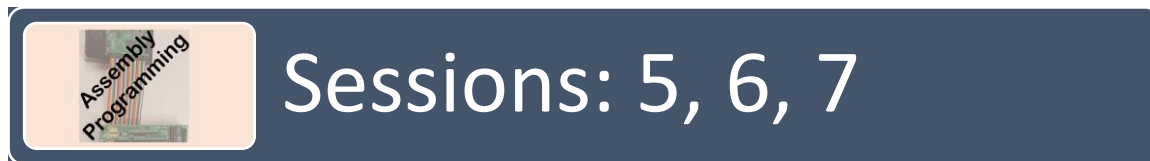


## Practical Module 3: Assembly Programming

*This document contains the background information and exercises for the third module of the 2IC30 Computer Systems practical.*



### Background and Introduction

In the previous practical module you gained an insight into how the internal architecture of a microprocessor functions when a particular machine code instruction is executed. As a result of this you should be aware that machine code instructions generally perform 1 (or more) of three operations:

- Moving Data (i.e. Memory to register, register to memory, register to register)
- Arithmetic and Logic Functions (e.g. Subtracting, rotating, comparing, AND-ing, OR-ing)
- Branching and Control (e.g. unconditional and conditional branches)

Programs are built up from multiple instructions with branching used to build the core control structures that are the backbone of all software; sequence, iteration, selection and subroutines.

The Raspberry PI is a System-On-A-Chip computer. The RPi, as given to you for this practical, runs a 32-bit version of the Debian operating system which is a Linux derivative. The operating system has an assembler, compiler and debugger built in. We will use these to develop assembly language programs to execute on the RPi. The RPi will be run in a headless mode (no keyboard or monitor), so all interaction with the device will be via tools on a Laptop/PC.

As well as being familiar with the materials listed in the Preparation column of the Schedule you should read through the activities and, where possible, prepare your programs *as much as possible in advance*. If you leave this till you arrive at the practical then you will not have enough time to implement and debug your code. Use the practical sessions for compiling and debugging your code.

### Session Schedule

Session	Preparation	Graded Exercises (points)
<a href="#">5</a>	Lectures 8 ,9 Chapter 8 of Course Text Install required software	<a href="#">5.9 The Write System Call (2 pts)</a> <a href="#">5.11 Correct the MinVal Function (2 pts)</a> <a href="#">5.12 The Write System Call – Again (2 pts)</a> <a href="#">5.15 Complete the gen_number Function (2 pts)</a> <a href="#">5.19 Unmapping and Closing (2 pts)</a> <a href="#">5.20 Adapting the Game (2 pts)</a> <a href="#">5.21 Making the Game Repeat (1 pt)</a>
<a href="#">6</a>	BCM2837 ARM Peripherals Datasheet, Chapter 8 of Course Text Gertboard User Manual (up to p24)	<a href="#">6.3 Playing with LEDs (2 pts)</a> <a href="#">6.4 Using a Delay to Blink an LED (2 pts)</a> <a href="#">6.6 Displaying a Binary Number (2 pts)</a> <a href="#">6.7 Playing With The Code (2 pts)</a>
<a href="#">7</a>	As above, plus significant code development before the session	<a href="#">7.1 Implementing Pulse-Width Modulation (3 pts)</a> <a href="#">7.2 Adapting the PWM Program (3 points)</a>

## Submissions

Working as a small group you will need to present your solutions to the Tutor or Student Assistants. A 'solution' for many of these exercise is a demonstration of working code. Unlike the logic module, presentation of solutions will require completing them in sequence.

## Grading

Exercises will be worth 1 to 3 points depending on complexity (as indicated in the schedule above). If the solution is not fully working then the grade will be reflective of this. Because you are working towards the completion of larger programs many exercises must be completed to progress through the session. Some exercises are indicated as non-dependent and may be missed.

For 1 point questions:

- 0 points : A solution is not submitted or does not compile/produces unexpected behaviour
- 1 point: The solution compiles and produces the expected behaviour

For 2 point questions:

- 0 points : A solution is not submitted
- 1 point: The solution compiles but produced unexpected behaviour
- 2 points: The solution compiles and produces the expected behaviour

For 3 point questions:

- 0 points : A solution is not submitted
- 1 point: The solution compiles but produced unexpected behaviour
- 2 points: The solution compiles and produces the expected behaviour
- 3 points: As above but the code is also well commented

Due to the nature of the work you will be carrying out, and in order to develop sound engineering and programming habits, the tutors and assistants will adopt a StackOverflow policy of assistance. Should you require assistance *try* have the following information to hand; it reduces the time tutors and Student Assistants require for assisting;

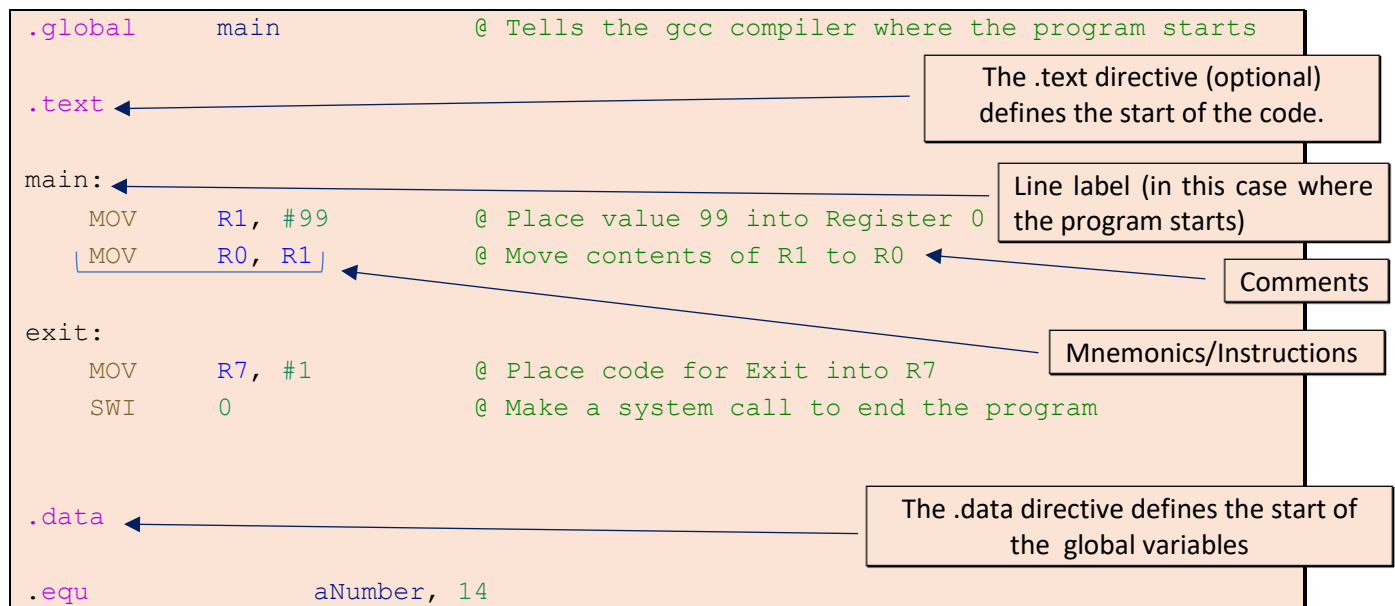
1. State which exercise you are working on.
2. Identify which function or lines of code are 'not working'.
3. State what you expect the code to do.
4. Have an idea as to what the code is actually doing i.e. Have you debugged the code and/or read relevant error messages and tried to understand them?

*Adopting this policy means you will become more self-sufficient, experienced and therefor quicker in solving problems!*

*You should aim to spend time between the practical sessions examining the exercises and preparing your code files in advance. In many cases you will be able to use the practical time for checking that your intended solution works, or debugging it, rather than actually developing it.*

## Source code file syntax

For quick reference an example assembly programming file is given below.



Use of tabs and indentation is important to maintain clarity of reading and to differentiate between labels, directives (e.g. `.data`) and mnemonics.

## Equipment/Software

You are required to use a headless RPi (no screen and no input peripherals) for development and execution of assembly code programs. Most programs are already partially completed, so you will need to add in the missing code and then assemble/compile the source code so that it may be executed. In order to speed up development, the RPi's you are provided with have a DHCP server installed and also allow for SSH connections to be made. This allows you to transfer files between the RPi and your laptop using only a network cable. Editing of files can be done either on your laptop or on the RPi. The tools you will need are;

- [PSFTP](#) command line client for transferring files to/from the RPi
- A command line terminal:
  - Windows: Powershell (Right-click Start icon(🪟) and select PowerShell)
  - Linux: Terminal
  - MacOS: Launchpad → Terminal
- [Visual Studio Code](#) for editing source code files. Note that Visual Studio Code (VS Code) contains a command line terminal.

The hardware supplied during the practical is:

- RPi + power supply + SD card + Network cable + Ethernet adapter
- (For later exercises) Gertboard extension board with connectors

Alternative software can be used e.g. WinSCP, FileZilla, Windows Terminal, Notepad++ and yes, it is possible to enable remote development with Visual Studio Code, but it is a *strong recommendation* that you develop your skills in using command line tools - this document assumes you are using the PSFTP and VS Code programs named above.

If you install PSFTP, then you can also use it from the command line in Powershell, by simply typing `PSFTP`.

## Common Errors

- Are you actually moving the files you think you are moving? Check the source and destination directories in the PSFTP terminal. Check the SSH directory you are currently in. When using a CLI, it is very common to get lost in directories if you are accustomed to graphical interfaces.
- Are you using sudo where required? If not, programs may not execute as expected, or may not execute at all.
- Do you know what a segmentation fault is? Once you do, it will help you to understand why your program is not executing as expected.
- You may not be able to SSH into the RPi. Did the error message mention a hosts file? If so, find that file and remove the contents.

## Session 5 – Assembly Programming Introduction

During this session you will become familiar with the following assembly programming concepts:

- Moving data into and between registers
- Loading data from variables into registers
- Making system calls
- Using Arrays
- Using functions with parameters and return values
- Using iteration (loops) and conditional execution

In order to do this you need to familiarise yourself with the process of assembly programming on the RPi. You will also be introduced to the GDB debugger which can be used to inspect the state of the processor (i.e. the register contents) during execution of a program to help find errors.

Using these basic techniques will allow you to write the code to complete a partially written number guessing game.

### Establish a Workflow

In this exercise we establish a routine for working with the RPi. You may need to refer back to this exercise until you are familiar with the process.

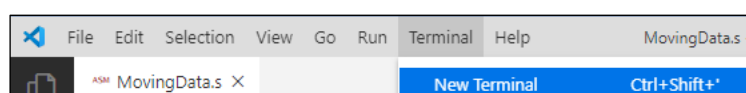
#### 5.1. Physical Connection

- i. Turn on the PC/laptop.
- ii. Ensure the SD card is inserted into the RPi.
- iii. Connect the power supply to the RPi.
- iv. Connect the RPi and the PC/Laptop using the network cable.

#### 5.2. Login to the RPi via SSH

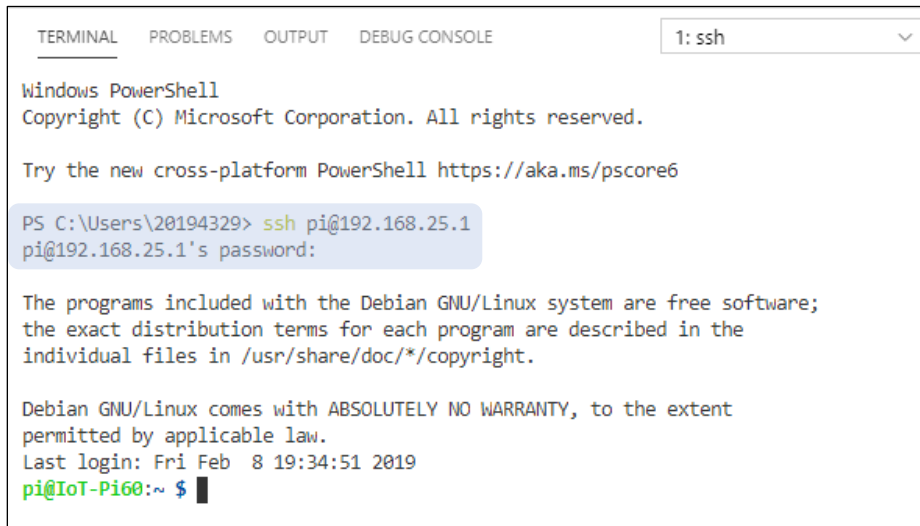
The RPi has a fixed IP address we can use to communicate with it.

- i. Start a terminal window (the image below shows this happening from within VS Code):



- ii. Type: `ssh pi@192.168.25.1`  
*This establishes a Secure Shell connection to the RPi using IP address 192.168.25.1 and the username pi*
- iii. At the password prompt type: `tue321`

From now on will use this SSH terminal window for compiling and executing files.



```
TERMINAL  PROBLEMS  OUTPUT  DEBUG CONSOLE  1: ssh
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\20194329> ssh pi@192.168.25.1
pi@192.168.25.1's password:

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Fri Feb  8 19:34:51 2019
pi@IoT-Pi60:~ $
```

Figure 1 Establishing an SSH connection with the Raspberry Pi

### 5.3. Create a Working Directory

On the RPi create a directory within `/home/pi`. Use your student username number `<username>` as the directory name. Do this by typing `mkdir` within the SSH terminal: e.g. `mkdir 20209876`

### 5.4. Transfer files to the RPi using PuTTY+PSFTP

- i. Download the ASMPProgramming directory from Canvas and unzip to a local directory on your Laptop/PC.
- ii. Download and install [PuTTY](#).
- iii. Execute PSFTP (found in the same windows menu as PuTTY).
- iv. Type `open pi@192.168.25.1` to connect to the RPi.
- v. Use the same password as for the SSH connection (`tue321`).
- vi. Use the `cd <directory name>` or `cd ..` command to navigate to the working directory you created on the RPi.
- vii. Use the `lcd <directory name>` or `lcd ..` command to navigate to ASMPProgramming directory – hint: copy/paste the full file path of the ASMPProgramming directory into the PSFTP terminal. Use quotes to delimit the path as directories/files often have spaces in their names, e.g. `lcd "C:\Users\22211112\My Files\ASMPProgramming"`
- viii. Use the `mput -r *` command to copy the entire ASMPProgramming directory and nested directories/files from the host PC to the RPi (Use `put <filename>` to move only one file).
- ix. Check that the `MovingData.s`, `Variables.s`, `Arrays` and `SystemCall.s` files are now on the RPi within the `WorkFlow` directory. These are the ones we will be using for the first part of this session.

More PSFTP commands are given in the reference sheet on Canvas.

*Programs such as WinSCP and FileZilla are essentially visual wrappers around SFTP terminal commands, so it will increase your proficiency and skills base if you can use the PuTTY command line interface.*

### 5.5. Assemble, Link and Execute a program

- i. Within the SSH terminal, change to the `Workflow` directory by using the `cd` command (see [Figure 2](#))
- ii. Ensure that the `MovingData.s`, `Variables.s`, `Arrays` and `SystemCall.s` files are now on the RPi within the `ASMPProgramming/WorkFlow` directory (use the `ls` command).
- iii. Compile the source file (assemble and link):  

```
gcc -g -o MovingData MovingData.s
```
- iv. Execute the `MovingData` program file:  

```
./MovingData
```
- v. Examine the contents of register `R0` (the program exit code) by typing:  

```
echo $?
```

#### Hint

In Windows you can copy text, e.g. the `gcc -g -o ...` command above, and paste it into the terminal with a right-click on the mouse. You can quickly repeat previous terminal commands using the **↑** cursor key.

```
pi@IoT-Pi60:~ $ cd 20194329/ASMPProgramming/WorkFlow
pi@IoT-Pi60:~/20194329/ASMPProgramming/WorkFlow $ gcc -g -o MovingData MovingData.s
pi@IoT-Pi60:~/20194329/ASMPProgramming/WorkFlow $ ./MovingData
pi@IoT-Pi60:~/20194329/ASMPProgramming/WorkFlow $ echo $?
99
```

Figure 2 Exercise 5.6 steps iii to v of the Workflow – Compile (Assemble, Link) and Execute a program

### 5.6. Edit a Program File

- i. Open the `MovingData.s` file (as located on the PC/laptop) within the code editor.
- ii. Edit the program so that a different number (0-255) is moved into `R0`.
- iii. Save the file.
- iv. Use PSFTP to replace the `MovingData.s` file on the RPi with the new version you have just saved.
- v. Compile and execute to verify that the updated program is working (use `echo$?` to check the value in `R0`).

#### Hint

Within the SSH terminal you can use the [nano](#) or [vi](#) text editors e.g.

```
>nano MovingData.s
```

This allows for files to be edited on the RPi directly thereby avoiding having to transfer them from the host machine to the RPi with every edit.

## 5.7. Clean-up and Exit

At the end of every session when you need to return the hardware you should remove ALL your practical files from the RPi. This is to ensure;

- You have a copy of your work for the next practical session.
- You don't leave your programs for other students to have a free ride.
- You don't clutter the RPi's file system.

The full procedure to finish your session is:

- If you have been editing files on the RPi directly and wish to keep them for further work then use PSFTP (the `mget -r *.* *` command) to copy all the files from the RPi to Laptop/PC (this will overwrite existing files with the same names on the Laptop/PC unless you change the directory). If you have been editing files on the Laptop/PC and then moving them to the RPi for compiling then you can skip this step.
- Close the PSFTP session by typing `exit`.
- Within the SSH terminal use `cd ~` to navigate to the `/home/pi` directory and then delete the `<username>` directory from the RPi using the command:

```
rm -r <username>
```

- The RPi **must** be shutdown using the command `sudo shutdown -h now`. This is done within the SSH terminal:

```
pi@IoT-Pi60:~ $ sudo shutdown -h now
Connection to 192.168.25.1 closed by remote host.
Connection to 192.168.25.1 closed.
PS C:\Users\20194329>
```

- When the RPi's green LED has stopped flashing, the RPi power supply and network cable can be disconnected. The PSFTP and SSH terminals can now be closed.

### Warning

If you unplug or switch off the power supply without issuing the shutdown command the SD card can become corrupted. You can use CTRL-C to stop any process that is blocking the terminal.

## Variables

We will now use a second assembly program for you to familiarise yourself with the process of loading data from a variable.

## 5.8. Moving Data

- If you have powered down the RPi and removed your files, then restart the Workflow to establish an SSH connection and copy your files from the laptop/PC to the RPi
- Open the `Variables.s` file in the code editor and try to understand what the code is doing. (Note the 2-step process to load a variable value; this process will be needed frequently!)
- Check that the `Variables.s` file is on the RPi (drag/drop in FileZilla if not)
- Compile the program and compare the output of the program (use the `echo $? *` command) with the source code to check it is the same.
- Within the code editor, change one of the variable names and the value.



- vi. Save the source code file.
- vii. Repeat steps iii and iv above.

#### Note

We are using the word compile in this document to refer to the process of assembling one or more assembly source code files into object code and linking them together.

## System Calls

System calls are crucial when developing any code that is to be run under the supervision of an operating system. We will look at one system call in detail to gain familiarity. In the rest of the sessions you will need to implement several different calls yourself.

### 5.9. The Write System Call (2 points)

- i. Examine the `SystemCall.s` file within the code editor. Read the comments that explain how the parameters(values) that are loaded into registers and how both the function and the system call are made. You should also use the Reference Sheet that explains the parameters.
- ii. Change the `SystemCall.s` file so that it displays a message using both variables on a *single* line e.g.

```
Hi, my name is Dorothy and I'm from Kansas
```

- iii. Note that you will also need to change the length of the strings. Count the letters (`\n` is a single character).
- iv. Save the file and transfer it to the RPi.
- v. Compile and execute the program to check it works. Edit the source file as necessary.

## Using the GDB Debugger

You will often make syntax and logical errors in your code when programming. Syntax errors are found at the assemble and link stage (during compiling), but logical errors (where programs may execute but give unexpected results) are not so easy to find. A debugging tool will allow you to step through a program seeing how register values change and program control moves between different functions. Being able to use debuggers is an important aspect to programming.

In this exercise we will recreate the first steps of the workflow, and then use the debugger to step through a short program to examine register values. The program will be used in the following exercise; read the introduction to the next exercise to get a feel for what the program is doing.

### 5.10. Some GDB Commands

- i. Open the `Arrays.s` program file (as on your Laptop/PC) and read through the explanation.
- ii. On the RPi you should compile and execute the program `Arrays.s`.
- iii. Note the return value of the program (use the `echo $? Command`).

We will now cover some of the basics of using GDB here. This is an extensive tool so we will only be covering the basics. A complete reference is [available online](#).

- iv. Start the debugger using the `gdb Arrays` command (assuming you named the executable `Arrays`)
- v. Type `list` to view the program source code.  
(Press enter repeatedly to skip through more of the code)
- vi. Type `b 1` to set a breakpoint on the 1st line of code.



- vii. Type `start` to begin execution (ignore the error message that appears and press 'n'). Note that execution stops and GDB displays the next instruction to be executed.
- viii. Type `info r` to view the contents of the registers.
- ix. Type `s` (step command) to execute the next instruction.
- x. Repeat the above 2 steps several times. Note how you can see the contents of the registers being changed at each step.
- xi. Type `print array1` to view the contents of the first element in the array.
- xii. Type `print array1@10` to view the contents of the whole array.
- xiii. Type `cont` to continue executing till the program stops or it hits another breakpoint.
- xiv. The complete list of GDB commands can be displayed by typing `help`.

#### 5.11. Correct the MinVal Function (2 points)

- i. As you can see from step iii above the function is 'not working'. See if you can fix the error by *changing a single line of code within the function*. Do not add or remove any lines of code. In order to test your fixed code you should change the array such that the smallest value is placed in different positions (beginning, middle and end of the array).

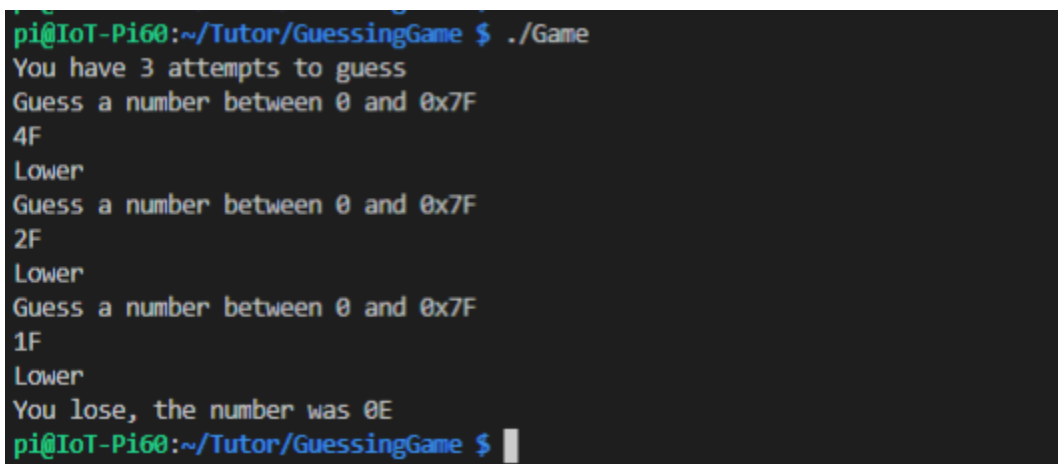
## Complete The Guessing Game

By now you should be familiar with the process of editing files, moving them to the RPi, and then assembling and executing. This exercise will concentrate on the core aspects of assembly programming such as putting values into registers, working with loops, conditionally executing code and calling functions (subroutines).

To do this a partially completed game (`Game_Template.s` in the `Game` directory) has been created. Although the game code provided will assemble and execute it is incomplete, so the main task is to fill in the template. Read over the file to familiarise yourself with the template and the function names.

The game will eventually generate a “random” value between 0 and 127 which the user must try to guess in a fixed number of tries. The game will display hints and if the user does not guess the number it is shown to them.

Note that the ARM instruction set lacks division (which would ease conversion between the base 10 user inputs and base 16 values the processor needs), so we will use hexadecimal numbers as input and output.



```
pi@IoT-Pi60:~/Tutor/GuessingGame $ ./Game
You have 3 attempts to guess
Guess a number between 0 and 0x7F
4F
Lower
Guess a number between 0 and 0x7F
2F
Lower
Guess a number between 0 and 0x7F
1F
Lower
You lose, the number was 0E
pi@IoT-Pi60:~/Tutor/GuessingGame $
```

Figure 3 Screenshot of how the completed game runs

Initially you may feel that by placing code into functions, the number of lines of code is larger as compared to placing the instructions within the main logic part of the game (you’re right!). However, the structure of the game itself is clearer and easier to understand. Also, quite a few of these functions can be reused in later sessions.

The list of functions that you will complete or use in the program is:

- `print` Write text to the terminal
- `read` Read input from the keyboard
- `atoi itoa` Convert ASCII characters to/from integer (hexadecimal) values
- `asctonum` Convert hexadecimal ascii values to integer values
- `numtoasc` Convert an integer value to ASCII characters
- `gen_number` Generate a random number from the time of day
- `print_hint` Display “Higher”, “Lower” or “Correct!” depending on the user’s input
- `print_lose` Display the correct answer if the user does not guess the number

## The ARM Calling Convention – A Reminder

Registers `R0-R3` are loaded with the parameters for the function. The function is called by the statement `'BL <function name>'` (Branch and Link). The processor adds 1 to the (`PC`) Program Counter, stores this value in `LR` (the Link Register which is an alias for `R14`), and replaces the current `PC` with the start address of the function.

The function must push all registers that it uses (except `R0-R3`) onto the stack. These values are restored before returning; this ensures that the code calling the function can assume the contents of all registers (except `R0-R3`) are unchanged after the function has been executed.

A function starts with a meaningful label that should describe its purpose, e.g. a function that calculates the distance travelled by an object in a certain amount of time could be named `'distance'`. It is good practice to document the function with comments, describing the parameters, the purpose and the return value. Actually, each line of code in an assembler program should have a comment as it can be difficult to understand code when returning to it after a period of time.

A function ends with the command `'MOV PC, LR'` which loads the `PC` register from the `LR`. The program will continue from the statement after the initial call. Notice that if your function calls another function (including a system call), you need to put `LR` on the stack as well (and reload it before returning).

### 5.12. The Write System Call – Again (2 points)

During execution of the game the program needs to write to the terminal a number of times. The function `print` has been created to encapsulate the `write` system call, but requires completing;

- i. Using the code from the [System Calls](#) exercise as a reference, complete the `print` function.
- ii. Locate the `.data` section and check that each string has the correct equivalent string length constant.
- iii. Identify the locations in code where the `print` function is called with the `BL print` instruction and check that each call to `print` is preceded by the correct parameters being set.
- iv. The first string the program displays is not very informative. Modify the prompt to:

```
"Guess a number between 0 and 0x7F:".
```

You will also need to change the length constant.

- v. Save the file, transfer it to the RPi, compile and execute to ensure the correct strings are displayed.

(Note that the code to enable the keyboard to be read is not yet implemented)

### 5.13. The Read System Call

During the game the user must make a guess which is a 2-digit hexadecimal number. The system call returns after the user presses Enter on the keyboard. If you wish to read 3 characters from the keyboard (by storing 3 in `R2`), the system call will store up to 3 ASCII characters at the address of the label in `R1`. If you enter  $n \leq 2$  characters and a return, the  $n$  characters and the newline character are stored at this address.

- i. We will create a variable `input` in the `.data` section in which to store the user entered characters. This is done as follows:

```
input: .space 3 @ TASK: Create user guess variable here
```

- ii. Within the main code locate the commented out `BL` read function call. Complete the two lines of code prior to the call. These should place the address of the `input` variable and the number of characters (3) into `R0` and `R1` respectively. These values are then passed as parameters to the `read` function.
- iii. Remove the `@` to allow the function call to execute.
- iv. Locate the `read` function and add the code to put the parameters into `R7` and `R0` needed for the `read` system call.
- v. Remove the `@` in the `read` function to allow the `SWI` instruction to execute.
- vi. Compile and execute your code.

Your program should almost be complete!

(The reason your program cannot show the correct value at the end is because we have not converted the numbers to ASCII text - we will fix this in the next stage).

#### 5.14. Iteration and ASCII conversion

All text as entered on the keyboard is actually an ASCII character - this includes the symbols 0-9. The next step is therefore to use the function `asctonum` (ASCII to number) to convert the hexadecimal characters entered by the user into a hexadecimal integer value.

The function `asctonum` requires a single parameter; the address of the string. This is passed in `R1`. This function iterates over the string and repeatedly calls the sub-function `atoi` which in turn converts a single character to an integer. `asctonum` returns the final result in `R0`.

The function `numtoasc` converts a number to a hexadecimal ASCII string. The parameters of this function are the value to convert (`R0`) and the address of the target string (`R1`). Note that multiple hexadecimal characters are required for numbers larger than 4 bits; so the sub-function `ittoa` is called.

- i. Within the `ittoa` function add in the code to convert a single integer value to the equivalent ASCII character.
- ii. Read through the comments for the above functions. Add the comments where required (indicated by `@ Task:` ) to show you understand how the conversion takes place. It may prove useful to use GDB to step through the function and keep an ASCII table ([such as this one](#)) handy so you can see what is happening in the registers at each step.

#### Pseudo Random Numbers

An easy way to get a more or less random value is from the time of day. This is known as a pseudo random number, and we can obtain this value by the `gettimeofday` system call. This system call needs address references to two structs (struct, abbreviated from structure) as arguments. The system call takes these references (see course text section 6.6) so it can write directly to them. One argument in `R0` is the address for the OS to write the time value and the parameter in `R1` is the address where it should write the time zone. We do not require the timezone data, so will pass a null reference and the system call will skip writing that value.

A struct can contain multiple items of data referenced by the address of the first item (similar to arrays). In this case the time struct consists of two words: the first word contains the number of seconds since January 1<sup>st</sup> 1970, the second word contains microseconds.

In the `.data` section, the following struct can be found;

```
time:          .space 4      @ Time (s) since Jan 1 1970
musecs:        .space 4      @ Time (ms)
```

In C syntax this is:

```
struct time
{
    int time;    // Time (s) since Jan 1 1970
    int musecs; // time (ms)
}
```

Once the system call has filled in the struct a more or less random value can be obtained from the word that contains the microseconds. It makes sense to give that word a separate label (`musecs`) so we can access it directly rather than having to use the address `time+4` as a reference.

### 5.15. Complete the `gen_number` Function (2 points)

- i. Locate the `gen_number` function and comment out the `MOV` instruction.
- ii. Underneath this line you should add four instructions;
  - The first is to load the address of (the reference to) the time struct into `R0`.
  - The next is to load 0 in `R1` to represent a null pointer to the time zone structure.
  - Now place the system call number for `gettimeofday` in `R7` (see the Reference Sheet).
  - Finally add the software interrupt instruction.
- iii. If you were to assemble and run the program now, what would be the value returned from the `genNumber` function?

In order to extract the microseconds value you need to perform 3 more tasks;

- iv. Load a register e.g. `R1` with the address of the reference to the `musecs` variable (similar to the method above).
- v. Then load `R0` with the actual value at the address contained in the above register (see `variables.s` as used in the [Moving Data](#) section for an idea on how to do this).
- vi. Finally we need to perform a logical `AND` of the value in `R0` with a bitmask of `0111 1111`. This sets the upper limit of any number now contained in `R0`. *Hint*: A similar process was performed in the `numtoasc` function.

In summary, the function should make a system call, retrieve the time, mask off the lowest 7-bits and return this value in `R0`. If this function does not work then you can always get your program working again by commenting out the code you have entered and un-commenting the `MOV` instruction at the top.

Hurrah - your program should be complete! Compile, execute and check that it is working.

### 5.16. Using Constants

Now that the program is working we can make some improvements to the code. This will make the code far more readable and it is always easier to remember names rather than numbers. We have defined some in the data section for use with the strings. But it is handy to define others at the top of the program, for example;

```
.equ    SYS_READ,    ?  
.equ    SYS_WRITE,  ?  
.equ    SYS_EXIT,    1  
.equ    SYS_GETTIME, ?  
  
.equ    STDOUT,      ?  
.equ    STDIN,        ?
```

Here is how to use a constant in your code;

```
_exit:  
    MOV R7, #SYS_EXIT      @ exit syscall  
    SWI 0
```

As you can see, it is far more readable!

- i. Find the numbers for the system calls shown in the code extract above (use your code or the Reference Sheets).
- ii. Add these numbers as constants at the top of your program.
- iii. Replace the numbers in your code with constants. To prevent too many errors make one change at a time and test that the program compiles and executes each time.
- iv. Identify any other places where [magic numbers](#) (un-named numerical constants) appear in your code and can be replaced with constants. Be careful, some constants can loaded with a MOV instruction, others may need an LDR.

## Understanding the RPi's Hardware

The Linux operating system on the RPi prevents user programs from accessing peripherals as directly addressable memory locations, instead treating them as files with access rights. The kernel maintains a byte-by-byte image of hardware devices in `/dev/mem`. It is possible, as you have seen, to use the read/write system calls to access devices, but the operating system overhead required for each system call is significant, especial when multiple system calls are used. Instead we can mirror (map) a portion of this image into our program's address space and use it to access the hardware directly.

As an introduction to this we are going to create a simple hardware random number generator program that returns a number in `R0`. Once this short program is working the code will then be used in the guessing game program to replace the system call you initially programmed. The hardware access method developed in this first set of exercises will then be used in other exercises when working with the Gertboard peripherals.

All hardware on the RPi is located in a block of physical addresses starting at `0x7E000000` [BCMAP, chapter 12]. We cannot access this address directly because the MMU (Memory Management Unit), which is used by the processor to access all memory locations, expects a virtual address.

The actual virtual base address for the RPi's hardware is `0x3F000000` and the System Timer has an address of `0x3F003000`. Register `CL0` within the timer contains the lower 32 bits of the timer (useful for our program). This register has on an offset of `0x4`. So if you want to read the ST's lower 32 bits you need to read at address `0x3F003004`. We cannot access the register as the Debian operating system (rightly) prevents programs directly accessing certain memory locations. The base address in [BCMAP] refers to the Raspberry Pi 1 and 2 Models. Broadcom have not released documentation for the RPi v3. With the exception of this base address the other information in the document is correct.

In order to access the ST (System Timer) registers it is necessary to have the following functions:

1. `open_mem` This declares to the operating system kernel that our program wishes to access a peripheral device (open the device rather like a file in memory). We do this by making the `open` system call with a location (`/dev/mem`) and permissions (read and/or write) as parameters.  
The kernel will then make a record of which program requested which level of access (so it can deny/allow other processes access to the same resource) and then provide a file descriptor (known as a *handle* in windows) in return which is passed back in register `R0`. We will store the file descriptor in a variable so we can use it later.
2. `map` Map (or mirror) the hardware peripheral address range to the Pi's memory space so it can be accessed. We do this by passing the file descriptor and other parameters to the `mmap2` system call. The kernel will map (or mirror) a subset of physical addresses (referenced by the file descriptor) directly to a range of virtual addresses and provide the starting address of the mapped memory in return (passed back in register `R0`).  
  
As our program does not know which areas of memory are free we have to ask the operating system to find some free memory within which to mirror the physical addresses. The starting address for this mirrored subset is returned by `mmap2` in `R0`. (A subset is used because the entire physical space can exceed 4GB and some of the virtual address space will be reserved for the operating system, other programs etc. The maximum and minimum size of any subsets is determined by the operating system.)  
  
We can then read/write to/from the memory as required by the program.
3. `unmap` Release the memory utilised to access the hardware peripheral. We do this by making the `munmap` system call.
4. `close_mem` Declare to the operating system kernel that we no longer require access to the memory. We do this by passing the file reference to the `close` system call.

Note that the system call used in [Complete the gen\\_number Function](#) to access the System Timer register is actually performing these above steps. We are now going to replicate how the operating system kernel functions with respect to hardware access.

### Warning!

Accessing memory/hardware using the above technique can reveal security information and the gives programs the ability to alter any aspect of the system. You will need to execute ALL your programs as super user with `sudo` e.g. `sudo ./guessing_game` and `sudo gdb guessing_game` when debugging.

Because the number of functions is growing it is helpful to split the code into multiple files. For this exercise there are two provided files in the `HardwareTimer` directory:

- `Hardware.s` contains the functions outlined above for accessing and mapping the hardware.
- `RandNum_HW.s` A short program for calling the above functions and then using the System Timer register values to generate a random number

The files provided will not assemble and link as they are. You will have to complete all the tasks in this Exercise first. In addition, because we are now working with multiple files they need to be assembled and 'linked' together. The `gcc` compiler makes this easy with the `.include` directive. Directives are



instructions for the compiler and not assembly language instructions. Look in the folder `MultipleFiles` for an example of how to use the `.include` directive. Add the directive when you are ready to compile the files together.

### 5.17. Opening a Memory Location

- i. Use the Reference Sheet to find the system call numbers for each of the system call constants that are listed at the top of `Hardware.s`. Add the values to the program constants.

To open the virtual memory location associated with the peripheral hardware and then retrieve the file descriptor we use the `open` system call. The system call takes two arguments (a third one is optional): a `const char *filename` and an `int flags`.

- ii. Within `RandNum_HW.s` add a string `mem` to your `.data` section for the filename that represents the hardware (it is a good idea to store string constants and variables after the word size constants and variables to prevent memory alignment problems):

```
dev_mem:      .asciz "/dev/mem"
```

### 5.18. Mapping a Memory Location

- i. Within `RandNumHW.s` add a constant `CLOCK_ADDR`. Use the address stated in the introductory text for this session.
- ii. Back in the `main` code block you should call the `map` function with the above value as the parameter (check the `map` function code to see which register you should pass the parameter in).
- iii. The `mmap2` system call returns the memory address to use for accessing the hardware clock. Check the `map` function to determine which register the return value is passed via. Within the `main` code block, just after the call to `map` add code to store this value to the `clockbase` variable.

### 5.19. Un-mapping and Closing (2 points)

The code to read the system timer value has been implemented for you, so at this point your program *should* work - however it is always good practice to clean up after yourself. There are two final functions, `unmap` and `close_mem`, that are called after you have retrieved the System Timer value. `unmap` will use the `munmap` system call to release the mapped memory back to the operating system kernel. The RPi hardware address of the clock is needed as an argument to the `unmap` call.

Having un-mapped the memory we then inform the operating system that access is no longer needed to the peripheral device. We do this by using the `close` system call.

Note - this code should be completed before we can start using the System Timer values. If we repeatedly execute and debug programs that are only opening and mapping without un-mapping and closing then we can cause problems!

- i. Within the `main` code block of `RandNumHW.s` make sure the required parameters are loaded into the registers before each of the calls to `unmap` and `close_mem`. Examine the `unmap` and `close_mem` code to determine what parameters are required.

Great - we can 'open' a memory location that relates to the peripheral timer chip and then map its memory location to one that we can access. Now we check that it is working! Execute the program several times

and use the `echo $?` command each time to check the number generated is different each time. Note also that there is a constant to determine the range of the random number. Can you work out the maximum value?

### 5.20. Adapting the Game (2 points)

*Subsequent exercises are not dependent on the completion of this exercise.*

Can you adapt the game to use the code from the previous exercises? Place a copy of `Hardware.s` and the completed `GameTemplate.s` file inside the empty `HardwareGame` directory. Use these copies for development. Within `GameTemplate.s` you will need to:

- Ensure the required variable and constant definitions are present
- Add a copy of the `gen_number_hardware` function and ensure you call this function instead of `gen_number`

### 5.21. Making the Game Repeat (bonus 1 point)

*Subsequent exercises are not dependent on the completion of this exercise.*

One other way in which we can improve the program is to ask the user if they wish to play again. This should happen after they either win or lose. The main program loop will need modifying and an extra function creating. You'll also need to add a couple of extra prompts and their lengths to the `.data` section. Here is the modified program loop:

```
BL    print_hint        @ Print a hint
CMP    R10, R8           @ If the guess was correct,
BEQ    extra_game       @ go to ask for an extra game.
SUBS    R9, #1           @ Reduce the remaining guesses (!)
BGT    next_guess       @ Try next guess if available
MOV     R0, R8           @ Pass 'hidden' number as argument.
BL     print_lose       @ No guess remaining, you lose.

extra_game:
BL     play_again        @ Ask for another game.
CMP     R0, #0           @ If answer isn't no,
BNE     main            @ restart the game.

_exit:
MOV     R7, #SYS_EXIT    @ exit
SWI     0
```

Here is the new function `play_again` function in C syntax:

```
int play_again()
{
    print( &message, message_len);    // print play again message
    input( &inputString, num_chars);   // read 3 chars from keyboard
    inputString[0] = toLower(inputString[0]);
    inputString[0] = inputString[0] - 'n'; // subtract 'n'
    return inputString[0];
}
```

And in assembly style pseudo code:

```
@ Ask the user if they wish to play again
@ if the user enters 'n' the return value is 0
play_again:
    @ Push used registers to the stack
    @ Load play again prompt reference
    @ Load play again prompt length
    @ Call print function
    @ Load the input reference
    @ Set the string length to read as 3
    @ Call the input function
    @ Load the first byte of the input string into R0
    @ Convert to lower case
    @ Subtract lowercase 'n'
    @ Pop used registers from the stack
    @ Return
```

### Summary of Session 5:

During this session you will have applied some of the knowledge from the lectures/course text and developed some related skills. A summary is given here:

#### Programming on remote machines:

When given step-by-step instructions you are able to edit and execute programs on remote machines. How?

- By using SSH to connect to remote machines
- By using PSFTP to transfer files to/from remote machines

#### Programming in ARM assembly + General Programming Skills:

When given a step-by-step low-level algorithm you are now able to identify the relevant ARM assembly language instructions needed to implement and test that algorithm. You *may* also be able to adapt code that solves a general problem (e.g. division) for use in a given program. How?

- By using gcc to compile source code files and using GDB to debug and inspect executables
- **By moving data from memory to/from registers**
- **By manipulating data within registers**
- **By applying the ARM calling convention to subroutines/functions**
- **By using the stack to store/retrieve register contents.**
- **By forming the core coding structures needed to solve all programming problems: iteration / selection / functions**
- **By iterating over arrays (including strings)**
- **By using (and understanding how and why) system calls are used to perform basic input/output operations**
- By using constants to replace 'magic numbers' and using comments to explain code
- In a processor system with an operating system you may need to use system calls to memory map peripheral devices and then used the mapped addresses in order to configure/read/write to/from those peripherals

**Note that the items in bold are examinable, so ensure you understand the concepts practiced during the sessions!**

During the following session you will extend these skills, but the instructions will be less detailed about the steps you need to follow. You may need to refer back to elements of this session.

**In order to complete session 6 within the scheduled time you will need to read through the exercises and complete as much coding as possible before the session starts.** You will also need to be familiar with locating relevant information in additional technical material such as [GUM] and [BCAMP].

## Session 6 - Accessing the RPi's Hardware

### Using the Gertboard - Background

The RPi has a block of 40 General Purpose Input Output (GPIO) pins used for connecting the RPi to various external hardware devices or which a subset permanently provide reference voltages (ground, +5v, +3.3v). The RPi board also has circuits to implement various communications protocols such as DPI and UART. A set of hardware registers within the RPi act like multiplexer controllers that not only connect the selected circuit (e.g. the DPI or UART) to the required GPIO pins but also configure the pins for input/output. This removes the need for the developer to set the pins manually. These registers also allow *some* of the GPIO pins to be configured manually to output a logic 1/0, or read a logic level. Simply connecting devices like buttons and lights to the GPIO pins is risky; over-voltage or over-current on the GPIO pins can damage the RPi board. An interface is needed.

*Now please read pages 4-10 of the Gertboard User Manual and keep figure 5 handy as a cross reference for the following reading.*

The Gertboard is a secondary hardware device designed to protect the GPIO interface by means of buffers that restrict the connection to 3.3v logic or low current (20mA) devices such as LEDs. Better to burn out a chip than a whole board! The Gertboard also allows for quick connection to devices such as relays, motors and an integrated analogue-to-digital converter. The Gertboard uses 32 of the 40 available GPIO pins which are connected via ribbon cable from the RPi to a header labelled J1 (see [Figure 4](#) below, or Figure 5 in [GUM] , the Gertboard User Manual. Some pins are reference signals (i.e. untouchable). The remaining signals are exposed on header J2 with pin labels GP0-GP25. Header J2 can thus be used for direct connection to the GPIO bus (not advised!).

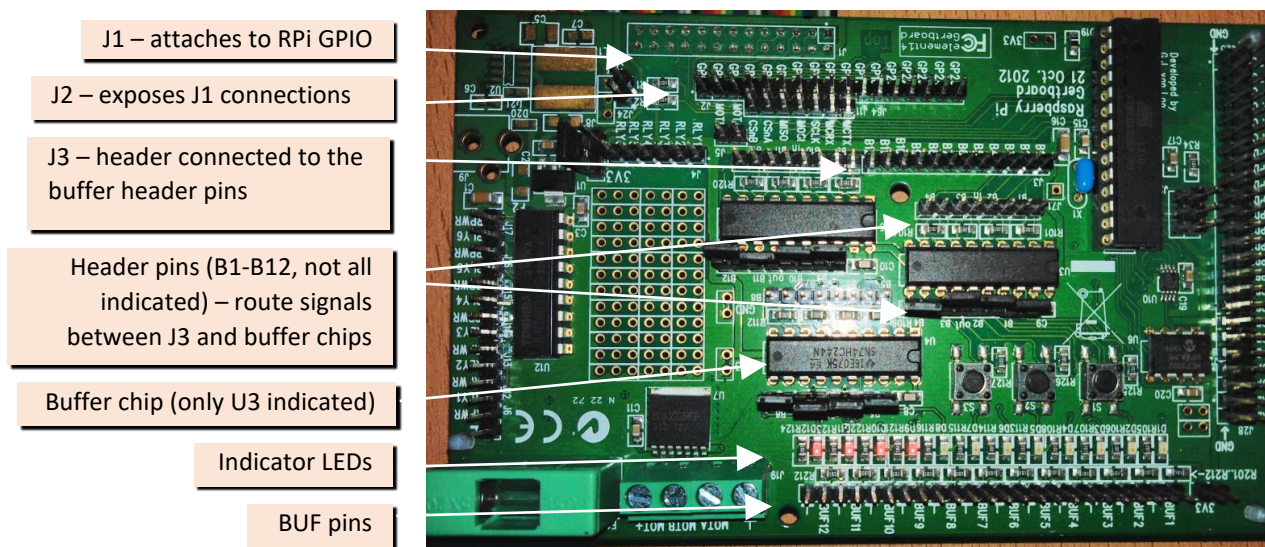


Figure 4 The Gertboard peripheral

As can be seen at the bottom of [Figure 4](#) the board has 12 input/output pins (BUF1 - BUF12) with LED indicators. These pins are connected to 3 buffer chips (labelled U3, U4, U6, each chip buffering 4 pins) by means of header jumpers. A jumper placed on the IN side of a chip (B1-B12 in/out) routes a signal from the relevant BUF pin to the relevant B1-B12 pin on header J3 (i.e. into the RPi). Placing the jumper on the OUT side of the chip routes the signal from the J3 header pin to the BUF pin (i.e. out from the RPi). Three push-button switches are connected to BUF1-BUF3. An explanation of the circuit is given in "Buffered I/O, LEDs, and Pushbuttons" and Figure 8 in [GUM].

If you have followed the above then you will see that to complete the routing of the signals from the RPi to the BUF pins we need to connect some of the GPIO connected pins of J2 to the buffer-header connected pins of J3. We will use pins GP 0,1,4,7,18,21-25 on header J2 for input/output. *Remember this information.*

With reference to [BCMAP table 6-2] it can be seen that there are 8 possible functions for each GPIO pin. Each pin requires 3 bits to act as control inputs to a multiplexer circuit that selects the function. These bits are stored in a control register. Each control register is 32-bits so capable of configuring 10 pins. 2 registers are needed to configure the 12 pins we require. These registers are called `GPFSSEL0` (pins 0-9) and `GPFSSEL1` (pins 10-19, of which we need only pins 10 and 11). As can be seen from table 6-2 in [BCMAP] writing `0000` to the lowest order bits of `GPFSSEL0` sets GPIO pin 0 to input and writing `0010` to it sets it as an output.

Having selected the function for each pin there are two further registers used to set the actual values that will be applied to the GPIO pins.

1. If a pin is configured as output then one needs to set the bit representing the pin concerned in a word that is then written to either `GPSET0` or `GPCLR0` [BCMAP, tables 6-8, 6-10].
2. If a pin is configured as input then one can read the pin value from the pin level registers `GPLEV0` for GPIO pins 0-31 (with the Gertboard, we cannot use pins 32-63). When reading from this register, the values of all the pins are returned [BCMAP tables 6-12, 6-13]. Using bitmasks will allow us to isolate the pins we need.

To complicate matters the GP pin numbers on the Gertboard are not aligned with the GPIO pin numbers on the RPi. By referring to Table 1 in [GUM] you can see the Gertboard pin number labels and the corresponding GPIO label (column "Port on RPi2"). You need this information for an exercise later.

Open the file `LED_Template.s` (in the `Gertboard_LED` directory) and try to understand how this program works – you have seen parts of it in the previous exercises (e.g. open and map the hardware location), but there are functions to implement functionality to set pin functions, to set and clear the output on pins.

The function `set_pin_function(parameters: pin, function)` first calls `check_pin(parameter: pin)` to verify that the pin number passed is available on the Gertboard and not equal to GPIO14 or GPIO15, which is set to UART mode by the Pi. Next, `set_pin_function` determines which `GPFSSEL` register is needed to set the function (lacking a division instruction, this is done by successively subtracting 10 and adjusting the offset of the `GPFSSEL` register with respect to the GPIO base address). Then the three bits corresponding to the pin are cleared (using an `LSR` operand) and set to the required function, which again involves a shift.

## 6.1. Preparing to use the Gertboard

- i. **Ensure the RPi is turned off.** Follow the appropriate workflow steps to do this.
- ii. **IMPORTANT:** Check that the jumper on J7 on the Gertboard is in place as described in [GUM], p.10, Figure. 7. When you connect the Gertboard to the RPi, different Gertboards may show a different state of the LEDs.
- iii. Connect the Gertboard to the RPi and check the alignment so that all the jumper pins are in the header socket as shown below in [Figure 5](#). **DO NOT SWITCH ON THE RPi yet.**



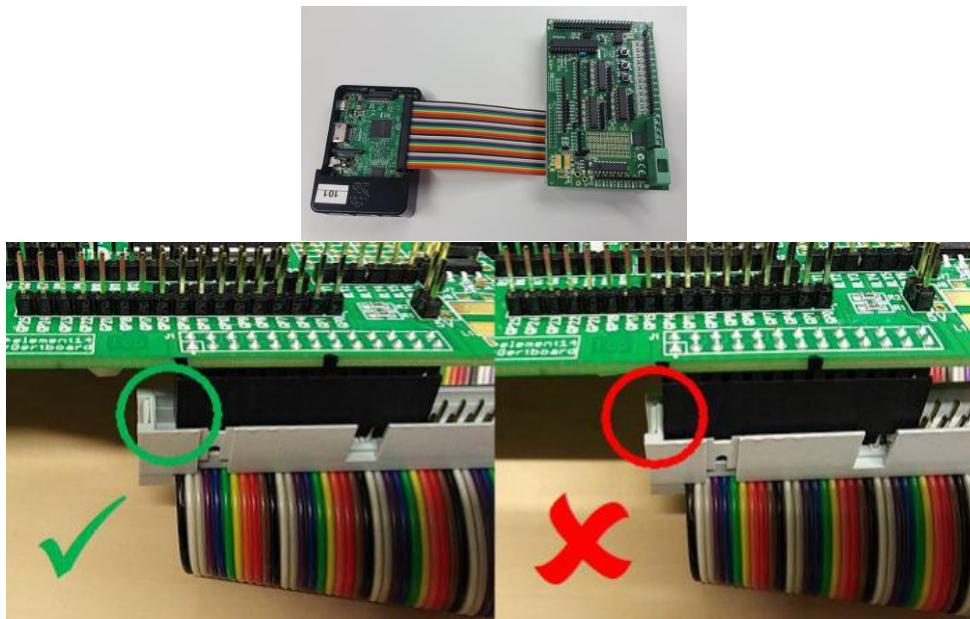


Figure 5 Alignment and connection of the Gertboard and the RPi

You should make sure that you connect the RPi and put the jumper on the Gertboard in the above manner for ALL the Gertboard exercises. Additional jumper and wire connections are as detailed in the exercises.

## 6.2. Switch a Gertboard LED on and Off

Keep [GUM], figure 5 to hand for the following:

- Use a single jumper wire to connect GP22 (header J2) and B4 (header J3).
- Put a jumpers on B4 out (output side of U3).
- Power on the RPi.
- Ensure the `Gertboard_LED` directory is on the RPi (if not then use PSFTP to transfer it).
- Place a copy of your `Hardware.s` file in the same directory as `LED_Template.s`.
- Compile `LED_Template.s` and execute the program (use the `sudo` command). What do you expect to see and what do you observe? Notice that writing a 1 to the bit corresponding to pin 22 in the `GPSET0` register (offset `0x1C` from the GPIO base address) sets this pin to 1. Before that, the pin has been configured for output with one of the `GPSELn` registers.
- Change the program so that it clears pin 22. For this, select the pin and write a 1 to the bit corresponding to pin 22 in the `GPCLR0` register (offset `0x28` from the GPIO base address).
- Assemble, link and execute the program (using `sudo`) as above so the LED is turned off.
- Add in the following constants to your program and make the necessary changes so your code uses them.

```
.equ    GPCLR0, 0x28      @ Value to set a GPIO pin to OFF
.equ    GPSET0, 0x1C      @ Value to set a GPIO pin to ON
.equ    GERT22, 22        @ RPi GPIO to gertboard mappings
```

- Create two versions of the program, one to turn the LED on and one to turn it off. Alternate executing each.



### 6.3. Playing with LEDs (2 points)

To gain confidence in your programming do the following;

- i. Change the programs so that a different LED is turned on or off. You will need to:
  - a. **Turn off the RPi** and connect another GP pin (on J2) to a buffer pin (on J3). The choice is yours (similar to step 6.8i above)
  - b. Add a jumper on the OUT side of the buffer chip (similar to step 6.8ii above).
  - c. Adjust the code so that the relevant GPIO pin is used. You can work out the GPIO pin using Table 1 in [GUM]. The relevant columns are mentioned in the [Gertboard Introduction](#).
- ii. Change the programs so that you can turn two or more LEDs on or off simultaneously. You will also need to make changes to the wiring/jumpers.

### 6.4. Using a Delay to Blink an LED (2 points)

This exercise will involve some advanced planning as you will have to implement loops and create functions given only the pseudo code. Within the loops we are going to implement delays using the system timer. (Can you think why a countdown loop is not practical when using an RPi with an operating system?). We also need to map both the System Timer and the GPIO ports, for this reason the file `Hardware2.s` has been provided.

- i. Put a copy of your final `LED_Template.s` from the previous exercise within the `Gertboard_Blink` directory. Rename it `BlinkTimer.s`.
- ii. At the start of your LED program replace the calls to `open_mem` and `map` with a single call to `map_io`, don't forget to add the necessary `.include`.
- iii. At the end of your LED program (just after the `exit:` line) replace the `unmap` and `close_mem` calls with a single call to `unmap_io`.
- iv. Examine the file `Wait.s`. You need to use the function in this file within your program, so use the `.include` directive again.
- v. Implement the following algorithm in your main function:

```
@ Set counter = 10
blink_loop:
    @ Turn an LED on
    @ Load the delay (ms) into R0
    @ Call the wait function
    @ Turn the LED off
    @ Load the delay (ms) into R0
    @ Call the wait function
    @ Decrement counter
    @ IF counter > 0
    @ THEN Branch to blink_loop
    @ ELSE End Program
```

*Does it flash 10 times? Congratulations! Now produce a version of the program that has a different total and speed of flashes.*

## Working with multiple LEDs

We will now extend the basic functionality to display a 10-bit binary number. The lowest order bit will be on GP12 (LED12), highest order on GP3 (LED3). LED1 and LED2 are not used. If GP12 (Jumper J3 is connected to GP0 on header J2) then GP3 will be connected to GP24 on header J2). The intermediate pin connections would be relatively straightforward, except that the GPIO pin numbers are not linearly related to the Gertboard pin numbers<sup>1</sup> on header J2. Refer to 6.1 for details on which GP pins on header J2 we will use and also how to find out which GPIO pins they are connected to. To replicate this map for easy use in code we will create an array where the index is the bit position of a binary number and the value is the GPIO port for that bit. The first element has address `disp_bits` and each element is a word in length;

```
disp_bits:
.word    0x4          @ bits[0]: GPIO2  represents bit 0 (GP0 on GB)
.word    0x8          @ bits[1]: GPIO3  represents bit 1 (GP1 on GB)
...
.word    0x8000000    @ bits[6]:GPIO27 represents bit 6 (GP21 on GB)
...
```

To understand how this mapping works, open the `Binary.s` file in the `Gertboard_Binary` directory and convert each of the hexadecimal values in the array to binary and look for the relationship between the binary form and the GPIO pin number. The array is not complete (yet!).

If we wish to represent the number 2 in binary then we need to turn on the LED for the 2<sup>nd</sup> bit (00 0000 0010). To determine the GPIO pin to use for the 2<sup>nd</sup> bit (index 1) of our binary number we look for the value at address `disp_bits+(1 x word)`, in this case the value is 0x8. If `disp_bits[i]` is written to the set (clear) register, the LED representing bit *i* is set (cleared).

Most numbers will require multiple bits to be set and therefore multiple words. Because we now have a map for bit position and GPIO pin we can create an iterative code block that loops over our binary number bit by bit and determines which GPIO pin will be used to represent each bit. OR-ing all these values together gives a single bit pattern for all the LEDs that need to be turned on. Look up the description in the data sheets to see why writing this to `GPSET0` will not clear the pins you write a 0 to.

### 6.5. Wiring the Gertboard for multiple LEDs

Keep [GUM] figure 5 to hand for the following:

- i. **Power down** the RPi and connect *GP0* and *B12*, *GP1* and *B11*, *GP4* and *B10*, *GP7* and *B9*.
- ii. Connect pins *GP17-GP24* and *GP25* (header J2) with *B8-B3* and *B1* (header J3) respectively
- iii. Place output jumpers for *B3-B12* (output side for *U3*, *U4*, *U5*).

This above connections are also stated in the top comments of `Binary.s`.

### 6.6. Displaying a binary number (2 points)

- iv. **Power on** the RPi. Copy your `Hardware2.s` file into the `Gertboard_Binary` directory.
- v. Within the `.text` section of the code in `Binary.s` complete the `disp_bits` array (partially implemented above). We have already given you an idea about where to find the details of the GPIO numbers for the GP pins on header J2 that we will use.
- vi. You will need to use a mask value to ensure unused bits are cleared. The mask value is the logical OR of all the values in the array. Create a constant for this value called `DISP_MASK`.

<sup>1</sup> For the original Raspberry Pi version, the GP pin labels of the Gertboard match the GPIO pins of the Raspberry. For Raspberry Pi versions 2 and 3, this is no longer the case. In particular GP0, GP1 and GP21 are mapped to GPIO2, GPIO3 and GPIO27, respectively

- vii. Add all the constants and variables in a `.data` section within `Binary.s` from previous exercises so that you can store the mapped hardware addresses.
- viii. The template file should now be complete and when compiled all the LEDs will be on. Why is the `MVN` instruction used to turn all LEDs on?
- ix. Play with the code to change the value to be displayed. Should you use `MOV` or `MVN`?

### 6.7. Playing with the code (2 points)

To practice using the functions you have so far, implement the following variations of the binary number program. Create multiple source files if needed e.g. `counter1.s`, `counter2.s`. You **must** comment your code for points to be awarded.

- i. (1 point) Create a program that counts from 0 to  $2^{10}-1$  repeatedly. The count is shown on the LEDs and there is a pause between each count increment. The program should stop after a fixed number of cycles (your choice of how many).
- ii. (1 point) The program waits for a keypress to begin, if the number 1 is pressed the program counts from 0 to  $2^{10}-1$  for a fixed number of cycles. If any other value is pressed the program counts from  $2^{10}-1$  to 0 for a fixed number of cycles (your choice of how many).

### Finishing the session

- As with the previous session you should remove all your files from the RPi.
- When the RPi has been shut down remove any jumper cables and pin jumpers and then carefully disconnect the header cable from the RPi (you may leave it attached to the Gertboard).

### Summary of Session 6

During this session you will have applied some of the knowledge from the lectures/course text and developed some related skills. A summary is given here:

#### Programming with peripherals:

- Integrating, and using peripherals, involves consideration of signal levels/types.

#### Programming in ARM assembly + General Programming Skills:

Some additional techniques were developed/reinforced in this session:

- **Iterating over arrays and indexing into arrays**
- **Using structs to store data**
- **Passing values and addresses (references) as parameters.**

**Note that the items in bold are examinable, so ensure you understand the concepts practiced during the sessions!**

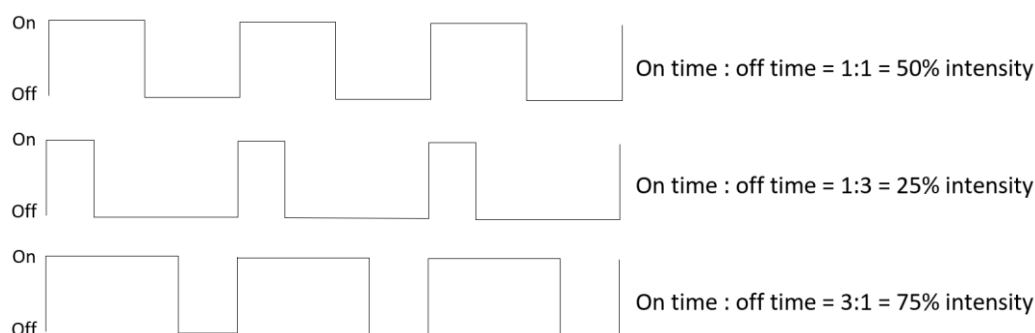
The following session will require you to invest in some preparation time. You should read through the instructions and complete as much of the coding as possible before the session starts. Note the submission requirements for the final exercises.

## Session 7 – LED Intensity Controller

In these last exercises you are going to draw together the techniques learnt in order to vary the intensity of an LED using PWM. In order to do this we will need to:

1. Understand Pulse Width Modulation.
2. Develop an algorithm that implements PWM in code.
3. Adapt our existing code to implement the algorithm.

It is possible to control the intensity of a light, such as an LED (even if that light can only be switched on or off) by means of a technique called Pulse-Width Modulation (PWM). This is illustrated in Figure 11 whereby the ration of time the light is on, to the time the light is off results in a given intensity. This is relative to the intensity of light had it been switched on all the time.



*Figure 6 Using PWM to adjust the intensity of an LED*

Our slow human eyes will not notice the switching, provided this occurs fast enough. If the light is switched on and off with a frequency of 100Hz, for example, this is more than fast enough to prevent our eyes seeing the flickering of the light, even when it is an LED. Thus, by varying this ration of one-time to off-time we can vary the apparent intensity of the light.

PWM is not only used to control lights - the so-called dimmers one finds in homes to control lights actually employ a form of PWM - but also circuits with higher power drains like electric motors. These can vary from the tiny motors in toy trains to the large traction motors found in (real) train locomotives. Actually, PWM is an energy efficient way to control the speed and power of such motors.

PWM can be realised very simply in dedicated digital circuits, as it does not require much more than an oscillator - a clock - and a counter or two. Our RPi processor, however, is pretty fast, fast enough, actually, to allow implementation of PWM completely in software. The timer in the RPi runs at 1 MHz, so it has a resolution of 1  $\mu$ s: the smallest possible timer step is 1  $\mu$ s. To obtain a repetition frequency of 1000 Hz requires 1000 such timer steps: 1000 times 1  $\mu$ s equals 1 ms, which is the period of a 1000 Hz signal.

PWM is obtained as follows: one period of the 1000 Hz signal consists of 1000 timer steps. At the beginning of each period an LED is switched on, and after  $n$  steps the LED is switched off, where  $0 \leq n \leq 1000$ . The LED is then aswitched off for 1000- $n$  steps. Variable  $n$  now determines the width of the pulses and, thus, the apparent intensity of the light emitted by the LED. If, for instance,  $n=0$  the LED will not be switched on at all, so they will remain dark. If  $n=1000$  the LED will remain switched on permanently (maximum brightness); and if  $n=500$ , the LED will be switched off halfway through the 1 ms period and will appear to have half the maximum intensity. Generally, the fraction of the time the LED is on equals  $n/1000$ , which is equal to  $n\%$ : this fraction is called the duty cycle of the pulse-width modulated signal. This way, the intensity of the LED equals  $n\%$  (of its maximum possible value) too. Because  $n$  is a natural number, the intensity of the LED thus can be controlled in 1000 steps, which is more than fine-grained enough for our purposes.

### 7.1. Implementing Pulse-Width Modulation (3 points max)

- a) Implement a PWM algorithm first for a single dimmed LED (2 points) and then extend this to multiple dimmed LEDs (3 points). In addition, a further single LED should always be on at full intensity. This LED will serve as a reference when viewing other LEDs to ensure that are actually dimmed. Use the `Gertboard_PWM` directory to contain your code. An example algorithm for the PWM control of an LED could be:

```
@ turn on reference LED
@ Set on_time = 500
@ Set off_time = 1024 - on_time
pwm_loop:
@ Turn PWM LED on
@ Wait on_time
@ Turn PWM LED off
@ Wait off_time
@ Branch to pwm_loop
```

Your program should not run indefinitely, can you think of a way to force the program to terminate after a fixed length of time?

### 7.2. Adapting the PWM Program (3 points)

- a) Create an array of 5 different brightness ratios. i.e. the array values should correspond to the on-time of the LED. Your program should prompt the user for a numerical value from 0-5 and set the brightness of the LED accordingly. Ensure that your program validates the user entry and deals with unexpected values accordingly.

## Summary of Session 7

### Congratulations!

You should now be familiar with a number of core aspects of microprocessor architecture, assembly programming and hardware interfacing.

- 1. Moving data in and out of registers. Data can come from memory locations (variables), constants, other registers, or the result of logical and mathematical instructions. Data can be moved to other registers or memory locations.**
- 2. Manipulating data in registers through mathematical and logical operations such as addition, subtraction, bit wise AND, OR etc. and logical shifts or rotates.**
3. Using masks to isolate a bit, or bits, within a value and also limit the value in a register.
- 4. Using functions to contain blocks of frequently used code. This aids in program design (abstraction and decomposition) and in keeping programs compact.**
- 5. Passing multiple parameters to functions and returning 1 or more values from functions. Parameters can be passed as values or as addresses (references).**
6. Using the stack to store values temporarily. Typically used on entry to a function so that scratch registers are preserved.
7. Using arrays and structs to group items of data together.

8. Using system calls to access and manipulate hardware. This involves setting parameters and issuing software interrupts.
9. **Using conditional branch statements to create the basic code structures; selection (if, if...else), iteration (while...do, do...while)**
10. **Using iteration structures to create counters (for loops) and iterate instructions.**
11. **Using iteration to access/process data within arrays**
12. Having a basic level of electronic understanding in order to interface with hardware (external to the processor system).

The items in bold are examinable – so make sure you revise these elements!