

*Real Time Operating Systems Design and Programming*

## **LAB 1**

### **OS Assembly Lab**

### **Processing Text in Assembly Language**

# Contents

<b>1</b>	<b>Overview.....</b>	<b>1</b>
<b>2</b>	<b>SOFTWARE.....</b>	<b>1</b>
2.1	Mixing Assembly Language and C Code.....	1
2.2	Main.....	1
2.3	Register Use Conventions.....	1
2.3.1	Calling functions and Passing Arguments.....	2
2.3.2	Temporary storage.....	2
2.3.3	Preserved Registers.....	2
2.3.4	Returning from Functions.....	2
2.4	String Copy.....	2
2.5	String Capitalisation.....	2
<b>3</b>	<b>Lab Procedure.....</b>	<b>3</b>

# 1 Overview

In this exercise you will execute assembly code on your board using the debugger in order to examine its execution at the processor level. This lab aims to get you familiar with the development environment and understand how the hardware “plumbing” is hard.

## 2 SOFTWARE

### 2.1 Mixing Assembly Language and C Code

We will use MDK to produce a C program, but add assembly language subroutines to perform the string copy and capitalisation operations. Some embedded systems are coded purely in assembly language, but most are coded in C and resort to assembly language only for time-critical processing. This is because the code *development* process is much faster (and hence much less expensive) when writing in C when compared to assembly language. Writing an assembly language function which can be called as a C function results in a modular program that offers us the best of both worlds: the fast, modular development of C and the fast performance of assembly language. It is also possible to add *inline assembly code* to C code, but this requires much greater knowledge of how the compiler generates code.

### 2.2 Main

First we will create the main C function. This function contains two variables (a and b) with character arrays.

```
int main(void)
{
    const char a[] = "Hello world!";
    char b[20];

    my_strcpy(a, b);
    my_capitalize(b);

    while (1)
        ;
}
```

### 2.3 Register Use Conventions

There are certain register use conventions which we need to follow if we want our assembly code to coexist with C code. They are not particularly important to this course, but the point is to show you how OS developer must be careful about such hardware details.

### 2.3.1 Calling functions and Passing Arguments

When a function calls a subroutine, it places the return address in the link register `lr`. The arguments (if any) are passed in registers `r0` through `r3`, starting with `r0`. If there are more than four arguments, or they are too large to fit in 32-bit registers, they are passed on the stack.

### 2.3.2 Temporary storage

Registers `r0` through `r3` can be used for temporary storage if they were not used for arguments, or if the argument value is no longer needed.

### 2.3.3 Preserved Registers

Registers `r4` through `r11` must be preserved by a subroutine. If any must be used, they must be saved first and restored before returning. This is typically done by pushing them to and popping them from the stack.

### 2.3.4 Returning from Functions

Because the return address has been stored in the link register, the `BX lr` instruction will reload the `pc` with the return address value from the `lr`. If the function returns a value, it will be passed through register `r0`.

## 2.4 String Copy

The function `my_strcpy` has two arguments (`src`, `dst`). Each is a 32-bit long pointer to a character. In this case, a pointer fits into a register, so argument `src` is passed through register `r0` and `dst` is passed through `r1`.

Our function will load a character from memory

```
_asm void my_strcpy(const char *src, char *dst)
{
loop
    LDRB    r2, [r0]      ; Load byte into r2 from mem. pointed to by r0 (src pointer)
    ADDS    r0, #1        ; Increment src pointer
    STRB    r2, [r1]      ; Store byte in r2 into memory pointed to by (dst pointer)
    ADDS    r1, #1        ; Increment dst pointer
    CMP     r2, #0        ; Was the byte 0?
    BNE     loop          ; If not, repeat the loop
    BX      lr            ; Else return from subroutine
}
```

## 2.5 String Capitalisation

Let's look at a function to capitalise all the lower-case letters in the string. We need to load each character, check to see if it is a letter, and if so, capitalise it.

Each character in the string is represented with its ASCII code. For example, 'A' is represented with a 65 (0x41), 'B' with 66 (0x42), and so on up to 'Z' which uses 90 (0x5a). The lower case letters start at

'a' (97, or 0x61) and end with 'z' (122, or 0x7a). We can convert a lower case letter to an upper case letter by subtracting 32.

```
__asm void my_capitalize(char *str)
{
    cap_loop
        LDRB    r1, [r0]      ; Load byte into r1 from memory pointed to by r0 (str pointer)
        CMP     r1, #'a'-1    ; compare it with the character before 'a'
        BLS     cap_skip      ; If byte is lower or same, then skip this byte



        CMP     r1, #'z'      ; Compare it with the 'z' character
        BHI     cap_skip      ; If it is higher, then skip this byte

        SUBS    r1, #32        ; Else subtract out difference to capitalize it
        STRB    r1, [r0]      ; Store the capitalized byte back in memory
    cap_skip
        ADDS    r0, r0, #1     ; Increment str pointer
        CMP     r1, #0         ; Was the byte 0?
        BNE     cap_loop      ; If not, repeat the loop
        BX      lr            ; Else return from subroutine
}
```

The code is shown above. It loads the byte into r1. If the byte is less than 'a' then the code skips the rest of the tests and proceeds to finish up the loop iteration.

This code has a quirk –the first compare instruction compares r1 against the character immediately before 'a' in the table. Why? What we would like is to compare r1 against 'a' and then branch if it is lower. However, there is no branch lower instruction, just branch lower or same (BLS). To use that instruction, we need to reduce by one the value we compare r1 against.

### 3 Lab Procedure

1. Copy the main.c file provided to your empty project.
2. Compile the code. 
3. Load it onto your board. 
4. Run the program until the opening brace in the main function is highlighted. Open the Registers window (View->Registers Window) What are the values of the stack pointer (r13), link register (r14) and the program counter (r15)?

**sp = 0x2000\_18F8, lr = 0x0800\_0217, pc = 0x0800\_0640.**

5. Open the Disassembly window (View->Disassembly Window). Which instruction does the yellow arrow point to, and what is its address? How does this address relate to the value of pc?

**SUB sp,sp,#0x28 is at address 0x0800\_0640, which is the value of pc. This is the next instruction which will be executed.**

6. Step one machine instruction forward using the F10 key while the Disassembly window is selected. Which two registers have changed (they should be highlighted in the Registers window), and how do they relate to the instruction just executed?

The stack pointer r13 has changed to 0x2000\_18D0, resulting from subtracting 0x28 from 0x2000\_18F8. The program counter r15 has changed to 0x0800\_0642, resulting from executing the subtract instruction (which is two bytes long).

7. Look at the instructions in the Disassembly window. Do you see any instructions which are four bytes long? If so, what are the first two?

Yes: BL.W my\_strcpy and BL.W my\_capitalize.

8. Continue execution (using F10) until reaching the BL.W my\_strcpy instruction. What are the values of the sp, pc and lr?

sp = 0x2000\_18D0, lr = 0x0800\_0217, pc = 0x0800\_0652.

9. Execute the BL.W instruction. What are the values of the sp, pc and lr? What has changed and why? Does the pc value agree with what is shown in the Disassembly window?

sp = 0x2000\_18D0, lr = 0x0800\_0657, pc = 0x0800\_0656. lr has changed because the bl.w instruction saved the return address (old value of PC + length of bl.w instruction + 1). pc has changed because the pc is loaded with the address of the subroutine to execute. Yes, the PC matches the disassembly window contents – the yellow arrow points to the instruction at 0x0800\_0656.

10. What registers hold the arguments to my\_strcpy (src and dst), and what are their contents?

src: register r0, value 0x2000\_18F5

dst: register r1, value 0x2000\_18E1

11. Open a Memory window (View->Memory Windows->Memory 1) for the address for src determined above. Open a Memory window (View->Memory Windows->Memory 2) for the address for dst determined above. Right-click on each of these memory windows and select ASCII to display the contents as ASCII text

12. What are the memory contents addressed by src?

Hello world!

13. What are the memory contents addressed by dst?

Null characters, displayed as ..... in ASCII mode.

14. Single step through the assembly code watching memory window 2 to see the string being copied character by character from src to dst. What register holds the character?

r2

15. What are the values of the character, the src pointer, the dst pointer, the link register (r14) and the program counter (r15) when the code reaches the last instruction in the subroutine (BX lr)?

`r2 = 0x0000_0000, src = r0 = 0x2000_18F5, dst = r1 = 0x2000_18E1, lr = 0x0800_0657, pc = 0x0800_0234`

16. Execute the BX lr instruction. Now what is the value of PC?

`pc = 0x0800_0656`

17. What is the relationship between the PC value and the previous LR value? Explain.

`pc` is `lr-1`. The processor resumes executing code at address `0x0800_0656`(`pc`), but the last bit of the `lc` is set to indicate the processor is executing in Thumb mode. This is due to the fact that the first bit of `pc` is always set to 0 so instructions are always aligned to word or halfword boundaries.

18. Now step through the `my_capitalize` subroutine and verify it works correctly, converting `b` from "Hello world!" to "HELLO WORLD!".

In embedded software, all these do not have to be handled by an OS (by compiler); however, OS usually does similar housekeeping and makes the development abstracted from the low level hardware details.