***Introduction to Robotic Systems Course***

**LAB 01**

**Programming with Mixed C and Assembly:**

**Processing Text in Assembly Language**

Contents

[1 Introduction 1](#_Toc31036059)

[1.1 Lab Overview 1](#_Toc31036060)

[2 Requirements 1](#_Toc31036061)

[3 Mixing Assembly Language and C Code 1](#_Toc31036062)

[4 Register Use Conventions 1](#_Toc31036063)

[4.1 Calling Functions and Passing Arguments 2](#_Toc31036064)

[4.2 Temporary Storage 2](#_Toc31036065)

[4.3 Preserved Registers 2](#_Toc31036066)

[4.4 Returning from Functions 2](#_Toc31036067)

[5 Task: Copy and Capitalize String 2](#_Toc31036068)

[5.1 Main Function 2](#_Toc31036069)

[5.2 String Copy 3](#_Toc31036070)

[5.3 String Capitalization 3](#_Toc31036071)

[6 Lab Procedure 4](#_Toc31036072)

# Introduction

## Lab Overview

This lab exercise presents an example code to introduce the concepts of mixed C and assembly code for modular code development. The example code presented performs two main actions. First, using pointers to address memory locations, it copies a string from a source to a destination memory location. Second, it capitalizes the copied string in the destination memory location. It achieves this by calling two subroutines written in assembly (string copy and string capitalization) in the main function.

# Requirements

The following hardware and software are required to complete this lab:

* **Hardware:** OpenCR1.0 Microcontroller Board and ULINK-ME debugger
* **Software:** Keil MDK. Version v5.28 was used when this course was developed.

# Mixing Assembly Language and C Code

Embedded systems code can be written purely in assembly language; however, most are written in C, and a programmer may resort to assembly language only for time-critical processing.

Code development process is much faster (and hence much less expensive) when writing in C compared to when writing in assembly language.

Writing an assembly language function that can be called in a C function results in a modular program that gives the best of both worlds: fast modular development in C and 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.

# Register Use Conventions

There are register use conventions that, when followed, allow assembly code to coexist with C code in the same program. More on this in the lecture module “C as implemented in Assembly Language.” The following points are worth noting.

## 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 not used to hold input 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

Since the link register holds the return address of a function, the instruction BX lr will reload the pc with the return address value on the lr. If the function returns a value, it will be passed through r0.

# Task: Copy and Capitalize String

In this task, you will write a program to perform string copy and capitalization operations. The main function in this program will be written in C language. You will also write assembly language subroutines, that will be called in the main function.

## Main Function

First, create the main C function. This function is the starting point of the program execution. The main function will contain two variables (“a” and “b”) with character arrays. Variable “a” is a constant that contains a string. The value of variable “a” will be copied into variable “b” from where the content can be modified.

The code snippet below shows the main function. This function calls the two functions **my\_strcpy** and **my\_capitalize** to perform the string copy and capitalization operations in this lab.

1. int main**(**void**)**
2. **{**
3. const char a**[]** **=** "Hello world!"**;**
4. char b**[**20**];**
5. my\_strcpy**(**a**,** b**);**
6. my\_capitalize**(**b**);**
7. **while** **(**1**);**
8. **}**

Figure 1: Main function.

## String Copy

The string copy function **my\_strcpy** shown in the code snippet below takes two arguments (src and dst). Each argument is a 32-bit long pointer to a character. Here, a pointer fits into a register, so argument src is passed through register r0 and dst is passed through r1.

This function will load a character from memory, save it into the destination pointer, and increment both pointers until the end of the string.

\_\_asm void my\_strcpy**(**const char **\***src**,** char **\***dst**)**

{

**loop**

LDRB r2**,** **[**r0**]** **//** Load byte into r2 from memory 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

}

Figure 2: Assemble Code – Copy String from source to destination.

## String Capitalization

The string capitalization function shown in the code snippet below takes one argument, a pointer to a memory location that contains the string to be capitalized. This function will capitalize all the lower-case letters in the string.

It does this by loading and checking each character in the string to ensure it is a letter before it is capitalized. Each character in the string is represented by its ASCII code. For example, the ASCII code for “A” is 65 (0x41), “B” is 66 (0x42), and so on up to “Z” that uses 90 (0x5a). The lower-case letters start at “a” (97, or 0x61) and end with “z” (122, or 0x7a). A lower-case letter can be converted to an uppercase 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

}

Figure 3: Assemble Code – Capitalize string.

In the code shown above, if the test byte loaded into r1 is less than “a” or higher than “z,” 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. Compile the code and load it onto your board.
2. Run the program until the opening brace in the main function is highlighted.
3. Open the Registers window (**View**->**Registers Window**) and note the values of the stack pointer (r13), link register (r14), and the program counter (r15).
4. Open the Disassembly window (**View**->**Disassembly Window**). Note the instruction and its address pointed to by the yellow arrow. How does this address relate to the value of pc?
5. Step through one machine instruction **(F10)** 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?
6. Continue stepping through one machine instruction **(F10)** until you reach the **BL.W my\_strcpy** instruction. What are the values of the sp, pc, and lr?
7. Step one line into the **my\_strcpy** function **(F11)**. What has changed and why? Does the pc value agree with what is shown in the Disassembly window?
8. What registers hold the arguments to **my\_strcpy**? Note the contents of these registers.

Open a Memory window (**View**->**Memory Windows**->**Memory 1**) and search with the noted address for **src**.

Open a Memory window (**View**->**Memory Windows**->**Memory 2**) and search with the noted address for **dst**.

Right-click on each memory window and select ASCII to display the contents as ASCII text.

1. What are the memory contents addressed by **src**?
2. What are the memory contents addressed by **dst**?
3. 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?
4. 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)?
5. Execute the BX lr instruction. Now what is the value of PC?
6. What is the relationship between the PC value and the previous LR value?
7. Now step through the **my\_capitalize** subroutine and verify it works correctly, that is all lowercase letters are replaced by their uppercase equivalent in **dst**.