

LAB1: PROCESSING TEXT IN ASSEMBLY LANGUAGE EXAMPLE

OVERVIEW

In this exercise you will write and compile assembly code and debug the program image on an mbed board (namely the Nucleo F401RE board) using the Keil MDK-ARM tool.

Before you embark on this lab, you need to go through the “Getting Started” document which includes instruction on how you can setup the necessary firmware.

LAB EXERCISE

MIXING ASSEMBLY LANGUAGE AND C CODE

We will use Keil MDK with a C program, but add assembly subroutines to perform string copy and capitalization operations.

Note that some embedded systems are coded purely in assembly language, but most are coded in C with assembly language used only for time-critical processing if at all. This is because the code *development* process is much faster (and hence less expensive) when writing in C when compared to assembly language.

Writing assembly code as functions, which can be called from C code as C functions, result in modular programs, which gives us the best of both worlds: the fast, modular development of C and the high performance of assembly code. It is also possible to add *inline assembly code* to C code, but this requires much greater knowledge of how compilers generate code.

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)
        ;
}
```

REGISTER USE CONVENTIONS

There are certain register use conventions which we need to follow if we would like our assembly code to coexist with C code.

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.

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.

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.

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`.

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
}
```

STRING CAPITALIZATION

Let's look at a subroutine to capitalize 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, capitalize 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.

LAB PROCEDURE

1. Open the asm project.
2. Compile the code.
3. Load it onto your mbed board.
4. Start the Debug Session (Ctrl + F5) and 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)? (see picture below).

(Please be aware, that register values and addresses shown in pictures might be different from what you get)

The screenshot displays the ARM debugger interface. On the left, the **Registers** window shows the values of several registers. On the right, the **Disassembly** window shows the assembly code. A yellow arrow points to the instruction at address 0x000002BC, which is `SUB sp, sp, #0x28`. A red box highlights this instruction and its address. A red arrow points from the text "Address 0x000002BC" to the instruction. Another red arrow points from the text "Register values" to the Registers window.

Register	Value
R0	0x1FFFF258
R1	0x20003000
R2	0x20003000
R3	0x00000000
R4	0x00003D2C
R5	0x00003D2C
R6	0x00000000
R7	0x00003D0B
R8	0x8FEFFBFA
R9	0x20000618
R10	0x00003D2C
R11	0x00003D2C
R12	0x00000000
R13 (SP)	0x20002FF8
R14 (LR)	0x00002C2B
R15 (PC)	0x000002BC

```

0x000002B4 1C6D ADDS r5,r5,#1
0x000002B6 1E76 SUBS r6,r6,#1
0x000002B8 D2F9 BCS 0x000002AE
0x000002BA BD70 POP {r4-r6,pc}
0x000002BC B08A SUB sp,sp,#0x28
0x000002BE A307 ADR r3,(pc)+2 ; @0x000002DC
0x000002C0 CB0F LDM r3,{r0-r3}
0x000002C2 9309 STR r3,[sp,#0x24]
0x000002C4 AB06 ADD r3,sp,#0x18
0x000002C6 C307 STM r3!,(r0-r2)
0x000002C8 A901 ADD r1,sp,#0x04
0x000002CA A806 ADD r0,sp,#0x18
0x000002CC F7FFFC8 BL.W _Z9my_strcpyPKcPc (0x00000260)
0x000002D0 A801 ADD r0,sp,#0x04
0x000002D2 F7FFFC8 BL.W _Z13my_capitalizePc (0x0000026E)
0x000002D6 BF00 NOP
0x000002D8 E7FE B 0x000002D8
0x000002DA 0000 DCW 0x0000
0x000002DC 6548 DCW 0x6548
0x000002DE 6C6C DCW 0x6C6C
0x000002E0 206F DCW 0x206F
  
```

```

main.cpp
16    CMP r2, #0 ; Was the byte 0?
17    BNE loop ; If not, repeat the loop
18    BX lr ; Else return from subroutine
19
20
21    asm void my_capitalize(char *str)
22    {
23    cap_loop
24        LDRB r1, [r0] ; Load byte into r1 from memory pointed to by r0 (str pointer)
25
26        CMP r1, #'a'-1 ; compare it with the character before 'a'
27        BLS cap_skip ; If byte is lower or same, then skip this byte
28
29        CMP r1, #'z' ; Compare it with the 'z' character
30        BHI cap_skip ; If it is higher, then skip this byte
31
32        SUBS r1,#32 ; Else subtract out difference to capitalize it
33        STRB r1, [r0] ; Store the capitalized byte back in memory
34    cap_skip
  
```

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? (see picture above)
6. Step one machine instruction 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? (see picture on the next page)

Stack pointer

Registers

Register	Value
R0	0x1FFFF258
R1	0x20003000
R2	0x20003000
R3	0x00000000
R4	0x00003D2C
R5	0x00003D2C
R6	0x00000000
R7	0x00003D08
R8	0xBFEFFBFA
R9	0x20000618
R10	0x00003D2C
R11	0x00003D2C
R12	0x00000000
R13 (SP)	0x20002FD0
R14 (LR)	0x00002028
R15 (PC)	0x000020BE
xPSR	0x61000000

Disassembly

```

0x000002B4 1C6D ADDS r5,r5,#1
0x000002B6 1E76 SUBS r6,r6,#1
0x000002B8 D2F9 BCS 0x000002AE {r4-r6,pc}
0x000002BA BD70 POP {r4-r6,pc}
maybe_terminate_alloc:
0x000002BC B08A SUB sp,sp,#0x28
0x000002BE A307 ADR r3,(pc)+2 ; @0x000002DC
0x000002C0 CB0F LDM r3,{r0-r3}
0x000002C2 9309 STR r3,[sp,#0x24]
0x000002C4 AB06 ADD r3,sp,#0x18
0x000002C6 C307 STM r3!,{r0-r2}
0x000002C8 A901 ADD r1,sp,#0x04
0x000002CA A806 ADD r0,sp,#0x18
0x000002CC F7FFFC8 BL.W Z9my_strcpyFKcPc (0x00000260)
0x000002D0 A801 ADD r0,sp,#0x04
0x000002D2 F7FFFC BL.W Z13my_capitalizePc (0x0000026E)
0x000002D6 BF00 NOP
0x000002D8 E7FE B 0x000002D8
0x000002DA 0000 DCW 0x0000
0x000002DC 6548 DCW 0x6548
0x000002DE 6C6C DCW 0x6C6C
0x000002E0 206F DCW 0x206F
  
```

main.cpp

```

34 cap_skip
35     ADDS r0, r0, #1 ; Increment str pointer
36     CMP r1, #0 ; Was the byte 0?
37     BNE cap_loop ; If not, repeat the loop
38     BX lr ; Else return from subroutine
39 }
40
41 int main(void)
42 {
43     const char a[] = "Hello world!";
44     char b[20];
45
46     my_strcpy(a, b);
47     my_capitalize(b);
48
49     while (1)
50     ;
51 }
52
  
```

Program counter

4 bytes long instructions

- 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? (see picture above)
- Continue execution (using F10) until reaching the *BL.W my_strcpy* instruction. What are the values of the SP, PC and LR? (see picture on the next page)

Registers

Register	Value
R0	0x20002FE8
R1	0x20002FD4
R2	0x2164C72
R3	0x20002FF4
R4	0x00003D2C
R5	0x00003D2C
R6	0x00000000
R7	0x00003D08
R8	0xBFEFFBFA
R9	0x20000618
R10	0x00003D2C
R11	0x00003D2C
R12	0x00000000
R13 (SP)	0x20002FD0
R14 (LR)	0x00002C28
R15 (PC)	0x00002CC
xPSR	0x10000000

Disassembly

```

0x000002BE A307 ADR r3,(pc)+2 ; @0x000002DC
0x000002C0 CB0F LDM r3,{r0-r3}
0x000002C2 9309 STR r3,[sp,#0x24]
0x000002C4 AB06 ADD r3,sp,#0x18
0x000002C6 C307 STM r3!,{r0-r2}
0x000002C8 A901 ADD r1,sp,#0x04
0x000002CA A806 ADD r0,sp,#0x18
0x000002CC F7FFFC8 BL.W _Z9my_strcpyFKcPc (0x00000260)
0x000002D0 A801 ADD r0,sp,#0x04
0x000002D2 F7FFFC8 BL.W _Z13my_capitalizePc (0x0000026E)
0x000002D6 BF00 NOP
0x000002D8 E7FE B 0x000002D8
0x000002DA 0000 DCW 0x0000
0x000002DC 6548 DCW 0x6548
0x000002DE 6C6C DCW 0x6C6C
0x000002E0 206F DCW 0x206F
0x000002E2 6F77 DCW 0x6F77
0x000002E4 6C72 DCW 0x6C72
0x000002E6 2164 DCW 0x2164
0x000002E8 0000 DCW 0x0000
0x000002EA 0000 DCW 0x0000
Reset Handler:

```

main.cpp

```

31
32 SUBS r1,#32 ; Else subtract out difference to capitalize it
33 STRB r1, [r0] ; Store the capitalized byte back in memory
34 cap_skip
35 ADDS r0, r0, #1 ; Increment str pointer
36 CMP r1, #0 ; Was the byte 0?
37 BNE cap_loop ; If not, repeat the loop
38 BX lr ; Else return from subroutine
39
40
41 int main(void)
42 {
43     const char a[] = "Hello world!";
44     char b[20];
45
46     my_strcpy(a, b);
47     my_capitalize(b);
48
49     while (1)

```

Stack Pointer, Link Register and Program Counter

9. Execute the *BL.W* instruction. What are the values of the SP, PS and LR? What has changed and why? Does the PC value agree with what is shown in the Disassembly window? (see picture on the next page).
10. What registers hold the arguments to *my_strcpy*, and what are their contents? (see picture on the next page)

The screenshot shows the ARM Disassembler interface. On the left, the 'Registers' window displays the state of various registers. On the right, the 'Disassembly' window shows the assembly code for the current function.

Registers Window:

Register	Value
R0	0x20002FE8
R1	0x20002FD4
R2	0x21646C72
R3	0x20002FF4
R4	0x00003D2C
R5	0x00003D2C
R6	0x00000000
R7	0x00003D08
R8	0xBFEBFBFA
R9	0x20000618
R10	0x00003D2C
R11	0x00003D2C
R12	0x00000000
R13 (SP)	0x20002FD0
R14 (LR)	0x00002D01
R15 (PC)	0x00000260
xPSR	0x61000000

Disassembly Window:

```

0x0000024E F002FE0D BL.W   _user_setup_stackheap (0x00002E6C)
0x00000252 4611     MOV     r1,r2
0x00000254 F7FFFFE3 BL.W   _rt_lib_init (0x0000021E)
0x00000258 F002FCE0 BL.W   main (0x00002C1C)
0x0000025C F002FF44 BL.W   exit (0x000030E8)
0x00000260 7802     LDRB    r2,[r0,#0x00]
0x00000262 3001     ADDS   r0,r0,#0x01
0x00000264 700A     STRB   r2,[r1,#0x00]
0x00000266 3101     ADDS   r1,r1,#0x01
0x00000268 2A00     CMP    r2,#0x00
0x0000026A D1F9     BNE    _Z9my_strcpyPKcPc (0x00000260)
0x0000026C 4770     BX     lr
0x0000026E 7801     LDRB    r1,[r0,#0x00]
0x00000270 2960     CMP    r1,#0x60
0x00000272 D903     BLS    0x0000027C
0x00000274 297A     CMP    r1,#0x7A
0x00000276 D801     BHI    0x0000027C
  
```

main.cpp:

```

9  loop
10  LDRB r2, [r0] ; Load byte into r2 from mem. pointed to by r0 (src pointer)
11
12  ADDS r0, #1 ; Increment src pointer
13  STRB r2, [r1] ; Store byte in r2 into memory pointed to by (dst pointer)
14
15  ADDS r1, #1 ; Increment dst pointer
16  CMP r2, #0 ; Was the byte 0?
17  BNE loop ; If not, repeat the loop
18  BX lr ; Else return from subroutine
19
20
21 _asm void my_capitalize(char *str)
22 {
23   cap_loop
24   LDRB r1, [r0] ; Load byte into r1 from memory pointed to by r0 (str pointer)
25
26   CMP r1, #'a'-1 ; compare it with the character before 'a'
27   BLS cap_skip ; If byte is lower or same, then skip this byte
  
```

Annotations:

- Red arrow from "R0: value of src pointer" points to R0 in the Registers window.
- Red arrow from "R1: value of dst pointer" points to R1 in the Registers window.
- Red arrow from "Address of the subroutine" points to the instruction `LDRB r2,[r0,#0x00]` in the Disassembly window.
- Red arrow from "Stack Pointer, Link Register and Program Counter" points to R13 (SP), R14 (LR), and R15 (PC) in the Registers window.

11. Watch the “Call Stack + Locals” window, to analyze the variable “a” and “b” (see picture below)

Call Stack + Locals		
Name	Location/Value	Type
_Z9my_strcpyPKcPc		
main		
a	0x20002FE8 "Hello world!"	auto - char[13]
b	0x20002FD4 ",="	auto - char[20]

12. What is the value of “a”?

13. What is the value of “b”?

14. Single step through the assembly code watching “Call Stack + Locals” window to see the string being copied character by character from a to b. What register holds the character?

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*)?

The screenshot displays an ARM disassembler interface. On the left, the 'Registers' window shows the state of the Core registers. R15 (PC) is highlighted with a red box and has a value of 0x0000026C. The 'Disassembly' window shows the instruction 'BX lr' at address 0x0000026C, which is highlighted in yellow. The 'Source' window shows the corresponding C code in 'main.cpp', where the 'loop' function is defined. The instruction 'BX lr' is at line 18 of the assembly, corresponding to the end of the 'loop' function.

Register	Value
R0	0x20002FF5
R1	0x20002FE1
R2	0x00000000
R3	0x20002FF4
R4	0x00003D2C
R5	0x00003D2C
R6	0x00000000
R7	0x00003D0B
R8	0xBFEBF8FA
R9	0x20000618
R10	0x00003D2C
R11	0x00003D2C
R12	0x00000000
R13 (SP)	0x20002FD0
R14 (LR)	0x000002D1
R15 (PC)	0x0000026C

```

0x00000260 7802  _asabi_memcpy4:
0x00000262 3001  LDRB  r2,[r0,#0x00]
0x00000264 700A  ADDS  r0,r0,#0x01
0x00000266 3101  STRB  r2,[r1,#0x00]
0x00000268 2A00  ADDS  r1,r1,#0x01
0x0000026A D1F9  CMP   r2,#0x00
0x0000026C 4770  BNE   _Z13my_capitalizePc (0x00000260)
0x0000026C 4770  BX    lr
0x0000026E 7801  _Z13my_capitalizePc:
0x00000270 2960  LDRB  r1,[r0,#0x00]
0x00000272 D903  CMP   r1,#0x60
0x00000274 297A  BLS   0x0000027C
0x00000276 D801  CMP   r1,#0x7A
0x00000278 3920  BHI   0x0000027C
0x0000027A 7001  SUBS  r1,r1,#0x20
0x0000027C 1C40  STRB  r1,[r0,#0x00]
0x0000027E 2900  ADDS  r0,r0,#1
0x00000280 D1F5  CMP   r1,#0x00
0x00000282 4770  BNE   _Z13my_capitalizePc (0x0000026E)
0x00000284 B570  BX    lr
0x00000284 B570  _asabi_memcpy4:
0x00000284 B570  PUSH  {r4-r6,lr}

main.cpp
9  loop
10  LDRB r2, [r0] ; Load byte into r2 from mem. pointed to by r0 (src pointer)
11
12  ADDS r0, #1 ; Increment src pointer
13  STRB r2, [r1] ; Store byte in r2 into memory pointed to by (dst pointer)
14
15  ADDS r1, #1 ; Increment dst pointer
16  CMP r2, #0 ; Was the byte 0?
17  BNE loop ; If not, repeat the loop
18  BX lr ; Else return from subroutine
19
20
21 _asm void my_capitalize(char *str)
22 {
23  cap_loop
24  LDRB r1, [r0] ; Load byte into r1 from memory pointed to by r0 (str pointer)
25
26  CMP r1, #'a'-1 ; compare it with the character before 'a'
27  BLS cap_skip ; If byte is lower or same, then skip this byte
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

16. Execute the *BX lr* instruction. Now what is the value of PC? (see picture on the next page).

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

The screenshot displays the ARM Disassembler and C source code. The Disassembler window shows assembly instructions, with the instruction `ADD r0, sp, #0x04` highlighted in yellow. The Registers window shows the PC register at address `0x00002D0`. A red arrow points from the PC register to the `PC` label. The C source code window shows the `main` function and the `my_capitalize` subroutine.

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

The screenshot displays the C source code and the Call Stack. The C source code window shows the `cap_loop` and `cap_skip` subroutines. The Call Stack window shows the current state of the program, with the `a` and `b` variables highlighted. The `a` variable contains `0x20002FE8 "Hello world!"` and the `b` variable contains `0x20002FD4 "HELLO World!"`.

Characters are capitalized one by one from `a` to `b`