM sources, such as robotic arms.

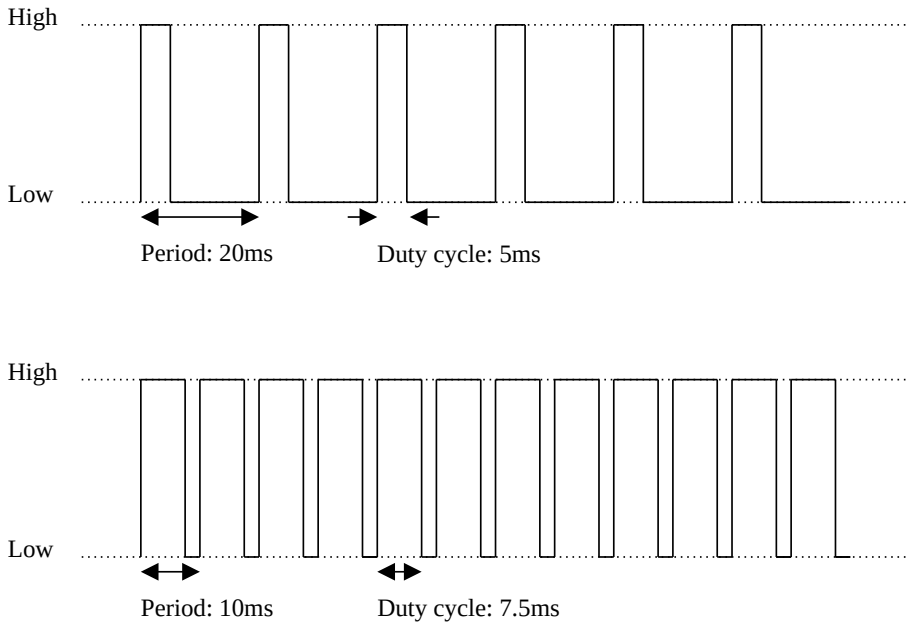


Figure 4.3: Period and duty cycle in PWM

In addition to the period and duty cycle, each PWM pin can be configured to work in one of two modes of operation. The first mode, referred to in the manual as M/S mode, works exactly as depicted in Figure 4.3, i.e., the GPIO is kept high for the duration of the duty cycle and then low for the remainder. The second mode spreads the duty cycle more or less evenly across the period, with several transitions from low to high and back. This mode more closely emulates an analog output and is preferred when such output is desired. On the other hand, M/S mode is used with devices that expect a signal with exact characteristics as a mode of communication. This is the case for servo motors, which will be described below.

4.4.2 Fading LED

For our first experiment with PWM, build the following circuit. This circuit is similar to the simple LED circuit from the first exercise, only this time we use GPIO 12, which is one of the pins that supports hardware PWM.

Now save this program as **pwm_led.py** and run it. You should see the LED fade in and out.

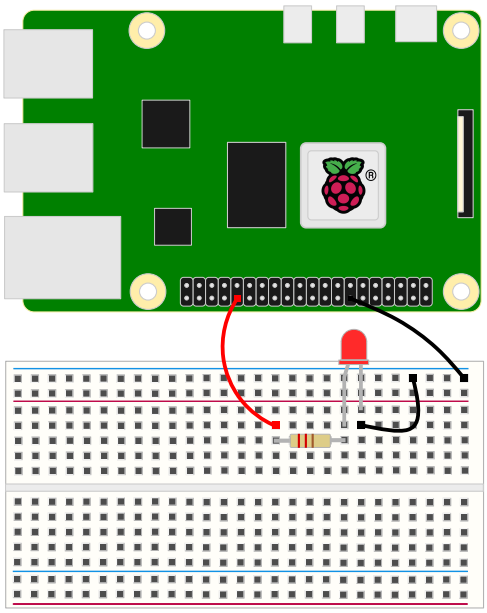


Figure 4.4: LED circuit connected to a PWM-enabled GPIO pin

pwm_led.py

```
1  import rpi_gpio as GPIO
2  import time
3
4  pwm = GPIO.PWM(12, 1000, mode = GPIO.PWM.MODE_PWM)
5
6  while True:
7      for dc in range(0, 100, 5):
8          pwm.ChangeDutyCycle(dc)
9          time.sleep(.1)
10
11     for dc in range(100, 0, -5):
12         pwm.ChangeDutyCycle(dc)
13         time.sleep(.1)
```

Line 4 creates a PWM object using GPIO 12 with a frequency of 1KHz and the standard PWM mode (not M/S). The loop on lines 7-9 changes the duty cycles from 0% to 100% in increments of 5% every 100 milliseconds, while the loop on lines 11-13 does the opposite.

4.4.3 Servo

Next, we will use PWM to control a servo motor. A servo is an electric motor that has a control mechanism that allows for exact positioning of the shaft. The angle of the shaft is communicated to the motor via PWM, where the period is fixed and the duty cycle is used to determine the angle.

There are many different types of servo motors, but one of the most common for beginners is the SG90, which is marketed under different brand names. This is an inexpensive, albeit weak, servo, but will do for the purpose of this exercise. The shaft can only be rotated 180 degrees between a minimum position and a maximum position.

The servo has three wires: brown (ground), red (power) and orange (control). Connect the servo as in the diagram:

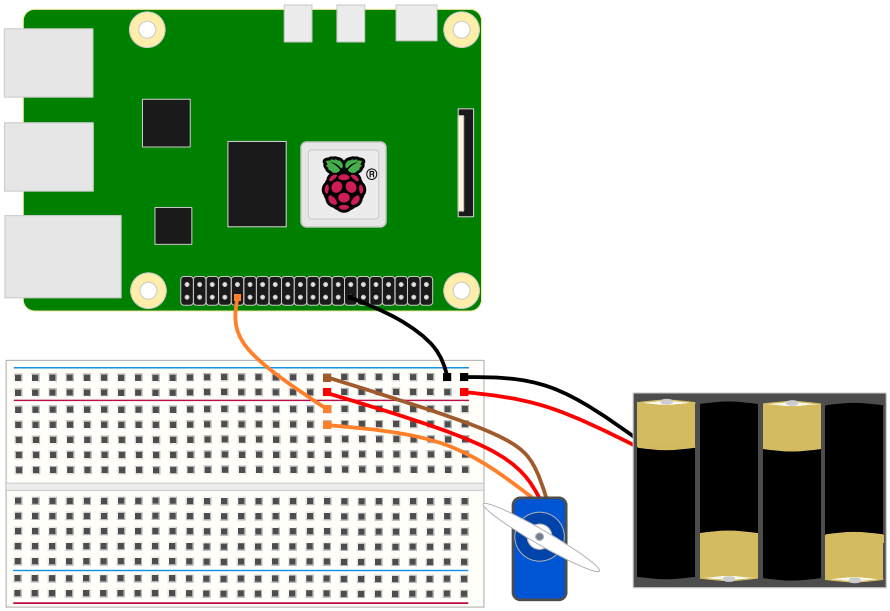


Figure 4.5: Servo circuit

Note that we have added an external power source (the diagram shows 1.5V batteries connected in series, but you can substitute a different DC source). The required voltage for the SG90 servo is between 4.8V and 6V. As mentioned before, do not use the Raspberry Pi 5V pin to power the servo, as it may exceed the current limit for this pin and damage the Raspberry Pi. It is important, however, to make sure that the ground terminal of the external power source is connected to a ground pin on the Raspberry Pi (in this case both are connected to the ground bus on the breadboard).

The orange wire is connected to GPIO 12 for control via PWM. To control the servo we have to use a period of 20 milliseconds (or a frequency of 50Hz). The servo's motor is in its minimum position (0°) when the duty cycle is 0.5 milliseconds (or 2.5%), and at its maximum position (180°) when the duty cycle is 2.5 milliseconds (or 12.5%). Intermediate duty cycle values position the motor in between these two angles.

> **Note**
>
> Some datasheets claim that the minimum duty cycle is 1ms and the maximum is 2ms. It is not clear whether that is a mistake, or that different vendors manufacture SG90 servos with different control parameters. You can experiment with various values to see that you get a full 180° motion. Just be careful not to exceed the minimum or maximum values for too long, to avoid strain on the gears.

Save this program as **pwm_servo.py** and run it. It should cause the servo's arm to rotate to one side and then the other, going through several angle changes.

pwm_servo.py

```
1  import rpi_gpio as GPIO
2  import time
3
4  pwm = GPIO.PWM(12, 50, mode = GPIO.PWM.MODE_MS)
5
6  MIN = 2.5
7  MAX = 12.5
8  dc = MIN
9  while dc <= MAX:
10     pwm.ChangeDutyCycle(dc)
11     dc += 0.5
12     time.sleep(.5)
13
14 while dc >= MIN:
15     dc -= 0.5
16     pwm.ChangeDutyCycle(dc)
17     time.sleep(.5)
```

Line 4 creates a PWM object using GPIO 12, with a frequency of 50Hz. Note that the mode must be set to M/S, as the control mechanism expects a continuous high level for the duty cycle in each period. Lines 8-12 move the shaft from 0° to 180° in regular increments every 500 milliseconds, and lines 14-17 do the opposite.

4.5 I2C

4.5.1 Background

All of the examples so far have only used a few GPIO pins. Once you start building less trivial systems you may find that a 40-pin header is simply not enough. Consider an LED matrix with dozens of diodes, a robotic arm that requires multiple servos (remember that the Raspberry Pi has only two PWM channels) or a sensor that provides a wide range of values. Instead of using a GPIO pin for each bit of data, the Raspberry Pi can connect to external devices using a communication bus. The simplest of these is the Inter-Integrated Circuit bus, commonly referred to as I2C.

The I2C bus requires just two pins - a serial data pin (SDA) and a serial clock pin (SCL). The data pin is used to communicate data to the device, by switching between low and high states, while the clock pin synchronizes these changes such that the target device

can interpret them as a stream of bits. The device can respond with its own stream of bits, for bi-directional communication. Pin 3 on the Raspberry Pi is the SDA pin, while pin 5 is the SCL pin.

Each I2C device has its own interpretation of this bit stream, of which the controlling device needs to be aware. This establishes a device-specific protocol on top of the I2C transport layer.

How does I2C solve the problem of replacing multiple GPIO pins? For one, the bit stream can be used to convey much more information between the Raspberry Pi and a device than a simple on/off state, or even PWM. For example, an LED matrix can be told to change the state of an LED at a given row and column, by interpreting the bit stream as command packets. An analog-to-digital converter replies to requests from the Raspberry Pi with bits that are interpreted as a value in a certain range.

Moreover, multiple devices can be connected to each other, forming a chain. Each device has an address (typically 7 bits, for a maximum of 128 addresses in a chain), and will respond only to communication that is preceded by its address (which is part of the bit stream sent by the Raspberry Pi).

The following sections show two examples of I2C devices and how to communicate with them. You may not have these devices (though both are cheap and readily available), but the principles should be the same for all I2C devices.

4.5.2 PCF8591 Digital/Analog Converter

PCF8591 is a simple 8-bit analog-to-digital and digital-to-analog converter. The device comes in the form of a 16-pin integrated circuit chip, though there are different packages (some with an extra carrier board). The device address is 72 (or 48 in hex), plus the value obtained from the chip's three address bit pins. Each of these address pins can be connected to ground for 0 or to the source voltage for 1, giving a total of 8 possible addresses between 72 and 79 (inclusive).

The device provides four analog inputs, which can be read by sending the byte 64 (or 40 in hex), plus the number of the input, between 0 and 3. Thus sending the byte 64 on address 72 reads the first analog input, byte 65 on address 72 the second input, etc. The value returned is between 0 and 255.

In this experiment we will use a 10KΩ potentiometer to change the value of input 0, read the result and print it to the console. The build is more complicated than in previous exercises, so make sure to check all connections. The PCF8591 chip should be inserted into the breadboard such that it straddles the middle trough, and with the semi-circular notch oriented as in the diagram:

Pin 1 from the Raspberry Pi, which provides a 3.3V source, is connected to one of the positive buses in the breadboard. That bus is then connected to the second positive bus on the other side of the breadboard. Similarly, pin 14 (ground) is connected to the first negative bus, which is then connected to the second negative bus. This configuration

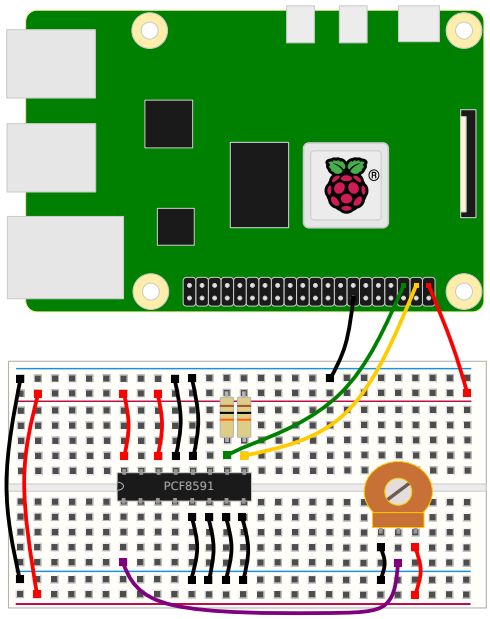


Figure 4.6: Analog-to-digital converter with a potentiometer

makes it easy to connect multiple pins from the PCF8591 chip to the source voltage and to ground.

The yellow and green wires in the diagram are connected to the SDA and SCL pins, respectively, on both the Raspberry Pi and the PCF8591 chip. These pins on the chip are also connected via 10KΩ pull-up resistors to the source voltage (or else they will not be able to detect changes on these pins). All of the address pins are connected to ground, which means that the device address is 72 (48 hex).

Finally, a 10KΩ potentiometer is connected to the source voltage, analog input 0 on the PCF8591, and ground.

Run the following program, and then turn the potentiometer to see the output.

adc.py

```
1  import smbus
2  import time
3
4  adc = smbus.SMBus(1)
5
6  prev_value = 0
7  while True:
8      value = adc.read_byte_data(0x48, 0x40)
9      if prev_value != value:
10         print('New value: {}'.format(value))
11         prev_value = value
12
13     time.sleep(0.1)
```

Line 4 establishes a connection to the I2C resource manager. The number 1 in the argument to `smbus.SMBus()` matches the I2C bus number, as assigned by the resource manager (not to be confused with the address assigned to the device, the ADC converter in this case, attached to that bus). If you get an error on this line check which device was registered by the resource manager:

```
qnxuser@qnxpi:~$ ls /dev/i2c*
/dev/i2c1
```

If the result is different than `/dev/i2c1` change the code to match the number in the device name.

Line 8 sends the command 0x40 on address 0x48 in order to request the value for analog input 0 from the device. It then prints the value if it has changed since the last time it was read. Turning the potentiometer all the way to one end should result in the value 0, the other end in the value 255, and intermediate positions in values between these two.

4.5.3 PCA9685 16 Channel PWM Controller

PCA9685 is a 16-channel, I2C-controlled integrated circuit. Each channel can be controlled individually by specifying two 12-bit values (i.e., values between 0 and 4095). The first value determines the time within the PWM period in which the channel turns on and the second value determines the time in which it is turned off. The period is determined by a common frequency value, which is derived from the internal oscillator running at 25MHz and a programmable pre-scale value. For example, for a frequency of 1KHz (i.e., a period of 1ms), if the first value is 300, and the second value is 1600, then the channel will turn on 73 microseconds after the beginning of the period and turn off 390 microseconds after the beginning of the period, resulting in a duty cycle of 31.7%.

Several vendors provide a board that contains a PCA9685 chip and 16 3-pin headers for connecting servos. Such a board is an easy and cheap way to control multiple servos. As mentioned before the Raspberry Pi only provides two PWM channels, which prevents the use of more than two independently-controlled servos at the same

time.

The board connects to the Raspberry Pi using 4 wires: 3.3V, ground, SDA and SCL. An independent power source is connected to the two power terminals, which is used to power the servos (recall that you cannot use the 5V output from the Raspberry Pi as it cannot sustain the current requirements of the servos). Finally, up to 16 servos can be connected to the 3-pin headers. Figure 4.7 illustrates a circuit with two servo motors.

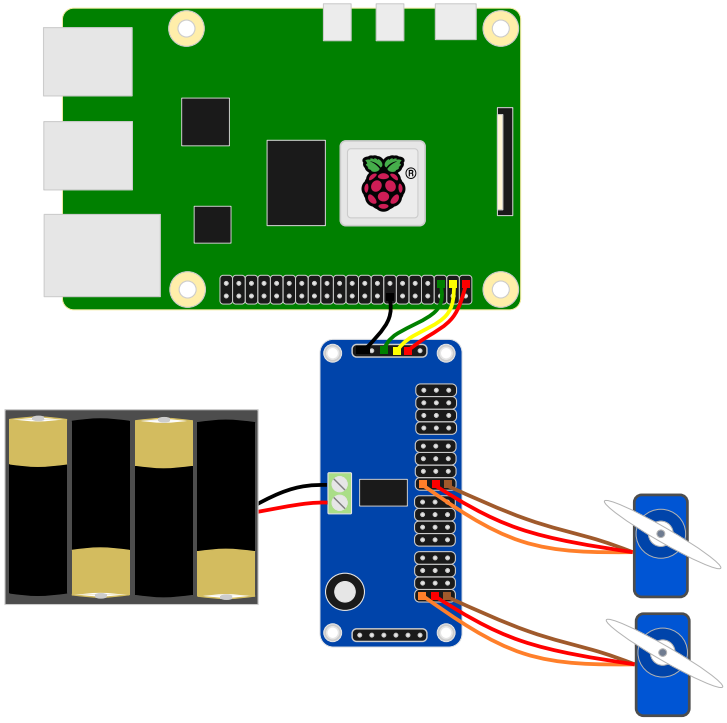


Figure 4.7: PCA9685 circuit with two servos

The chip is controlled by writing to I2C address 0x40 by default. It is possible to change this address by soldering one or more of the pads in the corner of the board, allowing for multiple devices to be connected on the same I2C bus. The two channel values are programmed with two bytes each, the first for the lower 8 bits, the second for the upper 2 bits. This means that each channel requires 4 bytes. These start at command number 6 for channel 0, and increment by four for each channel up to 15. Commands 0 and 1 are used to configure the chip, while command 254 configures the pre-scalar value to determine the frequency.

The following program controls the two servos, which are connected to channels 0 and 8, respectively, as in the diagram:

pca9685.py

```
1  import smbus
2  import time
3
4  smbus = smbus.SMBus(1)
5
6  # Configure PWM for 50Hz
7  smbus.write_byte_data(0x40, 0x0, 0x10)
8  smbus.write_byte_data(0x40, 0xfe, 0x7f)
9  smbus.write_byte_data(0x40, 0x0, 0xa0)
10 time.sleep(0.001)
11
12 # Servo 1 at 0 degrees
13 smbus.write_block_data(0x40, 0x6, [0x0, 0x0, 0x66, 0x0])
14
15 # Servo 2 at 180 degrees
16 smbus.write_block_data(0x40, 0x26, [0x0, 0x0, 0x0, 0x2])
17
18 time.sleep(1)
19
20 # Servo 1 at 180 degrees
21 smbus.write_block_data(0x40, 0x6, [0x0, 0x0, 0x0, 0x2])
22
23 # Servo 2 at 0 degrees
24 smbus.write_block_data(0x40, 0x26, [0x0, 0x0, 0x66, 0x0])
```

Lines 7-10 set the pre-scalar for a frequency of 50Hz.[1] This requires putting the chip to sleep first, and then restarting it, followed by a short delay. When restarting, the chip is also configured for command auto-increment, which means that we can write the per-channel 4-byte values in one I2C transaction. Line 13 writes the four byte values for channel 0, starting at command 6. These values say that the channel should turn on immediately at the beginning of a period, and turn off after 102 (0x66) steps out of 4096, giving a duty cycle of 2.5%. Line 16 configures channel 8, starting at command 38 (0x26), to turn on immediately at the beginning of the period and turn off after 512 steps out of 4096, giving a duty cycle of 12.5%. After one second each servo is turned the opposite way.

4.6 Towards Robotics: Motor Control

4.6.1 DC Motor with an H-Bridge

There are different kinds of motors used in robotics. We have already encountered servo motors, which provide precise control of the shaft angle. Servos, however, provide a limited motion range, and are not suitable for robot locomotion.[2] The simplest way to drive a robot is with a DC motor. The motor itself has no control mechanism: once connected to a voltage source and ground it moves at a constant speed in one direction. For a robot we would like to have control over the direction and the speed of the motor.

[1] According to the datasheet the pre-scalar value for 50Hz is calculated as 121, or 0x79. Nevertheless, an inspection with an oscilloscope revealed that a value of 0x7f is closer to the required frequency.

[2] Some servos have their internal gearbox modified to allow for continuous motion and then used for driving wheels. However, at that point they are really just standard motors.

> **Note**
>
> The typical DC motor is too fast and too weak to move a robot. It is therefore coupled with a gear box the slows the motor down while increasing its torque.

Controlling the direction can be done with a device called an *H-Bridge*, a fairly simple circuit that allows for the polarity of the connection to the motor to be switched. Thus, instead of one terminal being permanently connected to a voltage source and the other to ground, the two can change roles, reversing the direction of the motor.

While you can build an H-bridge yourself with a few transistors, it is easier to use an integrated circuit, such as L293D. This integrated circuit provides two H-bridges, which means it can control two motors at a time. Another popular choice is L298, which can work with higher voltage and is more suitable for bigger robots.

Figure 4.8 shows a circuit that combines a L293D chip with a 5V DC motor. The chip has two voltage source connections: one for the chip's logic, connected to a 5V pin on the Raspberry Pi, and the other for the motor, connected to an external power source. Inputs 1 and 2 of the L293D chip are connected to GPIOs 20 and 16, respectively, while outputs 1 and 2 are connected to the DC motor. GPIO 19 is connected to the pin that enables outputs 1 and 2. Note that the two ground buses are connected (by the left-most black wire) to ensure a common ground between the Raspberry Pi and the external power source. The four ground pins of the L293D chip are connected to the ground buses.

The motor is at rest when both inputs are the same (either high or low), rotates one way when input 1 is high and input 2 is low, and the other way when input 1 is low and input 2 is high. The following program demonstrates this.

1293.py

```
1  import rpi_gpio as GPIO
2  import time
3
4  GPIO.setup(20, GPIO.OUT)
5  GPIO.setup(16, GPIO.OUT)
6  GPIO.setup(19, GPIO.OUT)
7
8  GPIO.output(19, GPIO.HIGH)
9
10 GPIO.output(20, GPIO.HIGH)
11 GPIO.output(16, GPIO.LOW)
12 time.sleep(1)
13
14 GPIO.output(20, GPIO.LOW)
15 GPIO.output(16, GPIO.HIGH)
16 time.sleep(1)
17
18 GPIO.output(20, GPIO.LOW)
19 GPIO.output(16, GPIO.LOW)
20 GPIO.output(19, GPIO.LOW)
```

Line 8 enables the outputs. Lines 10-12 cause the motor to rotate one way for 1 second. Lines 13-16 reverse the motor's direction for one second. Then both outputs, as well

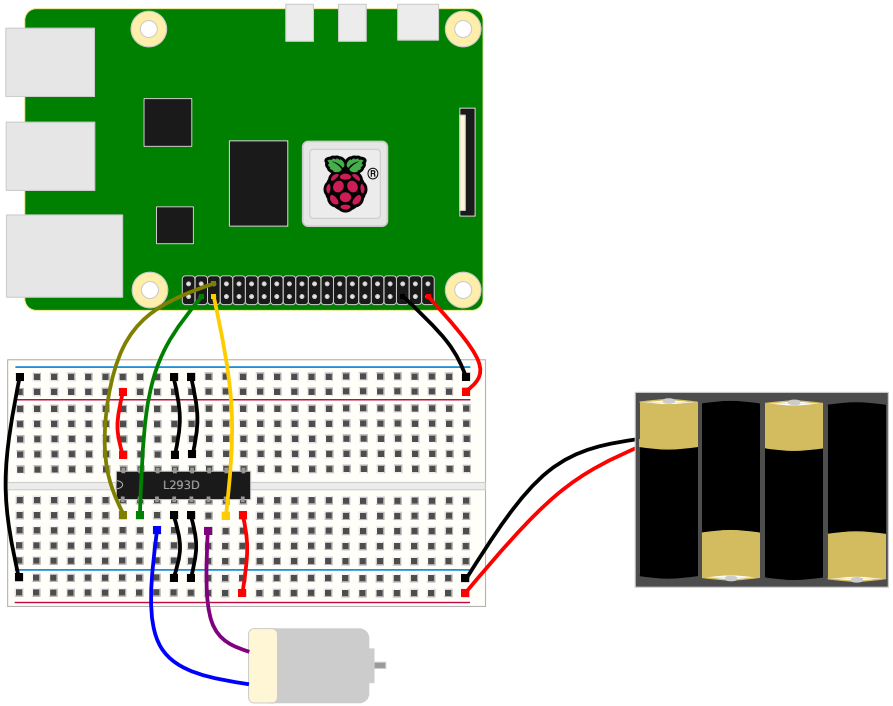


Figure 4.8: L293D circuit with a DC motor

as the enable pin, are turned off.

So far we have only used the L293D chip to control the direction of the motor. We can also control the speed by using PWM on the enable pin, which in our circuit is connected to GPIO 19. A duty cycle of 100% works as though the pin is always enabled, which means that the motor rotates at full speed. Lower values of the duty cycle slow the motor down. The next program starts the motor at full speed one way, slows it down to a halt, switches direction and then speeds up to full speed.

l293_pwm.py

```
import rpi_gpio as GPIO
import time

GPIO.setup(20, GPIO.OUT)
GPIO.setup(16, GPIO.OUT)
pwm = GPIO.PWM(19, 20, mode = GPIO.PWM.MODE_MS)

GPIO.output(20, GPIO.HIGH)
GPIO.output(16, GPIO.LOW)

for dc in range (100, 0, -1):
    pwm.ChangeDutyCycle(dc)
    time.sleep(0.1)

GPIO.output(20, GPIO.LOW)
GPIO.output(16, GPIO.HIGH)

for dc in range (0, 100):
    pwm.ChangeDutyCycle(dc)
    time.sleep(0.1)

pwm.ChangeDutyCycle(0)
GPIO.output(20, GPIO.LOW)
GPIO.output(16, GPIO.LOW)
```

4.6.2 Encoders

The level of control provided above is not sufficient for accurate positioning of a robot. Even if we set the speed of each wheel to a desired value and measure for a specific amount of time, there is no guarantee that the robot will move exactly as desired. The motors may take some time to reach the desired speed, and friction may affect movement. To solve this problem, we can use encoders, which tell us exactly how much each wheel has moved.

It is possible to purchase DC motors with built-in encoders. However, in the spirit of this book, we will create one from scratch using a pair diodes: an infrared LED and an infrared photodiode. First, construct a basic circuit to see these diodes at work, as depicted in Figure 4.9.

The clear diode is the infrared (henceforth IR) LED, while the black one is the photodiode. The IR LED is connected to 3.3V via a small resistor (27Ω in the example), which makes it permanently on. (An alternative is to connect the LED to a GPIO output and turn it on as needed). The photodiode's cathode (short leg) is directly connected to 3.3V (unlike an LED), while the anode (long leg) is connected via a large resistor

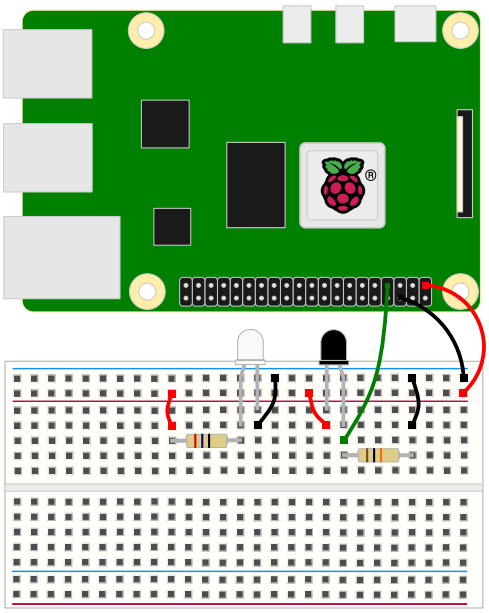


Figure 4.9: Infrared LED and photodiode circuit

(10KΩ in the example) to ground. A connection to GPIO 4 is made between the anode and the resistor.

If the photodiode does not detect IR light (which it should not in the configuration depicted in the diagram) then it does not pass any current. In this case the voltage level read by GPIO 4 is 0, as it is connected to ground via the resistor. However, when the photodiode detects IR light, which can be done by placing a white piece of paper over the heads to the two diodes, it passes current, which creates a voltage difference across the resistor. GPIO 4 detects this difference as a high input.

The next program reports how many times the photodiode detected IR light in the span of 5 seconds. Once you run it, move a white piece of paper back and forth at a small distance over the diodes, such that it alternates between covering and not covering these. The program should report how many times it transitioned between the non-covered and the covered states.

ir.py

```
import rpi_gpio as GPIO
import time

count = 0

def ir_detect(pin):
    global count
    count += 1

GPIO.setup(4, GPIO.IN, GPIO.PUD_DOWN)
GPIO.add_event_detect(4, GPIO.RISING, callback = ir_detect)

time.sleep(5)

print('Photodiode detected {} signals'.format(count))
```

To control a motor with IR light we add an *encoder wheel* to the motor. The encoder wheel is a disc with slots that can be detected by the photodiode as the motor rotates. Counting the slots allows the motor to be stopped by a program once the desired angle has been reached (which can be more than 360°). Using the angle and the circumference of the robot wheel attached to the motor (not the encoder disc) we can determine the distance that wheel has traversed. Figure 4.10 shows an encoder wheel with 9 slots.

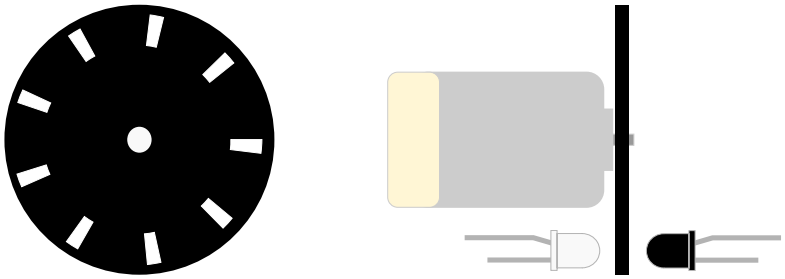


Figure 4.10: An encoder wheel

The slots can be holes in the disc, in which case the IR diodes are arranged as in Figure 4.10. Alternatively, the disc can have black stripes on a white background (or vice versa), in which case the diodes should be placed on the same side and use the reflection of the IR light for detection. For the purpose of this exercise you can use a 3D printer for creating an encoder wheel, or just make one out of cardboard. If opting for a slotted wheel, make sure that whatever material you use blocks IR light (printed plastic may need to be painted).

We will now combine the last two circuits for a full motor control device. The circuit, depicted in Figure 4.11 is somewhat complicated by the need to supply three different voltage sources: 5V from the Raspberry Pi to power the L293D chip, 3.3V from the Raspberry Pi for the IR LED and photodiode, and external power for the motor.[3]

[3] I was able to power the L293D from the 3.3V output of the Raspberry Pi, which simplifies the circuit. Nevertheless, the datasheet for the L293D specifies a minimum of 4.5V.

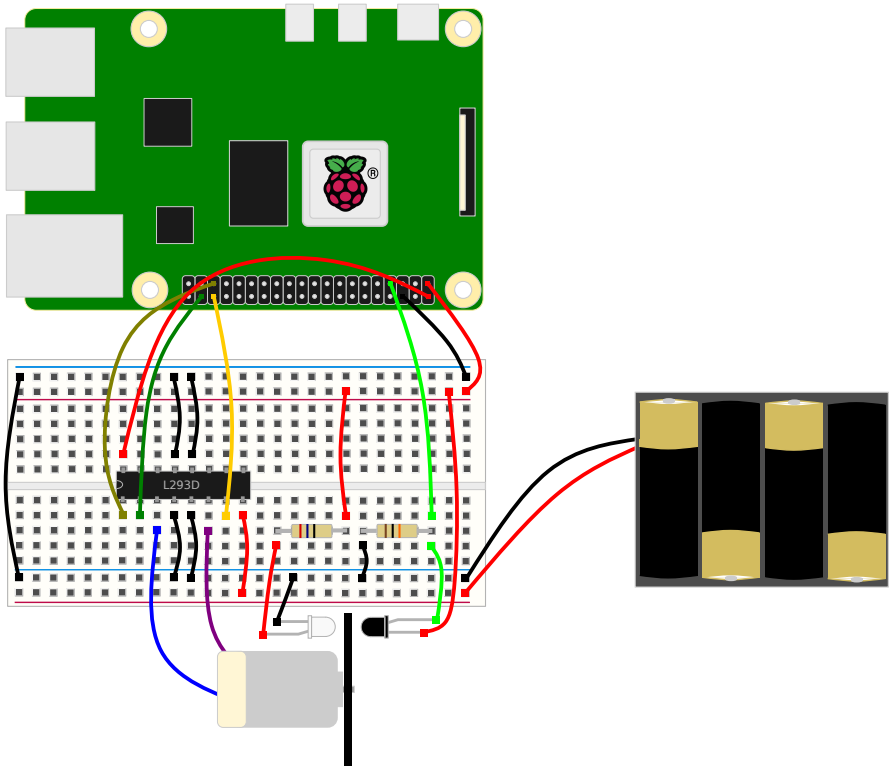


Figure 4.11: Motor and encoder circuit

The program for this circuit starts the motor at a 40% PWM duty-cycle (you can change this value depending on the rotation speed of your motor), and then stops it once 9 transitions from low to high have been detected. Assuming the encoder has 9 slots, as in the example, this code should stop the motor after one full rotation of the wheel.

encoder.py

```
import rpi_gpio as GPIO
import time

encoder_detect = 0

def stop_motor(pin):
    global encoder_detect
    encoder_detect += 1
    if encoder_detect == 9:
        GPIO.output(20, GPIO.LOW)
        GPIO.output(16, GPIO.LOW)
        print('Stopping motor')

GPIO.setup(4, GPIO.IN, GPIO.PUD_DOWN)
GPIO.add_event_detect(4, GPIO.RISING, callback = stop_motor)

GPIO.setup(20, GPIO.OUT)
GPIO.setup(16, GPIO.OUT)
pwm = GPIO.PWM(19, 20, mode = GPIO.PWM.MODE_MS)

GPIO.output(20, GPIO.HIGH)
GPIO.output(16, GPIO.LOW)

pwm.ChangeDutyCycle(40)

while encoder_detect < 9:
    time.sleep(0.1)

pwm.ChangeDutyCycle(0)
GPIO.output(20, GPIO.LOW)
GPIO.output(16, GPIO.LOW)
```

When running this program you may find that the motor is stopped well after the time it is supposed to. There are a couple of reasons for that:

1. Python is not very efficient. It is an interpreted language, with poor support for concurrency. By the time the thread that runs the callback function handles the events sent by the GPIO resource manager they may have been queued for quite some time.
2. Python's threads execute at the system's normal priority, which means that their scheduling is affected by that of other threads in the system (as well as each other).

We can alleviate the problem by using the `match` argument to `GPIO.add_event_detect()`, which allows the GPIO resource manager's event thread to keep track of changes, and notify the Python program only once the requested number of changes have occurred. This does not solve the problem fully, as the callback that stops the motor still runs at normal system priority. A better solution will be provided by running the same program using a high-priority thread for motor control, a subject that will be revisited in Section 5.3.

encoder_match.py

```
import rpi_gpio as GPIO
import time

encoder_detect = False

def stop_motor(pin):
    global encoder_detect
    GPIO.output(20, GPIO.LOW)
    GPIO.output(16, GPIO.LOW)
    print('Stopping motor')
    encoder_detect = True

GPIO.setup(4, GPIO.IN, GPIO.PUD_DOWN)
GPIO.add_event_detect(4, GPIO.RISING, match = 9, callback = stop_motor)

GPIO.setup(20, GPIO.OUT)
GPIO.setup(16, GPIO.OUT)
pwm = GPIO.PWM(19, 20, mode = GPIO.PWM.MODE_MS)

GPIO.output(20, GPIO.HIGH)
GPIO.output(16, GPIO.LOW)

pwm.ChangeDutyCycle(40)

while encoder_detect == False:
    time.sleep(0.1)

pwm.ChangeDutyCycle(0)
GPIO.output(20, GPIO.LOW)
GPIO.output(16, GPIO.LOW)
```

4.7 How Does It Work?

At this point you may be curious to know how does a Python program control the GPIO pins on the Raspberry Pi. (Of course, you may not be curious about it at all, in which case you can skip this section).

GPIO control in the Raspberry Pi QNX image is implemented by a resource manager (see Section 3.5) called **rpi_gpio**. The resource manager exposes a few files under the path **/dev/gpio**. We have already encountered this resource manager when we used the **echo** command to write to a file that controls a single GPIO pin from the shell. One of the files provided by the resource manager is **/dev/gpio/msg**, which can be opened by a client program. That program can then use QNX native message passing functions to request the resource manager to perform certain actions on its behalf.

The Python interpreter becomes such a client program when you import the `rpi_gpio` library (both the resource manager and the library have the same name, but are not to be confused). This library is implemented using Python's C extensions.[4] You can find the library file under **/system/lib/python3.11/lib-dynload/rpi_gpio.so**. When loaded, the library opens **/dev/gpio/msg**, keeping a file descriptor to the resource manager. It then uses `MsgSend()` to send messages to the resource manager and wait for it to reply. For example, the following code

[4] https://docs.python.org/3/extending/extending.html

implements the library's `output()` method, which was used in the LED example to turn the LED on and off:

```
1  static PyObject *
2  rpi_gpio_output(PyObject *self, PyObject *args)
3  {
4      unsigned    gpio;
5      unsigned    value;
6
7      if (!PyArg_ParseTuple(args, "II", &gpio, &value)) {
8          return NULL;
9      }
10
11     gpio = get_gpio(gpio);
12     if (gpio == 0) {
13         set_error("Invalid GPIO number");
14         return NULL;
15     }
16
17     if ((value != 0) && (value != 1)) {
18         set_error("Invalid GPIO output value");
19         return NULL;
20     }
21
22     rpi_gpio_msg_t  msg = {
23         .hdr.type = _IO_MSG,
24         .hdr.subtype = RPI_GPIO_WRITE,
25         .hdr.mgrid = RPI_GPIO_IOMGR,
26         .gpio = gpio,
27         .value = value
28     };
29
30     if (MsgSend(gpio_fd, &msg, sizeof(msg), NULL, 0) == -1) {
31         set_error("RPI_GPIO_WRITE: %s", strerror(errno));
32         return NULL;
33     }
34
35     Py_RETURN_NONE;
36 }
```

Lines 4-20 get the arguments passed to the function (the GPIO pin number and value, which can be either 1 for "HIGH" or 0 for "LOW"), and validate those. Lines 22-28 define a message, composed of a header and a payload. The header tells the resource manager that this message wants to write to a GPIO, while the payload tells it which GPIO pin to update and to what value. Finally, lines 30-33 send the message, wait for the reply and check for errors.

The resource manager runs a loop around the `MsgReceive()` call, which waits for client messages. When it gets such a message it uses the header to determine what type of message it is and then handles it appropriately. In the case of a `RPI_GPIO_WRITE` message the resource manager writes to the GPIO hardware registers a value that matches the GPIO pin specified in the payload of the message. If that value is 1 then it sets a bit in one of the GPSET registers, while for a value of 0 it sets a bit in one of the GPCLR registers. These registers are associated with physical addresses that the resource manager maps into its address space when it starts. More information about mapping hardware registers is provided in Section 5.6.

Chapter 5

Real-Time Programming in C

5.1 Building C Programs

Programs written in C need to be compiled and linked before they can be run on the Raspberry Pi. There are a few methods for doing that when using the QNX build tools. For the following discussion we will use the ubiquitous "Hello World" example:

hello.c

```c
#include <stdio.h>

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

5.1.1 Command Line

The compiler and linker front-end for QNX is called **qcc**. Front-end means that this tool is really a wrapper around the compiler and linker provided by the GNU project. As **qcc** can generate binaries and libraries for different targets it needs to be told the combination of target and tools.

```
$ qcc -Vgcc_ntoaarch64le -o hello hello.c
```

This command will build the code in **hello.c** into the executable **hello**. Once built, the executable can be copied to the Raspberry Pi and run:

```
$ scp hello qnxuser@qnxpi:
$ ssh -t qnxuser@qnxpi hello
```

or, from a QNX shell (after copying the executable to the Raspberry Pi):

```
qnxuser@qnxpi:~$ ./hello
Hello World!
```

Just as with other compilers, multiple source files can be specified on the command line to be linked together. Alternatively, the `-c` option can be used just to compile one source file into an object, and then the objects linked together.

5.1.2 Recursive Make

Of course, compiling multiple source files can quickly become tedious and error prone. The traditional UNIX `make` program can be used to keep track of multiple source files in a project, simplifying the task of building a complex executable. You can write your own make file using the standard language for such files. Alternatively, you can use the QNX recursive make system, a set of macros that facilitate the building of projects for QNX targets. While this system's greatest strength lies in its ability to create binaries for different targets from the same source code, it also simplifies the building of simple projects with just one target, by choosing the right tools and options.

The recursive make project for our hello world example consists of a top-level directory, an architecture-level directory and a variant-level directory. As we are only interested in building the program for the Raspberry Pi, the architecture level directory is **aarch64** and the variant level is **le**.[1] We could add a **linux-x86_64** directory at the architecture level with a **o** directory under it to build the same program for a Linux host.

```
$ tree hello
hello
|-- Makefile
|-- aarch64
|   |-- Makefile
|   `-- le
|       `-- Makefile
|-- common.mk
`-- hello.c
```

The **common.mk** file includes the files necessary for the recursive make system, along with definitions for the project as a whole (such as the name of the executable). A bare-bones file for our example looks like this:

[1] For simplicity, the OS-level directory was omitted, as all of our code is going to target QNX systems.

```
ifndef QCONFIG
QCONFIG=qconfig.mk
endif
include $(QCONFIG)

NAME=hello
USEFILE=

include $(MKFILES_ROOT)/qtargets.mk
```

The top-level make files simply tell make to continue down the hierarchy, while the bottom level file (the one under **hello/aarch64/le**) includes **common.mk**. It is the names of the directories that tell the recursive system what is the target for the binaries built under them.

We can now go into any of these directories and type `make` to build the executables for the targets covered by this level of the hierarchy. As we have only one target it doesn't matter which one we use.

5.1.3 VSCode

To use VSCode for QNX C/C++ development you can open an existing folder with pre-created make files (as in the example above), or you can create a new project directly in VSCode. To do that, open the command palette (Ctrl-Shift-P) and type "QNX: Create New QNX Project" (the command will be completed soon after you start typing). Choose a name, the make system to use and a default language. A project with a simple `main()` function is initialized and ready to be built.

Next, make sure that the Raspberry Pi is set as the default target. The list of targets is available in the QNX pane, which is made visible by clicking on the QNX plugin button. Once the default target is set you can launch the program on the Raspberry Pi. From the command palette choose "QNX: Run as QNX Application". The output should be visible in the debug console below. You can also use VSCode to debug the program.

5.2 Inter-Process Communication

Inter-Process Communication, or IPC, is a feature of operating systems that allows two processes, each running in its own private virtual address space, to interact. Shared memory, shared semaphores, pipes, sockets and even files accessible to multiple processes, can all be used as forms of IPC.

While there are use cases for inter-process communication on any operating system, IPC is an essential feature for an operating system based on a microkernel, such as QNX. No useful work can be done in such an operating system without communicating with the various services that are implemented by other processes. Allocating memory, writing to a file, sending a packet on the network, getting the coordinates from a USB mouse or displaying an image on the screen are all performed by talking to the various services that implement these features using IPC.

The most basic form of IPC in the QNX RTOS is synchronous message passing, also referred to as Send/Receive/Reply. Almost all of the scenarios mentioned above for the use of IPC in a microkernel operating system are implemented in QNX by the use of such messages. In fact, some of the other IPC mechanisms are built on top of synchronous message passing.

With synchronous message passing, one process, known as the *client*, requests a service from another process, known as the *server*, by sending it a message, and then waiting for a reply. The server receives the message, handles it, and then replies to the waiting client. Note that when we say that the client is waiting, we are only referring to one thread in the client that has sent the message to the server. Other threads in the client process can continue running while one thread is waiting for a reply.

The main advantage of synchronous message passing over other forms of IPC is that it does not queue messages. Consequently, there are no limits on the size of messages.[2] Also, synchronous message passing is self-throttling, making it harder for the client to flood the server with IPC requests at a higher rate than the server can handle. Finally, QNX message passing provides a scatter/gather feature, in which all the buffers used during the message pass (client's send and reply buffers, and the server's receive buffer) can be specified as arrays of sub-buffers, known as I/O vectors (IOV). Each of these sub-buffers consists of a base address and a length. With IOVs, messages can be assembled without first copying the components into a single buffer.

Another feature of synchronous message passing, which is absent from most other forms of IPC, is *priority inheritance*. When a client thread at priority 20 sends a message to a server, then the server thread that receives the message will have its priority changed to 20, for the duration of the time it handles the message. For more information on priorities see Section 5.3.

Before a client can send a message to a server, it needs to establish a communication conduit to that server. The server creates one or more *channels*, on which threads are waiting to receive messages. The use of channels as the end points of IPC rather than the server process itself allows the server to have multiple such end points, which it can then use either for different services (e.g., a file system process can have a separate channel for each mount point) or for different quality of service (e.g., separating services to privileged clients from those to non-privileged clients). Once a channel is created, via the `ChannelCreate()` kernel call, the client can establish a *connection* to that channel, using the `ConnectAttach()` kernel call.

The `ConnectAttach()` call takes the server's process ID and the channel ID, and returns a connection ID for use with calls to `MsgSend()`. However, the requirement to know the identifiers of the server process and its channel make it hard to write client code: these identifiers may be different on different systems, or even across boots of the same system. Consequently, we need a mechanism by which a server can advertise itself, and which the client can use to discover it. That mechanism is

[2] Well, there are limits. A single part in a message is limited to the size of the virtual address space, which is 512GB, and a message is limited to 512,000 parts, for a total limit of 256PB. If you run into this limit, please contact QNX support. We would love to see your system.

provided by paths: a server can associate its channel with a unique path name, which is registered and handled by the path manager (see Section 3.2). The client can then call `open()` on that path, which performs the following actions:

1. Sends a message to the path manager to inquire about the path. The path manager responds with the process and channel identifiers for the corresponding server channel.
2. Calls `ConnectAttach()` using the provided identifiers.

The resulting connection identifier is the QNX implementation of a *file descriptor*.

The following program demonstrates a client connecting to the GPIO server, and then using messages to control a GPIO. Rebuild the circuit from Section 4.2 to see the program turning an LED on and off.

led_client.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <fcntl.h>
4  #include <sys/neutrino.h>
5  #include <sys/rpi_gpio.h>
6
7  int
8  main(int argc, char **argv)
9  {
10     // Connect to the GPIO server.
11     int fd = open("/dev/gpio/msg", O_RDWR);
12     if (fd == -1) {
13         perror("open");
14         return EXIT_FAILURE;
15     }
16
17     // Configure GPIO 16 as an output.
18     rpi_gpio_msg_t msg = {
19         .hdr.type = _IO_MSG,
20         .hdr.subtype = RPI_GPIO_SET_SELECT,
21         .hdr.mgrid = RPI_GPIO_IOMGR,
22         .gpio = 16,
23         .value = RPI_GPIO_FUNC_OUT
24     };
25
26     if (MsgSend(fd, &msg, sizeof(msg), NULL, 0) == -1) {
27         perror("MsgSend(output)");
28         return EXIT_FAILURE;
29     }
30
31     // Set GPIO 16 to high.
32     msg.hdr.subtype = RPI_GPIO_WRITE;
33     msg.value = 1;
34
35     if (MsgSend(fd, &msg, sizeof(msg), NULL, 0) == -1) {
36         perror("MsgSend(high)");
37         return EXIT_FAILURE;
38     }
39
40     usleep(500000);
41
42     // Set GPIO 16 to low.
43     msg.value = 0;
44
45     if (MsgSend(fd, &msg, sizeof(msg), NULL, 0) == -1) {
46         perror("MsgSend(low)");
47         return EXIT_FAILURE;
48     }
49
50     return EXIT_SUCCESS;
51 }
```

Line 11 establishes the connection to the GPIO server. Line 18 declares a message structure with the type expected by this server. The first message, which is sent on line 26, uses the `RPI_GPIO_SET_SELECT` subtype and the value `RPI_GPIO_FUNC_OUT` to tell the server to configure GPIO 16 as an output. Lines 32-35 reuse the same message structure to tell the server to set GPIO 16 to high (note that the other structure fields are unchanged), while lines 43-45 use the same structure to set the GPIO to low.

It may seem from this example that messages must have a specific structure. The QNX kernel does not impose any such structure on messages, and treats these as raw

buffers to be copied from client to server and back. It is only the server that imposes semantics on these raw buffers, and each server can define its own expected structures for the various types of messages it handles.

As mentioned above, many of the standard library functions in QNX are implemented as message passes to the appropriate servers. The following example shows how to read a file directly with a message to the server that provides the file. When the call to `MsgSend()` returns, the read bytes are in the reply buffer. Of course, in this example there is no advantage to the direct use of `MsgSend()`, and code would normally call `read()` instead.

read_file.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/neutrino.h>
#include <sys/iomsg.h>

int
main(int argc, char **argv)
{
    if (argc < 2) {
        fprintf(stderr, "usage: %s PATH\n", argv[0]);
        return EXIT_FAILURE;
    }

    // Open the file.
    int fd = open(argv[1], O_RDONLY);
    if (fd == -1) {
        perror("open");
        return EXIT_FAILURE;
    }

    // Construct a read request message.
    struct _io_read msg = {
        .type = _IO_READ,
        .nbytes = 100
    };

    // Buffer to fill with reply.
    char buf[100];

    ssize_t nread = MsgSend(fd, &msg, sizeof(msg), buf, sizeof(buf));
    if (nread == -1) {
        perror("MsgSend");
        return EXIT_FAILURE;
    }

    printf("Read %zd bytes\n", nread);
    return EXIT_SUCCESS;
}
```

5.3 Threads and Priorities

5.3.1 What are Threads?

Every processor has a fixed set of registers, which store numeric values. One of these registers is the *instruction pointer* (also called *program counter*). When a processor

executes, it fetches the next instruction from memory based on the address stored in the instruction pointer, decodes it, updates registers according to the instruction (e.g., adds the values from two registers and stores the result in a third register), and then updates the instruction pointer such that it points to the next instruction. That next instruction is typically the one that follows the current instruction in memory, but it can also be in a different memory location if the processor has just executed a branch instruction.

The values stored in the processor's registers at any given point in time provide the processor with its *execution context*. We can think of the processor as executing a stream of instructions that updates this context as it goes along from one instruction to the next. Many simple micro-controllers only have just one such stream of execution, typically an infinite loop for the control of some device. The micro-controller follows that loop and does nothing else.

Computer systems that are not simple micro-controllers require multiple streams of execution. An example of the use of a separate execution stream is an exception: when the processor encounters an instruction such as a trap call, it jumps to a predefined location and executes the code in that location. It is expected that, once the exception is handled, the previous stream of execution resumes from where it left (though potentially the context is modified to reflect the results of exception handling). In order to resume the previous execution stream, the exception handler needs first to save, and then to restore, the execution context of the processor at the time the exception occurred.

A stream of execution that can be suspended and then restored is called a *thread*. In its most basic form, a thread provides storage for a processor's execution context. The system (typically the operating system kernel) copies the execution context from the processor to the thread's storage when a thread is suspended (such as when a processor jumps to an exception handler), and copies back the stored context to the processor when the thread is resumed.

The use of threads provides several important features in a computer system:

1. **Time Sharing** Different programs can make use of different threads in order to execute concurrently. The operating system alternates the execution of threads from all running programs, giving the illusion that the programs are running in parallel.[3]
2. **Processor Utilization** A program often needs to wait for some event to occur (e.g., a block read from a hard drive, user input, or a sensor detecting activity). Threads allow for a different execution stream to proceed while one is waiting, preventing the processor from going idle while the system has work to do.
3. **Non-Blocking Execution** Closely related to the previous point, a single program can make use of threads to turn a blocking operation into a non-blocking one. For example, a device that does not respond to a request until it is ready can

[3]Note that true parallelism is not possible on a single processor, but to a human observer fast-switching concurrency can be non-distinguishable from parallelism.

be interacted with using one thread, while another thread in the same program continues.

4. **Low Latency** Threads are crucial for a real-time operating system to provide timing guarantees on low-latency operations. When an event occurs, such as an interrupt raised from an external device, or a timer expiring, the current thread can be suspended, and a new one put into execution to handle the event. Such replacement of one thread by another in response to an event is called *preemption*. Without preemption event handling would have to wait until the current thread suspends itself, which may interfere with the timely response to the event.

> **Note**
>
> The discussion so far has made no reference to processes. Some textbooks refer to threads as light-weight processes, but that is an anachronism: old UNIX (and UNIX-like) systems had one stream of execution per process, and new processes had to be created for each of the use cases described above. The introduction of multiple such streams per process helped reduce the overhead in some of these cases, as the system no longer needed to allocate the full resources for a process just to have another stream of execution within the same logical program. But such a definition of a thread misses the essential point of this construct. Even in systems that do not support multiple threads per process there are still multiple threads, one for each process, and these threads are distinct entities. When the operating system scheduler puts a stream into execution, it schedules a thread, not a process.

5.3.2 Thread Scheduling

How does the system determine the next thread to execute? That is the task of the *scheduler*, which is a part of the operating system kernel.[4] The scheduler uses a scheduling policy to decide which, among all the threads that are currently ready for execution (i.e., threads that are not waiting for some event to occur and can proceed with their work), is the thread most eligible to run next, and load its stored context into the processor. Different operating systems provide different scheduling policies. Two common policies are:

- *FIFO* (first-in, first-out), in which the thread that has been ready for the longest time executes next, and then runs until it has to wait; and
- *round-robin*, which alternates between threads, allowing each a fixed time period to execute (its time slice).

Neither of these policies by itself is a good choice for a complex operating system. FIFO depends on a thread to relinquish the processor before any other activity can proceed, which means that other threads may have to wait indefinitely. Round-robin

[4] Putting threads into execution is the one task that even the smallest, purest microkernel has to do itself.

alleviates the problem only to a degree, but critical system work still has to wait for all the time slices of all the ready threads ahead of it.

QNX, like other real-time operating systems, supports *priority scheduling*. Each thread is associated with a priority value, which is a number between 0 (lowest) and 255 (highest). When a scheduling decision is performed, the scheduler chooses the thread with the highest priority that is ready to run. Within each priority level, threads are scheduled according to which has been ready for the longest time.

Priority 0 is reserved for the idle threads. These are the threads that run when the processor has nothing else to do. Typically these will execute the architecture-specific halt instruction, which can reduce the power consumption of the processor when it does not need to do anything. Priority 255 is reserved for the inter-processor interrupt (IPI) thread. IPIs are used in a multi-processor system to communicate between the different processors, and are required for distributing work. Additionally, one priority level is assigned to the clock interrupt handler, which handles timers. By default this priority is 254, but can be lowered with a command-line option to the kernel. If lowered, then the priorities between the clock thread and the IPI thread can be used for the lowest-latency threads, as long as these do not require software timers.

The assignment of priorities to threads is one of the most important tasks when designing a complete system. Clearly this cannot be a free-for-all, or all programs would assign the highest priority to all (or most) of their threads.[5] The following are some best practices when deciding on priorities:

- The threads with the highest priority in the system are those that require the shortest response time to events (i.e., the lowest latency). These are not (necessarily) the most *important* threads in the system, and are certainly not the most demanding in terms of throughput.
- The higher the priority of a thread, the shorter its execution time should be before it blocks waiting for the next event. High-priority threads running for long periods of time prevent the processor from doing anything else.
- The above restriction should be enforced by the system. In QNX the priority range is divided into a non-privileged range and a privileged range (by default 1-63 and 64-254). Only trusted programs should be given the ability to use privileged priorities. Also, the use of a software watchdog to detect a misbehaving high-priority thread is recommended.

The following example shows a program that creates 11 threads, in addition to the main thread. Ten of these threads are worker threads, which calculate the value of π, while another thread sleeps for one second and then prints the number of microseconds that have elapsed since the last time it woke up. Running this example should show that the high-priority thread executes consistently within one millisecond of the expected time.[6] Now comment out the call to `pthread_attr_setschedparam()` so

[5] Convincing programmers that the threads in the applications or drivers they work on should not have the highest priority in the system is one of the toughest jobs a system architect has to deal with.

[6] The one millisecond granularity is the result of the standard timer resolution. If needed, this example can be modified to use high-resolution timers.

that the high-priority thread is reduced to the default priority, and observe the effect.

threads.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <pthread.h>

static void *
worker(void *arg)
{
    double pi = 0.0;
    double denom = 1.0;

    for (unsigned i = 0; i < 100000000; i++) {
        pi += 4.0 / denom;
        pi -= 4.0 / (denom + 2.0);
        denom += 4.0;
    }

    printf("pi=%f\n", pi);
    return NULL;
}

static void *
high_priority(void *arg)
{
    unsigned long last = clock_gettime_mon_ns();
    for (;;) {
        sleep(1);
        unsigned long now = clock_gettime_mon_ns();
        printf("Slept for %luus\n", (now - last) / 1000UL);
        last = now;
    }

    return NULL;
}

int
main(int argc, char **argv)
{
    // Attribute structure for a high-priority thread.
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);

    struct sched_param sched = { .sched_priority = 63 };
    pthread_attr_setschedparam(&attr, &sched);

    // Create the high-priority thread.
    pthread_t tids[11];
    int rc;
    rc = pthread_create(&tids[0], &attr, high_priority, NULL);
    if (rc != 0) {
        fprintf(stderr, "pthread_create: %s\n", strerror(rc));
        return EXIT_FAILURE;
    }

    // Create worker threads.
    for (unsigned i = 1; i < 11; i++) {
        rc = pthread_create(&tids[i], NULL, worker, NULL);
        if (rc != 0) {
            fprintf(stderr, "pthread_create: %s\n", strerror(rc));
            return EXIT_FAILURE;
        }
    }

    // Wait for workers to finish.
    for (unsigned i = 1; i < 11; i++) {
        pthread_join(tids[i], NULL);
    }

    return EXIT_SUCCESS;
}
```

5.4 Timers

Keeping track of time is a requirement of many programs: refreshing the screen, retransmitting network packets, detecting deadlines and controlling devices with strict timing restrictions are all examples of such a requirement. Given that hardware timers are few and demand is high, the operating system needs to multiplex software timers on top of those provided by the hardware. A program can create multiple timers and associate each of those with an event to be emitted whenever that timer expires. The program can then ask the system, individually for each of the timers it created, to expire the timer at some point in the future.

The QNX RTOS (as of version 8.0) makes use of a per-processor hardware timer. When a program requests that one of its software timers expire in the future, that request is enqueued for the hardware timer associated with the processor on which the request was made. When the expiration time is reached, that hardware timer raises an interrupt, which is handled by the clock interrupt thread on the same processor. That thread then emits the event associated with the software timer, which notifies the program that the timer has expired.

Timers are created with the `timer_create()` function, which takes the clock ID and an event to be associated with the timer. The clock ID represents the underlying clock to which expiration times for the timer correspond. Two common clocks are `CLOCK_MONOTONIC` and `CLOCK_REALTIME`. Both of these are tied to the hardware clock, which ticks constantly, but the latter is affected by changes to the system's notion of the time of day, while the former is not. Another option for a clock is the runtime of a thread or a process (the latter is the accumulation of thread runtime for all threads in the process). This option is less common, and is typically used to monitor thread execution. The event associated with the timer can be any of the types that are allowed by the `sigevent` structure, including a pulse, a signal, posting a semaphore, creating a thread[7] or updating a memory location.

The standard timer resolution in a QNX system is the tick length, which is by default one millisecond. Software timers that use this resolution are referred to as *tick-aligned timers*. This means that these timers will expire on the next logical tick after their real expiration time. Aligning timers on a logical tick avoids excessive context switches as multiple timers expire in close succession.

> **Note**
>
> As of QNX 8.0, the system is tickless by default, which means that there is no recurring tick if no timer expires within the next tick period. Standard timers are still aligned to a *logical* tick, which is a multiple of the number of clock cycles in a tick period since the system's boot time.

The system also provides *high-resolution timers*, which expire as soon as the specified time was reached (subject to the granularity of the hardware timer). High-resolution

[7] Provided for POSIX compatibility. Strongly discouraged, especially when used with timers.

timers should be used sparingly[8] and with care, as frequent expiration of such timers can prevent the system from making forward progress. For this reason the creation of such timers is a privileged operation.

Once a timer has been created it can be programmed using `timer_settime()`. The function can be used to set the expiration time in either absolute or relative terms (both correspond to the clock used to create the timer), and as either a single-shot or a recurring timer. An example of using a timer will be given in the next section.

5.5 Event Loops

Real-time operating systems are often used to run programs for controlling external devices, such as robots, factory machinery or irrigation systems. All but the most simple devices provide multiple inputs, to which the control program needs to respond. One way to handle such multiple inputs is to have a separate thread for each one. A common alternative is to write the control program around an event loop: each input provides its own event, which the program handles in turn. These two approaches can also be combined in various ways. For example, a control program can have a pool of threads, each running an event loop.

An event loop consists of three phases:

1. Wait for any event to occur.
2. Decode the event.
3. Handle the event.

In C code, an event loop will have the following structure:

```
void
event_loop()
{
    event_t event;

    for (;;) {
        wait_for_event(&event);
        switch (event_type(event)) {
        case EVENT_TYPE_1:
            handle_event_1(event);
            break;

        case EVENT_TYPE_2:
            handle_event_2(event);
            break;

        ...
        }
    }
}
```

(This example does not reflect a real API, but simply illustrates the structure common to various event loop implementations. The switch statement may be replaced by

[8] There are only a few scenarios in which such timers are needed. As one safety expert at QNX is fond of saying, even the fastest car travels very little distance in one millisecond.

if-else blocks, or by a table of function pointers, but such choices do not alter the fundamental structure.)

Various operating systems have come up with different types of API data structures and functions for implementing event loops. In UNIX and UNIX-like operating systems the two common interface functions have traditionally been `select()`, and `poll()`. These functions have been recognized as deficient for a long time, both in terms of scalability and race conditions. Systems such as Linux and FreeBSD attempt to overcome their inherent limitations with less-standard approaches, such as `epoll` and `kqueue`, respectively.

While `select()` and `poll()` are available from the QNX C library, the inherent limitations of these functions make them less than ideal choices for implementing event loops. A much better choice is to build the loop around a call to `MsgReceivePulse()` and use pulses as the event notification mechanism. A *pulse* is a fixed-sized structure with a small payload (a one-byte code and an 8-byte value) that can be emitted in response to various events, including, but not limited to, an interrupt firing, a timer expiring, a pipe changing its state from empty to non empty, and a socket being ready to deliver the next packet. The program that implements the event loop requests to be notified via pulses by associating a pulse `sigevent` structure with each event source. We have seen in the previous section that a `sigevent` can be associated with a timer when that timer is created. Other functions that take such structures include `InterruptAttachEvent()`, `ionotify()` and `mq_notify()`. Various servers accept `sigevent` structures embedded in messages, which allow them to deliver these events when the conditions for delivery are met.

> **Note**
>
> As of version 7.1 of the QNX OS, `sigevent` structures passed to servers must first be registered with calls to `MsgRegisterEvent()`. Registration prevents server processes from delivering unexpected (or even malicious) events to clients. Events used with timers do not have to be registered, as these are delivered by the microkernel. Nevertheless, it is good practice to do so, and the example below does.

Pulses are queued for delivery by priority first and order of arrival second. Using priorities allows certain events to be handled first, resulting in lower latency for those events.

For the next example build the circuit depicted in Figure 5.1.

The program sets three event sources, one for each button and a timer that fires 5 seconds after the last event that was received. Each event is assigned a different pulse value to distinguish it. The timer event is used with a call to `timer_create()`. The button events are passed to the GPIO resource manager to be delivered by that server whenever a rising edge is detected on the respective GPIO.

The pulses are delivered to a local channel. As the channel is only used for pulse

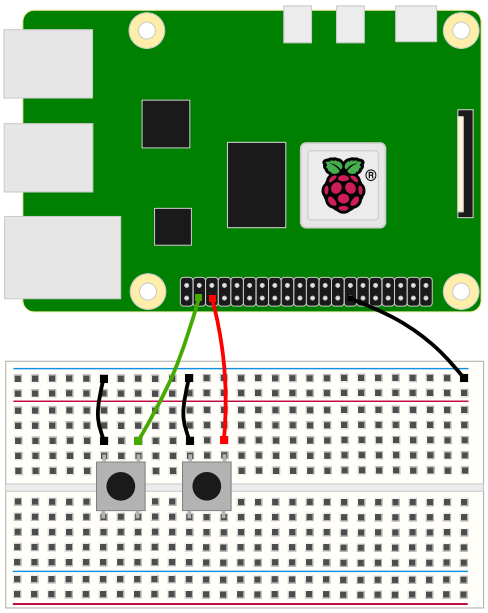


Figure 5.1: A circuit with two push buttons

delivery within the process, the channel can be, and should be, declared as private. This prevents other processes from connecting to this channel and sending messages or pulses.

The program creates a connection to the channel, which is then used by the system to deliver pulses. Note that while the `SIGEV_PULSE_INIT()` macro takes the connection ID used to deliver the pulse, `MsgRegisterEvent()` takes a connection ID that identifies the server that is allowed to emit the event. In the case of a timer the event is coming from the kernel, but in the case of the buttons the events are delivered by the GPIO resource manager. Any other process trying to deliver this event to the process will be prevented from doing so.

event_loop.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <fcntl.h>
#include <sys/neutrino.h>
#include <sys/rpi_gpio.h>

enum {
    EVENT_TIMER,
    EVENT_BUTTON_1,
    EVENT_BUTTON_2
};

static int chid;
static int coid;
static timer_t timer_id;

static bool
init_channel(void)
{
    // Create a local channel.
    chid = ChannelCreate(_NTO_CHF_PRIVATE);
    if (chid == -1) {
        perror("ChannelCreate");
        return false;
    }

    // Connect to the channel.
    coid = ConnectAttach(0, 0, chid, _NTO_SIDE_CHANNEL, 0);
    if (coid == -1) {
        perror("ConnectAttach");
        return false;
    }

    return true;
}

static bool
init_timer(void)
{
    // Define and register a pulse event.
    struct sigevent timer_event;
    SIGEV_PULSE_INIT(&timer_event, coid, -1, _PULSE_CODE_MINAVAIL,
                     EVENT_TIMER);
    if (MsgRegisterEvent(&timer_event, coid) == -1) {
        perror("MsgRegisterEvent");
        return false;
    }

    // Create a timer and associate it with the first pulse event.
    if (timer_create(CLOCK_MONOTONIC, &timer_event, &timer_id) == -1) {
        perror("timer_create");
        return false;
    }

    return true;
}
```

```c
static bool
init_gpio(int fd, int gpio, int event)
{
    // Configure as an input.
    rpi_gpio_msg_t msg = {
        .hdr.type = _IO_MSG,
        .hdr.subtype = RPI_GPIO_SET_SELECT,
        .hdr.mgrid = RPI_GPIO_IOMGR,
        .gpio = gpio,
        .value = RPI_GPIO_FUNC_IN
    };

    if (MsgSend(fd, &msg, sizeof(msg), NULL, 0) == -1) {
        perror("MsgSend(input)");
        return false;
    }

    // Configure pull up.
    msg.hdr.subtype = RPI_GPIO_PUD;
    msg.value = RPI_GPIO_PUD_UP;
    if (MsgSend(fd, &msg, sizeof(msg), NULL, 0) == -1) {
        perror("MsgSend(pud)");
        return false;
    }

    // Notify on a rising edge.
    rpi_gpio_event_t event_msg = {
        .hdr.type = _IO_MSG,
        .hdr.subtype = RPI_GPIO_ADD_EVENT,
        .hdr.mgrid = RPI_GPIO_IOMGR,
        .gpio = gpio,
        .detect = RPI_EVENT_EDGE_RISING,
    };

    SIGEV_PULSE_INIT(&event_msg.event, coid, -1, _PULSE_CODE_MINAVAIL,
                     event);
    if (MsgRegisterEvent(&event_msg.event, fd) == -1) {
        perror("MsgRegisterEvent");
        return false;
    }

    if (MsgSend(fd, &event_msg, sizeof(event_msg), NULL, 0) == -1) {
        perror("MsgSend(event)");
        return false;
    }

    return true;
}
```

```c
int
main(int argc, char **argv)
{
    if (!init_channel()) {
        return EXIT_FAILURE;
    }

    if (!init_timer()) {
        return EXIT_FAILURE;
    }

    int fd = open("/dev/gpio/msg", O_RDWR);
    if (fd == -1) {
        perror("open");
        return false;
    }

    if (!init_gpio(fd, 16, EVENT_BUTTON_1)) {
        return EXIT_FAILURE;
    }

    if (!init_gpio(fd, 20, EVENT_BUTTON_2)) {
        return EXIT_FAILURE;
    }

    struct itimerspec ts = { .it_value.tv_sec = 5 };
    timer_settime(timer_id, 0, &ts, NULL);

    for (;;) {
        struct _pulse pulse;
        if (MsgReceivePulse(chid, &pulse, sizeof(pulse), NULL) == -1) {
            perror("MsgReceivePulse()");
            return EXIT_FAILURE;
        }

        if (pulse.code != _PULSE_CODE_MINAVAIL) {
            fprintf(stderr, "Unexpected pulse code %d\n", pulse.code);
            return EXIT_FAILURE;
        }

        switch (pulse.value.sival_int) {
        case EVENT_TIMER:
            printf("Press a button already!\n");
            break;

        case EVENT_BUTTON_1:
            printf("Thank you for pressing button 1!\n");
            break;

        case EVENT_BUTTON_2:
            printf("Thank you for pressing button 2!\n");
            break;
        }

        timer_settime(timer_id, 0, &ts, NULL);
    }

    return EXIT_SUCCESS;
}
```

5.6 Controlling Hardware

We have already seen many examples of how to control external devices by communicating with the GPIO and I2C resource managers. Let us now examine how these resource managers, as well as other components such as the block device drivers used by the file system, or the network drivers, interact with the underlying hardware.

While every peripheral device has a unique way in which it is initialized and controlled (with manuals that sometimes reach thousands of pages), they all share common ways for software to interact with the device: memory-mapped registers, interrupts and DMA buffers. With the exception of some legacy devices on x86 systems, gone are the days of I/O ports.

The first thing a program implementing a device driver does is to map the device's control registers. Mapping requires knowing the physical address of the device and having permissions to do so. For most devices on the Raspberry Pi 4 these addresses are fixed and listed in the datasheet.[9] Things are more complicated when it comes to PCI devices, where the addresses can be determined only at boot time through PCI enumeration. Note that the addresses assigned to the devices do not correspond to parts of the system's memory (RAM). Device addresses are always distinct from the address ranges assigned to RAM.

> ⚠ Warning
>
> Mapping physical memory is an extremely dangerous operation. Using the wrong addresses creates bugs that are both critical and hard to find. Properly configured systems restrict every process that requires such mappings to just the physical addresses it needs (via the `PROCMGR_AID_MEM_PHYS` ability). An even better option is for the system to add named ranges to typed memory at startup, and then have the device drivers map these.

In the next exercise we will map the GPIO control registers and use those directly to change the state of a pin. First, rebuild the simple LED circuit from Section 4.2. Compile the following program.

[9] https://datasheets.raspberrypi.com/bcm2711/bcm2711-peripherals.pdf

gpio_map.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdint.h>
4  #include <unistd.h>
5  #include <sys/mman.h>
6  #include <aarch64/rpi_gpio.h>
7
8  static uint32_t volatile *gpio_regs;
9
10 int
11 main(int argc, char **argv)
12 {
13     // Map the GPIO registers.
14     gpio_regs = mmap(0, __PAGESIZE,
15                      PROT_READ | PROT_WRITE | PROT_NOCACHE,
16                      MAP_SHARED | MAP_PHYS, -1, 0xfe200000);
17     if (gpio_regs == MAP_FAILED) {
18         perror("mmap");
19         return EXIT_FAILURE;
20     }
21
22     // Configure GPIO 16 as an output.
23     gpio_regs[RPI_GPIO_REG_GPFSEL1] &= ~(7 << 18);
24     gpio_regs[RPI_GPIO_REG_GPFSEL1] |= (1 << 18);
25
26     int led_state = 0;
27     for (;;) {
28         if (led_state == 0) {
29             // Set GPIO 16 to high.
30             gpio_regs[RPI_GPIO_REG_GPSET0] = (1 << 16);
31         } else {
32             // Set GPIO 16 to low.
33             gpio_regs[RPI_GPIO_REG_GPCLR0] = (1 << 16);
34         }
35
36         led_state = !led_state;
37         sleep(1);
38     }
39
40     return EXIT_SUCCESS;
41 }
```

Note that you will need to run this program as root, as a normal user does not have the necessary privileges to map physical memory:

```
qnxuser@qnxpi:~$ su -c ./gpio_map
```

Lines 14-15 map the registers at physical address 0xfe200000 and then assign these to an array of 32-bit integers called `gpio_regs` (the control registers are 32-bit and need to be accessed as such). Note the use of `PROT_NOCACHE`: if the virtual address assigned by the memory manager for this mapping is marked as cached then reads and writes will not propagate immediately to the registers.[10] Also note the use of the `MAP_PHYS` flag, which tells the memory manager that the last argument is actually a physical address rather than an offset for the underlying object. Finally, `MAP_SHARED` is crucial here: a private mapping would create a copy of the register contents in

[10]Since the physical address is not part of RAM, the use of `PROT_NOCACHE` results in strongly-ordered device memory (or nGnRnE, in AArch64 terminology), which also ensures that writes are not reordered.

memory rather than back the mapping with the registers themselves.

Lines 22-23 configure GPIO 16 as an output, first by clearing bits 18-20 in the GPFSEL1 register and then setting these bits of 0b001. Lines 27-33 set GPIO 16 to high by setting bit 16 in GPSET0 or to low by setting bit 16 in GPCLR0. These registers are described in GPIO chapter of the datasheet.

5.7 Handling Interrupts

5.7.1 What is an Interrupt?

An interrupt is an asynchronous notification delivered to the processor outside of its current stream of execution. Interrupts are typically used by devices (both internal and external to the processor) to inform the processor, and through it any software interested in such events, that something has happened that requires attention. For example, a timer generates an interrupt when its counter reaches some value. A network device may generate an interrupt when a new packet arrives, or a serial device when its buffer has room for more data.

The main benefit of interrupts is that they allow software handling devices to avoid constantly checking whether an event has happened. Such checking, known as *polling*, can waste processing time if performed too often and the device does not require any handling, or can cause latency if not performed often enough as devices are left unattended.

An event that causes the processor to switch away from its current execution stream is called an *exception*. It is important to differentiate between synchronous and asynchronous exceptions, with an interrupt being of the latter type. A synchronous exception occurs as a direct result of the last instruction executed by the processor. Examples of synchronous exceptions include kernel calls, page faults, and floating point processing errors. Interrupts, on the other hand, are generated by the devices irrespective of current instructions, though of course software influences the generation of interrupts: a timer device needs to be programmed by software and its interrupt capability enabled before it can generate an interrupt. There are other types of asynchronous exceptions, though they are less common, such as bus errors observed some time after the responsible instruction was executed.

Typically, different devices generate different interrupts, so that when an interrupt is delivered to the processor, software can determine which device it came from and allow the relevant device driver to handle the interrupt. Things are more complicated if the same interrupt is shared by multiple devices. Nevertheless, modern hardware typically allows for enough interrupts to service a large number of devices, and even different functions within the same device.

From a hardware point of view, an interrupt involves three components:

1. the device that generates the interrupt (a timer, a serial device, a network card, a graphics processor, etc.);

2. one or more interrupt controllers, connected both to the device and to the processor;
3. the processor.

In a multi-processor system, the interrupt controller can be configured to deliver specific interrupts to specific processors, or (depending on the controller) to a subset of the processors, though a single processor is the usual case.

The following conditions must be met for an interrupt to be seen by the processor:

1. the device is configured to generate an interrupt upon meeting some condition;
2. the condition is met;
3. the specific interrupt is enabled in the interrupt controller;
4. the processor to which the interrupt is routed allows for interrupts to be delivered.

When referring to a specific interrupt being disabled or enabled in the interrupt controller we say that the interrupt is *masked* or *unmasked*, respectively. For a processor, we talk about interrupts (in general) being enabled or disabled.

5.7.2 Processing an Interrupt

Any software that supports interrupts needs to follow a protocol with the three hardware components that are involved in interrupt generation. In the following description the term *device driver* will be used for the software that manages a specific device, while the term *kernel* will be used for the software that handles the processor's switch from running the normal instruction stream to servicing an interrupt. Note, however, that different systems can compose these components in different ways. In a monolithic kernel, the device driver is part of the kernel (perhaps loaded as a kernel module). In a microkernel system, such as QNX, the device driver is typically a stand-alone user process. In a simple executive there is no real separate kernel, but the system still needs to provide dedicated code for the processor to execute when it switches to servicing an interrupt.

As mentioned above, the device driver must first configure the device to generate an interrupt when various events of interest occur. A device driver may choose not to configure the device for interrupts for some, or any, of the events associated with the device, and resort to polling instead.

When an interrupt is generated by the device and the interrupt controller notices it, the controller pokes the processor. Assuming the processor has interrupts enabled, it immediately jumps to a pre-defined code address, where it expects the kernel to have the code for handling interrupts. Typically the kernel will store the state of the processor that reflects the current thread executing, so that it can be resumed later.

Next, the kernel queries the interrupt controller to find out which specific interrupt caused the processor to be poked (in some simple systems, this may already be known by the address to which the processor jumped, but modern hardware has too many interrupt sources to provide a different entry point for each). Once the interrupt source

has been determined the kernel can notify the device driver so that it can take the necessary action.

When the device driver is done with the interrupt, it lets the interrupt controller know, so that a new interrupt can be generated once the next event occurs. Note that the specific interrupt is blocked between the time the processor jumps to the kernel's entry routine and the time the device driver is done handling it, to prevent a never-ending sequence of jumps to the entry routine. Such blocking can occur either at the processor level (in which case all interrupts to that processor are blocked), or at the controller level (which allows for the masking of a specific interrupt).

5.7.3 Handling an Interrupt in the QNX RTOS

Most interrupts in a QNX-based system are handled by device drivers that are stand-alone user-mode processes. Exceptions to these include inter-process interrupts (IPIs) and a per-processor clock interrupt, which are handled by the kernel.

Each interrupt to be handled is associated with a thread, referred to as the *Interrupt Service Thread*, or IST. This is true even for the interrupts handled by the kernel, and you can see the IPI and clock ISTs for each core listed by **pidin** when looking at the kernel process.

As mentioned in the previous section, the device driver must configure the device to generate interrupts before it can start handling them. How the device is configured is specific to each device and is covered by the hardware manual. The routing of the device's interrupts to different processors is not up to the device driver (as it is not up to the device), and is configured by the **startup** executable that is part of the board-support package for the specific board. The **startup** program also configures a global source ID for every interrupt in the system, which the device driver needs to know in order to attach to the right interrupt. The association of these source ID values to interrupts is provided in the documentation for the board-support package.

The device driver may choose between two methods for being notified about interrupts. The first is to create a thread dedicated to the handling of the interrupt (an IST). That thread calls `InterruptAttachThread()` with the source ID of the relevant interrupt, which creates an association between that interrupt and the thread in the kernel. The thread can now invoke the `InterruptWait()` call to block until the interrupt is delivered. When the call returns (without an error) the interrupt is masked in the controller. It is up to the device driver to unmask the interrupt once it has handled it and is ready to accept new interrupts. Unmasking can be done with a call to `InterruptUnmask()`. However, as the thread will typically employ a loop around `InterruptWait()` that needs to unmask right before the call, there is a flag `_NTO_INTR_WAIT_FLAGS_UNMASK` to `InterruptWait()` to cause it to unmask the interrupt and then block, which saves on the number of kernel calls required to handle an interrupt.

The following code snippet shows the typical structure of an interrupt handling loop in an IST:

```
int id = InterruptAttachThread(interrupt_number,
                               _NTO_INTR_FLAGS_NO_UNMASK);

for (;;) {
    if (InterruptWait(_NTO_INTR_WAIT_FLAGS_UNMASK, NULL) != -1) {
        // Service interrupt
    } else {
        // Handle errors
    }
}
```

Notes:

- The interrupt number passed to `InterruptAttachThread()` is the global interrupt source identifier as provided by the BSP documentation. The value returned from the call is a process-specific identifier that can then be passed to calls to mask, unmask or detach an interrupt.
- Since the loop above uses the shortcut flag for unmasking an interrupt before blocking, the call to `InterruptAttachThread()` must not itself unmask the interrupt. The kernel keeps track of how many times an interrupt is masked and unmasked, and it is an error to unmask too many times.
- Each thread can attach to at most one interrupt. It is possible to attach multiple threads to the same interrupt. When the interrupt is delivered, the kernel notifies all the threads attached to that interrupt. However, attaching multiple threads to the same interrupt is rarely useful, and can be a source of trouble, as the controller is only told to unmask the interrupt once all threads are done handling it and unmask the interrupt. If one thread does not do that, or if it takes a long time to handle the interrupt, the other threads will not be able to handle new events from the device.
- The IST is a regular thread. Other than the requirement to unmask the interrupt when it is done there are no special restrictions imposed upon it. The IST can invoke any kernel call or any library routine. An IST that takes a long time to service an interrupt affects only that particular device and not the system as a whole.
- The flag `_NTO_INTR_WAIT_FLAGS_FAST` can be used in a call to `InterruptWait()` to reduce the overhead of the kernel call. The downside of this flag is that it cannot be used in combination with a timeout on the blocking call. If a device driver does not need to enforce a timeout on the call then this flag provides lower latency.
- The latency of handling a specific interrupt is determined by the priority and processor affinity of the IST. As a general rule the IST should have its processor affinity set to the same processor to which the interrupt is routed. The higher the thread's priority the lower the latency, though a high priority also requires shorter work bursts from the thread to reduce the impact on the system.

A second method for handling interrupts is by attaching an action represented by a signal event to the interrupt. The device driver then waits for the event and processes the interrupt. The actions associated with signal events include delivering pulses to a channel, emitting a signal, posting a semaphore, creating a thread and changing a

memory value. In practice, however, only pulse and semaphore events should be associated with interrupts. A call to `InterruptAttachEvent()` associates an interrupt number with a signal event.

Earlier we said that each interrupt handled by the system must be associated with a thread. This is true for the case of attaching an event to an interrupt. A call to `InterruptAttachEvent()` creates a thread that itself calls `InterruptAttachThread()` and then runs a loop that dispatches the requested event whenever `InterruptWait()` returns. It should be clear that using events with interrupts does not provide any functionality that using ISTs cannot, and the latter provides lower overhead and better control over interrupt handling. The sole purpose of the event API is to allow for easy migration of code from previous versions of the QNX RTOS that did not provide the thread-attaching API.

5.7.4 What about ISRs?

People familiar with earlier versions of the QNX RTOS, or with other operating systems, may wonder at this point about the absence of any mention above of interrupt service routines (ISRs). An ISR is code for handling an interrupt that runs in the context of the kernel routine that is jumped to by the processor when the interrupt is delivered. In a monolithic kernel, an ISR is simply a function registered by the device driver and invoked by the kernel. In a microkernel, the ISR may be some form of byte code interpreted by the kernel, or, as in the case of previous versions of the QNX RTOS, a function pointer in the user-mode driver process that the kernel invokes after switching to the address space of the process.

ISRs provide very good interrupt latency, but do so at the expense of severe limitations, as well as safety and security risks. An ISR must be kept very short. It cannot invoke any kernel call, which means that synchronization with other parts of the driver cannot use blocking objects such as mutexes and semaphores. Any bug or exploit in the ISR can compromise the entire system, as the code is executed with the full privileges of the kernel. Moreover, any bug or exploit in the driver can lead to a bug or exploit in the ISR. For all these reasons ISRs were removed from (or, more accurately, were not designed for) the QNX kernel as of version 8.0 of the operating system.

5.7.5 Example

For the next example, rebuild the button and LED circuit described in Section 4.3. In order to run the program, you will first need to kill the GPIO resource manager, which itself attaches to the GPIO interrupt. While it is possible for more than one process to attach to the same interrupt, the result is often interference, and in fact the resource manager will reset the hardware's event registers before our example program sees the interrupt.

```
qnxuser@qnxpi:~$ su -c slay rpi_gpio
qnxuser@qnxpi:~$ su -c ./rpi_interrupt
```

gpio_interrupt.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/neutrino.h>
#include <aarch64/rpi_gpio.h>

static uint32_t volatile *gpio_regs;

static bool
init_gpios()
{
    // Map the GPIO registers.
    gpio_regs = mmap(0, __PAGESIZE,
                     PROT_READ | PROT_WRITE | PROT_NOCACHE,
                     MAP_SHARED | MAP_PHYS, -1, 0xfe200000);
    if (gpio_regs == MAP_FAILED) {
        perror("mmap");
        return false;
    }

    // Configure GPIO 16 as an output.
    gpio_regs[RPI_GPIO_REG_GPFSEL1] &= ~(7 << 18);
    gpio_regs[RPI_GPIO_REG_GPFSEL1] |= (1 << 18);

    // Configure GPIO 20 as a pull-up input.
    gpio_regs[RPI_GPIO_REG_GPFSEL2] &= ~(7 << 0);
    gpio_regs[RPI_GPIO_REG_GPPUD1] &= ~(3 << 8);
    gpio_regs[RPI_GPIO_REG_GPPUD1] |= (1 << 8);

    // Clear and disable all events.
    gpio_regs[RPI_GPIO_REG_GPEDS0] = 0xffffffff;
    gpio_regs[RPI_GPIO_REG_GPEDS1] = 0xffffffff;
    gpio_regs[RPI_GPIO_REG_GPREN0] = 0;
    gpio_regs[RPI_GPIO_REG_GPREN1] = 0;
    gpio_regs[RPI_GPIO_REG_GPFEN0] = 0;
    gpio_regs[RPI_GPIO_REG_GPFEN1] = 0;
    gpio_regs[RPI_GPIO_REG_GPHEN0] = 0;
    gpio_regs[RPI_GPIO_REG_GPHEN1] = 0;
    gpio_regs[RPI_GPIO_REG_GPLEN0] = 0;
    gpio_regs[RPI_GPIO_REG_GPLEN1] = 0;

    // Detect falling and rising edge events on GPIO 20.
    gpio_regs[RPI_GPIO_REG_GPREN0] |= (1 << 20);
    gpio_regs[RPI_GPIO_REG_GPFEN0] |= (1 << 20);

    return true;
}
```

```c
int
main(int argc, char **argv)
{
    if (!init_gpios()) {
        return EXIT_FAILURE;
    }

    // Attach to the GPIO interrupt without unmasking.
    int intid = InterruptAttachThread(145, _NTO_INTR_FLAGS_NO_UNMASK);
    if (intid == -1) {
        perror("InterruptAttachThread");
        return EXIT_FAILURE;
    }

    for (;;) {
        // Unmask the interrupt and wait for the next one.
        if (InterruptWait(_NTO_INTR_WAIT_FLAGS_FAST |
                          _NTO_INTR_WAIT_FLAGS_UNMASK, NULL) == -1) {
            perror("InterruptWait");
            return EXIT_FAILURE;
        }

        // Check for an event on GPIO 20.
        if ((gpio_regs[RPI_GPIO_REG_GPEDS0] & (1 << 20)) != 0) {
            if ((gpio_regs[RPI_GPIO_REG_GPLEV0] & (1 << 20)) == 0) {
                // GPIO 20 is low, set GPIO 16 to high.
                gpio_regs[RPI_GPIO_REG_GPSET0] = (1 << 16);
            } else {
                // GPIO 20 is high, set GPIO 16 to high.
                gpio_regs[RPI_GPIO_REG_GPCLR0] = (1 << 16);
            }
        }

        // Clear any detected events before unmasking the interrupt.
        gpio_regs[RPI_GPIO_REG_GPEDS0] = 0xffffffff;
        gpio_regs[RPI_GPIO_REG_GPEDS1] = 0xffffffff;
    }

    return EXIT_SUCCESS;
}
```

The example attaches interrupt 145, which is the GPIO interrupt on Raspberry Pi 4, to the main thread. While it is common to have a dedicated IST in a program, this example is simple enough to have the main thread service the interrupt. After configuring the GPIOs such that the hardware detects both rising and falling edges on GPIO 20 (the button), the program goes into an infinite loop, waiting for the interrupt. Once the call to `InteruptWait()` returns, indicating that interrupt 145 has fired, the code checks that the reason for the interrupt was indeed a change to GPIO 20. If so, it reads the current value of the GPIO, and updates the output on GPIO 16 (the LED) accordingly. It is important to reset the event registers before going back to `InterruptWait()`, to prevent the interrupt from firing again immediately.

Chapter 6

The GPIO Header

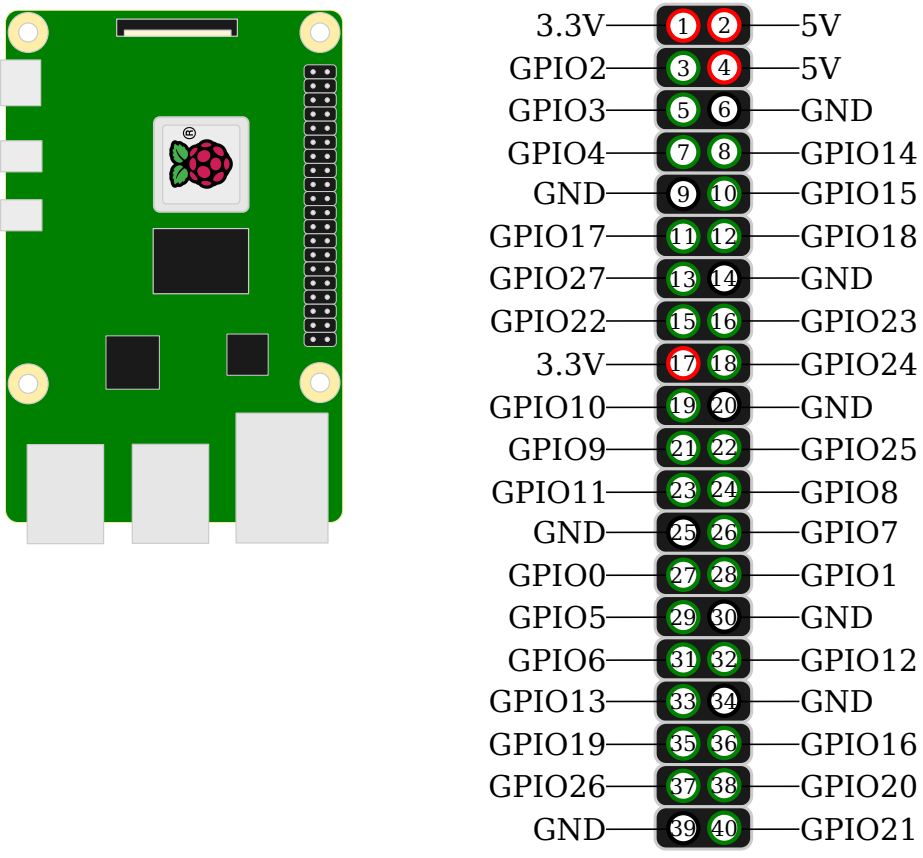


Figure 6.1: GPIO Layout on the Raspberry Pi