ECEN 5803

Mastering Embedded System Architecture

(Fall-2023)

PROJECT-1 : MODULE 1

TO C OR NOT TO C

Guided By

Prof. Timothy Scherr

Submitted By

Ajay Kandagal

Kshitija Dhondage

Que 1] Starting with the results from Lab_Exercise_1 in the Homework 2-Practical, create another Keil project, call it M1String 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?

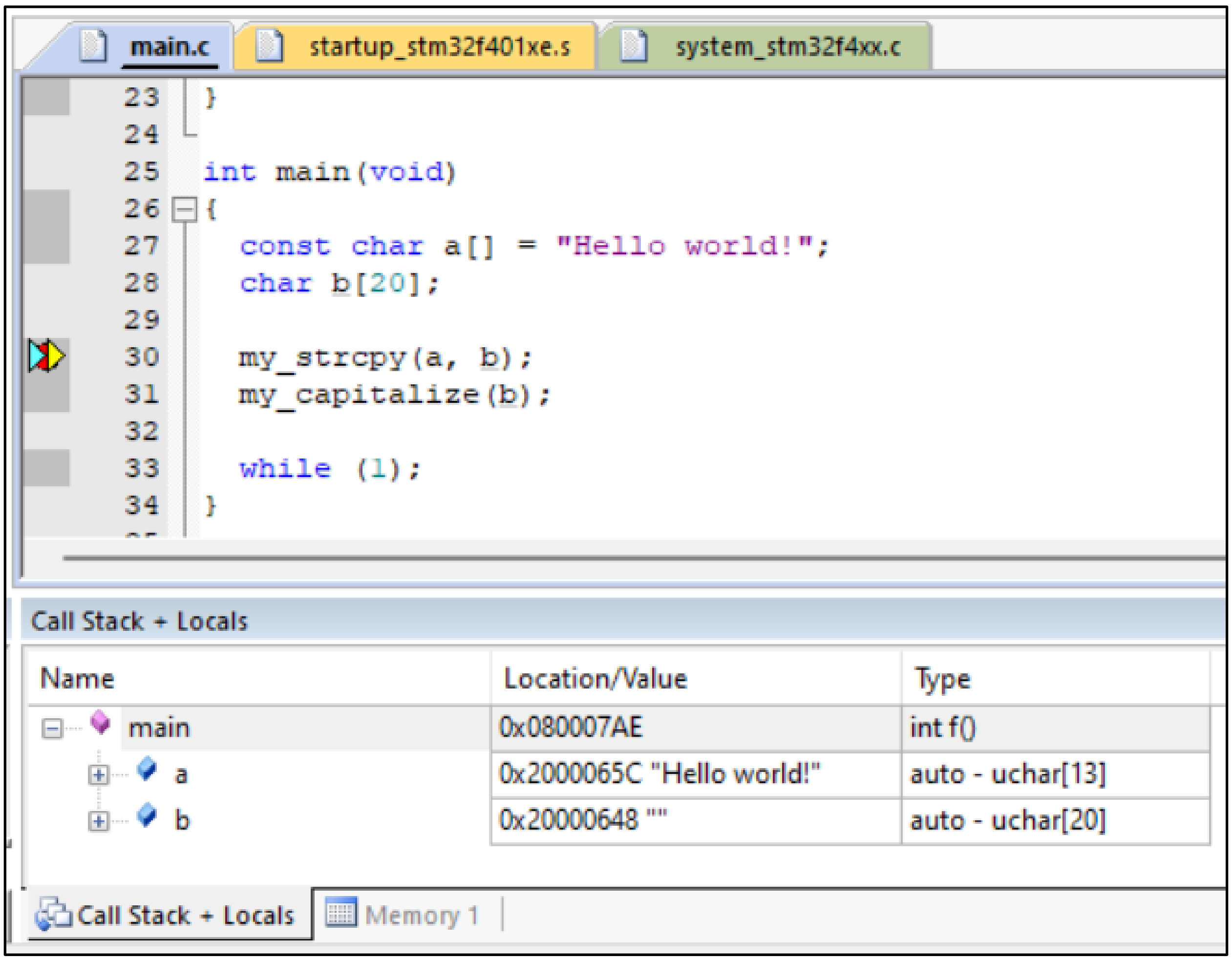
Ans:

C code write,compile and run:

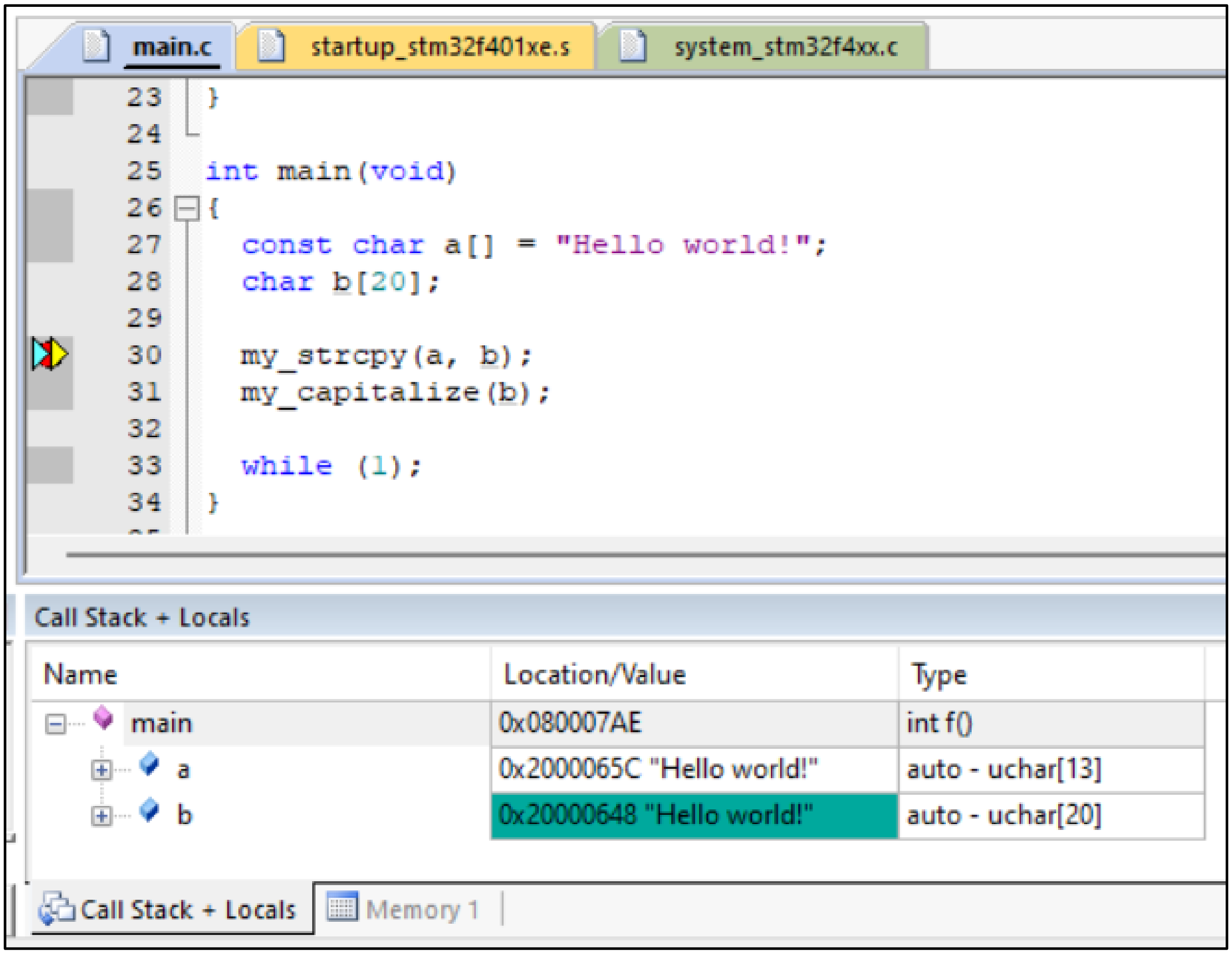
Note: please find the attached M1String project also the code is updated in the Appendix(end of this document)

Code output:

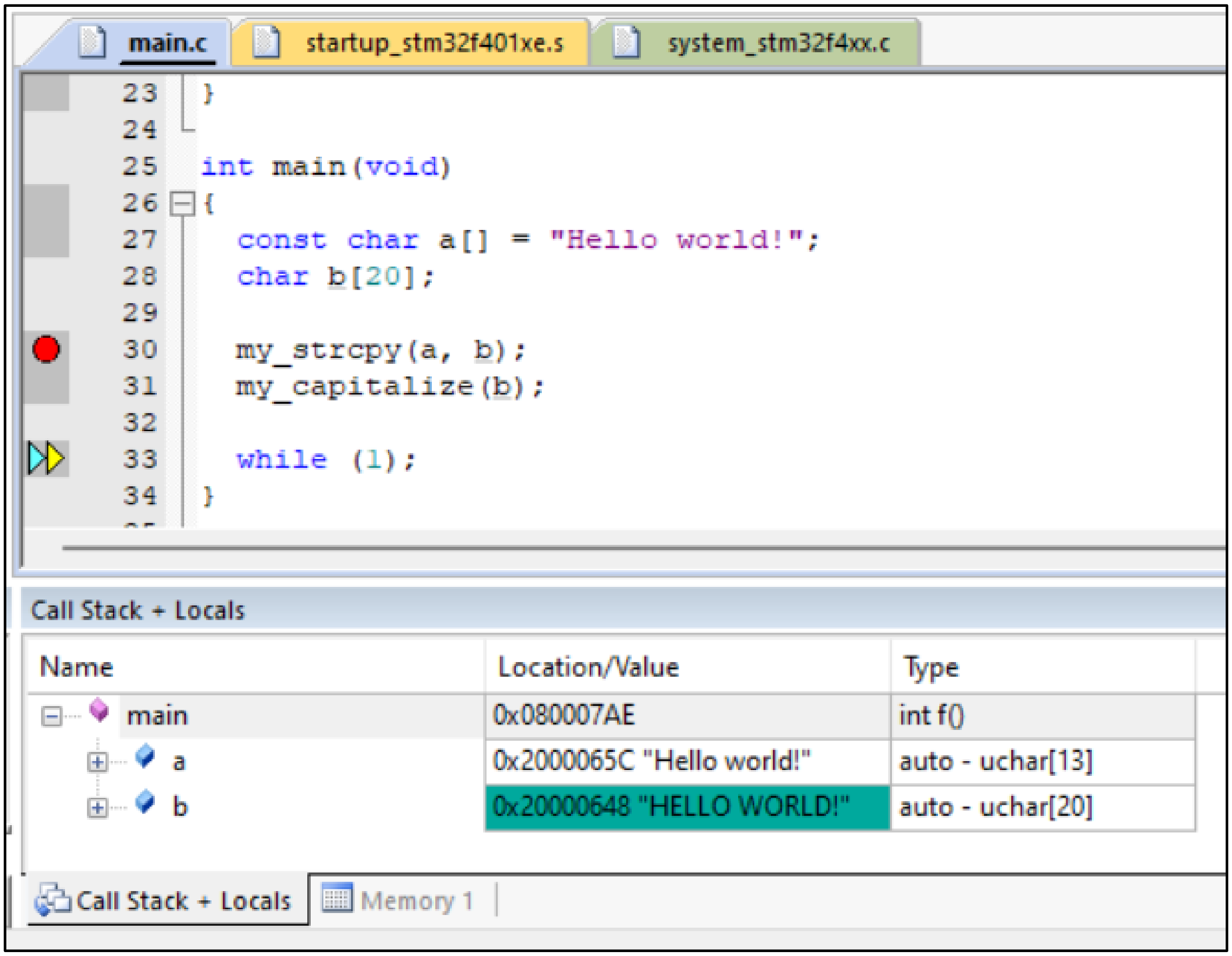
string copy:



string before copy



string after copy



Memory usage comparison between the assembly language function project and the C function project:

Assembly language Code Memory Usage:

```
Build Output
compiling timer.c...
compiling i2c.c...
compiling main.c...
compiling stm32f4xx_adc.c...
compiling adc.c...
compiling comparator.c...
compiling stm32f4xx_gpio.c...
compiling gpio.c...
compiling stm32f4xx_rcc.c...
compiling stm32f4xx_i2c.c...
compiling uart.c...
assembling startup_stm32f401xe.s...
compiling system_stm32f4xx.c...
linking...
Program Size: Code=1696 RO-data=420 RW-data=0 ZI-data=1656
".\Objects\M1String.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:01
```

1.1 Keil project asm build output

```
==============================================================================


    Code (inc. data)   RO Data    RW Data    ZI Data      Debug

    1696         42        420          0       1656      32411   Grand Totals
    1696         42        420          0       1656      32411   ELF Image Totals
    1696         42        420          0          0          0   ROM Totals

==============================================================================

   Total RO  Size (Code + RO Data)                 2116 (   2.07kB)
   Total RW  Size (RW Data + ZI Data)              1656 (   1.62kB)
   Total ROM Size (Code + RO Data + RW Data)       2116 (   2.07kB)

==============================================================================
```

1.2 asm code .map file

Assembly code uses 1696 bytes of code memory with optimization set to -O0 as shown below.

C language Code Memory Usage:

```
Build Output
Rebuild started: Project: M1String
*** Using Compiler 'V6.19', folder: 'C:\Keil_v5\ARM\ARMCLANG\Bin'
Rebuild target 'Target 1'
compiling stm32f4xx_i2c.c...
compiling stm32f4xx_adc.c...
compiling stm32f4xx_usart.c...
compiling i2c.c...
compiling main.c...
compiling stm32f4xx_gpio.c...
compiling comparator.c...
compiling stm32f4xx_rcc.c...
compiling timer.c...
compiling adc.c...
compiling gpio.c...
compiling uart.c...
compiling system_stm32f4xx.c...
assembling startup_stm32f401xe.s...
linking...
Program Size: Code=1776 RO-data=420 RW-data=0 ZI-data=1656
".\Objects\M1String.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:02
```

1.3 Keil project M1String build output

```
==============================================================================


      Code (inc. data)   RO Data    RW Data    ZI Data      Debug

      1776         42        420          0       1656      32699   Grand Totals
      1776         42        420          0       1656      32699   ELF Image Totals
      1776         42        420          0          0          0   ROM Totals

==============================================================================

    Total RO  Size (Code + RO Data)                 2196 (   2.14kB)
    Total RW  Size (RW Data + ZI Data)              1656 (   1.62kB)
    Total ROM Size (Code + RO Data + RW Data)       2196 (   2.14kB)

==============================================================================
```

1.4 M1String code .map file

C code uses 1776 bytes of code memory with optimization set to -O0 as shown below.

Conclusion:

So, from the above map file, we can see that the Assembly implementation uses 2.07 kB of memory on the other hand, the C implementation uses 2.14 kB of memory.

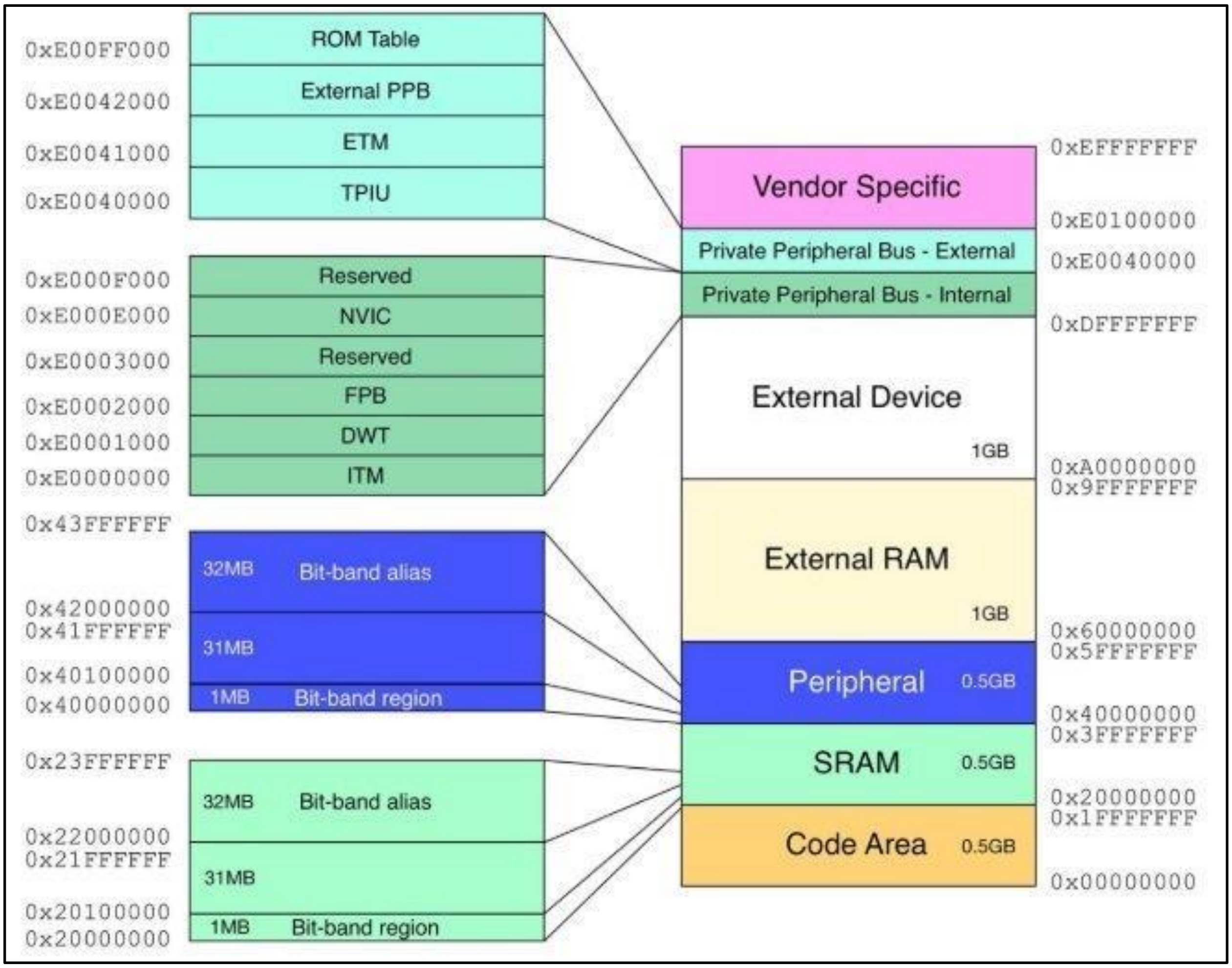
The C language code uses more memory than assembly language code for the same functionality.

Due to the absence of compiler-generated overhead, the assembly language project is more memory-efficient compared to the C

Que 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.

Ans:

Memory Model of ARM Cortex-M4:



ARM Cortex-M processors use 32-bit processor cores. The number of bits in the processor's design tells us how much memory it can work with.

In the case of ARM Cortex-M4 microcontrollers, they can manage up to 4 gigabytes (4GB) of memory.

The ARM Cortex-M4 architecture has a specific memory model that divides memory into different regions to serve different purposes.

Memory regions are as follows:

The ARM Cortex-M4 has a specific memory model that organizes memory into different regions for various purposes:

1. Code Memory

- The region of code memory is 0x0000_0000 to 0x1FFFF_FFFF

- This is where the program code is stored.

- Static initialized data can also be placed here.

- It spans a memory range of 0.5 GB.

2. Data Memory:

- The region of data memory is 0x2000_0000 to 0x3FFF_FFFF

- SRAM (Static Random-Access Memory) is located in this region.

- It is used for temporary data storage, including the heap, stack, and temporary function variables.

- Additionally, this area contains a bit-banded alias and a bit-band region.

- It spans a memory range of 0.5 GB.

3. Peripherals:

- The region allocated for peripherals is 0x4000_0000 to 0x5FFF_FFFF.

- Peripheral registers, such as GPIO (General-Purpose Input/Output) ports, UART (Universal Asynchronous Receiver/Transmitter), I2C (Inter-Integrated Circuit), Timers, and more, are located here.

- These registers control and communicate with various hardware peripherals.

- This region spans 0.5 GB of memory.

4. IRQ Handlers:

- The region allocated for IRQ handlers is Internal memory space from 0xE000E100 to 0xE000ECFF

- IRQ (Interrupt Request) handlers are stored in the NVIC (Nested Vector Interrupt Control) part of memory.

- NVIC maps to the Private Peripheral Bus.

- IRQ handlers manage interrupts, allowing the processor to respond to external events or requests efficiently.

5. External RAM:

- This part of memory, from 0x6000.0000 to 0x9FFF.FFFF.

- This is used for connecting and using things like SD cards or external flash drives.

Reference:

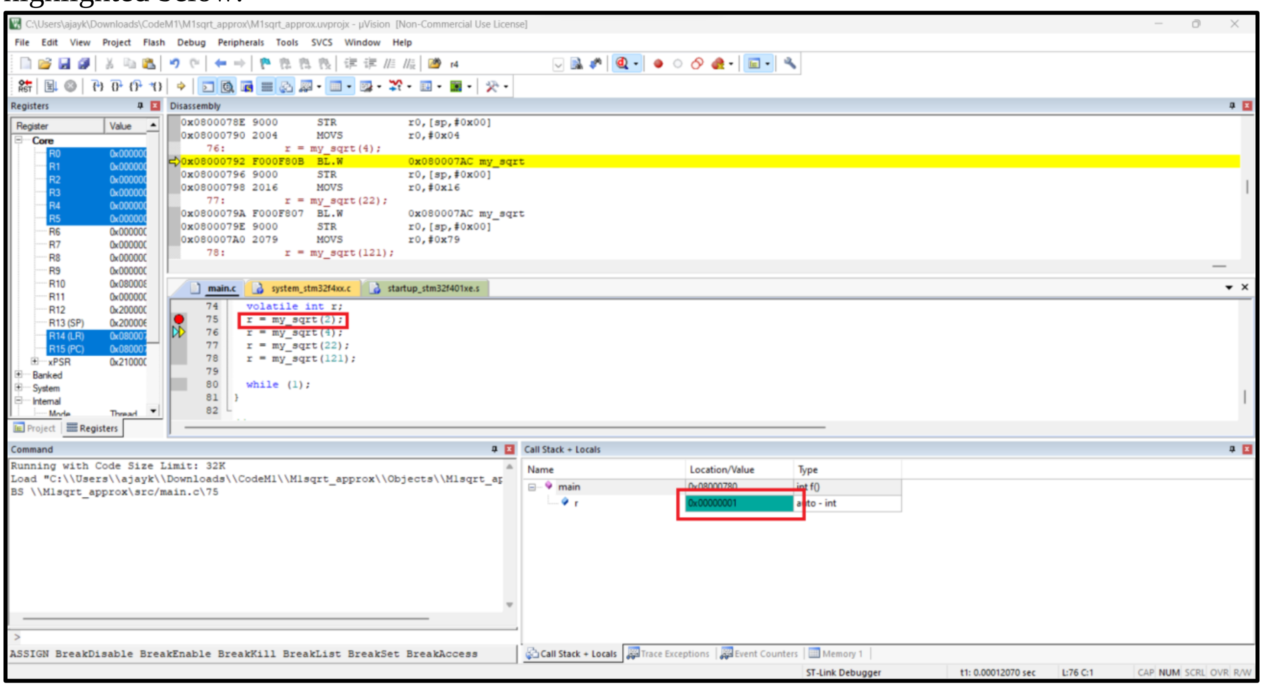
Cortex_m4_architecture : Above used memory model diagram.

Cortex-M4_reference_Manual : Explanation of memory model.

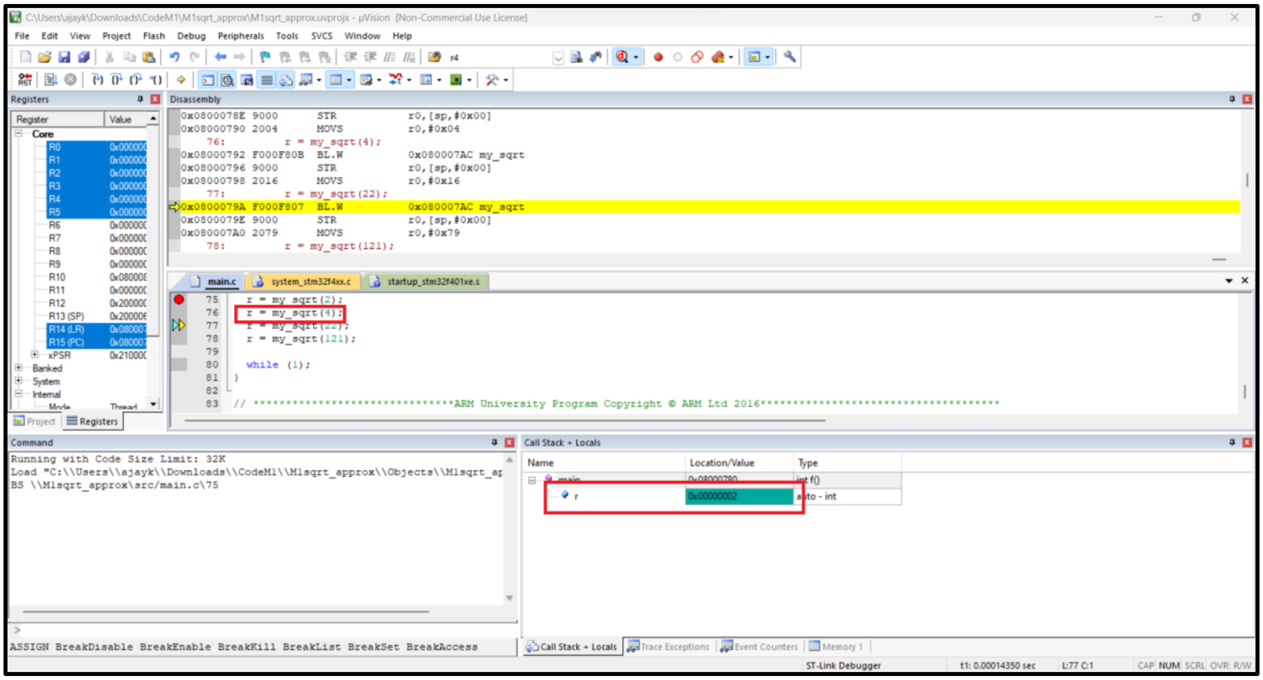
Que 3] Test your code with these inputs: 2, 4, 22, and 121. Record the results.

Ans:

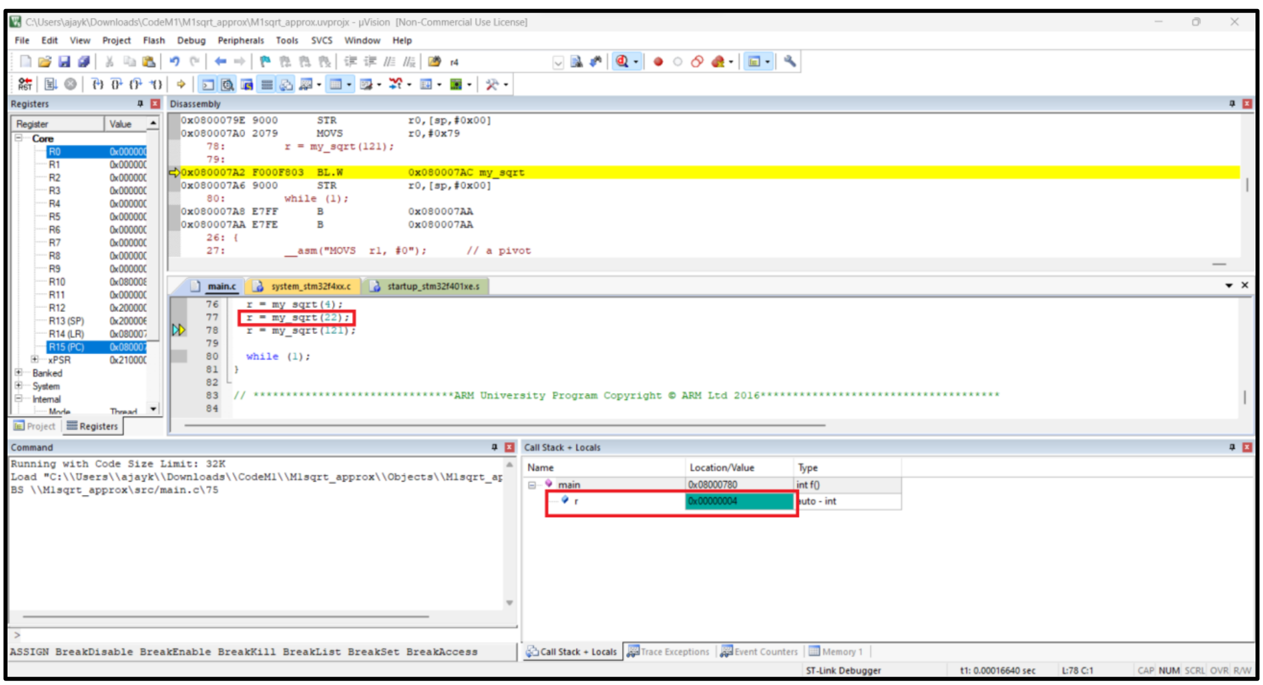
For input **2**, we get a square root approximation in integer as **1** as indicated in variable **r** highlighted below.



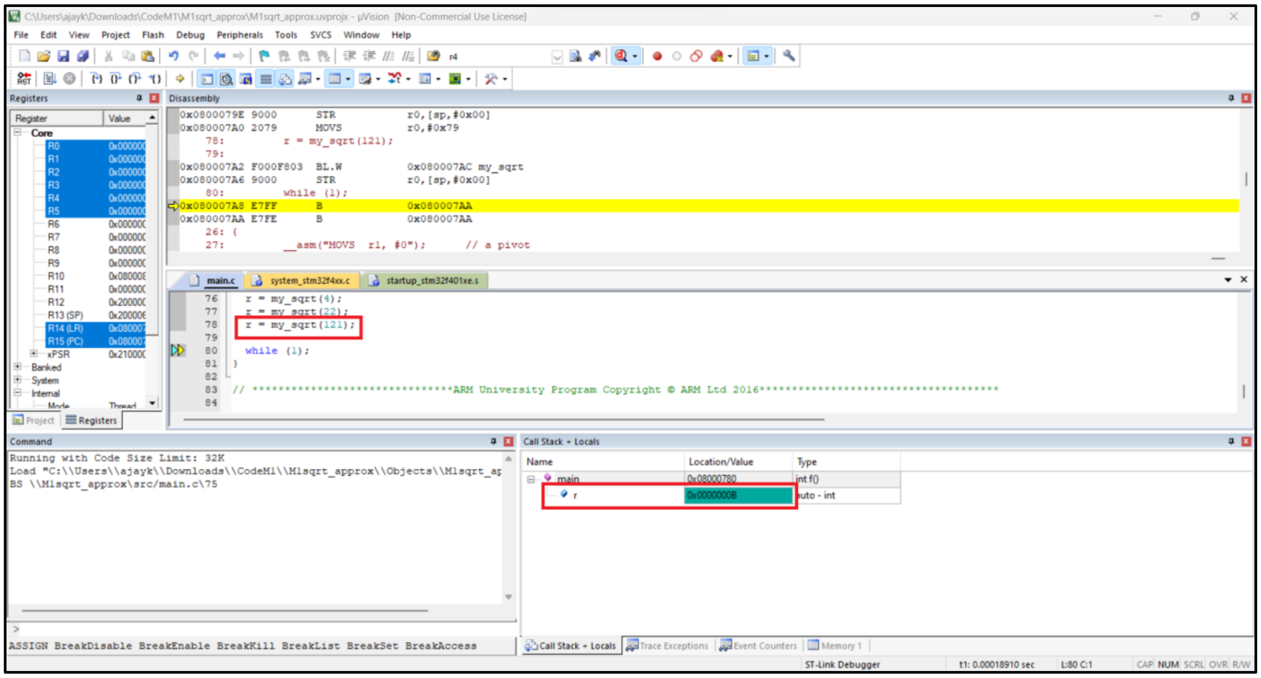
For input **4**, we get a square root approximation in integer as **2** as indicated in variable **r** highlighted below.



For input **22**, we get a square root approximation in integer as **4** as indicated in variable **r** highlighted below.



For input **121**, we get square root approximation in integer as **0x0B,** which equates to **11** in decimal as indicated in variable **r** highlighted below.



Que 4] Estimate the number of CPU cycles used for this calculation.

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

Ans:

The following analysis is done to approximate the number of CPU cycles:

To count the number of loop iterations, r6 is used, and the code is modified the r6 to increment inside the loop.

- **Three instructions** for setting the a, b, and c pivots initially.
- In the loop, **ten instructions** when a and b pivots are being set during approximation.
- **Seven instructions** when the previous approximation matches the current approximation and exit the loop.
- **One instruction** to copy the approximated square root value to r0.

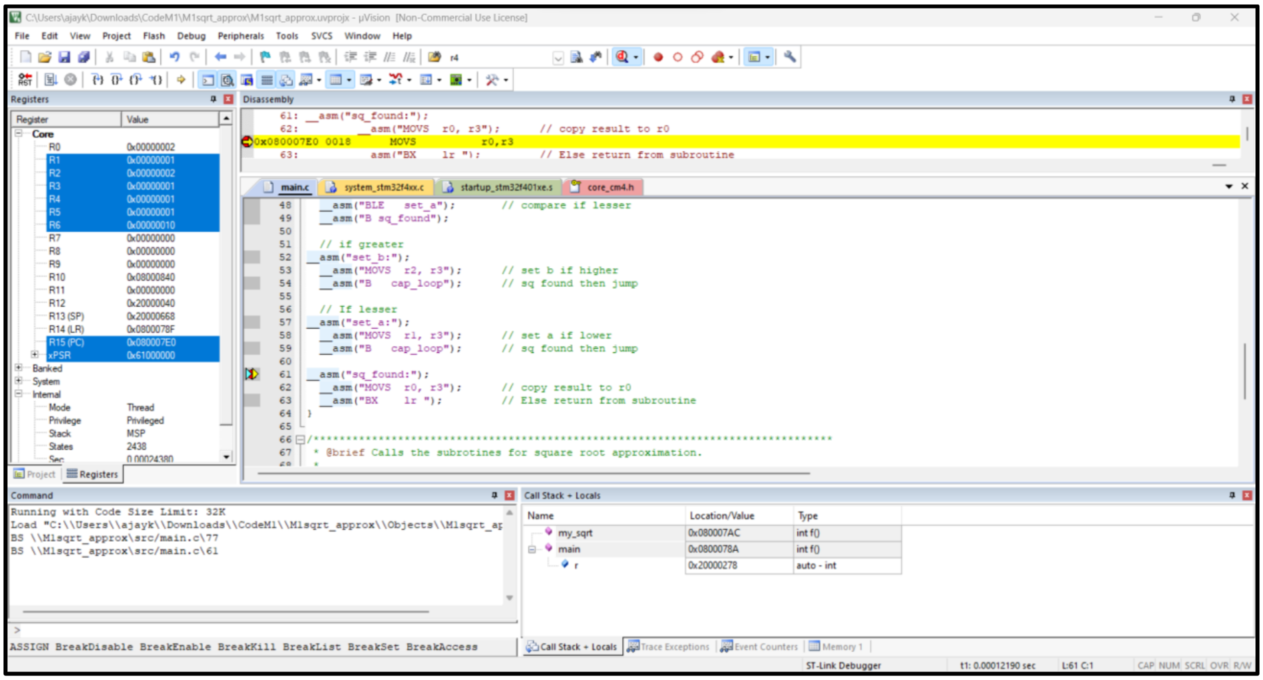
Below is the screenshot when running the square root approximation for input **2.** As indicated, the r6 value (loop iterations) is 0x10, which in decimal is 16.

· In 16 iterations, 15 iterations are used for adjusting a and b pivots. Hence, the total CPU cycles used during this is 15 * 10 = **150**

· The last iteration is used to exit the loop, which takes **seven** instructions.

· Also, 4 total instructions are used to set the pivot values at the beginning and copy the result to r0.

In total, **161 CPU cycles** are used while approximating the square root for input 2.



The code uses 1688 bytes of code memory with optimization set to -O0 as shown below.

Build Output

```
compiling uart.c...
assembling startup_stm32f401xe.s...
compiling system_stm32f4xx.c...
linking...
Program Size: Code=1688 RO-data=420 RW-data=0 ZI-data=1656
".\Objects\M1sqrt_approx.axf" - 0 Error(s), 4 Warning(s).
Build Time Elapsed:  00:00:02
```

Que 3] Auto-generate documentation using Doxygen. Provide either an HTML directory or PDF file documenting your codebase.

Ans: Generated doxygen is included in the .zip submission.

Appendix

1. **M1String.c code :**

```
/*
 * @file_name :  main.c
 *
 * @authors :    Kshitija Dhondage
 *
 * @brief :      This file contains the  main, my_strcpy (string copy),and my_capitalize
(string captalize) function definitions
 *
 */

 /**
 * @brief Copy the string from source to the destination
 *
 * @param[in] *str
 * String pointer (source) from which the string is to be copied.
 *
 * @param[in] *dst
 * String pointer (destination) to which the string is to be copied.
 *
 * @return
 * None
 */
void my_strcpy(const char *src, char *dst)
{
			char *dst_ptr = dst;
			char *src_ptr = (char*)src;

    do {
        *dst_ptr = *src_ptr;
        src_ptr++;
        dst_ptr++;
    } while (*src_ptr != '\0');
}

/**
```

```
 * @brief Convert the string to the uppercase.
 *
 * @param[in] *str
 * String pointer pointing to the string that is to be converted to the uppercase.
 *
 * @return
 * None
 */
void my_capitalize(char *str)
{
	int i = 0;

  while (str[i]) {
		if (str[i] >= 'a' && str[i] <= 'z')
			str[i] = str[i] - 'a' + 'A';

		i++;
	}
}

/**
 * @brief Main function Entry point function
 *
 * @param[in] Void
 *
 * @return
 * int
 */
int main(void)
{
	const char a[] = "Hello world!";
	char b[20];

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

	while (1);
}
```

****

1. **M1sqrt_approx code:**

```
/*****************************************************************************
*********
 * @file    main.c
 * @brief   Code for Project 1 Module 1 Square root approximation
 * @version v1.00
 * @date    25, September, 2023
 * @author  Ajay Kandagal (ajka9053@colorado.edu)
 *
 * @details Contains ARM assembly subroutines for square root approximation using
 *          bisection method.

*****************************************************************************
********/


/*****************************************************************************
*********
 * @brief Assembly subroutine for integer square root approximation.
 *
 * @details The logic uses the Bisection Method for square root approximation. The
 * size of the registers is assumed to be 32-bit for setting the upper pivot. The
 * approximation runs as long as the exact square root is found or the previously
 * approximation is the same as the current approximation.
 *
 * @param sq_number Number for which square root needs to be calcualated.
 *
 * @return Returns integer approximated square root value.
 *
```

```
*******************************************************************************/
__attribute__((naked)) int my_sqrt(int sq_number)
{
	__asm("MOVS  r1, #0");				// a pivot
	__asm("MOV  r2, #46431");	// b-pivot: Max possible sqrt for 32-bit unsigned int
	__asm("MOVS  r3, #-1");			// c pivot

__asm("cap_loop:");
	__asm("MOVS  r4, r3");			// c prev

	__asm("ADD  r3, r1, r2");	// add a_pivot and b_pivot
	__asm("LSR  r3, r3, #1");	// divide by 2

	__asm("MUL  r5, r3, r3");	// square

	__asm("CMP   r4, r3");	// check if old approximation is same new approximation
	__asm("BNE   set_pivots");
	__asm("B sq_found");

__asm("set_pivots:");
	__asm("CMP   r5, r0");	// check if equal to given number
	__asm("BHI   set_b");	// compare if greater
	__asm("BLE   set_a");	// compare if lesser
	__asm("B sq_found");

	// if greater
```

```
__asm("set_b:");
        __asm("MOVS  r2, r3");          // set b if higher
        __asm("B   cap_loop");

        // If lesser
__asm("set_a:");
        __asm("MOVS  r1, r3");          // set a if lower
        __asm("B   cap_loop");

__asm("sq_found:");
        __asm("MOVS  r0, r3");          // copy result to r0
        __asm("BX    lr ");       // return from subroutine
}

/*************************************************************************
*********
 * @brief Calls the subrotines for square root approximation.
 *
 * @details The function tests the square root subroutines by passing some sample
 * values. The square root approximated value can be monitored using the debugger and
 * adding break points at subroutine calls.
 *

*************************************************************************
*******/
int main(void)
{
        volatile int r;
        r = my_sqrt(2);
```

```
        r = my_sqrt(4);
        r = my_sqrt(22);
        r = my_sqrt(121);

        while (1);
}
```