7: ROS Ins and Outs

When we use computers and operating systems such as Raspberry Pi OS, we take an awful lot for granted. We type commands at the keyboard, these get 'actioned' and more often than not, provide output in the form of information or results by way of what is displayed on the screen. There's a lot going on.

Consider a couple of what would seem relatively simple tasks, typing a command at the keyboard and then getting a response on the screen. These are things we do every time we interact with the Raspberry Pi OS command line. The question is then, how do we get input from the keyboard and write information to the screen in our machine code programs?

In the strictest sense you do it yourself. But this involves a good deal of knowledge about the various hardware components of the Raspberry Pi, because to write a message to the screen for instance, we have to know exactly where the hardware that drives the screen is located within the computer's memory and, in turn how to write the information to it. Equally, to read input from the keyboard we need to understand how the keyboard is mapped and how to read that matrix to identify which keys are being pressed.

Reading and writing to the hardware to do this is often termed bare metal programming, because you are 'talking' to the computer hardware directly. Whilst this is potentially exciting, it is rather an advanced topic and not necessarily the domain of a beginner's book such as this. Equally though unless you are specifically bare metal programming as an exercise there is absolutely no need for you to do it. Instead we can access the operating systems own routines to do this.

SWI and SVC Commands

The SWI instruction allows you as the programmer to gain access to predefined routines or libraries of operating systems functions. SWI stands for SoftWare Interrupt because when it is encountered it causes the flow of your program to be paused and handed over to the appropriate routine. Once the SWI instruction has been completed, control is handed back to the calling program which can continue on its way. The SWI command is also often referred to as SVC or SuperVisor Call as this is a mode of operation that is invoked in the ARM chip when called.

You will probably recall that we have used a SWI command in all our assembler programs so far. We used it to exit the code back to the command line prompt. This use took the form:

```
MOV R7, #1
SWI 0
```

All SWI calls are executed with SWI 0 (or SVC 0 can be used instead). The actual function to be performed is determined by the number held in register R7. This is called the Operating System Call or 'Syscall' for short, number. In addition, other registers may also have to be seeded with information, so a call to SWI 0 often requires some setting up before being executed. For example, to write a string of characters to the screen three other items of information must be placed in specific registers.

To use these SWI calls effectively then, we need to know what they do, what information must be passed and into what registers. Information may be passed back by the SWI call and in such cases, we need to know what information and in what registers.

Appendix B contains a list of the Syscalls available in the Raspberry Pi OS. A detailed description of all the SWI calls is not provided, but the more common and useful ones are described at various points in this book. No official list of Syscalls exist but there are various sources on independent websites.

Let's look at what are arguably the two most important Syscalls at this stage in our learning —printing to the screen and reading from the keyboard. These are important as we will use them a lot in the program examples in the rest of this book. In using them we will need to look at a few more features of the GCC assembler and use a few assembly language techniques that we won't learn about in detail until later.

Writing to the Screen

To write a sequence or string of ASCII characters to the screen we need to use the 'write' function. This is Syscall 4. The parameters required by Syscall 4 are as follows:

```
R0= determines output stream, 1 for the monitor
R1= the address of the string of characters
R2= the number of characters to be written
R7= the number of the Syscall, so R7=4
```

(Incidentally, ASCII stands for American Standard Code for Information Interchange, and an ASCII code is a simple number used to represent the character. Appendix A contains the ASCII character table, and this is universal in acceptance as a standard.)

The GCC assembler provides us with a facility to store an ASCII string of characters within the body of our machine code file. Program 7a illustrates the setup for this. The key here is to note the GCC assembler directive '.ascii' on

the last line. This directive informs the assembler that an ASCII string of characters follows the string that is enclosed by quotes. You will also notice '\n' at the end of the character string, but within the quotes. The backslash character signifies that the next character is a 'control-character' and as such has an action. Here '\n' means generate a new line. A label is used to mark the start of the location of the string — in this case I have been original and called it 'string'.

Program 7a. Syscall 4 to write a string to the screen.

```
/* How to use Syscall 4 to write a string */

     .global  _start
_start:

     MOV R7, #4             @ Syscall number
     MOV R0, #1             @ Stdout is monitor
     MOV R2, #19            @ string is 19 chars long
     LDR R1,=string         @ string located at string:
     SWI 0

_exit:
                            @ exit syscall
     MOV R7, #1
     SWI 0

.data
string:
.ascii "Hello World String\n"
```

Create, assemble, and link the program and try it out for yourself. The instruction:

```
LDR R1,=string
```

Can be read as:

```
LoaD Register R1 with the address of the label string:
```

When Syscall 4 is made it identifies the output stream, the 1 passed in R0 defines the standard output device, the monitor. It then extracts the length of the string from R2 and prints that number of characters out starting at the address held in R1. The number of characters held in R2 includes spaces and any punctuation. The final '\n' character is regarded as one character. Try altering the value loaded into R2 and see if you can predict the result. For example, try:

```
    MOV R2, #11
```

You will also note that there is an extra directive in the program:

```
    .data
```

This informs the assembler that what follows should be treated as a subsection containing data, as opposed to assembly language code.

The data subsection could have been placed at the start of the source file had we desired. We would then have needed to signify the start of the assembly language subsection by using a directive thus:

```
    .text
```

How you structure your files is entirely a matter of which way you wish to work. I tend to prefer placing data and data areas at the end of programs to avoid any alignment problems. You may recall from Chapter 5 that ARM machine code must be assembled on four-byte word boundaries, in other words start at an address that is directly divisible by four. This may not be the case if a string of 10 characters was used, for example. This can be corrected by using an align directive, something discussed later.

Reading from the Keyboard

To read a sequence (or string) of ASCII letters from the keyboard we need to use the 'read' function. This is Syscall 3. The parameters required by Syscall 3 are similar to Syscall 4 and are as follows:

```
    R0=  input stream, this is 0 for the keyboard
    R1=  the address of the buffer for the string of
         read characters to be placed
    R2=  the number of characters to be read
    R7=  the number of the Syscall, so R7=3
```

You can use Program 7a as a basis for the new program. You can make a copy of it at the command line by using the cp command as follows:

```
     cp prog7a.s prog7b.s
```

Now edit the source file to contain the new _read routine. The entire program is given below.

Program 7b. Syscall 3 to read from the keyboard.

```
/* How to use Syscall 3 to read from keyboard */

    .global _start
_start:
_read:
                        @ read syscall
    MOV R7, #3          @ Syscall number
```

```
    MOV R0, #0          @ Stdin is keyboard
    MOV R2, #5          @ read first 5 characters
    LDR R1,=string      @ string placed at string:
    SWI 0

_write:
                        @ write syscall
    MOV R7, #4          @ Syscall number
    MOV R0, #1          @ Stdout is monitor
    MOV R2, #19         @ string is 19 chars long
    LDR R1,=string      @ string located at string:
    SWI 0

_exit:
    @ exit syscall
    MOV R7, #1
    SWI 0

.data
string:
.ascii "Hello World String\n"
```

Here we still need to define the ASCII string. I have purposely left the original text in place so that you can see what results from using the function. The label string: points to what is effectively a buffer or place for the input read from the keyboard to be placed. We could have just defined an empty string, for example:

```
    .ascii "                                        "
```

(There are other ways to reserve empty spaces in memory in programs and these will be discussed later.)

R2 is now used to hold the number of characters we want from the read process. It is important to remember that this is not the number of characters that can be typed. When _read: is executed it accepts all input at the keyboard until the Return key is pressed. Only at that stage does it extract the first x characters as defined by the value in R2. Thus typing:

```
    123456789
```

At the keyboard would see 12345 (the first five characters) placed into the string buffer. The rest would then be dealt with as though a Bash command had been entered and therefore generates an error message. (Bash being the name given to the Raspberry Pi OS command line shell you have been working within.) The '_write:' routine would print out the newly created string which in this instance would be:

```
12345 World String
```

12345 having overwritten 'Hello'.

Run the program again and just type in:

```
12
```

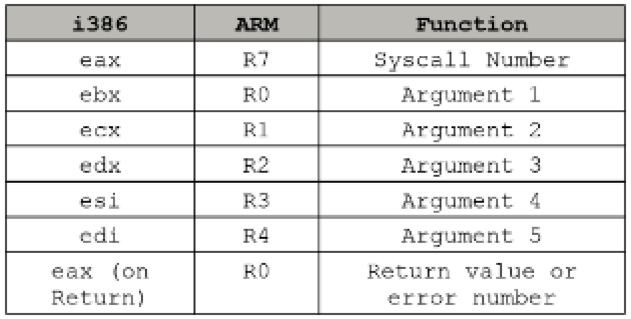
Now the string printed is:

```
12
lo World String
```

Note here that a newline has been generated. This is because the <Return> was inserted into the string buffer as well. We will come back to Syscalls in Chapter 18.

eax and Others

Much of the Syscall documentation you come across with have been written with non-ARM machines in mind and specifically i386 processor systems. As such you will find yourself dealing with an alien set of register references. Figure 7a lists these registers and their ARM equivalents which should assist you in breaking down what needs to go where.



i386	ARM	Function
eax	R7	Syscall Number
ebx	R0	Argument 1
ecx	R1	Argument 2
edx	R2	Argument 3
esi	R3	Argument 4
edi	R4	Argument 5
eax (on Return)	R0	Return value or error number

Figure 7a. 386 v ARM registers for Syscalls.

Makefile

So far as we have developed new source files and we have assembled and linked the files by typing the commands at the command line. This is repetitive but made easier by the Terminal history feature. By using the up and down arrow keys while in Terminal you can scroll through previously entered commands, which in turn can be edited.

GNU also provide a very clever piece of software called 'Make'. This is a tool that allows programmers to control the generation of executable files from a single controlling file. When you see a piece of software installing on your

computer then chances are that the whole process is bring controlled by a Make file. Make is a very sophisticated tool and you can find out more about it in detail from the GNU website.

Program 7c is a source file (although you save it without the '.s' suffix) that will automate the whole assemble and link process for you. It is extremely flexible and can deal with most possibilities. (Note the '#' characters are equivalent of the '@' in assembler files. They allow comments to follow.)

Program 7c. Automate assembly and linking with Make.

```
PROGRAMS = prog7a prog7b

# If we've supplied a goal on the command line
# then set it as the list of programs we already know about.
ifneq ($(MAKECMDGOALS),)
     ifneq ($(MAKECMDGOALS),clean)
          PROGRAMS = $(MAKECMDGOALS)
     endif
endif

# The default rule if none specified on the command line
all: $(PROGRAMS)

# Make knows how to compile .s files, so all
# we need to do is link them.
$(PROGRAMS): % : %.o
        ld -o $@ $<

clean:
        rm -f *.o $(PROGRAMS)
```

Create the above file and call it 'makefile' - there is no need to append an '.s' to the filename, just plain 'makefile'. Ensure this file is aved in the same directory as your source file. Also, note that the two lines:

```
ld -o $@ $<
```

and

```
rm -f *.o $(PROGRAMS)
```

must be indented by a *single tab character* for Make to work.

Run the makefile by typing, at the command line prompt:

```
make
```

The variable PROGRAMS (first line in makefile) is being used to hold the names of the source files to be assembled and linked. You can enter one or as many names as you like here, with each being separated by a space. And effectively that is all you need to do. The rest of the program will assemble each of the files, create the object files and then link them.

In the listing above this would mean the source files called prog7a and prog7b. Note the '.s' suffix is implied, and you do not need to include it.

```
PROGRAMS = prog7a prog7b
```

This also implies the makefile exists in the same directory as your source code files. If the source files have previously been assembled and linked they will be overwritten provided the target files are older than their associated source files. If the files do not exist, then make will complain with an error message. You can 'force' the re-make by using:

```
make -B
```

If you want to assemble and link a specific file or files, you can enter the file name after the commands thus:

```
make prog7a
```

This would assemble and link the source file called 'prog6a' provided it existed. This would be in preference to any filenames listed on that first makefile line.

At the command line, typing:

```
make clean
```

Will delete the '.o' files from the directory based on initial definition for PROGRAMS.

It makes good sense to include a makefile in each and any directory where you create and save your source files that need to be assembled and linked.

The 'make' utility is very versatile is and can be utilised in several ways. If you wish to study 'make' in more detail the GNU website has plenty of manuals and examples at:

```
www.gnu.org/software/make/
```

Some further 'makefile' examples are provided at further points in the book, and the source files that can be downloaded as previously mentioned include relevant makefile examples in the respective directory, where appropriate.

If you downloaded the program files from my website, you would find that a makefile (or similar) is included for each of the programs if appropriate.