ECEN 5803 Mastering Embedded System Architecture

Project 1 Module 1

## Question 1 : Starting with the results from Lab\_Exercise\_1 in the Homework 2-Practical, create another Keil project, call it M1-String and replace the string copy and string capitalization assembly language functions with C functions written by your or your partner. Compile and run this project. Compare the memory usage between the assembly language function project and the C function project – which uses less memory?

* **Memory Usage for Assembly Code :**

|  |  |
| --- | --- |
| Section | Size (Bytes) |
| Code | 1392 |
| RO-data | 436 |
| RW-data | 40 |
| ZI-data | 1632 |

Table : Memory usage of a assembly code

A screenshot of a computer

Description automatically generated

Figure : Compiled Assembly code showing memory usage

* **Memory Usage for C Code :**

|  |  |
| --- | --- |
| Section | Size (Bytes) |
| Code | 1408 |
| RO-data | 436 |
| RW-data | 40 |
| ZI-data | 1632 |

Table 2 : Memory usage of a C code

A screenshot of a computer

Description automatically generated

Figure : Compiled C code showing memory usage

* **Size Analysis between Assembly and C Code :**

Looking into the two screenshots with just 16 extra bytes added in the code section, the assembly implementation code size is little less than the C implementation's. The three types of RO, RW, and ZI data memory sections remain unchanged between the two implementations.

This shows that while the assembly fundamental logic optimization concludes in a small space reduction, the overall data structures and constants are unchanged.

## Question 2 : Look at the .map file for this project. Explain the memory model of ARM Cortex-M4 with respect to the code memory, data memory, IRQ handlers and peripherals. Explain with the help of a diagram where required.

* **Basic Idea of ARM Cortex-M4 memory model**   
  The ARM Cortex M4 is a processor core developed by ARM Holdings organisation for microcontroller use, notably in embedded systems. It features a Memory Model that is suited very well to meet the needs of high-performance, low power devices. Below is a high level overview of the memory model with respect to the mentioned aspects.

**1. Code Memory (Flash):**

* + Code memory is a non-volatile and stores the program executable code. It retrieves and retain data even when the power is off.

**2. Data Memory (RAM):**

* + Data memory is a volatile and stores data as well as variables that the program uses during execution run. It forgets its content when the power is turned off.

**3. Interrupt Vector Table:**

* + Interrupt vector table contains the addresses of Interrupt Service Routines (ISR) or IRQ Handlers. Each of these IRQ’s corresponding to a specific interrupt or exception. It allows the processor to handle requests from peripherals efficiently.

**4. Peripherals:**

* + Peripherals are the components like timers, communication ports like USART, and ADCs, enabling interaction with the outside world or performing certain tasks. They are used and controlled through memory mapped Special Function Registers(SFR).

A diagram of a computer program

Description automatically generated

Figure : Memory Map Model From ARM Official Website

* **Our .map Analysis:**

**Code Memory :**

The code memory in embedded systems is where the executable code is stored. For the ARM Cortex-M4, which, this means that code and data coexist in the same memory space. This specific memory, often known as flash memory, is non-volatile, meaning it retains its data even without power. According to our .map file as seen in figure 4, this memory region is designated as the Load Region LR\_IROM1 with a base address of 0x08000000 and a size of 0x0000075c. Within this, the Execution Region ER\_IROM1, where the code actually runs, has the same base address, and a slightly smaller size of 0x00000734. The maximum allowable size for this region is 0x00080000.

A screenshot of a computer

Description automatically generated

Figure : Code memory from .map file

**Data Memory:**

The data memory in embedded systems is used to store variables and run-time data. Within the context of the Cortex-M4, this memory region is predominantly comprised of Random Access Memory (RAM). According to the .map file, the data memory region is denoted by the Execution Region RW\_IRAM1. The base address for this region is 0x20000000. The size of RW\_IRAM1, as specified in the .map file screenshot in figure 5 , is 0x00000688. This memory also accommodates individual sections like .data for initialized variables and .bss for uninitialized ones. Additionally, dedicated sections for HEAP and STACK are present, which are integral for dynamic memory allocation and function call management respectively.

A screenshot of a computer

Description automatically generated

Figure : data memory section from .map file

**IRQ Handlers:**

Interrupt Request (IRQ) handlers are responsible for dealing with events or signals from peripherals or internal system components that need immediate attention. In the context of the Cortex-M architecture, these handlers are typically part of the vector table at the beginning of the code memory. In the .map file for the STM32F401xE microcontroller, the IRQ handlers are functions designed to address specific system or peripheral events. Looking at .map file in screenshot below in figure 6, we can see that IRQ for each handers has been assigned a specific address. Many share the address 0x0800024f, suggesting they're aliased to a default function in the absence of specific implementations.

A screenshot of a computer

Description automatically generated

Figure : IRQ's from .map file

**Peripherals:**

Peripherals represent the various hardware components integrated into the microcontroller, such as GPIOs, UARTs, ADCs, and timers. The memory associated with peripherals is usually mapped to a specific region in memory and is often called the peripheral memory space. In our .map file of microcontroller projects, peripheral-related data doesn’t have explicit sections named after the peripherals. Instead, they can be found in the symbol table with names resembling peripheral functions, in memory sections detailing memory-mapped I/O, or via cross-references linking object files to symbols. The absence of peripherals could be because they are not initialised or used in main.c code.

References :

* + Arm Developer. (n.d.). Architectural Overview. Retrieved September 25, 2023, from https://developer.arm.com/documentation/den0001/latest/
  + MikroE. (n.d.). Memory Organization. Retrieved September 25, 2023, from http://download.mikroe.com/documents/compilers/mikroc/arm/help/memory\_organization.htm

## Question 3 :

## Testing Approximate square root with bisection method without Q16.16 format code with these inputs: 2, 4, 22, and 121.

## Estimate the number of CPU cycles used for this calculation.

## Auto-generate documentation using Doxygen. Provide either an HTML directory or PDF file documenting your codebase.

* Testing input: 2

Result as seen in R0 register = 0x00000001

A screenshot of a computer

Description automatically generated

Figure : Result of normal code output with input 2

* Testing input: 4

Result as seen in R0 register = 0x00000002

A screenshot of a computer

Description automatically generated

Figure 8 : Result of normal code output with input 4

* Testing input: 22

Result as seen in R0 register = 0x00000004

A screenshot of a computer

Description automatically generated

Figure 9 : Result of normal code output with input 22

* Testing input: 121

Result as seen in R0 register = 0x0000000B

A screenshot of a computer

Description automatically generated

Figure 10 : Result of normal code output with input 121

* CPU Cycle

After going through the code computing CPU cycles for each instruction the final CPU cycle time can be computed as follows. Assuming that branching is not taken and I being number of iterations.

Total Cycles = 11 (Setup) + [8 + (2 (for either Update A or B)) + 5]\*i (i Iterations) + 8 (Clean-up)

References:

* + Arm Developer. (n.d.). Cortex-M4 Instructions: Instruction Set Summary. Retrieved September 25, 2023, from https://developer.arm.com/documentation/ddi0439/b/Programmers-Model/Instruction-set-summary/Cortex-M4-instructions

## Doxygen Documentation :

HTML and Latex Generated.

Path in Submission (Folder): Doxygen Documentation.

## Question 4 : (Bonus Question)

## Testing Approximate square root with bisection method with Q16.16 format code with these inputs: 2.0, 4.0, 22.0, and 121.0.

## Estimate the number of CPU cycles used for this calculation and also the size of the code in memory.

## Auto-generate documentation using Doxygen. Provide either an HTML directory or PDF file documenting your codebase.

* Testing input: 2.0

Result as seen in R0 register = 0x00016A00(1.41 in Q16.16 format)

A screenshot of a computer

Description automatically generated

Figure : Bonus Lab output with input 2.0

* Testing input: 4.0

Result as seen in R0 register = 0x00020000 (2.0 in Q16.16 format)

A screenshot of a computer

Description automatically generated

Figure : Bonus Lab output with input 4.0

* Testing input: 22.0

Result as seen in R0 register = 0x0004B000(4.68 in Q16.16 format)

A screenshot of a computer

Description automatically generated

Figure 10 : Bonus Lab output with input 22.0

* Testing input: 121.0

Result as seen in R0 register = 0x000B0000(11.0 in Q16.16 format

A screenshot of a computer

Description automatically generated

Figure 11 : Bonus Lab output with input 121.0

* CPU Cycle

After going through the code computing CPU cycles for each instruction the final CPU cycle time can be computed as follows. Assuming that branching is not taken and I being number of iterations.

Total Cycles = 12 (Setup) + [9 + 1 (cond\_less\_equal if branched) + 3]\*i (i Iterations) + 6 (Clean-up)

* **Memory Usage for Bonus Lab Code :**

|  |  |
| --- | --- |
| Section | Size (Bytes) |
| Code | 1948 |
| RO-data | 436 |
| RW-data | 36 |
| ZI-data | 1636 |

## Doxygen Documentation :

Refer the attached pdf auto generated from doxygen using “make pdf ” command in latex folder.

HTML and Latex also generated.

Path in Submission (Folder): Doxygen Documentation

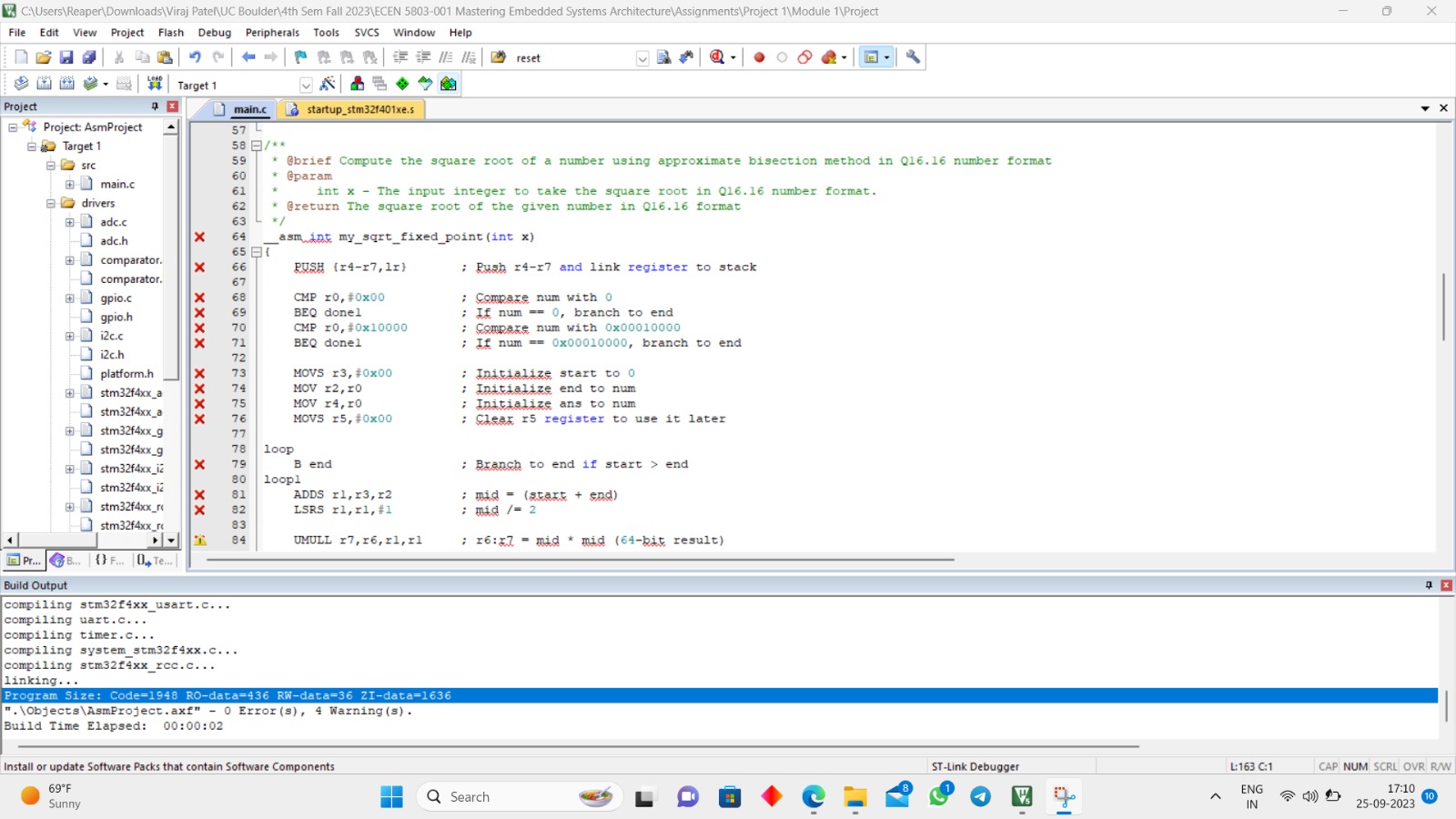


Figure 12 : Bonus Lab Size of Memory

References:

* + Arm Developer. (n.d.). Cortex-M4 Instructions: Instruction Set Summary. Retrieved September 25, 2023, from https://developer.arm.com/documentation/ddi0439/b/Programmers-Model/Instruction-set-summary/Cortex-M4-instructions

## Code :

### Square Root approximation :

/\*\*

\* @file main.c

\* @brief Square Root Approximation using Integer Approach - Project 1 Module 1

\*

\* @author Viraj Patel, Kiran jojare

\* @see <a href="https://developer.arm.com/documentation/ddi0439/b/Programmers-Model/Instruction-set-summary/Cortex-M4-instructions?lang=en">ARM Cortex-M4 Instructions</a>

\* @see PDF obtained from professor "Approximate Square Root Bisection Method"

\*/

// include standard header files

#include <string.h>

#include <assert.h>

/\*\*

\* @brief Compute the integer square root of a number using approximate bisection method.

\* @param

\* int x - The input integer to take the square root.

\* @return The integer square root of the given number.

\*/

\_\_asm int my\_sqrt\_int(int x)

{

PUSH {R4-R8, LR} ; Save registers that will be used in the subroutine

MOV R4, #0 ; Initialize a = 0

LDR R5, =0x4000 ; Initialize b = 2^16

MOV R6, #-1 ; Initialize c = -1

MOV R8, #0 ; Initialize done = 0

loop\_start

MOV R7, R6 ; c\_old <- c

ADD R6, R4, R5 ; R6 = a + b

ASR R6, R6, #1 ; c <- (a+b)/2

MUL R3, R6, R6 ; R3 = c\*c

CMP R3, R0 ; Compare c\*c with x

BEQ done ; If c\*c == x, branch to done

BLT update\_a ; If c\*c < x, branch to update\_a

B update\_b ; Otherwise, branch to update\_b

update\_a

MOV R4, R6 ; a <- c

B loop\_check ; jump to loop\_check

update\_b

MOV R5, R6 ; b <- c

B loop\_check ; jump to loop\_check

loop\_check

CMP R6, R7 ; Compare c with c\_old

BEQ done ; If c == c\_old, branch to done

CMP R8, #1 ; Compare done with 1

BEQ done ; If done == 1, branch to done

B loop\_start ; Otherwise, loop back to start

done

MOV R0, R6 ; Move result into R0 (return value)

POP {R4-R8, LR} ; Restore saved registers

BX LR ; Return

}

/\*\*

\* @brief Compute the square root of a number using approximate bisection method in Q16.16 number format

\* @param

\* int x - The input integer to take the square root in Q16.16 number format.

\* @return The square root of the given number in Q16.16 format

\*/

\_\_asm int my\_sqrt\_fixed\_point(int x)

{

PUSH {r4-r7,lr} ; Push r4-r7 and link register to stack

CMP r0,#0x00 ; Compare num with 0

BEQ done1 ; If num == 0, branch to end

CMP r0,#0x10000 ; Compare num with 0x00010000

BEQ done1 ; If num == 0x00010000, branch to end

MOVS r3,#0x00 ; Initialize start to 0

MOV r2,r0 ; Initialize end to num

MOV r4,r0 ; Initialize ans to num

MOVS r5,#0x00 ; Clear r5 register to use it later

loop

B end ; Branch to end if start > end

loop1

ADDS r1,r3,r2 ; mid = (start + end)

LSRS r1,r1,#1 ; mid /= 2

UMULL r7,r6,r1,r1 ; r6:r7 = mid \* mid

SUBS r7,r0,r7 ; Subtract result from num

SBCS r6,r5,r6 ; Subtract with carry the result from 0

BCC cond\_less\_equal ; If unsigned lower or same, branch to cond\_less\_equal

ADDS r3,r1,#1 ; start = mid + 1

MOV r4,r1 ; ans = mid

B loop

cond\_less\_equal

SUBS r2,r1,#1 ; end = mid - 1

end

CMP r3,r2 ; Compare start with end

BLS loop1 ; If start <= end, branch to loop1

LSLS r0,r4,#8 ; Return ans << 8

done1

POP {r4-r7,pc} ; Pop r4-r7 from stack and return

BX LR ; Return

}

/\*\*

\* @brief Application Entry Point.

\* @param void

\* @return nothing

\*/

int main(void)

{

volatile int r, j = 0;

// Assert that my\_sqrt\_int returns the correct value for each test input

r = my\_sqrt\_int(0);

assert(r == 0); // assert that the square root of 0 is 0

r = my\_sqrt\_int(25);

assert(r == 5); // assert that the square root of 25 is 5

r = my\_sqrt\_int(133);

assert(r == 11); // assert that the square root of 133 is 11

// Testing requested from lab exercise question

// 3. Test your code with these inputs: 2, 4, 22, and 121. Record the results

r = my\_sqrt\_int(2);

assert(r == 1); // assert that the square root of 2 is 1

r = my\_sqrt\_int(4);

assert(r == 2); // assert that the square root of 4 is 2

r = my\_sqrt\_int(22);

assert(r == 4); // assert that the square root of 22 is 4

r = my\_sqrt\_int(121);

assert(r == 11); // assert that the square root of 121 is 11

// Iterate from 0 to 9999 and compute square root of 'i' using 'my\_sqrt\_int', then accumulate the results in 'j'.

for (int i = 0; i < 10000; i++) {

r = my\_sqrt\_int(i);

j += r;

}

// Assert that the accumulated result 'j' is 661650

assert(j == 661650);

// Testing requested from bonus lab exercise question

// 1. Test your code with these inputs: 2.0, 4.0, 22.0, and 121.0. Record the results

r = my\_sqrt\_fixed\_point(0x00020000); // 2.0 in Q16.16 format

assert(r == 0x00016a00); // assert that square root of 2.0 is 1.41 in Q16.16 format

r = my\_sqrt\_fixed\_point(0x00040000); // 4.0 in Q16.16 format

assert(r == 0x00020000); // assert that square root of 4.0 is 2.0 in Q16.16 format

r = my\_sqrt\_fixed\_point(0x00160000); // 22.0 in Q16.16 format

assert(r == 0x0004b000); // assert that square root of 22.0 is 4.68 in Q16.16 format

r = my\_sqrt\_fixed\_point(0x00790000); // 121.0 in Q16.16 format

assert(r == 0x000b0000); // assert that square root of 121.0 is 11.0 in Q16.16 format

// Code execution ends here

while(1);

}

## Code :

### String Capitalize and Copy (C Code):

/\*\*

\* @file: main.c

\* @brief: String Copy And Capitalize In C To Compare Memory Usage With Assembly Code - Project 1 Module 1

\*

\* @author: Viraj Patel, Kiran jojare

\* @references:

\* - Professors Assembly Code in Code1Nucleo.zip

\* from Homework2(Practical)

\*/

/\*\*

\* @brief: Copy A Null Terminated String From Source Pointer To Destination Pointer

\* @param:

\* const char \*src - Source String Pointer

\* const char \*dst - Destination String Pointer

\* @return: Nothing

\*/

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

{

// While Source Is Not Null

while (\*src != '\0') {

// Copy Source To Destination

\*dst = \*src;

// Increment Source And Destination Pointer

src++;

dst++;

}

// Terminate Destination With Null

\*dst = '\0';

}

/\*\*

\* @brief: Capitalize Every Letter In A Null Terminated String

\* @param:

\* char \*str - Pointer To String To Capitalize

\* @return: Nothing

\*/

void my\_capitalize(char \*str)

{

// While String Is Not Null

while (\*str != '\0') {

// Check If Character Between a-z

if ((\*str >= 'a') && (\*str <= 'z')) {

// Convert To Uppercase

\*str = \*str-32;

}

// Increment String Pointer

str++;

}

}

int main(void)

{

const char a[] = "Hello world!";

char b[20];

my\_strcpy(a, b);

my\_capitalize(b);

while (1);

}

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*ARM University Program Copyright © ARM Ltd 2016\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## Code :

### String Capitalize and Copy (Assembly Code):

/\*----------------------------------------------------------------------------

LAB EXERCISE 5.1 - PROCESSING TEXT IN ASSEMBLY LANGUAGE

----------------------------------------------

Examine program execution at the processor level using the debugger

\* @file: main.c

\* @brief: String Copy And Capitalize In Assembly - Project 1 Module 1

\*

\* @author: Viraj Patel, Kiran jojare

\* @references:

\* - Professors Assembly Code in Code1Nucleo.zip

\* from Homework2(Practical)

\*----------------------------------------------------------------------------\*/

/\*\*

\* @brief: Copy A Null Terminated String From Source Pointer To Destination Pointer

\* @param:

\* const char \*src - Source String Pointer

\* char \*dst - Destination String Pointer

\* @return: Nothing

\*/

\_\_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

}

/\*\*

\* @brief: Capitalize Lowercase Letters In A Null Terminated String

\* @param:

\* char \*str - String Pointer

\* @return: Nothing

\*/

\_\_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

}

/\*----------------------------------------------------------------------------

MAIN function

\*----------------------------------------------------------------------------\*/

int main(void){

const char a[] = "Hello world!";

char b[20];

my\_strcpy(a, b);

my\_capitalize(b);

while (1);

}

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*ARM University Program Copyright (c) ARM Ltd 2014\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*