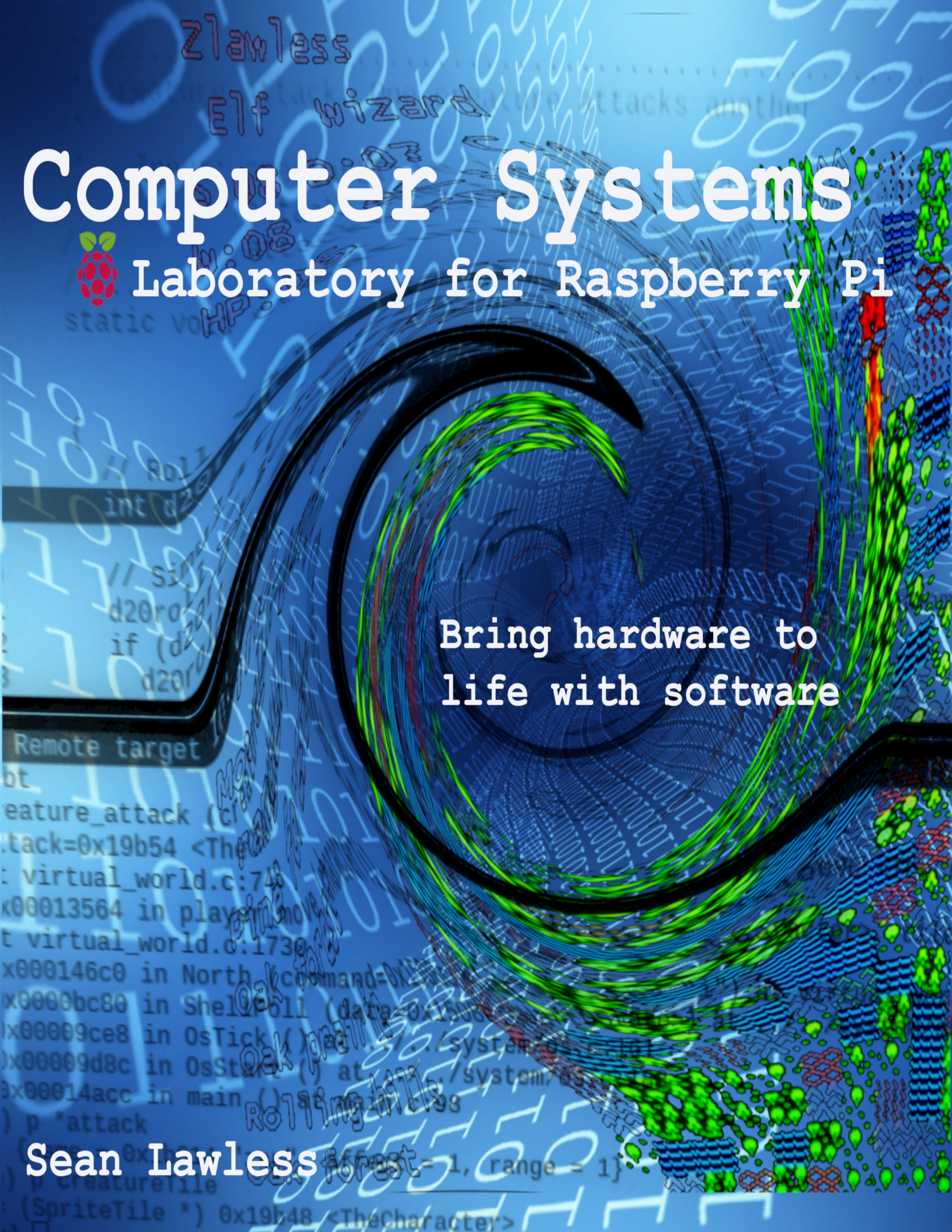
Computer Systems

Laboratory for Raspberry Pi

Bring hardware to life with software

Sean Lawless



Computer Systems Laboratory

for Raspberry Pi

Sean Lawless

This book is for sale at http://leanpub.com/computersystems_lab_rpi

This version was published on 2019-08-27

Contents

Lab 1: Raspberry Pi

1.0 Disclaimer

Raspberry Pi is a trademark of the Raspberry Pi Foundation. All attempts were made to comply with the Raspberry Pi Marks copyright agreement when using the name Raspberry Pi and the Raspberry Pi logo. Reference: https://www.raspberrypi.org/trademark-rules/

1.1 Why the Raspberry Pi?

The Raspberry Pi platform, or RPi for short, is unique in many ways. A low cost combined with video capabilities, it can become a viable development environment when running a Linux OS. At the same time, its simplicity and documentation make it ideal for systems engineering projects. All that is needed for a complete systems engineering experience is two Raspberry Pi's, one running a Linux OS and used to edit, create and debug the software that executes on a second RPi. The GPIO pin out is also very helpful, eliminating the need for any soldering or third party hardware in order to connect the two RPi's.

Not to be overlooked for the beginning systems software creator however, is the removable SD card that contains the binary image that the RPi will automatically execute upon start up. Having the boot image on a removable SD card greatly simplifies initial system engineering development. Those new to software development can take their time to build up to leaning how to move a compiled binary to another hardware system for execution. With the RPi it is possible to move the SD card to the PC used to create the executable binary and copy the file in a way familiar to most computer users. On RPi power up a file on the SD card is executed automatically. To execute our software creations, we rename the original file (Linux kernel) and copy our own creation with a matching name, and then power up the RPi to execute our creation. This makes the Raspberry Pi series of laboratory assignments the simplest for introductory students.

If not all of the previous discussion was understood, no worries. We will get to each in time. This chapter starts the journey by describing the details of RPi B+/2/3 hardware, identifying the CPU, RAM and peripherals as well as the external connections (USB ports, etc.). This is a general overview but by the end of this chapter the reader should know the names of the major peripherals on the motherboard as well as the locations of the RAM, CPU and peripheral connections.

All software created in this laboratory book, as well as in the companion main book, are available to the public at the authors GitHub page, https://github.com/sean-lawless/computersystems. This repository holds the latest source code, chapter by chapter, which are meant to be used to follow along with the chapters of this book.

1.2 Raspberry Pi Hardware

The Raspberry Pi 2/3 and B+ have mostly the same hardware except the CPU, GPU (Graphics Processing Unit) and RAM are upgraded in the RPi 2. The RPi 3 has a further upgraded CPU and RAM as well as a separate chip for WiFi and Bluetooth. Let us view a picture of the hardware now, with the majority of the peripherals named.

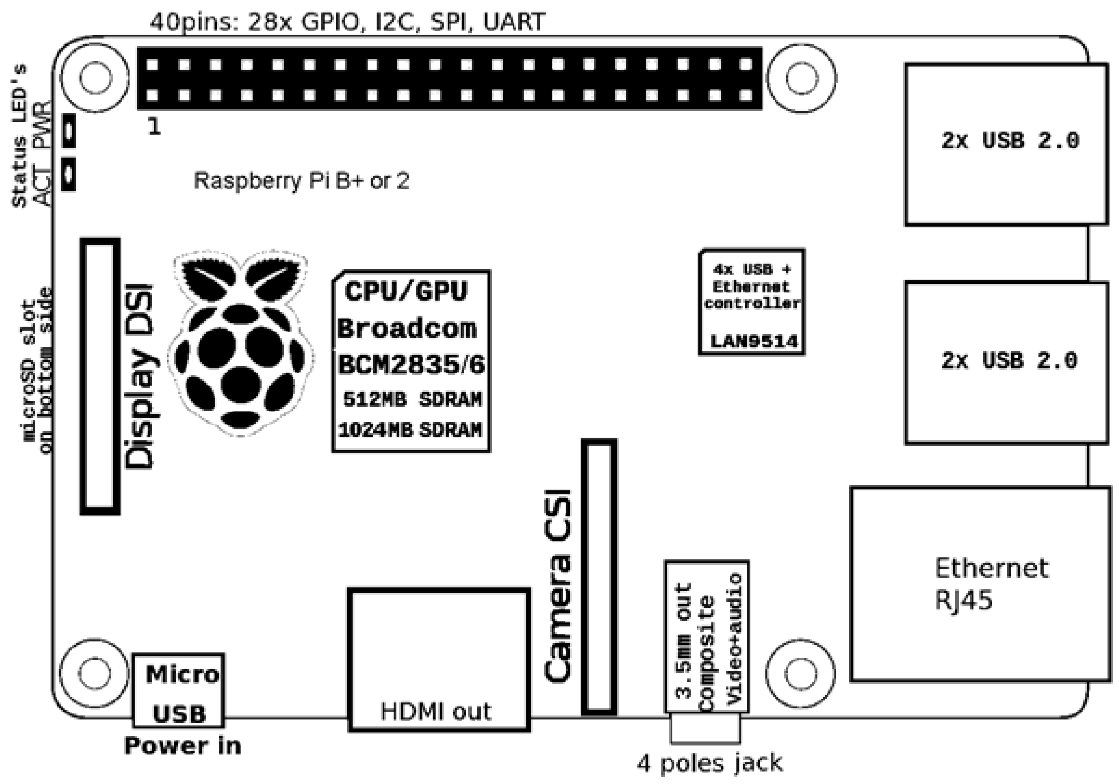

40pins: 28x GPIO, I2C, SPI, UART
Status LED's
ACT PWR
1
Raspberry Pi B+ or 2
microSD slot on bottom side
Display DSI
CPU/GPU Broadcom BCM2835/6 512MB SDRAM 1024MB SDRAM
4x USB + Ethernet controller LAN9514
2x USB 2.0
2x USB 2.0
Ethernet RJ45
Camera CSI
3.5mm out Composite Video+audio
4 poles jack
HDMI out
Micro USB
Power in

Fig 1: Raspberry Pi hardware

The core computer is contained within the CPU (Central Processing Unit), GPU (Graphics Processing Unit) and RAM (Random Access Memory) that is packaged as a single physical chip, known as a System on a Chip (SoC). The BCM2835 is the SoC for the RPi B+, the BCM2836 the SoC for the RPi 2, and the BCM2837 for the RPi 3. The power supply is a micro USB, allowing many USB chargers to power the RPi. Another important interface is the HDMI out, which can be connected to any monitor or TV with an HDMI in jack. Or the LCD display DSI port can be used with compatible LCD displays. The USB connections allow the user to add a keyboard and mouse and interact with the computer, and Ethernet connects to the Internet. An important connection to system engineering is the microSD slot on the left underside of the board. The microSD card slot must have a SD card connected containing a valid executable image (Linux kernel or otherwise) or no software will execute when the hardware is powered on. It is this SD card that we will copy our initial software creations onto in order to execute them.

The lab assignments incrementally build an understanding of the development process, not just the result. If questions arise, do not get distressed immediately but instead continue reading at least one chapter ahead. If the question still remains, reread and/or ask a knowledgeable peer, teacher or the Internet for more help. With incremental development, the next step often reinforces the understanding of the last.

This is a very concise introduction to systems engineering and the C language at the same time. They should both reinforce the other and to understand one requires understanding the other. However to the casual reader it will appear as if you have been thrown in the deep end of the pool. Remain calm with an open mind and you may surprise yourself to learn that it can be understood.

1.3 Protect the Pi

The next project is to protect the Raspberry Pi. If you have purchased or been provided a protective case, please follow the provided instructions for installing the Pi. For those with a bare system in a box, do not remove the board from the box until a case is available. It is simple enough to create a case out of cardboard, plastic or other sturdy material. This lab assignment is to use your imagination and create a case to protect your Pi, leaving holes to connect the USB, Ethernet, microSD and HDMI connections. It is important for the Pi hardware to be accessible as the case will need to be opened and expanded upon in the future as the reader advances chapters. Be sure to leave the RPi board in the static bag while measuring or sketching an outline of it.

The goal of this lab is to become creative and understand the case is for protection and style. Creating a case is empowering and adds individuality which increases the bond between the user and the computer. Drawing, painting and decorating are fun, but so is designing the case so that it keeps the board off the bottom for ventilation. For younger readers it may be appropriate to use stickers to decorate an existing case to avoid damaging the computer. The quickest solution is to use the box the RPI arrived and add elevating posts, snip open the ports and reinforce with tape.

Important

Adult supervision required. Do not paint or decorate the inside of the case as some paints and stickers, etc. are conductive and can short circuit the board. Never use metal, or even cardboard with a metallic surface, for the inside of the case. During initial operation, constantly observe and check the temperature of any new case that is flammable, such as cardboard or other organic material. Test how hot the RPi gets in the case and create airflow vents. Note that both sides of the motherboard require ventilation. Feet to lift the bottom of the motherboard above the bottom of the case is often required. Cut up sticky pads and use at the board corners, or design the case to keep the board up. A case that could hold the RPi up sideways often ventilates nicely.

Below is an outline for a box to be used as a Raspberry Pi case. This is an upgraded design for the RPi B+/2, based on by James Delgarno's original design for the RPi A/B models. The case can open and close with the tabs and slits. The following picture may not be the correct scale, please be sure to verify the dimensions before creating a case using this template for your Pi. Cardboard cereal boxes are an easy cardboard to work with. If you are so lucky, check if the inside of the cover of your book is a case design for the Raspberry Pi. If so, detach the cover carefully and follow the instructions within the outline.

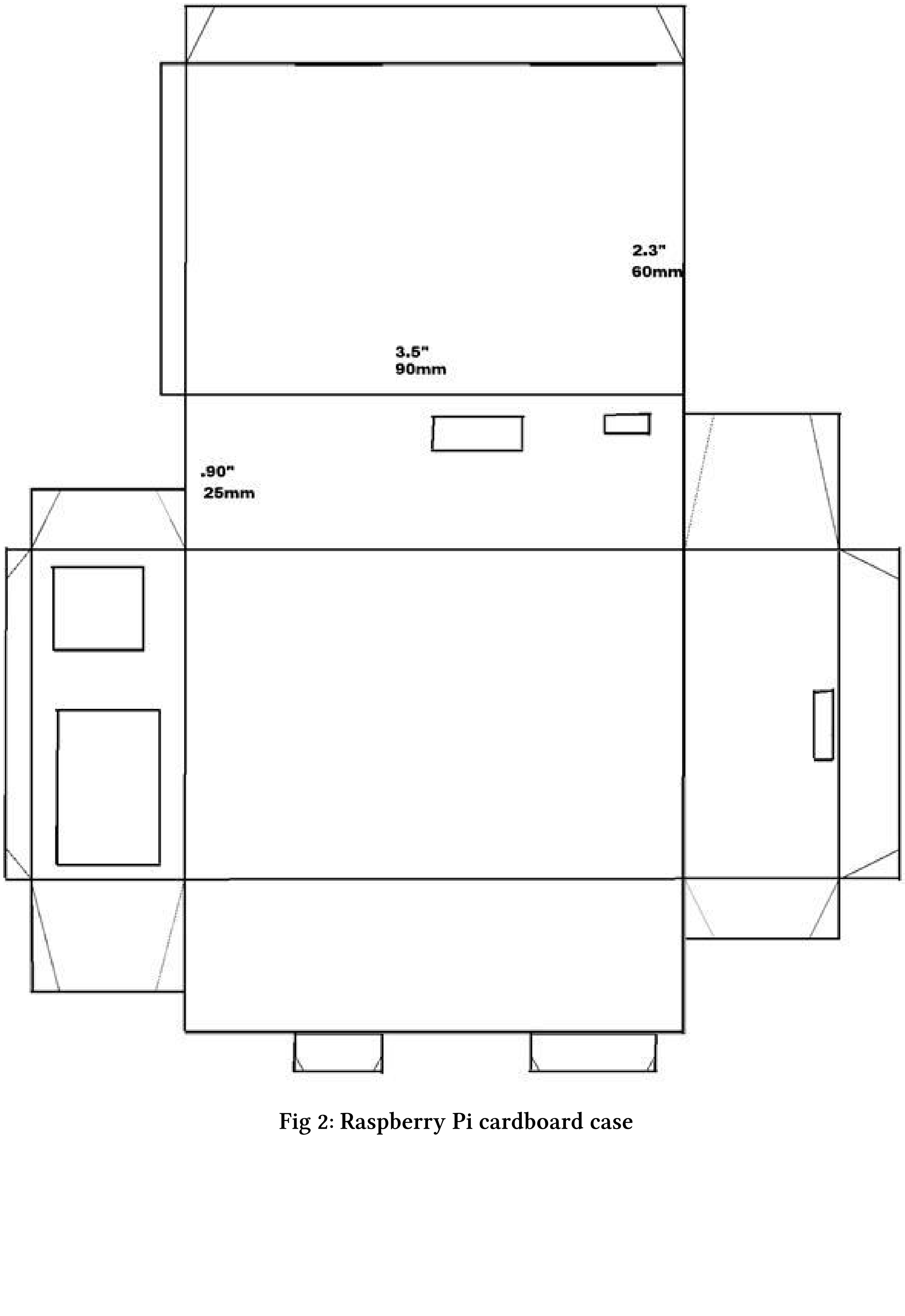


Fig 2: Raspberry Pi cardboard case

1.4 Connect the Pi

Practice performing the following sequence of actions with the board in its case and unpowered.

1. Connect and disconnect a USB keyboard to the Raspberry Pi.
2. Insert and remove microSD card into Raspberry Pi.
3. Connect and disconnect an Ethernet cable to the Raspberry Pi.
4. Connect and disconnect an HDMI cable to the Raspberry Pi.

All connectors must first be oriented before they will connect. User should identify the orientation of the connection on the board and orient the cable/connector before attempting connection. This process is expected to familiarize the user with the various connectors and their required orientation before connection. It may also require adding or adjusting holes in the case if homemade.

1.5 Create a Linux Installer

Important

Less experienced students should be provided preinstalled SD cards and skip this section or adults with prior experience can carefully direct and supervise.

To run Linux on the RPi we must first burn a Linux installer to the microSD card. It is recommended to install Raspbian, not NOOBs. Details change between releases so to keep this timeless, please just follow the installation directions on the raspberrypi.org website below. Windows PC users can use Win32DiskImager (or an equivalent utility) to burn the Raspbian distribution installer onto the SD card. For Linux OS users, consult the latest information on the Raspberry Pi website. In general though, download the installable image from the website and use the installation procedure to write the image to the microSD card connected to the PC with a USB adapter. Then insert the SD card into the RPi and it will install Linux on the next power up. See this link for burning an installer image to the microSD card to execute on the Raspberry Pi: https://www.raspberrypi.org/downloads/

1.6 Power the Pi

Time to power up the Raspberry Pi! Perform the following actions in order, starting with nothing connected.

Warning

Be sure to do them in this order as removing the microSD card while the board is powered on can corrupt the SD card.

1. Connect USB keyboard and mouse to the Raspberry Pi.
2. Insert microSD that was pre-installed with Raspbian.
3. Connect HDMI cable to Raspberry Pi and a Monitor or TV.
4. [OPTIONAL] Connect an Ethernet cable from Raspberry Pi to home router (Internet access).
5. Attach a 5 volt, 2 amp minimum USB charger, the USB end into the Raspberry Pi first, before plugging the other end into a power strip or outlet.

Important

If this lab involves children, adult supervision may be needed here regarding running Linux on the Raspberry Pi, especially with powering on and off. Be clear with the procedure and use power strips to make it easier to remove the microSD card only while the board is powered off.

The procedure above should power the board and start the Raspbian Linux operating system installer (or other image previously burned to the microSD card). If the SD card is empty or not properly imaged, the board will still have the red (power) LED lit. The yellow LED may flicker briefly when power is originally applied, this is good as it indicates the hardware found the startup files needed to perform pre-boot. After the Linux installer completes, the Raspbian OS will run and the next power up will start Linux directly. Do not power down yet until performing the shutdown procedure in the next section.

Important

For initial Raspberry Pi development where SD card removal is common, it is very helpful to use a power strip to turn the device on and off. If not, at least pull the plug from the wall instead of the USB connection. This is important to prevent wear on the USB power connection as well as the case (if cardboard).

1.7 The development system

Once Raspbian is installed, it should boot into a graphical interface. The user must open a system Terminal window to be able to enter commands. Linux variants differ on where to find this, but it can usually be found from the start menu, in system utilities or accessories. I find it helpful to create a shortcut to the Terminal on the desktop so I only have to find it once. On Raspbian the 'Terminal' is currently located in the top left 'Accessories' menu. Right click to add the 'Terminal' icon to the Desktop.

Opening the terminal, system shell, command prompt, etc. is the end of this chapter, but Linux will not be happy if we just turn off the power. Linux and other modern operating systems prefer that the user tell the operating system that it wishes to shut down. This is accomplished on Linux systems with the "shutdown" command. Enter the following command to shut down a Linux computer and attempt to power off hardware.

```
$ shutdown
```

If you see an error similar to 'Permission denied' you will need superuser privileges to use the 'shutdown' command. To execute, or 'do', a command as the 's'uper 'u'ser, enter 'sudo' before the command, like this example.

```
$ sudo shutdown
```

If Raspbian is installed then click the Shutdown… option in the upper left drop down start menu. Other graphical Linux systems typically have a shutdown option available from the start menu. If you cannot find it, the command line above will always works. Here is the new order for power cycling the Raspberry Pi as a development board.

Check connections of the Raspberry pi

1. Check USB keyboard, mouse and Ethernet connections to the Raspberry Pi.
2. Insert or check insertion of microSD installed with Raspbian Linux.
3. Check HDMI cable connection between the Raspberry Pi and Monitor.
4. Check attached 5 volt, 2 amp minimum USB charger at the USB end going into the Raspberry Pi.

Power on

1. Check or plug into power strip or wall outlet.
2. Turn on power strip (if present and off).
3. Linux OS will boot. Use Linux until shutdown is performed.

Power off

1. After 'shutdown' command, wait for all activity lights stop and monitor goes into power save mode indicating the Linux OS has completely shutdown.
2. Turn off power strip or unplug from wall outlet.

Lab 2: Command Shell and Editors

2.1 Why the Command Prompt or Shell?

This lab begins the discussion on how to create software for a computer system. These days there are a variety of graphical Integrated Development Environments, or IDEs, to choose from. However, these are often complex and can hide the details, requiring a large investment of time to configure and an incomplete understanding while using. To efficiently learn to create system software, the development environment should be step by step and easy to understand so that the process becomes comfortable and familiar. A familiar and repeatable process to create system software gives confidence to the developer.

Software development, especially system software, should be carefully created, one step at a time. Issuing commands in the command terminal will be explored as the first step in understanding the details of creating software. This knowledge can be directly applied to IDE configurations if/when the reader is ready to learn them.

The last laboratory assignment should have introduced the reader to the system terminal or command shell, which is historically the simplest form of development environment. This lab describes how to navigate file systems and use source code editors from the command shell. These skills are necessary to create the software in future laboratory assignments.

2.2 Exploring file systems from the command line

The command prompt on Unix and Windows systems allows the user to navigate the file system. A file system has a top down (hierarchical), structure and the term ‘directory’ is used to describe each layer. The top directory is the root of the file system.

Each directory contains files and other directories, with each directory containing more files and directories, on and on as desired. Upon opening the command shell the default directory is most often the users home directory.

The ‘pwd’ command can be used to display the complete path of the current working directory (‘pwd’ stand for Print Working Directory). The ‘pwd’ command, outputs the current file system path, or current working directory. The directory path is all the parent directories leading up to and including the current working directory. Every command prompt or system shell begins in a default location in the file system, usually the home directory for the user. Some Unix systems display the current working directory as part of the command prompt itself. Commands executed in the terminal use the current working directory when applying to or looking for files, etc.

To display the contents of a directory from the command prompt, the list directory, or ‘ls’, command can be used on Linux systems or the MinGW shell or PowerShell executing on Windows. More detailed information on the contents of a directory is available with the ‘ls -la’ command. In the output of ‘ls -la’ the files and directory names are displayed in the right column, while details of each is on the left. If the details on the far left begins with the letter ‘d’ then the name is associated with a directory, not a file.

The ‘ls’ command alone might have color coding to indicate directories vs. files and even executables, but not always. Linux, Mac and Windows based command terminals very in their user color friendliness, but ‘ls -la’ always works. Some commands available in the command shell usually provide information on usage if passed the ‘–help’ option. For example, more information about the list directory command can be viewed using ‘ls –help’.

Changing directories can be done with the ‘cd’ command. This command can take the user deeper into the file system hierarchy, or out of the file system entirely. For example, ‘cd source’ will move the current directory location one level down into the ‘source’ directory (if it exists in the current directory, or an error otherwise). To move down multiple directories with a single command add the ‘/’ character between the directories, for example ‘cd source/apps’. To move to an absolute path, start the path with the ‘/’ character, for example ‘cd /home/MyUserName’. To move up out of the current directory and into the parent of the current directory, the double dot notation is used, for example, the ‘cd ..’ command is used move up to the parent directory. To move back up the files system more than one level requires multiple or stacked double dots. For example, to move back two directories one should use the ‘cd ../..’ command.

Using the ‘pwd’, ‘ls’ and ‘cd’ commands allows one to navigate the directories of a file system of any depth.

It is best to create a specific location on the development system where the system software is created. The make directory command, or ‘mkdir’, creates a new directory within the existing directory. From the users home directory let us make a new ‘source’ directory with the ‘mkdir source’ command. This directory is where we can put the source code we are going to create. Then ‘cd source’ will move the command prompt into this directory. If lost, remember the ‘pwd’ command can be used to display the current directory.

2.3 Viewing and editing source code files

The simplest editor that can be used in the command line is ‘pico’. This editor is very basic and simple, allowing users to view and edit text files such as C source code. The ‘pico’ interface is similar to other text editors. Use the arrow keys to move the cursor and add text wherever the cursor is located. The delete and backspace keys will remove a character in front of or behind the cursor. The ‘pico’ editor also accepts control characters to perform actions such as saving and exiting. All the commands supported are shown at the bottom of the editor. The ^ character indicates to hold the Ctrl key down while pressing the next character.

Knowing a few commands are required to use ‘pico’. For example, to exit the editor requires holding down the Ctrl key and pressing X. This control sequence is commonly referred to as Ctrl-X or ^X. Ctrl-O writes Out the file, or saves it. To move text, there is Ctrl-K to cut text, and Ctrl-U to uncut text or paste. Ctrl-A or ^A brings up a menu of the commands, so Ctrl-A and then X exits the editor, similar to Ctrl-X. If you forget all other commands, remember Ctrl-A as that will bring up the menu of commands.

The cut and paste commands affect entire lines only but multiple lines can be combined, so three cuts and one uncut will paste three lines at the cursor location. This works great for moving and/or organizing the source code, but is a bit limited. Cut the lines, move the cursor to the new location and paste the lines to move blocks of source code. Let us use the ‘pico’ command line editor now to create the new file add.s.

```
$ pico add.s
```

Now it is time to write some ARM assembly code. Let us review the differences between the ARM assembly code representation of the pseudo code were learned and created in the companion book. Note that instead of the MOV instruction there is, 'ldr' and 'str', or load register and store register. Also, all the ARM assembly instructions are defined in lower case. For GCC assembler, comments follow any semicolon ampersand ';@'. Each line of code below is commented with information to aid in reading and understanding the assembly code.

```
ldr  r1,=Var1         ;@ load R1 register with address in RAM of Var1
ldr  r4, [r1]         ;@ load value of Var1 by referencing R1 address
add  r4, r4, #1       ;@ add one to the value
str  r4, [r1]         ;@ store the result to Var1 in RAM
```

The ARM **assembly** code above starts at the load register instruction 'ldr' and goes through to the store register instruction 'str'. To create the new file add.s, navigate to the directory where the file should be created ('source' folder) and type the command 'pico add.s'. Then enter the above ARM **assembly** into the new file add.s using the pico editor, pressing Ctrl-X and Enter to save the new file upon exit. This **assembly** language file creates a variable named 'Var1' and assigns it an initial value of zero (0).

2.4 Command line assembling

The command for **assembling assembly** files is called the **assembler** and can be invoked at the command line with the 'as' keyword followed by the name of the file to **assemble**. The **assembler** command 'as' requires options and therefore must be executed from the command prompt.

Important

For users developing on a PC (Windows or Linux), the examples for this chapter require a compiler for the ARM architecture be installed, not one for the x86 architecture commonly used for PCs. Please jump ahead to Section 3.2 and install the GCC tool set for ARM on the PC development system. Then replace the 'as' and 'objdump' commands below with the 'arm-none-eabi-as' and 'arm-none-eabi-objdump' commands respectfully for the remainder of this chapter. Raspbian users can use the default Linux assembler 'as' installed with Raspbian until it is replaced in Section 3.3.

Let us take a look at using the command 'as' to compile the file 'add.s', created in the last section.

```
$ as add.s
```

After executing this command with the assembly file named, the compiler creates a binary file named 'a.out'. You can see this new file if you use the 'ls' command to list the current directory contents. This 'a.out' file is not a very helpful name and things can get confusing quickly if compiling more than one assembly file. The '-o' option is available to the assembler that allows you to name the resulting output file. Here is the another example of compiling the 'add.s' file into the binary output file 'add.o'.

```
$ as add.s -o add.o
```

2.5 Command line disassemble

Once the assembly file is compiled into a binary output file, the object dump command, or 'objdump', can be used to read the binary executable and disassemble the contents back into human readable **assembly** language. Let us examine the recently compiled 'add.o' to understand what the **assembler** is doing in more detail.

{linenos=off,lang="bash"}} $ objdump -D add.o

```
add.o:     file format elf32-littlearm

Disassembly of section .text:

00000000 <.text>:
   0:   e59f1008        ldr     r1, [pc, #8]    ; 10 <.text+0x10>
   4:   e5914000        ldr     r4, [r1]
   8:   e2844001        add     r4, r4, #1
   c:   e5814000        str     r4, [r1]
  10:   00000000        andeq   r0, r0, r0

Disassembly of section .data:

00000000 <Var1>:
   0:   00000005        andeq   r0, r0, r0

Disassembly of section .ARM.attributes:

00000000 <.ARM.attributes>:
   0:   00001341        andeq   r1, r0, r1, asr #6
   4:   61656100        cmnvs   r5, r0, lsl #2
   8:   01006962        tsteq   r0, r2, ror #18
   c:   00000009        andeq   r0, r0, r9
  10:   01080106        tsteq   r8, r6, lsl #2arm
```

Note that the core of the .text section is close to what is in the ‘add.s’ so it appears the assembler is working correctly. Also note that a new .data section was created containing the variable ‘Var1’.

Reviewing the dump of the binary above we see that while we did not declare the ‘Var1’ variable in the .s file, it was declared in the binary. To understand better why this is happening, let us add the variable declaration statement for ‘Var1’ to the original assembly source and recompile.

```
.section .data
Var1: .int 0            ;@ declare Var1 as an integer with value zero

.section .text
ldr  r1,=Var1           ;@ load R1 register with address in RAM of Var1
ldr  r4, [r1]           ;@ load value of Var1 by referencing R1 address
add  r4, r4, #1         ;@ add one to the value
str  r4, [r1]           ;@ store the result to Var1 in RAM
```

After compiling again, dump the object file again and note that the object dump files are identical. What is happening is that the assembler is being helpful by declaring the variable automatically as a integer type (.int) once used. This is helpful but can also lead to unintended consequences as all undefined variables will be declared as integer type and the initial value is not guaranteed. For clarity of reading the assembly as well as to ensure operation, it is best practice to always declare and initialize variables at the top of the .s file.

To summarize, the 'as' command **assembles** an **assembly** language file and creates a binary output file as a result, while the 'objdump' command **disassembles** a binary output file, showing the **assembly** language that the binary file represents. System software requires an accurate and reliable **assembler** to build a software binary correctly. A reliable and verified **assembler** is vital to the success of any software project. **Assembling** and dumping an **assembly** file is a good way to check what the **assembler** is doing. In the .text section of the 'objdump' output above is the assembly language the compiler created from 'add.s'. As expected, the binary generated by the assembler looks very similar to the assembly file we created to add one to a variable with a default value of zero. The reader is encouraged to modify the assembly language in 'add.s', recompile and examine the resulting changes with 'objdump'.

2.6 Assembly language branch instructions

The main branch assembly instructions for ARM CPU are branch 'b', branch less than or equal 'ble', and branch greater than 'bgt'. Labels are places in the assembly language where a branch instruction can jump to. Labels are defined in the GCC assembler as a single line containing the label name followed by a colon (:). Let us review the ARM specific assembly version of the generic assembly that was reviewed at the end of

Chapter 2. This assembly stores/loads the 'Count' variable value into a register and then loops, adding 1 to this register every loop. After every ten (10) times through the loop the value is saved back to the variable 'Count'. Remember that the '.section .data' is used to declare variables, and ';@' used to indicate a comment, in GCC assembly.

```
;@ Variables
.section .data
Count:                    ;@ declare Count as an Integer with value zero
.int 0

.section .text
ldr  r1,=Count            ;@ load R1 register with address in RAM of Count
ldr  r3, [r1]             ;@ load value of Var1 by referencing R1 address

_save:
str  r3, [r1]             ;@ store the result to Var1 in RAM
mov  r2, #1               ;@ initialize the save count to one

_loop:
add  r3, r3, #1           ;@ add one to the value
add  r2, r2, #1           ;@ add one to the save count
cmp  r2, #10              ;@ compare the store count
bgt  _save
b    _loop
```

2.7 Integer variable roll overs

Remember from Chapter 2 of the Computer System book that when an unsigned integer value increases beyond the maximum for the size of the variable, it rolls back to zero. For example, if an unsigned integer variable of size 32 bits and value 0xFFFFFFFF is increased by one, the value becomes 0x00000000. Modify the assembly language from the last section, adding a second variable to be increased by one only once 'Count' has reached its maximum value. So when increasing 'Count' by one, check if it results in a roll over to value zero (0). If so, 'Count2' should be incremented. To do this, initialize ‘Count’ and ‘Count2’ to zero and check ‘Count’ after every increment. If ‘Count’ is zero it rolled over so increment ‘Count2’ by one.

```
;@ Variables
.section .data
Count:  .int 0          ;@ declare Count as Integer of value zero
Count2: .int 0          ;@ declare Count2 as Integer of value zero

.section .text
ldr  r1,=Count          ;@ load R1 register with address of Count
ldr  r2,=Count2         ;@ load R2 register with address of Count2

ldr  r3, [r1]           ;@ load Count1 value into R3 by referencing R1
_loop:
add  r3, r3, #1         ;@ add one to the value
str  r3, [r1]           ;@ store the result to Var1 in RAM
cmp  r3, #0             ;@ compare count to zero
ble  _rollover          ;@ branch to rollover if less or equal
b    _loop              ;@ branch to add one again to Count1
_rollover:
ldr  r3, [r1]           ;@ load Var1 into R3 by referencing R1
add  r3, r3, #1         ;@ add one to the value
str  r3, [r1]           ;@ store the result to Var1 in RAM
b    _loop              ;@ branch to add one again to Count1
```

Lab 3: GCC Tool Chain for ARM

3.1 Why GNU C for the Raspberry Pi?

The C programming language uses a **compiler** to create system software. It must create accurate and predictable software from the source code. The GNU C Compiler, or GCC, is a common compiler for building Linux systems as well as bare metal systems. Other C compilers are available but GCC is open source, integrated with the Linux OS, supports most CPU types and has cross compiler support for most major PC Operating Systems. The Raspberry Pi hardware is well supported by the GCC team and the ARM open source community in general.

3.2 Create a cross compiled executable

The command for compiling with GCC is 'gcc', while the command to link is 'ld'. However, on Linux development systems this default compiler and linker are designed for compiling Linux applications which can bring on a whole lot of dependencies and assumptions that are not valid for bare metal system software. Therefore the system engineer requires the 'arm-none-eabi-gcc' compiler instead of the default Linux GCC compiler installed by the OS. For a Linux development system, such as the Raspberry Pi running Raspbian, this version of the GCC compiler can be installed by issuing the 'sudo apt-get install gcc-arm-none-eabi' command with Internet access. This install includes the whole range of tools built specifically for bare metal ARM CPU compiling, for example 'arm-none-eabi-gcc', 'arm-none-eabi-as', 'arm-none-eabi-ld', 'arm-none-eabi-objdump', and 'arm-none-eabi-objcopy'. The different naming convention allows your Linux OS can keep the default compiler in place for building Linux applications.

```
$ sudo apt-get install gcc-arm-none-eabi
```

The 'arm-none-eabi-gcc' compiler and 'arm-none-eabi-ld' linker are designed for bare ARM systems without Linux. For Windows PC developers it is recommended to install

3rd party cross compiler tools for ARM. Presently, the GCC installer created by ARM Developer group at developer.arm.com is recommended for ARM cross compiler on Windows. Search online references for ARM cross compiling tools if you desire an alternative compiler or this information becomes out of date. Search for ‘arm-none-eabi-gcc Windows download’ for example. Presently for Windows a reliable and tested source can be found here ‘https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm/downloads’.

Information

If no online cross compiler is available for your OS, one can be built from the GCC source code. This is far from trivial however and beyond the scope of this introduction, but could be an option for those with more experience.

The newly installed compiler and linker commands can be executed from the command line, such as a Terminal in Linux or the Command Prompt in Windows. With the command line we can execute the compiler and linker commands, including options, to compile the file ‘main.c’ from Chapter 3 of the companion book. The example below compiles (executes arm-none-eabi-gcc at the command line) the file ‘main.c’ into main.o and then links (arm-none-eabi-ld) main.o into an Executable and Linkable Format (.elf) file. Finally the .elf is converted (arm-none-eabi-objcopy) into binary file led.bin which is then copied (cp or copy in Windows) to ‘kernel.img’, which is the name of the file the RPi automatically executes on power up.

Let us briefly review the options used with the GCC compiler. The ‘-c’ is used to tell the compiler to compile and assemble the C source into an output file defined with the ‘-o’, in the example below this would be ‘main.o’. The remaining option is the ‘-I’ option which tells the compiler which directory contains C header (.h) files #included within the C files. Option ‘-I.’ tells the compiler to look in directory ‘.’, which is the current directory. Here is an example of the steps entered into the Linux command shell to compile, link and prepare the binary executable.

```
$ arm-none-eabi-gcc -I. -o main.o main.c
$ arm-none-eabi-ld -o app.elf main.o
$ arm-none-eabi-objcopy app.elf -O binary app.bin
$ cp led.bin kernel.img
```

We will briefly mention the ‘app.elf’ linker file first to get it out of the way. Consider

ELF (.elf) files to be a form of formatted binary files, often required by ‘objdump’ and debuggers, but not directly executable by the CPU. It is important to understand that the linker (arm-none-eabi-ld) creates ELF files and this ELF is expected for object dump (arm-none-eabi-objdump) and debuggers such as GDB (see next chapter). The object copy (arm-none-eabi-objcopy), with ‘-O binary’ option, can convert the ELF file into a raw executable binary file that can only then be directly executed. The last step of the sequence above copies (cp) this binary executable file to the ‘kernel.img’ file, which is the name needed for it to execute on the Raspberry Pi’s Linux boot microSD card when powered on. The next step is to put this created ‘kernel.img’ file onto the microSD card and execute it.

3.3 Execute a binary on the Raspberry Pi

The last step before testing is to move the binary executable kernel.img to the boot file system. For the RPi, this is the file named kernel.img (or kernel7.img for RPi 2/3) in the root directory of the boot sector of the microSD card. Using a USB to microSD card adapter, move the microSD card to the development PC and then move/rename the existing Linux kernel (named above) to a new file, such as kernel_linux.img. If using a Linux system with automount, such as Raspbian, two folder partitions ‘boot’ and ‘root’ will be discovered and pop up when the USB adapter with microSD card is connected. The ‘kernel.img’ that the RPi executes at power on is in the ‘boot’ drive folder.

The Linux kernel we previously named kernel_linux.img will need to be renamed to kernel.img (or kernel7.img if RPi 2/3) before Linux will boot again. It is important to have a PC or other device available which can access the microSD card after this change in order to rename the Linux kernel back to kernel.img, otherwise the RPi cannot boot back into Linux. Whatever is in kernel.img the RPi will execute every boot. In a group environment will all RPi’s, keep one system running Raspbian dedicated to this so no one has to wait for an available system.

The command sequence below will first save the Linux kernel and then copy the bare metal program we created and run it. Afterward, commands are shown to restore the Linux kernel and boot back into Linux on the RPi. The Raspberry Pi running Linux is using the same /boot folder and the Linux kernel was renamed and replaced with the bare metal image. After executing our creation it is required to delete or rename our creation and rename the Linux kernel back to kernel.img (or kernel7.img for RPi 2/3) before the RPi can boot into Linux again.

Warning

Only after renaming or backing up the real Linux kernel should you copy the newly compiled bare metal program into the root directory of the 'boot' driver of the microSD card to replace it. A second PC is required to restore the name the Linux kernel back to 'kernel.img' in order to boot into Linux again on the RPi.

The sequence below is for those developing and testing on the same RPi, while using another PC running a Linux OS to recover the Linux kernel of the RPI afterward on the microSD.

```
[AFTER COMPILING ON DEVELOPMENT RPI RUNNING LINUX]
$ mv /boot/kernel.img /boot/kernel_linux.img
$ cp kernel.img /boot/kernel.img
$ shutdown -h now
[POWER OFF]
[POWER ON]
<* Bare Metal Executable runs *>
[POWER OFF]
[PHYSICALLY REMOVE SD AND PLUG INTO SECOND LINUX PC]
$ sudo mount -t vfat /dev/sda1 /mnt/usb
$ mv /mnt/usb/kernel.img /mnt/usb/kernel_app.img
$ mv /mnt/usb/kernel_linux.img /mnt/usb/kernel.img
$ sudo umount /mnt/usb
[MOVE SD BACK TO RPI]
[POWER ON]
$
```

The sequence below is for those developing and testing on the same RPi while using another PC running a Windows OS to recover the Linux kernel of the RPI afterward on the microSD.

```
[AFTER COMPILING ON DEVELOPMENT RPI RUNNING LINUX]
$ mv /boot/kernel.img /boot/kernel_linux.img
$ cp kernel.img /boot/kernel.img
$ shutdown -h now
[POWER OFF]
[POWER ON]
<* Bare Metal Executable runs *>
[POWER OFF]
[PHYSICALLY REMOVE SD AND PLUG INTO WINDOWS PC AS DRIVE E:]
$ mv e:\kernel.img e:\kernel_app.img
$ mv e:\kernel_linux.img e:\kernel.img
[MOVE SD BACK TO RPI]
[POWER ON]
$
```

3.4 Cross compiler development PC

After Chapter 5 it will no longer be possible to use the same RPi system for development as well as to execute the compiled system software. The system software will soon open a physical communications channel with the development system, so both a development PC and an RPi are required. This development PC is used to edit, compile and distribute the software and to communicate with the RPi during system software execution. It is recommended to use two RPi's, with one system running Raspbian Linux as the development system and the other running the bare metal application and communicating with the RPi system under development. Windows, Linux or MacOS can be used as the development system. All that is needed is a compatible USB to micro SD card adapter, USB TTL (see chapter 6), source file editor, command shell and the correct 'arm-none-eabi' cross compiler installed on the OS being used as the development system.

Shutting down Linux in order to test a bare metal application, and then restoring the Linux kernel on the SD card, can quickly test ones patience. To become more efficient with this process requires we transition to compiling for the RPi with a cross compiler on a separate development PC. The USB adapter to the SD card can be used to copy the compiled image that we wish to execute on the RPi. Here is a command line example of mounting and copying to a USB device. This overwrites the existing kernel.img, assuming it was not the linux kernel but the last test. This type of procedure is common when the microSD card is dedicated to the RPi used for testing, and the Linux kernel has already been archived and or renamed.

```
$ ls /dev/sd*
[INSERT USB TO SD CARD ADAPTER]
$ ls /dev/sd*
[THE SD ADAPTER IS THE NEW ONE]
[EXAMPLE BELOW USES /dev/sda1 CHANGE WITH ABOVE IF DIFFERENT]
$ sudo mount -t vfat /dev/sda1 /mnt/usb
$ cp kernel.img /mnt/usb/kernel.img
$ sudo umount /mnt/usb
[REMOVE SD ADAPTER]
[MOVE SD ADAPTER TO SECOND TEST RPI]
[POWER ON SECOND TEST RPI]
<* Bare Metal Executable runs *>
[POWER OFF SECOND TEST RPI]
[REPEAT]
```

This example assumes '/mnt/usb' is a valid empty directory and the Linux system did not already automount the USB drive. With Raspbian and automount, the procedure to copy the file ultimately becomes one 'cp' line once the mount location is known. Below is the sequence of commands needed to build and copy the executable to the microSD card on a Linux OS with automount, for executing on the RPi. Optionally the the File Manager GUI application can be used to rename and copy and paste the kernel.img file to the mounted /boot folder.

```
$ arm-none-eabi-gcc -I. -o main.o main.c
$ arm-none-eabi-ld -o app.elf main.o
$ arm-none-eabi-objcopy app.elf -O binary app.bin
$ cp app.bin kernel.img
[INSERT USB TO SD CARD ADAPTER]
$ cp kernel.img /mnt/usb/boot/kernel.img
[MOVE SD ADAPTER TO SECOND TEST RPI]
[POWER ON SECOND TEST RPI]
<* Bare Metal Executable runs *>
[POWER OFF SECOND TEST RPI]
[MOVE SD CARD OF TEST RPI TO USB ADAPTER]
[REPEAT]
```

On Windows systems with an 'arm-none-eabi' cross compiler installed, the same sequence of commands, including options, are needed to compile the file 'main.c', link 'main.o' into an Executable and Linkable Format (.elf) file and finally into the binary files 'led.bin' and 'kernel.img'. However, Windows automount will activate a new drive letter, for example drive E: for the SD card 'boot' partition.

```
[MOVE SD TO PC, E: DRIVE]
$ cp kernel.img e:\kernel.img
[DISCONNECT SD/USB IN OS]
[MOVE SD BACK TO RPI]
[POWER ON]
<* Program runs *>
[POWER OFF]
[REPEAT]
```

With a Linux OS supporting automount (Raspbian, etc.), or Windows, the folders of the SD card open upon connecting the USB device. An alternate procedure to those above is to drag and drop, or copy and paste, the 'kernel.img' from the location where it was created and onto the connected boot drive. On Windows there is only one drive letter that is accessible when the USB adapter is plugged in so copy the kernel.img file to the root directory of that drive. If Raspbian was previously installed on the microSD card there will be two drives that open when USB is connected. Copy the 'kernel.img' file to the 'boot' drive.

3.5 Consistent cross compiler development

To compile, link and execute a program requires fewer steps when using a dedicated development PC. This cross compiling on a fast development PC will be used in all the examples going forward. Using two RPi's, one for the development Linux OS and the other for the system under test, is the recommended option. What is needed now is a simpler way to build the software. The majority of steps in the previous section were to compile and link the software.

To aid ease of use and ensure repeatability in the compilation and linking process it is recommended to use a build script. The shell commands can be stored and executed one at a time in sequence using a simple file. Create a file named 'build.sh' (or 'build.bat' if Windows) and put the compilation and linking commands above in it, one command per line. Then to compile, instead of typing each line, enter a single command at the command prompt to execute the script './build.sh' (or '.\build.bat' for Windows).

```
$ ./build.sh
arm-none-eabi-gcc -c -I. -o main.o main.c
arm-none-eabi-ld -o app.elf main.o
arm-none-eabi-objcopy app.elf -O binary app.bin
```

The build script is vital to creating system software in a consistent and repeatable way. If you see a 'permission denied' error on Linux it is because the file is not executable. The 'chmod +x build.sh' command will make the 'build.sh' file executable.

3.6 Compile and link executable for bare metal

Using the cross compiler on the development system, we first compile the file 'main.c', then link main.o into an '.elf' file and finally into the binary files 'app.bin' and 'kernel.img'. Let us expand upon the 'build.sh' from last section and introduce some options to the GCC compiler so that it will create a more targeted and optimized executable. One feature we will use is the '-O', or optimize, option. There are different types of optimizations available for '-O' option. Option '-O0' indicates to the compiler to not optimize at all and this is helpful for debugging or reviewing the resulting assembly language. Option '-Os' tells the compiler to optimize for size, creating the smallest binary possible from the source code. Option '-O2' tells the compiler to optimize for a combination of size and speed and is the option recommended.

The option '-ffreestanding' tells the compiler that the compiled binary is free standing and will not be running on Linux or using the GCC standard library, or stdlib, which is normally linked automatically when creating a binary. The freestanding option creates what is known as a 'bare metal' executable that will be completely independent of any other software. The '-ffreestanding' option tells the compiler not to create the default _start assembly entry point to the software. With this option it is required to put main.c first in the link order so that main() will be the entry point.

```
arm-none-eabi-gcc -c -O2 -ffreestanding -I. -o main.o main.c
arm-none-eabi-ld -o app.elf main.o
arm-none-eabi-objcopy app.elf -O binary app.bin
cp app.bin kernel.img
```

To review, executing this program requires overwriting the kernel.img on the microSD card. Here is the sequence events again when using a Windows PC for development.

```
[MOVE SD TO PC E: DRIVE]
$ cp e:\kernel.img e:\kernel_last.img
$ cp app.bin e:\kernel.img
[DISCONNECT SD/USB IN OS]
[MOVE SD BACK TO RPI]
[POWER ON]
<* Program runs *>
```

Using the example source code and procedure, test that the application will execute the loop and run forever. Run this application and compare the LED status with powering on the RPi without a microSD card present. By viewing the LED during application execution, how can you tell the difference between the application execution and running the RPi without an SD card? Were you able to verify that the software binary executes the loop forever? Change the source to loop a million times and then exit main. If you recompile and test, does the LED behave differently? Depending on the version of RPi you are executing on, your observations may vary. One piece of the puzzle is still missing.

Let us update our ‘build.sh’ now, using a script comment (#) to add comments on how to configure. The example below assumes you are building for the RPi B+ but this can be changed by changing the ‘-DRPI=’ value to the number version of your RPi; 1, 2, 3 or 4.

```
#options -ffreestanding ensures no linkage to GCC external libraries
#option -c compiles/assembles, -O2 optimizes, -ggdb adds debug symbols

#Compile main.c for RPI 1 with debug symbols and no optimizations
arm-none-eabi-gcc -c -DRPI=1 -O0 -ffreestanding -I. -o main.o main.c

#Use ld to link with RPi memory map
arm-none-eabi-ld -T memory.map -o app.elf main.o

#strip elf into binary and copy to kernel.img
arm-none-eabi-objcopy app.elf -O binary app.bin
cp app.bin kernel.img
```

Information

DEPRECATED. The following is historic and not necessary for current versions of GCC. The CPU specific options below can be enabled to allow the compiler to do CPU specific optimizations in conjunction with -O2 and -Os options.

The GCC compiler, including ‘arm-none-eabi-gcc’, have compile options that will generate assembly code correctly for specific ARM CPU’s. For the RPi, these changes are different depending on whether the hardware is an RPi B+, RPi 2, 3 or 4. For the RPi B+, add the option ‘-mcpu=arm1176jzf-s’ to the ‘arm-none-eabi-gcc’ compiler. For the RPi 2, add the options ‘-march=armv7-a’ and ‘-mtune=cortex-a7’ to create an executable specific to the RPi 2 CPU. For the RPI 3 the options are ‘-march=armv8-a+crc’ and ‘-mtune=cortex-a53’ for the 64 bit 4 core ARM Cortex A53 CPU within the SoC.

Now the application should build and execute specifically for the RPi hardware you have. However, with the simple application we created there is no way know if the software executes correctly. In the next chapter we will enable the activity LED on the RPi so we can physically see our software in action.

Lab 4: GPIO and LED

4.1 Why the GPIO and LED?

Using a GPIO interface to enable an LED is often the simplest system engineering challenge that will result in a physical change visible to the user. While the RPi has an onboard LED that is connected to the GPIO interface, it is the same software procedure to turn on or off any of the RPI GPIOs if they were connected to an external LED. GPIOs can connect to LEDs, motors or other electronics and/or a separate breadboard connected to one or more external electrical circuits. The internal activity LED (the yellow one) is connected and controlled by the RPi GPIO number 47 (for RPI B+ through 2, the RPI 3 B+ uses GPIO 29 and RPI 4 uses GPIO 42).

With some carefully written lines of code it should be possible to turn the GPIO on or off, changing the status of the onboard LED so as to be visible to the user. Those wishing to use external electronics with the Raspberry Pi must take note that the GPIO signals use 3.3V, which is not compatible with 5V peripherals, such as some motors, etc. The RPi GPIO pins should never be connected to 5V inputs without a proper voltage divider to achieve 3.3V. Failure to follow this warning will destroy your RPi. An RPi with GPIO connected to a 5V output has undefined behavior and can also damage the RPi (although maybe not with as much certainty as an input).

The previous laboratory procedures have built up the experience needed for the reader to now write system software that performs a visible change to the hardware. Let us carefully understand the remainder of this laboratory procedure. Concepts missed or misunderstood now can lead to confusion going forward, however do not let yourself get hung up on any one issue before continuing as major concepts are further clarified in future chapters.

4.2 Raspberry Pi memory map

The RPI hardware has a fixed location memory map configuration for peripherals. The interface to the RPi B+ memory map for the peripheral registers begins at address

0x20000000, at the top of the memory map. At this address begins the peripherals that can be accessed directly by software. The RPi 2 and 3 peripherals begin at address 0x3F000000 in the memory map. The RPi 4 has more RAM so the peripheral registers do not begin until address 0xFE000000. Here is a picture of the RPi B+ hardware memory map showing a software program loaded for execution at address 0x00008000 to 0x01000000. In this picture the alternate peripheral register begin address of 0xE0000000 is defined, but this location is no longer used and 0x20000000 should be used instead.

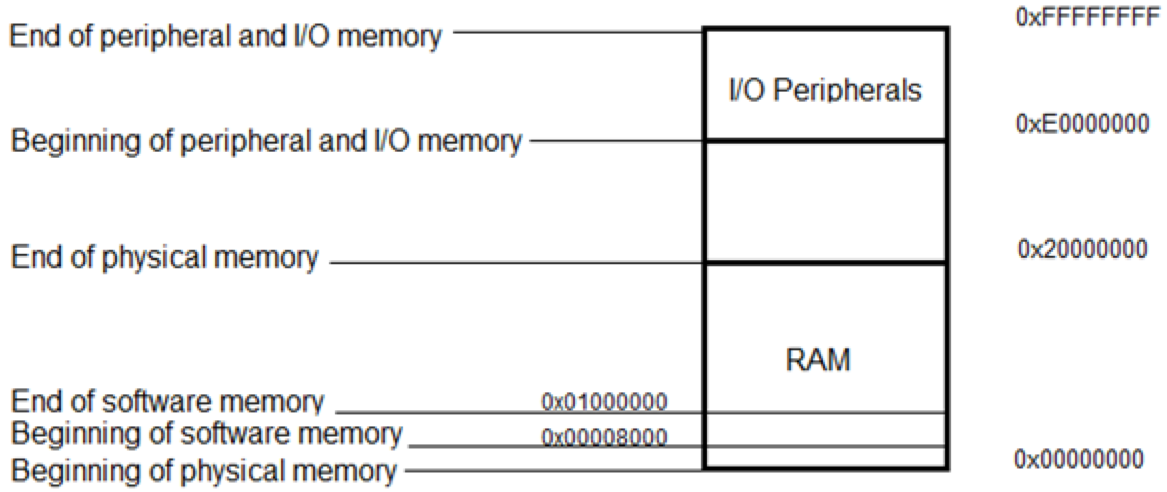

End of peripheral and I/O memory
Beginning of peripheral and I/O memory
End of physical memory
End of software memory
Beginning of software memory
Beginning of physical memory
0x01000000
0x00008000
I/O Peripherals
RAM
0xFFFFFFFF
0xE0000000
0x20000000
0x00000000

Fig 3: Raspberry Pi hardware memory map

Before writing any code let us review the memory map file 'memory.map' created in the the main book. For the RPi we must make a small modification to the linker script 'memory.map' created in Chapter 4 of the companion book. Since the RPi, upon boot up, expects to execute a Linux kernel, and a Linux kernel typically begins execution at address 0x8000 (for various reasons), we must modify the ORIGIN field to reflect this. Please modify the 'memory.map' file at this time, setting the ORIGIN to 0x8000.

```
MEMORY
{
    ram : ORIGIN = 0x8000, LENGTH = 0x1000000
}
```

We also need to change the build script to use this memory map. Open the 'build.sh/build.bat' file created at the end of chapter 3 and change the linking command to use the memory map.

```
#Use ld to link with RPi memory map
arm-none-eabi-ld -T memory.map -o app.elf main.o ..\..\boards\rpi\boar\
d.o
```

It would also be advisable, although not required, to change the name of the application from ‘app’ to ‘led’, as the application we are going to create in this chapter will active the LED.

4.3 Create/Edit board.h definitions file

To put the system software together, let us start by using the definitions in the system.h header file created and updated in this same chapter of the main book. First we should define the specific RPi register addresses from the memory map so they are easy to read and understand when writing and reading the source code. These register address definitions are board specific, so let use create a new file name ‘board.h’ and define them now. For reference, the peripheral base addresses described above, defined as RPI_BASE below, is documented at the following link: https://www.raspberrypi.org/documentation/hardware/raspberrypi/ peripheral_addresses.md

The GPIO register offsets are defined in Chapter 6 of the RPi hardware documentation file, BMC2835-ARM-Peripheral.pdf. Note that the documentation for this chapter uses addresses starting at 0x7E000000, yet neither the code or the link above uses this address. It should be stated that the BCM2835 documentation has many errors and out of date sections, but it is the best the RPi community has.

The following are the initial definitions defined to use the RPI_BASE define, so that they one change can allow the code to be used for various versions of the RPi. These source code definitions use the Field Name’s in the documentation, for example, GPIO set register 0 is defined in Chapter 6, and below, as GPSET0. This should make looking up register names in the documentation simpler. For now we are only interested in the Pin Output Set, Pin Output Clear and GPIO Pin Pull-up/down Enable registers. Let use create and review this board.h file now, based on the documentation.

```
/*.....................................................................*/
/* Configuration                                                       */
/*.....................................................................*/
/*
 * Base of peripheral memory map
*/
#if RPI == 1
#define RPI_BASE         0x20000000  /* RPi B+ */
#elif RPI == 4
#define RPI_BASE         0xFE000000 /* RPi 4 */
#else
#define RPI_BASE         0x3F000000  /* RPi 2/3 */
#endif

/*
 * GPIO SET, CLEAR and PULL UP/DOWN ENABLE register addresses
*/
#define GPSET0          (RPI_BASE | 0x0020001C)
#define GPSET1          (RPI_BASE | 0x00200020)
#define GPCLR0          (RPI_BASE | 0x00200028)
#define GPCLR1          (RPI_BASE | 0x0020002C)
#define GPPUD           (RPI_BASE | 0x00200094)
#define GPPUDCLK0       (RPI_BASE | 0x00200098)
#define GPPUDCLK1       (RPI_BASE | 0x0020009C)
```

Be sure to use the correct RPI_BASE definition for the specific RPi being used to execute the created software. For example, if compiling this system software for the RPi B+, add the -DRPI=1 compiler options to the correct value from the header file will be used. The -D (define) option to GCC is used to define hardware type to compile the executable for. Below is an updated build.sh file that uses the -DRPI=1 option to compile the LED executable for RPi 1 B+ hardware. If using different RPI hardware, change the -DRPI= value.

```
#options -ffreestanding ensures no linkage to GCC external libraries
#option -c compiles/assembles, -O2 optimizes, -ggdb adds debug symbols

#Compile main.c for RPI 1 with debug symbols and no optimizations
#  change -DRPI to the version of RPI system used to execute
arm-none-eabi-gcc -c -DRPI=1 -ggdb -ffreestanding -I. -o main.o main.c
#Compile main.c for RPI 1 with no debug symbols and optimizations
#arm-none-eabi-gcc -c -DRPI=1 -O2 -ffreestanding -I. -o main.o main.c

#Use ld to link with RPi memory map
arm-none-eabi-ld -T memory.map -o led.elf main.o

#strip elf into binary and copy to kernel.img
arm-none-eabi-objcopy led.elf -O binary led.bin
cp led.bin kernel.img
```

4.4 Configuring the GPIO Pull up/down Enable register

The Raspberry Pi allows software to configure the GPIO type, and depending on this type the GPSET and GPCLR registers change the internal pull up or down switches differently. A GPSET will apply voltage to the GPIO and the GPCLR will remove voltage from the GPIO. How the voltage behaves depends on the configuration of the GPIO and whether a pull up or pull down resistor is present. Review the BCM2835 documentation again, pages 102 and 103 and note the Pull column.

The GPIO Pull-up/down Register (GPPUD) configures the use of internal physical resistors for each GPIO, for example, pull up or pull down resistors. On page 101 of the BCM2835 document it describes the GPPUD, GPPUDCLKn and how to use them. Note that it is possible to configure pull up, pull down or none (disable pull up/down) for each GPIO. In general, if an RPi GPIO is being used as described on page 102 or 103, it is best to configure the GPIOs with the GPPUD and GPPUDCLKn to the state defined in the Pull column. If the GPIO is being repurposed for a different use then it can and most likely should be reconfigured. An LED is an output GPIO and we see from the documentation that it should be configure as Pull High, or as a pull up GPIO.

Let us review and add the code below to the top of main() in main.c. This codes uses the GPPUD and GPPUDCLK1 to configure GPIO 47 to pull up for the activity LED. The RPI 3 and RPI 4 GPIO configuration will be shown at the end but the principle is the same. The pull up resistor helps trap the voltage to keep the LED lit and without flutter, even if the voltage supplied to the GPIO line is not constant over time.

```
/*
** GPPUD can be 0 (disable pull up/down)
** (1 << 0) enable pull down (low)
** (1 << 1) enable pull up (high)
*/

/* Set pull up (high) for LED GPIO 47. */
REG32(GPPUD) = (1 << 1); /* bit one is pull up */

/* Loop to wait until GPPUD assignment persists. */
for (i = 0; i < TOGGLE_LOOP_CNT / 1000; ++i)
  select = REG32(GPPUD);

/* Assign GPPUD settings to GPIO 47 in GPPUDCLK1. */
REG32(GPPUDCLK1) = (1 << (47 - 32)); /* GPIO 47 */

/* Loop to wait until GPPUD clock assignment persists. */
for (i = 0; i < TOGGLE_LOOP_CNT / 1000; ++i)
  select = REG32(GPPUDCLK1);
```

4.5 Operating a GPIO with software on the RPi

GPIOs are CPU controlled electrical circuits that can be turned on (supplied voltage) or turned off (grounded) with software. A GPIO can be used to supply voltage (turn on) or ground (turn off) an LED or small motor. Software can turn a GPIO on or off with the GPSET and GPCLR registers. GPSET0 contains the bits to turn on (drive high or apply voltage) the first 32 GPIOs (0-31) while GPSET1 contains the bits to turn on the remaining 32 GPIOS (32 - 63). The same is true for the GPCLR0 and GPCLR1, except these turn off (drive low or ground) the GPIOs. Turn on (drive high) the 47th

GPIO by setting GPSET1 bit 15 to one, since 47 minus the 32 GPIOs used by GPSET1 results in bit 15. Hopefully this example clearly explains how GPSET1 and GPCLR1 service the GPIOs 32 through 63.

```
  /* Other RPIs have LED at GPIO 47, so set GPIO 47. */
  REG32(GPSET1) = 1 << (47 - 32);
```

Now let us review an example of how to turn on the LED (RPi 1 or 2) using the GPIO interface. For each GPIO there is a set and a clear register which uses one bit per GPIO to turn it on (GPSET) or off (GPCLR).

```
#include <system.h>
#include <board.h>

/*...................................................................*/
/* Global Function Definitions                                       */
/*...................................................................*/

/*...................................................................*/
/*        main: Application Entry Point                              */
/*                                                                   */
/*     Returns: Exit error                                           */
/*...................................................................*/
int main(void)
{
  unsigned int i, reg;

  /*
  ** GPPUD can be 0 (disable pull up/down)
  ** (1 << 0) enable pull down (low)
  ** (1 << 1) enable pull up (high)
  */

  /* Set pull up (high) for LED GPIO 47. */
  REG32(GPPUD) = (1 << 1);
  /* Loop to wait until GPPUD assignment persists. */
  for (i = 0; i < TOGGLE_LOOP_CNT / 1000; ++i)
    select = REG32(GPPUD);

  /* Assign GPPUD settings to GPIO 47. */
  REG32(GPPUDCLK1) = (1 << (47 - 32)); /* GPIO 47 */

  /* Read back GPPUDCLK1 in a loop to stall/hold the change above. */
  for (i = 0; i < TOGGLE_LOOP_CNT / 1000; ++i)
    reg = REG32(GPPUDCLK1);

  /* RPI 1 has LED at GPIO 47, so set GPIO 47. */
  REG32(GPSET1) = 1 << (47 - 32);

  return 0;
}
```

To turn the LED off, replace the code comment with “turn on LED” with this.

```
  /* RPI 1 has LED at GPIO 47, so clear GPIO 47. */
  REG32(GPCLR1) = 1 << (47 - 32);
```

If you build and execute the example above it may or may not work, depending the version of RPi hardware and even the version of the boot up software on the microSD card. There is still one piece of the puzzle missing to ensure it will always work for all RPi revisions and boot configurations.

4.6 Peripheral and GPIO sharing with Function Select

Many times a hardware system has more peripherals than can be made available within the hardware address space. In these cases, a level select is used. This means a register acts as a selector and when modified can select a different set of peripherals and/or GPIOs per level, allowing different hardware configurations of the GPIO pins. The level select interface is used by software to access different peripherals with software that all share the same hardware address space. There is typically a default level selected at power on and software needs to change the level select infrequently. On the RPi, these level select registers are referred to as GPio Function Select registers, or GPFSELn registers.

Let use review the GPIO alternate functions that can be configured with the function or level select registers. In BMC2835-ARM-Peripheral.pdf, page 102 and 103, is a nice table showing all the GPIOs and their six (6) different alternate functions for each GPIO. We know from the GPFSELn definition on page 91 of the BCM2835 documentation that each GPIO is represented in the function select registers as 3 bits each, or the integer value 0 through 7. On page 92 there is this breakdown of the 3 bits.

```
000 = GPIO Pin 9 is an input
001 = GPIO Pin 9 is an output
100 = GPIO Pin 9 takes alternate function 0
101 = GPIO Pin 9 takes alternate function 1
110 = GPIO Pin 9 takes alternate function 2
111 = GPIO Pin 9 takes alternate function 3
011 = GPIO Pin 9 takes alternate function 4
010 = GPIO Pin 9 takes alternate function 5
```

The RPi hardware defines/allocates each General Purpose Input Output (GPIO) line with three (3) configuration bits, so each 32 bit function select register holds the settings for up to ten (10) GPIOs (30 bits with 2 unused bits). To repeat, the 32 bit select register can contain settings for 32 / 3, or 10 GPIOs plus two unused bits. For example, on the RPi with five level select registers (GPFSEL0, GPFSEL1, GPFSEL2, GPFSEL3 and GPFSEL4) would be able to control up to 50 GPIO pins. To access the GPIO 22 configuration will require using the level select register two (GPFSEL2). GPFSEL2 starts with the configuration for GPIO 20 at bit 0, so the configuration for GPIO would starts at bit six (6 = 2 x 3). Let us add the function select registers to the ‘board.h’ file now, above the definition for GPSET0.

```
/*
 * Base of peripheral memory map
*/
#if RPI == 1
#define RPI_BASE        0x20000000 /* RPi B+ */
#else
#define RPI_BASE        0x3F000000 /* RPi 2/3 */
#endif
#define GPIO_BASE       (RPI_BASE | 0x200000)

// GPIO function select (GFSEL) registers have 3 bits per GPIO
#define GPFSEL0         (GPIO_BASE | 0x0) // GPIO select 0
#define GPFSEL1         (GPIO_BASE | 0x4) // GPIO select 1
#define GPFSEL2         (GPIO_BASE | 0x8) // GPIO select 2
#define GPFSEL3         (GPIO_BASE | 0xC) // GPIO select 3
#define GPFSEL4         (GPIO_BASE | 0x10)// GPIO select 4
#define   GPIO_INPUT         (0 << 0) // GPIO is input      (000)
#define   GPIO_OUTPUT        (1 << 0) // GPIO is output     (001)
#define   GPIO_ALT0          (4)      // GPIO is Alternate0 (100)
#define   GPIO_ALT1          (5)      // GPIO is Alternate1 (101)
#define   GPIO_ALT2          (6)      // GPIO is Alternate2 (110)
#define   GPIO_ALT3          (7)      // GPIO is Alternate3 (111)
#define   GPIO_ALT4          (3)      // GPIO is Alternate4 (011)
#define   GPIO_ALT5          (2)      // GPIO is Alternate5 (010)

// GPIO SET/CLEAR registers have 1 bit per GPIO
#define GPSET0          (GPIO_BASE | 0x1C) // set0 (GPIO 0 - 31)
#define GPSET1          (GPIO_BASE | 0x20) // set1 (GPIO 32 - 63)
#define GPCLR0          (GPIO_BASE | 0x28) // clear0 (GPIO 0 - 31)
#define GPCLR1          (GPIO_BASE | 0x2C) // clear1 (GPIO 32 - 63)

// GPIO Pull Up and Down Configuration registers
#define GPPUD           (GPIO_BASE | 0x94)
#define   GPPUD_OFF       (0 << 0)
#define   GPPUD_PULL_DOWN (1 << 0)
#define   GPPUD_PULL_UP   (1 << 1)
```

```
#define GPPUDCLK0        (GPIO_BASE | 0x98)
#define GPPUDCLK1        (GPIO_BASE | 0x9C)
```

To clarify, if GPIO's are numbered starting from 0, and select registers are numbered starting from 0, GPIO's 0 through 9 are in select register 0, or GPFSEL0. Then GPIO's 10 through 19 would be in select register 1 or GPFSEL1. Below is a C source code example of assigning the fourth level select register (GPFSEL4) to configure GPIO 47. With GPFSEL4 now defined to a valid address in the hardware memory map this will configure GPIO 47. Noting the definitions above, for value of the GPFSEL4 if configured for GPIO 47 to an output (GPIO_OUTPUT) is to set bit 0, or (1 << 0). Since GPIO 47 is a simple LED it should be configured as an 'output' GPIO.

Reminder

Do not directly use the "or equal" (|=) or "and equal" (&=) assignment operations with the REG32() macro.

```
/* Enable LED. 3 bits per GPIO so 10 GPIOs per select register. */
/* GPIO 47 is 7th register in GPFSEL4, so 7 * 3 bits or bit 21. */
select = REG32(GPFSEL4);

/* 3 bits per GPIO, input (0), output (1) and alternate select */
/* 0 through 5. */

/* Configure the LED (GPIO 47) starting at bit 21, as output (1). */
select |= (GPIO_OUTPUT << 21);
REG32(GPFSEL4) = select;
```

The above source code first copies or saves the value of the GPIO select register to the variable 'select'. Then 'select' is OR'ed with the value one (1) shifted twenty one (21) times (1 << 21). This second assignment exactly sets bit 21 high, leaving the rest of the variable 'select' the same. The expression (1 << 23) is equal to the binary number 100000000000000000000, but is more understandable to the hardware documentation reader when represented as the above expression of bit offsets. The last statement uses the REG32() macro to assign the new value to the hardware register in the memory map.

Each GPIO has three bits used for control which can be assigned a value of 0 through 7. The previous example works great only if all 3 bits of this GPIO are already zero. To create reliable system software it is often required to first clear the bits in question, before assigning a new value. Remember that the C language allows various assignment operations, such as +=, -=, |= and &=. These operations can be used to replace repeating the assigned variable. So 'select += ' can replace 'select = select +'. Below is the improved solution, which first clears and then assigns the correct GPIO select to enable the LED at GPIO 47 as an 'output' GPIO.

```
/* Enable LED. 3 bits per GPIO so 10 GPIOs per select register. */
/* GPIO 47 is 7th register in GPFSEL4, so 7 * 3 bits or bit 21. */

/* Clear the 3 bit range (7) starting at bit 21 */
select = REG32(GPFSEL4);
select &= ~(7 << 21);

/* 3 bits per GPIO, input (0), output (1) and alternate select */
/* 0 through 5. */

/* Configure the LED (GPIO 47) starting at bit 21, as output (1). */
select |= (GPIO_OUTPUT << 21);
REG32(GPFSEL4) = select;
```

4.7 Turn on or off the LED

Remember this statement from last chapter, "when main() finishes, the program will end and the hardware will halt". This is probably not what we want if our intention is to turn on the LED and leave it on. For the examples below we can achieve a wait by looping to assign the level select over and over. This should do nothing practical except cause a wait in the application, as this level select for the GPIO is already assigned.

```
/*....................................................................*/
/*       main: Application Entry Point                                */
/*                                                                    */
/*    Returns: Exit error                                             */
/*....................................................................*/
int main(void)
{
  unsigned int i, select;

  /*
  ** Enable LED. 3 bits per GPIO so 10 GPIOs per select register.
  ** GPIO 47 is 7th register in GPFSEL4, so 7 * 3 bits or bit 21
  */

  /* Clear the 3 bit range (7) starting at bit 21 */
  select = REG32(GPFSEL4);
  select &= ~(7 << 21);

  /* 3 bits per GPIO, input (0), output (1) and alternate select
   * 0 through 5. */

  /* Configure the LED (GPIO 47) as an output (1) */
  select |= GPIO_OUTPUT << 21;
  REG32(GPFSEL4) = select;

  /*
  ** GPPUD can be 0 (disable pull up/down)
  ** (1 << 0) enable pull down (low)
  ** (1 << 1) enable pull up (high)
  */

  /* Set pull up (high) for LED GPIO 47. */
  REG32(GPPUD) = GPPUD_PULL_UP;

  /* Read back the GPFSEL4 in a loop to stall/hold the change above. */
  for (i = 0; i < 5000; ++i)
```

```
    reg = REG32(GPFSEL4);

  /* Assign GPPUD settings to GPIO 47. */
  REG32(GPPUDCLK1) = (1 << (47 - 32)); /* GPIO 47 */

  /* Read back GPFSEL4 in a loop to stall/hold the change above. */
  for (i = 0; i < 5000; ++i)
    reg = REG32(GPFSEL4);

  /* Set GPIO 47 to turn on LED. */
  REG32(GPSET1) = 1 << (47 - 32);

  /* Set GPIO 47 to turn off LED. */
//  REG32(GPCLR1) = 1 << (47 - 32);

  /* Loop forever assigning the select register (doing nothing). */
  for (;;)
    REG32(GPIO_SELECT4) = select;

  return 0;
}
```

Compile the source code below and compile, the application should turn on the LED and then wait forever. Change the source code to comment out turning on the LED and uncomment the code that turns the LED off. After a recompile the application should turn off the LED and wait forever when executed. Change the source to not wait after enabling the LED (remove the loop to assign the level select and the end), and return from main() right after turning on or off the LED. By carefully observing the LED immediately upon execution, can you tell during execution between the application that turns the LED on and exits vs. the one that turns off the LED?

4.8 Blinking LED

Using the previous examples, create an additional application that turns the LED on, loops for a while to pause, and then turns the LED off, looping again to pause.

Surround this code with another loop to run forever to create an application that will blink the LED on and off. To do this correctly requires adjusting the loop count until the LED on and off patterns are visible. The specific values depend on the hardware and compiler and this exercise is to use trial and error to figure how many times to loop between LED toggles before the LED flashing is visible.

Do not be impatient during test execution as the TOGGLE_LOOP_CNT may be too large for the hardware, etc. For example, an LED that changes after 10 minutes (or 60) can be fine-tuned by lowering the TOGGLE_LOOP_CNT to blink every second or so. It is also important that the loops access a volatile hardware register, and return the value from main(), otherwise the compiler might optimize the loop away entirely if it determines the loop is not needed for the application.

Notice that this version is the final version for this chapter and contains code for all RPI versions 1, 2, 3 and 4 hardware. The difference between RPI's is the GPIO used for the LED. Reviewing the code and comparing the differences between 'RPI == 3' code is a good way to understand the GPIO interface. RPI 4 uses GPIO 42, RPI 3 uses GPIO 29 and the other RPI's use GPIO 47 for the activity LED. The configuration of the GPIO for pull up is the same regardless of the GPIO number used.

```
/*...................................................................*/
/* Configuration                                                     */
/*...................................................................*/
/*
 * Configure number of loops reading HW register to wait one second
*/
#define TOGGLE_LOOP_CNT  5000000 /* 5MHz is about 1 second on B+ */
                                 /* and -O2 compiler optimizations. */

/*...................................................................*/
/* Global Function Definitions                                       */
/*...................................................................*/

int main(void)
{
  unsigned int i, select;

#if RPI == 4
```

```
  /* GPIO 42 is 2nd register in GPFSEL4, so 2 * 3 bits or bit 6. */
  /* Clear the 3 bit range (7) starting at bit 6 */
  select = REG32(GPFSEL4);
  select &= ~(7 << 6);

  /* Configure the LED (GPIO 42) starting at bit 6, as output (1). */
  select |= (GPIO_OUTPUT << 6);
  REG32(GPFSEL4) = select;
#elif RPI == 3
  /*
  ** Enable LED. 3 bits per GPIO, so 10 GPIOs per select register means
  ** GPIO 29 is select register two number 9. 3 bits per GPIO so 9
  ** starts at bit 27.
  */

  /* Clear the 3 bit range (7) starting at bit 27 */
  select = REG32(GPFSEL2);
  select &= ~(7 << 27);

  /* 3 bits per GPIO, input (0), output (1) and alternate select */
  /* 0 through 5. */

  /* Configure the LED (GPIO 29) starting at bit 27, as output (1). */
  select |= (GPIO_OUTPUT << 27);
  REG32(GPFSEL2) = select;
#else
  /* Enable LED. 3 bits per GPIO so 10 GPIOs per select register. */
  /* GPIO 47 is 7th register in GPFSEL4, so 7 * 3 bits or bit 21. */

  /* Clear the 3 bit range (7) starting at bit 21 */
  select = REG32(GPFSEL4);
  select &= ~(7 << 21);

  /* 3 bits per GPIO, input (0), output (1) and alternate select */
  /* 0 through 5. */
  /* Configure the LED (GPIO 47) starting at bit 21, as output (1). */
  select |= (GPIO_OUTPUT << 21);
  REG32(GPFSEL4) = select;
#endif

  /* GPPUD - GPio Pin Up Down configuration */
  /*   (0) disable pull up and pull down to float the GPIO */
  /*   (1 << 0) enable pull down (low) */
  /*   (1 << 1) enable pull up (high) */

  /* Always pull up (high) for LEDs as they require voltage. */
  REG32(GPPUD) = GPPUD_PULL_UP;

  /* Loop to wait until GPPUD assignment persists. */
  for (i = 0; i < TOGGLE_LOOP_CNT / 1000; ++i)
    select = REG32(GPFSEL4);

#if RPI == 4
  /* Push GPPUD settings to GPPUDCLK1 GPIO 42. */
  REG32(GPPUDCLK1) = (1 << (42 - 32)); /* GPIO 42 */
#elif RPI == 3
  /* Push GPPUD settings to GPPUDCLK0 GPIO 29. */
  REG32(GPPUDCLK0) = (1 << 29); /* GPIO 29 */
#else
  /* Push GPPUD settings to GPPUDCLK1 GPIO 47. */
  REG32(GPPUDCLK1) = (1 << (47 - 32)); /* GPIO 47 */
#endif

  /* Loop to wait until GPPUD clock assignment persists. */
  for (i = 0; i < TOGGLE_LOOP_CNT / 1000; ++i)
    select = REG32(GPFSEL4);

  /* Loop turning the activity LED on and off. */
  for (;;)
  {
    /* Turn on the activity LED. */
#if RPI == 4
    /* RPI 4 has LED at GPIO 42, so set GPIO 42. */
    REG32(GPSET1) = 1 << (42 - 32);
#elif RPI == 3
    /* RPI 3 has LED at GPIO 29, so set GPIO 29. */
    REG32(GPSET0) = 1 << 29;
#else
    /* Other RPIs have LED at GPIO 47, so set GPIO 47. */
    REG32(GPSET1) = 1 << (47 - 32);
#endif

    // Loop to wait a bit
    for (i = 0; i < TOGGLE_LOOP_CNT; ++i) /* loop to pause LED on */
      select = REG32(GPFSEL4);

    /* Turn off the activity LED. */
#if RPI == 4
    /* RPI 4 has LED at GPIO 42, so clear GPIO 42. */
    REG32(GPCLR1) = 1 << (42 - 32);
#elif RPI == 3
    /* RPI 3 has LED at GPIO 29, so clear GPIO 29. */
    REG32(GPCLR0) = 1 << 29;
#else
    /* Other RPIs have LED at GPIO 47, so clear GPIO 47. */
    REG32(GPCLR1) = 1 << (47 - 32);
#endif

    // Loop to wait a bit
    for (i = 0; i < TOGGLE_LOOP_CNT; ++i) /* loop to pause LED off */
      select = REG32(GPFSEL4);
  }
  return select;
}
```

Compile and execute this code and the LED should blink on and off for the RPi executing the software. Note that the TOGGLE_LOOP_CNT defined above is for the

RPi B+ and that this loop count is used for the hold wait when configuring the GPIO pull up and pull down registers. Congratulations on your success! It was not easily earned but has laid the foundation for all things that follow.

4.9 GDB and OpenOCD

The GCC ecosystem has GDB (GNU De-Bugger) for **debugging** executables over a remote JTAG connection to the RPI. On PC development systems it is required to purchase, configure and use a JTAG controller that supports the RPI (ARM) hardware and Windows/Linux OS. JTAG controller configuration procedures are product specific and beyond the scope of this document. Tutorials exist that demonstrate JTAG debugging with the RPI using different off the shelf commercial JTAG compatible controllers. For Linux development PCs, especially for an RPI development PC running Raspbian, there is a unique solution that allows us to use OpenOCD and wire the GPIOs directly between the RPI's.

Regardless of the JTAG adapter and physical connection between the development PC and the **remote target** RPI, the development PC must have the bare metal (arm-none-eabi) version of GDB installed. On Linux this can be accomplished with the following command.

```
$ sudo apt-get install gdb-arm-none-eabi
```

OpenOCD, or the Open On-Chip Debugger, is an open source tool that supports JTAG and GDB. A BCM2835 GPIO JTAG bit bang driver was created that allows an RPI development PC to use the GPIOs on the RPI board to bit bang a JTAG controller. This effectively turns the development PC into a JTAG controller at the same time it runs Raspbian. This BCM2835 driver for OpenOCD is still in development but is functional enough to be highly recommended as it requires no additional hardware to purchase. The BCM2835 driver is not included in the OpenOCD official release but can be built into the project at compile time. The following procedure will download the OpenOCD source tree, configure it for the BCM2835 GPIO driver, build the OpenOCD software and then install it on the Raspbian development PC.

First, be sure your Raspbian development PC is up to date.

```
$ sudo apt-get update
$ sudo apt-get upgrade -y
$ sudo apt-get dist-upgrade -y
```

Next be sure the dependencies necessary to build OpenOCD are installed before using Git to clone OpenOCD and prepare it for building.

```
$ sudo apt-get install git autoconf libtool
$ sudo apt-get install pkg-config libusb-1.0-0 libusb-1.0-0-dev
$ git clone git://git.code.sf.net/p/openocd/code openocd
$ cd openocd
$ ./bootstrap
```

Now configure, make and install the OpenOCD software on the Raspbian development PC.

Information

The following instructions were tested and are known to work on an RPI development PC running Raspbian.

```
$ ./configure --enable-bcm2835gpio --enable-sysfsgpio
$ make
$ sudo make install
```

4.10 Wiring and Configuring JTAG

With the OpenOCD BCM2835 GPIO JTAG controller installed from the previous chapter, we next need to wire together the RPI development PC with the **remote system** RPI to enable debugging. The JTAG GPIOs on the RPI are defined beginning with ARM_ in Section 6.2 (page 102) of the BCM2835-ARM-Peripherals.pdf document. This page shows the complete GPIO map for the Raspberry Pi, including all the alternate functions. The JTAG pins are designated through alternate 4 (Alt 4) of GPIO 22 (ARM_TRST), 23 (ARM_RTCK), 24 (ARM_TDO), 25 (ARM_TCK), 26 (ARM_TDI) and 27 (ARM_TMS). These GPIOs map to pin numbers and the picture shows the all the connections between the two RPi's exposed pins. I the picture the development PC is on top while the **remote systems**, or target to be debugged, is on the bottom.

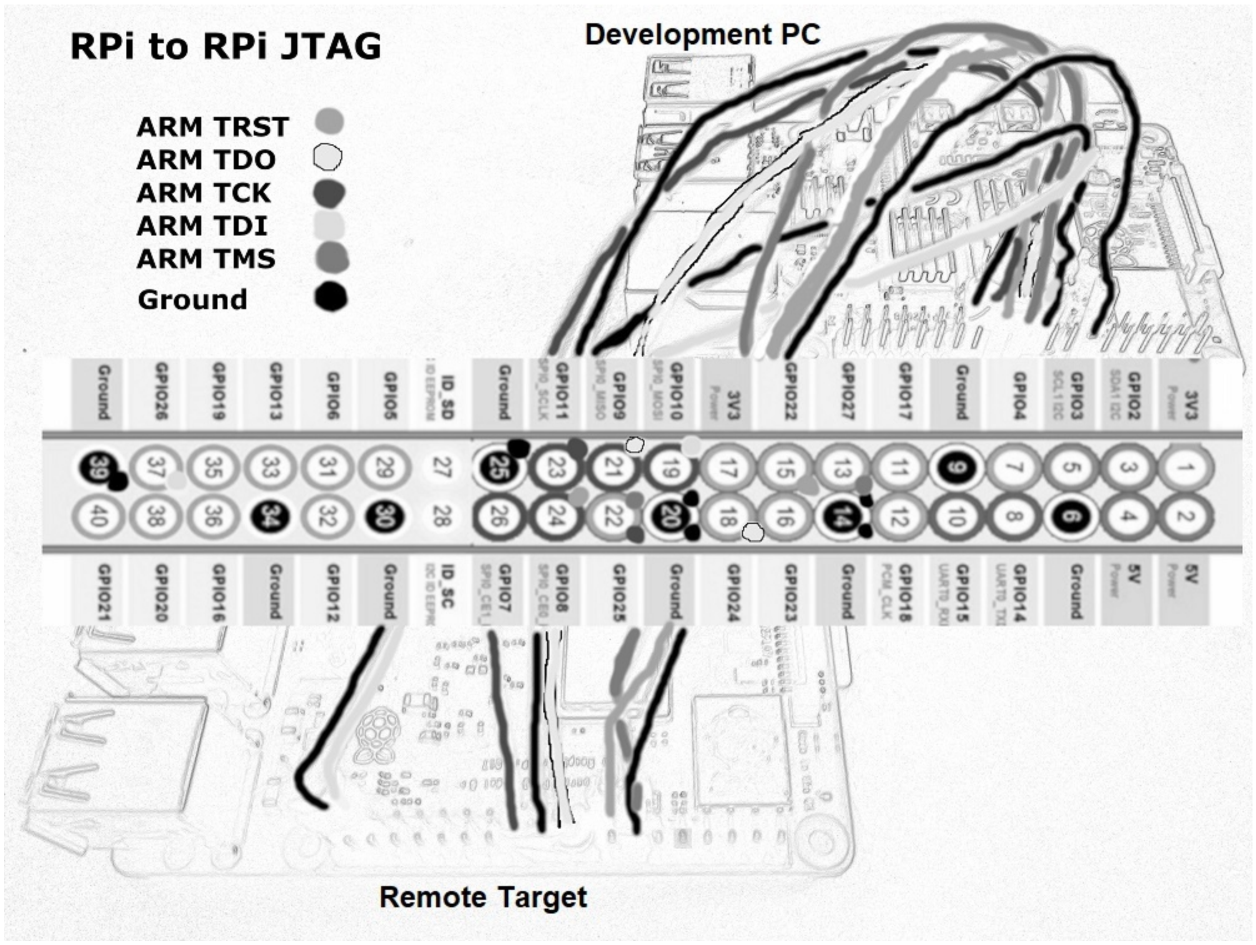


Fig 4: Raspberry Pi GPIO pinout

For the RPI development PC running Raspbian and using the OpenOCD bit bang JTAG controller, the default configuration file uses the following RPI GPIOs for the JTAG signals, GPIO 7 (ARM_TRST), ? (ARM_RTCK), 9 (ARM_TDO), 11 (ARM_TCK), 10 (ARM_TDI) and 25 (ARM_TMS). The following table shows the JTAG wiring between the RPI development PC (left) and the **remote target** RPI (right).

Fig 5: JTAG pin map for RPI development PC to RPI target

GPIO	Pin	Pin	GPIO	JTAG Signal	Color
7	24	15	22	ARM_TRST	Grey
?	?	16	23	ARM_RTCK*	-
9	21	18	24	ARM_TDO	Outline
11	23	22	25	ARM_TCK	Almost Black
10	19	37	26	ARM_TDI	Light Grey
25	22	13	27	ARM_TMS	Dark Grey
-	14	14	-	Ground	Black
-	20	39	-	Ground	Black
-	25	20	-	Ground	Black

Warning

Both RPIs should be powered down before connecting the JTAG GPIOs.

The **target system** software must configure these GPIOs to use select alternate 4 and then physically connect the GPIOs above to the development system PC. The following code needs to be added to main.c in order to select alternate four (4) operation of the GPIO pins to enable JTAG on the **remote system**.

```
/*
 * Enable the JTAG GPIOs
*/

/* Disable pull up/down for the next configured GPIO. */
REG32(GPPUD) = GPPUD_OFF;

/* Loop to wait until GPPUD assignment persists. */
for (i = 0; i < TOGGLE_LOOP_CNT / 1000; ++i)
  select = REG32(GPFSEL4);

// Apply to all the JTAG GPIO pins
REG32(GPPUDCLK0) = (1 << 22) | (1 << 23) | (1 << 24) | (1 << 25) |
                   (1 << 26) | (1 << 27);
```

```
/* Loop to wait until GPPUD clock assignment persists. */
for (i = 0; i < TOGGLE_LOOP_CNT / 1000; ++i)
  select = REG32(GPFSEL4);

// Select level alternate 4 to enable JTAG
select = REG32(GPFSEL2);
select &= ~(7 << 6); //gpio22
select |= GPIO_ALT4 << 6; //alt4 ARM_TRST
select &= ~(7 << 9); //gpio23
select |= GPIO_ALT4 << 9; //alt4 ARM_RTCK
select &= ~(7 << 12); //gpio24
select |= GPIO_ALT4 << 12; //alt4 ARM_TDO
select &= ~(7 << 15); //gpio25
select |= GPIO_ALT4 << 15; //alt4 ARM_TCK
select &= ~(7 << 18); //gpio26
select |= GPIO_ALT4 << 18; //alt4 ARM_TDI
select &= ~(7 << 21); //gpio27
select |= GPIO_ALT4 << 21; //alt4 ARM_TMS
REG32(GPFSEL2) = select;
```

Information

Some versions of RPI and boot code may interfere with the JTAG signals. To ensure the RPI boots with JTAG support edit the /boot/config.txt of the RPI executing the JTAG code above and add enable_jtag_gpio=1 to the end of the file.

Finally double check all the wiring and execute the created software on the **remote system** by copying the kernel.img file to the SD card boot folder as previously described in detail in Chapter 3. With the latest LED application executing and the JTAG wired to the **remote system**, it is now time to test the OpenOCD bit bang JTAG controller. Create file boards/rpi/openocd_rpi_jtag.cfg and add the following lines of configuration.

```
#
# Do not forget the ground connections
#

interface bcm2835gpio

bcm2835gpio_peripheral_base 0x3F000000

# RPI 2 (900 MHz clock)
bcm2835gpio_speed_coeffs 146203 36
# RPI 3 (1200 MHz clock)
#bcm2835gpio_speed_coeffs 194938 48

# Each of the JTAG lines need a gpio number set: tck tms tdi tdo
# RPI header pin numbers: 23 22 19 21
bcm2835gpio_jtag_nums 11 25 10 9

# RPI to RPI has trst pins only
bcm2835gpio_trst_num 7
reset_config trst_only trst_open_drain
#reset_config none
```

This configuration tells the OpenOCD driver the information it needs to control the JTAG signals on the GPIO lines that were previously connected. The final piece of the puzzle is to create the OpenOCD configuration file for the **remote system**. The following example is for the RPI 1, other configuration files exist in the companion files of the laboratory. Let us create boards/rpi/rpi1_jtag.cfg.

```
# Raspberry Pi 1

telnet_port 4444
gdb_port 3333

transport select jtag
adapter_khz 125
jtag_ntrst_delay 400

if { [info exists CHIPNAME] } {
   set  _CHIPNAME $CHIPNAME
} else {
   set  _CHIPNAME raspi
}

reset_config trst_only trst_open_drain
#reset_config none

if { [info exists CPU_TAPID ] } {
   set _CPU_TAPID $CPU_TAPID
} else {
   set _CPU_TAPID 0x07b7617F
}
jtag newtap $_CHIPNAME arm -irlen 5 -expected-id $_CPU_TAPID

set _TARGETNAME $_CHIPNAME.arm
target create $_TARGETNAME arm11 -chain-position $_TARGETNAME

$_TARGETNAME configure -event reset-assert-post { gdbinit }
$_TARGETNAME configure -event gdb-attach { halt }
```

Now execute OpenOCD, configuring it at run time with the two configuration files.

```
$ sudo openocd -f ./openocd_rpi_jtag.cfg -f ./rpi1_jtag.cfg
Open On-Chip Debugger 0.10.0+dev-00809-g7ee61869 (2019-05-11-00:02)
Licensed under GNU GPL v2
For bug reports, read
  http://openocd.org/doc/doxygen/bugs.html
BCM2835 GPIO: peripheral_base = 0x3f000000
BCM2835 GPIO: speed_coeffs = 146203, speed_offset = 36
BCM2835 GPIO config: tck = 11, tms = 25, tdi = 10, tdo = 9
BCM2835 GPIO config: trst = 7
trst_only separate trst_open_drain
adapter speed: 125 kHz
jtag_ntrst_delay: 400
trst_only separate trst_open_drain
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : BCM2835 GPIO JTAG/SWD bitbang driver
Info : JTAG only mode enabled (specify swclk and swdio gpio to add SWD
Info : clock speed 125 kHz
Info : JTAG tap: raspi.arm tap/device found: 0x07b7617f (mfg: 0x0bf (Br
Info : found ARM1176
Info : raspi.arm: hardware has 6 breakpoints, 2 watchpoints
Info : Listening on port 3333 for gdb connections
```

If you see the output below, or something similar, congratulations you are now ready to debug with GDB. Leave this OpenOCD console running and open a new console window in Raspbian. In this new console let us install and execute GDB now.

```
$ sudo apt-get install gdb-arm-none-eabi
$ arm-none-eabi-gdb led.elf
GNU gdb (7.10-1+9) 7.10
Copyright (C) 2015 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gp
...
Reading symbols from ./led.elf...done.
(gdb)
```

Now at the GDB prompt, let us try the OpenOCD JTAG connection with the 'target remote' command.

```
(gdb) target remote localhost:3333
0x00008068 in ?? ()
(gdb)
```

If you see this it is great news. The final step is to recompile the LED application with debugging symbols in it. In build.sh add the -ggdb option to the list of options to arm-none-eabi-gcc command (arm-none-eabi-gcc -c -ggdb …). This enables debug symbols so the code can be debugged. Now rebuild the LED and do the SD card dance to get the new software to execute again on the **remote system** RPI. OpenOCD in the other console should be spitting out errors once the **remote system** is power cycled. Ctl-C in the OpenOCD window to stop the program until the **remote system** is executing again and then issue the command to start OpenOCD again.

With the **remote system** now executing the latest software, and OpenOCD JTAG controller running successfully in another console window of the Raspbian development PC, let us retry GDB.

```
$ arm-none-eabi-gdb led.elf
...
Reading symbols from ./led.elf...done.
(gdb) target remote :3333
Remote debugging using :3333
main () at main.c:104
104       for (i = 0; i < TOGGLE_LOOP_CNT; ++i) /* loop to pause LED off
(gdb)
```

It would dramatically speed of the incremental build and test process if there was a way to load the executable without removing the SD card. It so happens that GDB can load ELF (.elf) files into system RAM for execution. We must first use GDB to connect over JTAG and stop execution of the current system software. Then we can issue the 'load led.elf' command. First, follow the previous instructions to open a new terminal and run OpenOCD on the development PC running Raspbian. Then build your application and open it with GDB, as below.

```
(gdb) load led.elf
Loading section .text, size 0x390 lma 0x8000
Start address 0x8000, load size 912
Transfer rate: 7296 bits in <1 sec, 912 bytes/write.
(gdb) continue
Continuing.
```

Now after 'continuing' in GDB, our newly created executable is running on the remote target. With this procedure we no longer need to swap out the SD card and copy the new kernel.img file to execute and test our new software creations. As long as the SD card is running in a loop and has enabled the JTAG GPIO's to select alternate 4, GDB and OpenOCD can be used from the Raspbian development PC over the physical GPIO wires. For GDB to connect remember to use the 'target remote' command ((gdb) target remote :3333) and then load ((gbd) load led.elf) the new executable. Then the continue ((gdb) continue) command will execute the newly loaded executable.

If this is working congratulations. If not, recheck the connections and source code as JTAG is fickle. If everything is not perfect, nothing will work. The quality of wire has been shown to make a difference with JTAG and the speed can be increased with the OpenOCD adapter_khz setting if high quality wire and connectors are used.

Reminder

Some combinations of RPI and boot code may interfere with the JTAG GPIOs. To ensure the RPI boots with JTAG support edit the /boot/config.txt of the RPI executing the JTAG code above and add enable_jtag_gpio=1 to the end of the file.

4.11 Debugging with GDB

GDB supports an interactive Text User Interface, or TUI. Let us explore the 'step' and 'breakpoint' commands. The 'step' commands take no options and advances the execution to the next line of source code. The 'break' command takes a function name or file name with a colon ':' and line number.

```
$ arm-none-eabi-gdb -tui led.elf

   ┌─main.c─────────────────────────────────────────────────────────┐
   │185                                                             │
   │186         /* Turn off the activity LED. */                    │
   │187     #if RPI == 4                                            │
   │188         /* RPI 4 has LED at GPIO 42, so clear GPIO 42. */   │
   │189         REG32(GPCLR1) = 1 << (42 - 32);                     │
   │190     #elif RPI == 3                                          │
   │191         /* RPI 3 has LED at GPIO 29, so clear GPIO 29. */   │
   │192         REG32(GPCLR0) = 1 << 29;                            │
   │193     #else                                                   │
   │194         /* Other RPIs have LED at GPIO 47, so clear GPIO 47. *│
   │195         REG32(GPCLR1) = 1 << (47 - 32);                     │
   │196     #endif                                                  │
   │197                                                             │
   │198         // Loop to wait a bit                               │
  >│199         for (i = 0; i < TOGGLE_LOOP_CNT; ++i) /* loop to pause│
   └────────────────────────────────────────────────────────────────┘
remote Remote target In: main                          L199  PC: 0x80cc
Reading symbols from led.elf...done.
(gdb) target remote :3333
Remote debugging using :3333
main () at main.c:200
(gdb) next
(gdb) break main.c:101
Breakpoint 1 at 0x8014: file main.c, line 101.
(gdb)
```

Stepping through our source code creations with a debugger and using **breakpoints** is a great way to reinforce our knowledge and build confidence in our emerging abilities as a systems software creator.

OpenOCD and GDB work better on some versions of RPI hardware than others. From the authors tests, the best RPI to use for a development PC is an RPI 2 or higher as these have multiple cores for Linux to utilize. The best RPI to use for the target system is an RPI B+ as it has full GDB support over OpenOCD. The RPI 2 as a target system could not be debugged with OpenOCD and GDB, while the RPI 3 B+ had limited debugging functionality. This appeared to be because OpenOCD does not completely support these versions of Cortex processors yet, or possibly because the GDB used with OpenOCD is dated. Specifically the development RPI running OpenOCD 0.10.0 and GDB 7.10 connecting ('target remote :3333') to the RPI 3 B+ target system would not 'step' or 'next' but would allow **break points** to be set. However, once a **break point** hit the stack and variables would be inaccessible until the user removed the **break points**, continued and then broke into the **debugger** with Ctl-C. Below is an example of the commands used and responses received from GDB/OpenOCD.

```
(gdb) break LedOn
Breakpoint 1 at 0x8188: file ../../boards/rpi/board.c, line 126.
(gdb) continue
Continuing.
cannot read system control register in this mode

Breakpoint 1, LedOn () at ../../boards/rpi/board.c:126
(gdb) bt
#0  LedOn () at ../../boards/rpi/board.c:126
#1  0x00008220 in microsleep (microseconds=0) at ../../boards/rpi/boar
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
(gdb) delete breakpoints
Delete all breakpoints? (y or n) y
(gdb) continue
Continuing.
target rpi3.a53.0 was not halted when resume was requested
cannot read system control register in this mode
The target is not running when halt was requested, stopping GDB.

Program received signal SIGINT, Interrupt.
LedOn () at ../../boards/rpi/board.c:126
(gdb) bt
#0  LedOn () at ../../boards/rpi/board.c:126
#1  0x00008094 in main () at main.c:76
(gdb)
```

Lab 5: System Timers

5.1 Why is a timer important?

The hardware timer is the foundation on which a systems engineer can create software for more complex hardware interfaces. A software timer has many uses when developing system software. Combined with the ability to turn on the LED, a timer interface can be used to blink detailed error codes with a sequence of LED flashes. When first bringing up new hardware it is important to verify and integrate the hardware timer into a software interface. Oftentimes a system problem is the result of a timer or timing problem. A reliable timer with an easy to use interface is the foundation of quality system software.

The Raspberry Pi has a memory mapped interface to a hardware clock running at microsecond, or 1MHZ, precision. Referencing the BCM2835-ARM-Peripheral.pdf document, section 12.1, this chapters goal is to understand the Raspberry Pi timer interface (12.1 System Timer Registers). This BCM2835 document, and others like it, can be confusing because they are authored for such a wide audience. From a systems software perspective the key to deciphering these documents is to focus on the software interface, or hardware registers exposed to software. The 32 bit timer status, as well as low and high count, are the only registers required to create a complete system interface for the hardware timer.

5.2 Configure board.h for Raspberry Pi timer

The RPi hardware has a fixed memory map configuration so the offset address of the peripherals has not yet changed across hardware revisions. The timer offset address is 0x003000, relative to the peripheral base address which depends on the RPI hardware version. Add this definition to board.h, just above the GPIO_BASE defined in the previous chapter, if it is not there already. This value was discovered in the Raspberry Pi hardware documentation, BCM2835-ARM-Peripherals.pdf page 172.

Also for readability, we are changing the name RPI_BASE to PERIPHERAL_BASE and using it when assigning the address for each hardware register.

```
#define TIMER_BASE      (PERIPHERAL_BASE | 0x003000)
```

Next we must use the TIMER_BASE to define the specific timer registers we will use to create our timer functions. Let use define these at the bottom of board.h now.

```
/*
 * Timer registers
*/
#define TIMER_CS        (TIMER_BASE | 0x00) // clock status
#define TIMER_CLO       (TIMER_BASE | 0x04) // clock low 32 bytes
#define TIMER_CHI       (TIMER_BASE | 0x08) // clock high 32 bytes
```

5.3 System Software Interface to LED

Using the LED source code from the previous chapter as an example, create a system software interface for the LED. The software interface should split the functionality in the Chapter 4 main.c into three global functions, BoardInit(), LedOn() and LedOff() and it should declare the functions global in system.h. These global functions are hardware specific so the functions themselves should be created in the file 'boards/rpi/board.c'. For BoardInit() we put the level select and GPIO configure code from the previous chapter that initializes the LED. The turning on and off of the LED should go in the global LedOn() and LedOff() functions. This should be the resulting file structure:

```
Lab5 Timed LED
    applications
        errorstatus
            main.c
    boards
        rpi
            board.c
            board.h
    include
        system.h
```

While reorganizing is good for the long run, it does make the compilation process a bit more complicated as header files and source files are now in different directories. After creating the the new files and directory structure, the build script will also need to be updated.

The updated ‘build.bat/build.sh’ is below. Note the option ‘-I’ adds an include file path which define the paths, relative to the application directory. It moves back two directories (‘../..’) to get to the ‘include’ or ‘boards’ directory since the build.bat script is executing now in the ‘applications/errorstatus’ directory.

```
#options -ffreestanding prevent linking in GNU stuff
#option -c compiles/assembles, -O2 optimizes

# RPI 2
# options -DRPI=2 for RPi 2
# RPI B+
#option -DRPI=1 for RPi B+

arm-none-eabi-gcc -c -DRPI=3 -ffreestanding -I. -I../../include -I../.\
./boards/rpi -o main.o main.c
arm-none-eabi-gcc -c -DRPI=3 -ffreestanding -I. -I../../include -I../.\
./boards/rpi -o ..\..\boards\rpi\board.o ..\..\boards\rpi\board.c

#Use ld to link with RPi memory map
arm-none-eabi-ld -T ../../boards/rpi/memory.map -o errorcodes.elf main\
.o ../../boards/rpi/board.o
```

```
#strip elf into binary and copy to kernel.img
arm-none-eabi-objcopy errorcodes.elf -O binary errorcodes.bin
cp errorcodes.bin kernel.img
```

Here are the generic board global function declarations to add to system.h.

```
/*
 * Board interface
*/
void BoardInit(void);
void LedOn(void);
void LedOff(void);
```

Add/move the code in main.c from past chapter to the new function BoardInit() within board.c. Create the file and BoardInit() function now if you have not already done so. Afterward, add the LedOn() and LedOff() functions below to the end of the board.c file.

```
/*..................................................................*/
/*      LedOn: Turn on the activity LED                             */
/*                                                                  */
/*..................................................................*/
void LedOn(void)
{
  /* Turn on the activity LED. */
#if RPI == 4
  /* RPI 4 has LED at GPIO 42, so set GPIO 42. */
  REG32(GPSET1) = 1 << (42 - 32);
#elif RPI == 3
  /* RPI 3 has LED at GPIO 29, so set GPIO 29. */
  REG32(GPSET0) = 1 << 29;
#else
  /* Other RPIs have LED at GPIO 47, so set GPIO 47. */
  REG32(GPSET1) = 1 << (47 - 32);
#endif
}

/*..................................................................*/
/*     LedOff: Turn off the activity LED                            */
/*                                                                  */
/*..................................................................*/
void LedOff(void)
{
  /* Turn off the activity LED. */
#if RPI == 4
  /* RPI 4 has LED at GPIO 42, so clear GPIO 42. */
  REG32(GPCLR1) = 1 << (42 - 32);
#elif RPI == 3
  /* RPI 3 has LED at GPIO 29, so clear GPIO 29. */
  REG32(GPCLR0) = 1 << 29;
#else
  /* Other RPIs have LED at GPIO 47, so clear GPIO 47. */
  REG32(GPCLR1) = 1 << (47 - 32);
#endif
}
```

Combine this change with the timer and sleep functions created in the main book to create an application that loops forever, alternating between turning the LED on for one second and off for one second. The timer functions created in the book require one modification, the TIMER_CLOCK defined is generic and must be changed to be RPi specific. Please change the code to use the RPI low timer register name TIMER_CLO instead of TIMER_CLOCK. Then create a new main.c as below that should blink the LED activity light upon execution.

```
/*
 * Loop, turning the led on and off.
*/
for (;;)
{
  LedOn();
  Sleep(1);
  LedOff();
  Sleep(1);
}
```

5.4 New application “errorcodes”

System software now exists to turn the LED on or off as well as to keep time. Using these functions, create an algorithm to turn the single LED on and off in a sequence that can represent a one byte value (0 - 255). The function should repeat the sequence repeatedly in a manner so that the observer will know when the sequence begins.

One way to do this is repeat the sequence of the error number. A long delay between repeats and a shorter delay between digits could allow a user to count two consecutive nibble values, allowing an entire byte to be communicated as an informative error code.

There are many ways to create this solution. The answer below splits the error code into two hexadecimal digits, or nibbles, and flashes the two numbers in the range of 0 through 15, one after the other with a short pause between. After both numbers are blinked, there is a longer pause before the sequence is repeated. Please review the code and comments now.

main.c

```
/*..................................................................*/
/*        main: Application Entry Point                             */
/*                                                                  */
/*     Returns: Exit error                                          */
/*..................................................................*/
int main(void)
{
  u8 i, code, tmp;

  /* Initialize the hardware. */
  BoardInit();

  /* Loop, testing different error codes in increments of one. */
  for (code = 1;; ++code)
  {
    /* Shift high order nibble of code into tmp and clear rest. */
    tmp = ((code >> 4) & 0xF);

    /* Loop, blinking LED the number of times of the nibble value. */
    for (i = 0; i < tmp; ++i)
    {
      LedOn();
      usleep(MICROS_PER_SECOND / 4); /* quarter second on */
      LedOff();
      usleep(MICROS_PER_SECOND / 4); /* quarter second off */
    }

    /* Wait one second to indicate to user the next nibble. */
    Sleep(1);

    /* Put low order nibble of code into tmp. */
    tmp = code & 0xF;

    /* Loop, blinking LED the number of times of the nibble value. */
    for (i = 0; i < tmp; ++i)
    {
      LedOn();
      usleep(MICROS_PER_SECOND / 4); /* quarter second on */
      LedOff();
      usleep(MICROS_PER_SECOND / 4); /* quarter second off */
    }

    /* Wait three seconds to indicate to user the next byte. */
    Sleep(3);
  }
  return 0;
}
```

Most of the code is straight forward and described clearly in the comments, however let us review the creation of the low and high order nibbles. The code 'code & 0xF' ANDs the error code variable 'code' with a nibble of all ones (0xF), causing the high order nibble to be removed from the byte and only the low order nibble to remain. The code '((code >> 4) & 0xF)' uses the right shift operator '>>' to shift the high order 4 bit nibble to the right 4 bits and into the low order position. It then does the same '& 0xF' to ensure only the low order nibble remains.

5.5 Using Make to create an executable binary

Make can greatly simplify and add reliability to the compiling and linking process. As the number of files to compile grows the more complicated the build.sh script becomes. The first step to using Make is to ensure it is installed on the development PC and install it if not. Open a system shell (Terminal or Command Prompt) and try the command 'make'. If the command is not found, Make must be installed. On Linux this can usually be performed with this command:

```
$ sudo apt-get install make
```

Windows users are encouraged to use the PowerShell, or install MinGW and MSYS which provides a system shell with many extra Unix commands including make (see chapter 2 lab). Please take the time now to install 'Make' and check that it is usable. With 'Make' installed, let us create a new file named 'Makefile' and copy the example

'Makefile' from the main book, section 5.4, into this file. Two options need to be added to the 'Makefile' for GCC to build correctly for the specific RPi hardware, the EXTRAS and ASFLAGS definitions. Also, the changes are different depending on whether the hardware is an RPi B+, 2 or 3. First let us review the RPi B+ changes, adding -DRPI=1 to the EXTRAS definition. These options were discussed in chapter 3 and used in the build shell script. Now we need to be sure to add them to the Makefile and enable the specific options for the RPi hardware we are creating software for.

```
EXTRAS = -DRPI=3 -ffreestanding
ASFLAGS =
```

Next let us review the RPi 2 changes needed to the base Makefile to reflect the difference in CPUs.

```
EXTRAS = -DRPI=2 -ffreestanding
ASFLAGS =
```

Now that Make is installed and the Makefile is configured for the RPi, let us review the new simplified process to create system software that is executable on bare metal. Here a single make command is used to compile and link the entire bare metal application. While not a huge time saver now, this will become increasingly helpful as the number of files grow. It is also very helpful to have the 'make clean' command available, to clean out the old compiled binaries so we can perform a fresh build. Alternatively to the procedure below, the software creator may use GDB and JTAG/OpenOCD to 'load' the newly created application and 'continue' to execute it, instead of copying to the SD card.

```
$ make
arm-none-eabi-gcc -c -Wall -O2 -DRPI=3 -ffreestanding -I. -o main.o ma\
in.c
arm-none-eabi-ld -T memory.map -o app.elf main.o
arm-none-eabi-objcopy app.elf -O binary app.bin
cp app.bin kernel7.img

$
[MOVE SD TO PC, E: DRIVE]
```

```
$ cp app.bin e:\kernel7.img
[DISCONNECT SD/USB IN OS]
[MOVE SD BACK TO RPI]
[POWER ON]
<* Program runs *>
```

Make checks time stamps of the source files so if a source code file (.c) has been modified more recently then the associated output file (.o), Make will compile this file. However, Make does not by default check for header file changes, so if a common header file changes it is very important to recompile all source code files. The easy way to do this is with the "make clean" command. Let us review the last example for this laboratory assignment that cleans the application and compiles it again from scratch.

It is not at all required to understand the entirety of the Makefile syntax. However, the INCLUDES definition is very important since it contains a list of file system paths that are to be searched, in order, for header files needed during the compilation process. Any header file in any of these directories will be available to a .c file during compilation, for example '#include <board.h>'. Otherwise a full path (absolute or relative to the application) path enclosed in quotes is needed when including header files '#include "../../include/board.h"' within a .c file.

Another very important definition is the OBJECTS definition which includes all the source objects that should be compiled to create the resulting executable. In this example we have created a new board.c file so it needs to be added to the Makefile. Since the first code to be executed is the first to be linked, main.c should be the first object in the OBJECTS list. This will be explained and expanded upon in more detail in the following chapters.

```
$ make clean
rm -f main.o
rm -f *.bin
rm -f *.elf
rm -f *.img

$ make
arm-none-eabi-gcc -c -Wall -O2 -DRPI=3 -ffreestanding -I. -I../../incl\
ude -I../../boards/rpi -o main.o main.c
arm-none-eabi-gcc -c -Wall -O2 -DRPI=3 -ffreestanding -I. -I../../incl\
ude -I../../boards/rpi -o ../../boards/rpi/board.o ../../boards/rpi/bo\
ard.c
arm-none-eabi-ld -T memory.map -o errorcodes.elf ../../boards/rpi/boar\
d.o main.o
arm-none-eabi-objcopy errorcodes.elf -O binary errorcodes.bin
cp errorcodes.bin kernel7.img
$
```

Let us review the completed Makefile now. Note that this Makefile includes RPi revision optimization flags for building and the EXTRAS and ASFLAGS lines need to be commented out and the existing commented lines used instead for the RPi 2/3.

```
#
# Makefile for errorcodes application
#

##
## Commands:
##
CP   = cp
RM   = rm
C = arm-none-eabi-gcc
CC   = arm-none-eabi-gcc
LINK   = arm-none-eabi-ld
PLINK= arm-none-eabi-objcopy
```

```
##
## Definitions:
##
APPNAME = app

# Define the RPI hardware version: 1 through 4
EXTRAS = -DRPI=3 -ffreestanding
ASFLAGS =

##Warnings about everything and optimize for size or speed (-Os or -O2)
#CFLAGS = -Wall -O2 $(EXTRAS)

##Warnings and debugging build, with GDB and no optimizations (-O0)
CFLAGS = -Wall -ggdb -O0 $(EXTRAS)
INCLUDES = -I. -I../../include -I../../boards/rpi

##
## Application
##
OBJECTS = main.o \
          ../../boards/rpi/board.o \

##
## Define build commands for the C and assembly source file types
##
.c.o:
  $(CC) -c $(CFLAGS) $(INCLUDES) -o $@ $<

.s.o:
  $(AS) $(ASFLAGS) -o $@ $<

##
## Targets
##
all:  object
```

```
object: $(OBJECTS)
  $(LINK) -T ../../boards/rpi/memory.map -o $(APPNAME).elf $(OBJECTS)
  $(PLINK) $(APPNAME).elf -O binary $(APPNAME).bin
  $(CP) $(APPNAME).bin kernel7.img

clean:
  rm -f $(OBJECTS)
  rm -f *.bin
  rm -f *.elf
  rm -f *.img
```

5.6 Complete 64 bit timer for RPi

In the companion book we introduced an algorithm that had software maintain the roll over of a single 32 bit time register. While that was a good exercise and is required on some hardware, the RPi includes two hardware registers, one for the low order 32 bits of the timer, and other for the high order 32 bits of the counter. Because of this there is no need for software to maintain the roll over. Let us upgrade our timer routines to use both low and high order timer registers and thus simplify the algorithm.

First let us be sure not to change the public interface so that usleep() remains the same, and the parameters and return values of TimerRegister() and TimeRemaining() are the same. Let us review the changes to TimerRegister and TimerRemaining() now.

```
/*...................................................................*/
/* TimerRegister: Register an expiration time                        */
/*                                                                   */
/*      Input: microseconds until timer expires                      */
/*                                                                   */
/*    Returns: resulting expiration time                             */
/*...................................................................*/
struct timer TimerRegister(u64 microseconds)
{
  struct timer tw;
  u64 now;
  /* Retrieve the current time from the 1MHZ hardware clock. */
  now = REG32(TIMER_CLO);  /* low order time */
  now |= ((u64)REG32(TIMER_CHI) << 32); /* high order time */

  /* Calculate and return the expiration time of the new timer. */
  tw.expire = now + microseconds;

  /* Return the created timer. */
  return tw;
}

/*..................................................................*/
/* TimerRemaining: Check if a registered timer has expired          */
/*                                                                  */
/*      Input: expire - clock time of expiration in microseconds    */
/*             unused - unused on RPi as it has 64 bit precision    */
/*                                                                  */
/*    Returns: Zero (0) or microseconds until timer expiration      */
/*..................................................................*/
u64 TimerRemaining(struct timer *tw)
{
  u64 now;

  /* Retrieve the current time from the 1MHZ hardware clock. */
  now = REG32(TIMER_CLO); /* low order time */
  now |= ((u64)REG32(TIMER_CHI) << 32); /* high order time */

  /* Return zero if timer expired. */
  if (now > tw->expire)
    return 0;

  /* Return time until expiration if not expired. */
  return tw->expire - now;
}
```

```
/*...................................................................*/
/*   TimerNow: Return the current time in clock ticks                */
/*                                                                   */
/*...................................................................*/
u64 TimerNow(void)
{
  u64 now;

  /* Retrieve the current time from the 1MHZ hardware clock. */
  now = REG32(TIMER_CLO); /* low order time */
  now |= ((u64)REG32(TIMER_CHI) << 32); /* high order time */

  /*
  ** Return the current time.
  */
  return now;
}
```

Lab 6: UART Peripheral

6.1 Why the UART?

The UART peripheral is a simple and versatile hardware peripheral available on the RPi and the next step to creating functional system software. A UART can send data at moderate speeds in either direction between the development PC and the RPi executing our software creations. A UART is commonly used for a terminal or command shell, so that the development PC can issue commands to execute on the other system and receive responses. It is the creation of this shell interface that is the goal of this chapter, as connecting the development PC to an RPi running a UART command shell allows many execution and diagnostic options to speed up future software development efforts. For example, error strings can be sent over the UART to be displayed on the development PC in order to speed up the discovery and resolution of future problems.

An easily expandable command line shell interface executing on the Raspberry Pi is a great tool to aid system software development. With the physical UART connection, the development PC can enter a shell environment and execute any commands the application has made available. Command results, along with logging and error messages are also possible. Let the journey be as interesting as the destination.

6.2 Raspberry Pi UART Register Address Map

Chapter 6 of the companion book created a generic UART system software interface designed for the common 16C650 compatible UART hardware interface. The Raspberry Pi primary UART is a PL011 UART which is partially 16C650 compatible. With some additional configuration, the UART interface created in the main book should be ready to run and test on the RPi. First, reference the BCM2835-ARM-Peripherals.pdf, Chapter 13 UART. In section 13.4 is the UART Address Map.

The first coding step is to add the UART base definition to board.h, after the TIMER_-BASE definition.

```
#define UART0_BASE      (PERIPHERAL_BASE | 0x201000)
```

The next goal is to review the UART Address Map and define names and address locations for the hardware registers in the memory map to represent software pointers, for example '#define UART0_DATA (UART0_BASE + 0x00)'. Please add UART0_ to each register name so it is recognized as a memory mapped register of the UART. For clarity, please rename RSRECR register defined in the BCM2835 document to UART0_RX_STATUS, FR to UART0_STATUS, LCRH to UART0_LINE_CTRL and CR to UART0_CONTROL. This makes the registers easier to read and understand, as well as portable to other hardware systems that do not use the BCM2835 syntax for register naming. These are also the names used in the main book so the code can be used with minimal changes. Let us review these defined register addresses now.

```
/*
 * UART (16C650) register map
*/
#define UART0_DATA       (UART0_BASE + 0x00)
#define UART0_RX_STATUS  (UART0_BASE + 0x04)
#define UART0_STATUS     (UART0_BASE + 0x18)
#define UART0_ILPR       (UART0_BASE + 0x20)
#define UART0_IBRD       (UART0_BASE + 0x24)
#define UART0_FBRD       (UART0_BASE + 0x28)
#define UART0_LINE_CTRL  (UART0_BASE + 0x2C)
#define UART0_CONTROL    (UART0_BASE + 0x30)
#define UART0_IFLS       (UART0_BASE + 0x34)
#define UART0_IMSC       (UART0_BASE + 0x38)
#define UART0_RIS        (UART0_BASE + 0x3C)
#define UART0_MIS        (UART0_BASE + 0x40)
#define UART0_ICR        (UART0_BASE + 0x44)
#define UART0_DMACR      (UART0_BASE + 0x48)
#define UART0_ITCR       (UART0_BASE + 0x80)
#define UART0_ITIP       (UART0_BASE + 0x84)
#define UART0_ITOP       (UART0_BASE + 0x88)
#define UART0_TDR        (UART0_BASE + 0x8C)
```

6.3 Raspberry Pi UART Register Details

The goal of this section is to analyze the registers and create sub definitions for each of the configuration and status bits that the system software needs to access. First let use review the register details for UART0_DATA (page 179 of BCM2835-ARM-Peripherals.pdf) and create these definitions for the 32 bit data register. The first 8 bytes are the data followed by error bits. The final definition combines all errors into a single definition DATA_ERROR so all errors can be checked at once. RX_DATA can be used with the 'And' (&) operation in C to extract the 8 bits of data from the 32 bit UART0_DATA register, for example 'data = REG32(UART_DATA) & RX_DATA'.

```
#define UART0_DATA       (UART0_BASE + 0x00)
#define   RX_DATA                (0xFF << 0)
#define   DR_FRAME_ERROR         (1 << 8)
#define   DR_PARITY_ERROR        (1 << 9)
#define   DR_BREAK_ERROR         (1 << 10)
#define   DR_OVERRUN_ERROR       (1 << 11)
#define     DATA_ERROR   (DR_FRAME_ERROR | DR_PARITY_ERROR | \
                          DR_BREAK_ERROR | DR_OVERRUN_ERROR)
```

Next we do the same with UART0_RX_STATUS (page 180 of BCM2835-ARM-Peripherals.pdf). The UART0_RX_STATUS defines four different errors and an RX_-ERROR define that is a combination of all four errors, similar to the DATA_ERROR definition above.

```
#define UART0_RX_STATUS  (UART0_BASE + 0x04)
#define   RX_FRAME_ERROR         (1 << 0)
#define   RX_PARITY_ERROR        (1 << 1)
#define   RX_BREAK_ERROR         (1 << 2)
#define   RX_OVERRUN_ERROR       (1 << 3)
#define     RX_ERROR     (RX_FRAME_ERROR | RX_PARITY_ERROR | \
                          RX_BREAK_ERROR | RX_OVERRUN_ERROR)
```

Next move on to the UART0_STATUS (page 181 of BCM2835-ARM-Peripherals.pdf). The UART status register contains important information, such as whether the UART Rx FIFO buffer is empty, which means there is no data available to read. Also, to avoid

Tx errors it is important for the system software to wait before sending data if the Tx FIFO is full. The UART status is the dashboard for the UART controller. The use of these UART registers is an important step in understanding most other hardware to software interfaces.

```
#define UART0_STATUS     (UART0_BASE + 0x18)
#define   CLEAR_TO_SEND          (1 << 0)
#define   BUSY                   (1 << 3)
#define   RX_FIFO_EMPTY          (1 << 4)
#define   TX_FIFO_FULL           (1 << 5)
#define   RX_FIFO_FULL           (1 << 6)
#define   TX_FIFO_EMPTY          (1 << 7)
```

Next is the UART0_LINE_CTRL (page 184 of BCM2835-ARM-Peripherals.pdf) register. The UART line control register contains the configuration options for the UART serial channel. It can be used to set the word size, parity and stop bits. While it is highly encouraged to leave size, parity and stop bits at the default configuration, the ENABLE_FIFO bit will turn on Rx and Tx FIFO queues that increase speed and reliability of the UART hardware interface. A hardware FIFO queue allows system software to send multiple bytes at a time (Tx FIFO) and for hardware to receive multiple bytes (Rx FIFO) before requiring service by software. Turning on ENABLE_FIFO should increase speed and/ reduce data corruption errors at high speeds.

```
#define UART0_LINE_CTRL  (UART0_BASE + 0x2C)
#define   BREAK                  (1 << 0)
#define   PARITY                 (1 << 1)
#define   EVEN_PARITY            (1 << 2)
#define   TWO_STOP_BITS          (1 << 3)
#define   ENABLE_FIFO            (1 << 4)
#define   BYTE_WORD_LENGTH       (3 << 5)
#define   STICK_PARITY           (1 << 7)
```

Last is the UART0_CONTROL (page 185 of BCM2835-ARM-Peripherals.pdf) register. The UART control register contains the configuration options for the UART hardware controller. Included are an Rx and Tx enable as well as an overall Enable bit. If the

UART requires reconfiguring, the UART enable bit should be cleared (set to zero) to disable the UART. It is good design to check and wait until the UART0_STATUS is not Busy before disabling or outgoing data may be lost/corrupted. Once disabled, the UART can be reconfigured and then restarted by setting the UART enable bit to one (1). RTS and CTS are hardware flow control options that require additional GPIO wires on the physical UART connection. Hardware flow control can add reliability to the data transfers but may not eliminate all loss or transfer errors. Let us define these bits now for completeness and to aid this future effort.

```
#define UART0_CONTROL    (UART0_BASE + 0x30)
#define   ENABLE                 (1 << 0)
#define   LOOPBACK               (1 << 7)
#define   TX_ENABLE              (1 << 8)
#define   RX_ENABLE              (1 << 9)
#define   RTS                    (1 << 11)
#define   RTS_FLOW_CONTROL       (1 << 14)
#define   CTS_FLOW_CONTROL       (1 << 15)
```

6.4 GPIO configuration for RPi UART

The interface to the primary UART on the Raspberry Pi is through the GPIO header pins on the motherboard. These pins need to be configured for the UART, which means setting the GPIO function select alternate number and disabling the pull up or pull down resistors for the UART Rx and Tx GPIOs. The pull up/down GPIO resistors must be disabled so that the UART hardware controller can take control of the GPIO pins without the interference of a resistor. With a pull up or pull down resistor in place the UART controller will have difficulty driving the Rx and Tx data transfers. Add this code to UartInit() created in Chapter 6.3 of the book, in between disabling the UART and configuring the baud rate divisor and enabling. Replace example baud rate divisor code with the RPI specific code below.

```
  /* Select Alternate 0 for UART over GPIO pins 14 Tx and 15 Rx.*/
  gpio = REG32(GPFSEL1);
  gpio &= ~(7 << 12); /* clear GPIO 14 */
  gpio |= GPIO_ALT0 << 12; /* set GPIO 14 to Alt 0 */
  gpio &= ~(7 << 15); /* clear GPIO 15 */
  gpio |= GPIO_ALT0 << 15; /* set GPIO 15 to Alt 0 */
  REG32(GPFSEL1) = gpio;

  /*
  ** GPPUD can be 0 (disable pull up/down)
  ** (0) disable pull up and pull down to float the GPIO
  ** (1 << 0) enable pull down (low)
  ** (1 << 1) enable pull up (high)
  */

  /* Disable pull up/down for next configured GPIOs so they float. */
  REG32(GPPUD) = GPPUD_OFF;
  usleep(MICROS_PER_MILLISECOND); /* hold time */

  /* Apply above configuration (floating) to UART Rx and Tx GPIOs. */
  REG32(GPPUDCLK0) = (1 << 14) | (1 << 15); /* GPIO 14 and 15 */
  usleep(MICROS_PER_MILLISECOND); /* hold time */

  /* Set the baud rate. */
#if RPI >= 3
  // RPI 3/4 has a 48MHz clock as it was intended to work with BT
  //(48000000 / (16 * 115200) = 26.042
  //(0.042*64)+0.5 = 3
  //115200 baud is int 26 frac 3
  REG32(UART0_IBRD) = 26; /* Integer baud rate divisor */
  REG32(UART0_FBRD) = 3; /* Fractional baud rate divisor */
#else
  // RPI 1/2 has a 3MHz clock by default
  //(3000000 / (16 * 115200) = 1.627
  //(0.627*64)+0.5 = 40
  //115200 baud is int 1 frac 40
  REG32(UART0_IBRD) = 1; /* Integer baud rate divisor */
  REG32(UART0_FBRD) = 40; /* Fractional baud rate divisor */
#endif
```

Notice there are often usleep calls, or a hold time, to the GPIO configuration procedure. This hold time is used to ensure that the previous register assignment has completed and the peripheral is ready for the next assignment. Without the working timer and sleep software interface we created in chapter 5, it would not be possible to consistently initialize and configure the UART. This hold time requirement is defined on page 101 of the BCM2835 document.

The default UART clock on the RPi 1 and 2 is 3MHz, the same as the example in the companion book. In fact, RPi UART can use the code in the companion book directly, except for the RPi 3 and 4 which have a faster clock. The baud divisor must be configured so as to drive the UART data transfer at the correct baud rate. With the RPi 1 and 2 it is always recommended to double check the boot settings of the UART. In the boot file system of the RPi being used for testing (the one we copy the kernel.img to), check that in the /boot/config.txt file that the 'init_uart_clock=3000000' option is set (not commented out). If a different UART clock rate is configured only garbage will appear over the UART when connecting with the development PC, as the clock divisor depends on the shared system clock connected to it.

6.5 Command line application

The next step is to create a new application named 'console' that can be used to send commands and receive responses. In the applications directory create a directory named 'console' and copy all the files from the previous chapters application, 'errorcodes', to the new directory. Open Makefile and change APPNAME from 'errorcodes' to 'console' and add '../../boards/rpi/uart0.o' and '../../system/shell.o' to the OBJECTS declaration.

```
##
## Error code application
##
OBJECTS = ../../boards/rpi/boot.o \
          ../../boards/rpi/board.o \
          ../../boards/rpi/uart0.o \
          ../../system/shell.o \
          main.o \
```

Now create the file 'boards/rpi/uart0.c' and copy all the UART functions from Chapter 6 to this file (UartInit(), UartPutc(), UartPuts(), UartRxCheck() and UartGetc()). Next copy the additional RPi specific UART configuration code above (level select and disabling of GPIO pull up/down) to the middle of the UartInit() function. Finally, create the 'system/shell.c' file copy the system shell code from the end of Chapter 6 to this file. The goal of this section is to put together all of the source code and create a 'console' application that presents a command line shell interface over the UART peripheral. Combined with the Makefile changes above, the 'console' application should now compile without errors or warnings.

6.6 USB to serial TTL connection

So the system software now has UART support, but how can the development PC use this? The first requirement is to have the appropriate hardware connection. There are many options for using the UART but this book will discuss two. The first option is the simplest, the Raspberry Pi has a 3.3 volt TTL interface to the UART through the GPIO pins. If you have two Raspberry Pi systems then the UART GPIO pins can be connected directly with female to female jumper wires. The second option is for those with only one Raspberry Pi or those who want to use a different PC for the software development. For full Linux or Windows development systems (desktop, laptop, etc.) there is typically no TTL UART available on GPIO pins and if there is it may not be accessible or documented for the motherboard. So it is recommended to use a USB to serial TTL adapter to connect the Raspberry Pi to the development PC, whether using a Linux OS or Windows OS.

Warning

The RPi UART GPIO pins are 3.3V TTL and if connected directly to a standard serial (DB9) cable typical of legacy PC systems it will destroy the Raspberry Pi. This is because the serial DB9 connection uses 5 volts and this voltage is too high.

Before we discuss the specific instructions for each method of connecting the UART, let us discuss some commonalities. First, the UART serial communication model is serial cross over in that the Rx from one side of the connection must be connected to the Tx side on the other end. So when the RPi sends, the development PC is receiving. And vice versa the other way so when the development PC sends, the RPi is receiving. This configuration is commonly referred to as a crossover connection. With this in mind, see Section 6.2 (page 102) of the BCM2835-ARM-Peripherals.pdf document for the complete GPIO map for the Raspberry Pi, including all the alternate functions. The primary UART interface is highlighted in red and named RXD0 and TXD0 and available on select alternate zero (Alt 0). Note that TXD0 is GPIO 14, and RXD0 is GPIO 15 and located at pin 8 and 10 respectively on the exposed male pins of the RPi. Here is a picture created by Jamie Bainbridge that shows all the GPIOs of the RPi as exposed by the male pins.

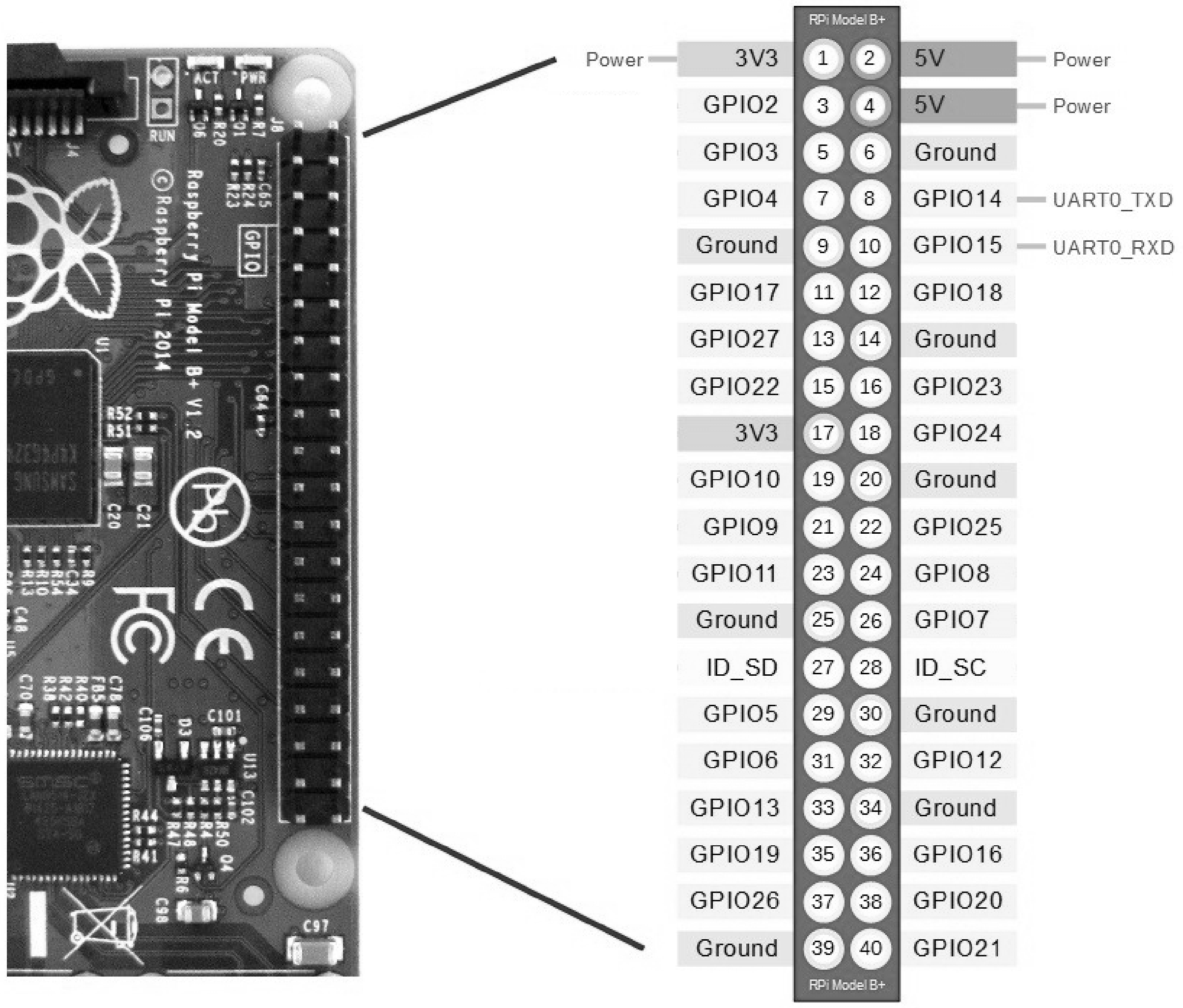


Fig 6: Raspberry Pi GPIO pinout

6.6.1 Connect UART between two RPi's

Wiring the TTL UART between two different Raspberry Pi's is relatively straightforward. First, power off both boards just to be safe (if you bump live wires together, etc. it could damage one or both RPi's). Second, using a female to female jumper wire, connect the ground (GND) nearest the UART GPIO pins (this would be the Ground (GND) at pin 6) with a gray or black wire. It is always good practice to use gray or black for GND and red or yellow for power (PWR) and the other colors for everything else. It is also good practice to always connect the ground (GND) of the peripheral to the ground nearest the GPIO connectors the peripheral is going to use. Grounding nearby GPIO pins avoids static build up between the systems and may reduce errors during data transfers.

Fig 7: UART pin map for RPI development PC to RPI target

GPIO	Pin	Pin	GPIO	UART line
15	8	10	14	TX to RX
14	10	8	15	RX to TX
-	6	6	-	Ground

Once the USB to TTL UART is grounded to the RPi, connect the UART0 TX (pin 8) of the first RPi to the UART0 RX (pin 10) of the second RPi. Do the same for the UART0 RX (pin 10) of the first RPi, connecting it to the UART0 TX (pin 8) of the second RPi. Now boot up the development RPi and the UART should be operational as the device '/dev/serial0' or '/dev/ttyAMA0'.

Information

Recent versions of RPI hardware often disable the primary UART. To ensure the RPI boots with primary UART edit the /boot/config.txt of the RPI executing the UART code and add enable_uart=1 to the end of the file. Reboot the development RPI with this change and see if it not works.

6.6.2 UART software for Linux development PC

Linux users should check to ensure they have 'minicom' installed on the development RPi and if not, issue the command 'sudo apt-get install minicom' to install it. With 'minicom' installed let us connect the two RPi UART's physically as described in section 6.6.1, or with a USB to TTL UART as described in section 6.6.3 below if you have not done so yet. Then copy the 'console' binary file named 'kernel.img' onto the microSD card of the RPi that will execute it (or use GDB to 'load' it). Then open 'minicom' on the '/dev/serial0' device on the Linux development PC with this command.

```
$ sudo minicom -D /dev/serial0

Welcome to minicom 2.7

OPTIONS: I18n
Compiled on Feb  7 2016, 14:00:34.
Port /dev/serial0, 12:32:56

Press CTRL-A Z for help on special keys
```

After connecting physically and starting minicom, power on the RPi (or invoke 'continue' from GDB) to execute the software we created. The interactive shell should output its introduction splash through minicom to the development PC.

```
$ sudo minicom -D /dev/serial0

Welcome to minicom 2.7

OPTIONS: I18n
Compiled on Feb  7 2016, 14:00:34.
Port /dev/serial0, 12:32:56

Press CTRL-A Z for help on special keys

Console application running on Raspberry Pi
  Copyright 2015 Sean Lawless, All rights reserved.

Check timer. Sleep 1s loop. Press key to exit.
1234

Check timer. Sleep 1/10s loop. Press key to exit.
12345678901234567890123456

Entering the system shell
shell>
```

If you connected after powering on it is no big deal, but you will miss the introductory splash so you will probably witness the numbers counting by in second increments until you press a key. If you see garbage characters, the baud rate divisor and/or UART clock are not configured correctly.

Important

Often the Linux OS installed on the RPi development PC (Raspbian, etc.) will by default configure and use the primary UART for a Linux shell console. This means that the RPi development PC's serial device '/dev/serial0' will be in use as a Linux command shell after boot up. This will interfere with the development RPi when we connect to it. To avoid this interference, be sure to disable the Linux console on the serial0 device at boot up on the development. For Raspbian on the RPi, remove the 'console=…' line from the /boot/cmdline.txt file so Linux will not use the UART as a command line console. Reboot the development RPI with this change and the UART should works better.

The last step is to add privileges to the Linux OS user to use 'minicom' so that the 'sudo' command is not needed. 'Sudo' elevates the user privilege to the superuser which is not a good security practice. The serial devices, such as '/dev/serial0', are only accessible to the superuser as well as the user group 'dialout'. To avoid using 'sudo' with the '/dev/serial0' device, add the 'dialout' group to your user group settings. This can be done with this command, replacing <username> with the name of your user.

```
$ sudo usermod -a -G dialout <username>
```

Then log out from your user account (or reboot) and upon the next log in the user will be able to use 'minicom' with the '/dev/serial0' device without superuser privilege.

```
$ minicom -D /dev/serial0

Welcome to minicom 2.7

OPTIONS: I18n
Compiled on Feb  7 2016, 14:00:34.
Port /dev/serial0, 12:32:56

Press CTRL-A Z for help on special keys
```

6.6.3 Connect UART from RPi to PC

Using a Windows or Linux system for development requires additional hardware, specifically a USB to TTL serial cable to connect between the development PC and the Raspberry Pi. The easiest to use solution has a USB connection on one end, and two to five female jumper wires on the other end. The jumper wires connect to the Raspberry Pi GPIOs directly exposed on the RPi board as male pins. Only two are really needed, Rx and Tx, but the ground should also be connected if available. As of this writing, there are USB to UART TTL's for under $10 USD online. Make sure it is for TTL (3.3 volts) and the USB device is compatible with the development PC's Operating System. It also should have individual wires on the non-USB end to connect to the RPi, preferably female connectors.

Important

Remember this is a cross over connection, so connect the Tx wire coming from the USB adapter to the RX pin on the RPi, and the RX of the USB adapter to the RPi's TX pin.

Advanced users, or users who plan to be moving beyond this book, may wish to consider a more advanced USB to serial adapter, such as the FTDI FT2232. This part has a micro USB port for the development PC and two 3.3 volt TTL serial ports with male connectors. With some female to female jumper wires, and careful reading of the FT2232 data sheet, the FT2232 can be connected to Raspberry Pi as described above. Like the dedicated cable, the FT2232 can use three wires to connect; ground, Rx and

Tx. Be sure to cross over the RX and TX when connecting to the RPi pins and connect at least one ground on the FT2232 (nearest the UART being used) with the RPi ground nearest the UART pins (pin 6). The ground is NOT optional when using the FT2232.

With Linux OS as the development environment refer to section 6.5.2 for instructions on how to connect to the serial TTL UART with ‘minicom’. For USB adapters, the serial TTL device of the USB adapter will appear in Linux under the ‘/dev’ folder after connecting. It is not possible to know ahead of time which device (/dev) will be created/used for the USB adapter. However ‘/dev/ttyUSB0’ is common. If ‘/dev/ttyUSB0’ is not present or does not work, unplug the USB adapter and list the /dev directory (‘ls /dev’) before and after connecting the USB cable to the Linux development PC. Any new ‘/dev/tty*’ device is the one to use. On older Linux OS’s the ‘/dev/ttyS0’ may be used by the OS.

The FT2232 also supports JTAG and can be used instead of using an RPi development PC running Raspbian and OpenOCD JTAG bit bang controller as discussed in Chapter 4. If you are not using an RPi development PC, it is recommended to connect both the RPi target UART and JTAG to the FT2232 to enable both the UART and JTAG debugging with the development PC.

6.6.4 UART Software for Windows development PC

This section explores how to install and use an application running on the development PC to connect to the Raspberry Pi over the serial UART on a Windows development PC. First the reader should search for and download serial terminal software, such as TeraTerm (or ExtraPuTTY) from the Internet.

With the USB to TTL adapter (FT2232 or otherwise) connected, run TeraTerm or ExtraPuTTY and from the Setup menu select Serial Port. In the Port field use the down arrow key to show all serial COM ports. If you have more than one and do not know which one is the Raspberry Pi, open the Windows Control Panel, then Hardware and Sound and click on Device Manager. Open Ports (COM & LPT) to see the connected Serial Ports. If you still do not know, try disconnecting the USB adapter and seeing which COM port(s) goes away in the device manager. Then configure TeraTerm or ExtraPuTTY for the Port and Baud rate of 115200. Leave the Data, Parity, Stop and Flow control settings to 8 bit, none, 1 bit (8N1) respectfully. Select Ok to connect with the Raspberry Pi.

If the ‘console’ application was already running before the development PC connected, then the Raspberry Pi has already sent the introductory splash and command prompt. Pressing the enter button will send the empty command and if connected to the Raspberry Pi running the ‘console’ application, it will receive back further characters or the command prompt ‘>’. The goal of this laboratory assignment is to establish the terminal shell connection with the Raspberry Pi. Congratulations if this is working. If it is not yet working, see the troubleshooting in the next section.

6.7 UART Troubleshooting

Like JTAG, there is a lot that can go wrong with the UART, from the physical connection to the software configuration on both the system software and development PC. The usual checklist is to double check the wiring and software configurations. Pay particular attention to the development PC and be sure you are connecting to the correct device (/dev/ttyX or Windows COM port), and with the correct settings, 112500 baud at 8N1. It helps to use a development system whose UART is known to work with other boards so you can limit any investigation to a single side. Below are some common problems and steps to take to solve them.

If nothing appears over the UART during the start of the ‘console’ applications, and repeated key presses also show nothing, there is a hardware misconfiguration. Either the TTL is not physically connected correctly, or the development PC terminal software (minicom, TeraTerm, etc.) is not using the correct physical device. There could also be a serious issue with the compiled image, such as it was compiled for a different RPI, such that board.h is pointing the PERIPHERAL_BASE to the RPi 2 even though you are executing on the RPi B+.

If garbage appears when the ‘console’ application begins, we know the correct device is being used by the development PC. Try pressing keys and see if more garbage appears. Double check the software configuration on both sides and be sure both are configured for 112500 baud and 8N1. Double check the ‘/boot/config.txt’ file of the RPi executing the ‘console’ application and be sure the ‘init_uart_clock=3000000’ is not commented out or set to a different value than 3MHZ.