

Tunku Abdul Rahman University College

BAME 3104

RTOS for Embedded System

***Real Time Operating System***

***with Preemptive Scheduler***

|  |  |  |
| --- | --- | --- |
| Name: | Foong Ghin Yew |  |
| ID: | 15 WAU 09184 |  |
|  |  |  |
| Programme: | RMB 3 |  |
|  |  |  |
| Tutor: | Dr. Poh Tze Ven |  |

TABLE OF CONTENTS

[LIST OF FIGURE i](#_Toc458532482)

[LIST OF TABLE ii](#_Toc458532483)

[CHAPTER 1: INTRODUCTION 1](#_Toc458532484)

[Objective: 1](#_Toc458532485)

[Brief Background: 1](#_Toc458532486)

[CHAPTER 2: METHODOLOGY 3](#_Toc458532487)

[Behavior Investigation: 3](#_Toc458532488)

[Behavior of Assembly Function 3](#_Toc458532489)

[Investigation on Mixing Assembly with C 6](#_Toc458532490)

[Behavior of Interrupt Handler 9](#_Toc458532491)

[RTOS Implementation: 12](#_Toc458532492)

[A. Initialization 12](#_Toc458532493)

[B. Task Switching 12](#_Toc458532494)

[CHAPTER 3: RESULT 14](#_Toc458532495)

[Result & Discussion: 14](#_Toc458532496)

[CHAPTER 4: EXTENSION 23](#_Toc458532497)

[CHAPTER 5: CONCLUSION 25](#_Toc458532498)

[REFERENCE 29](#_Toc458532499)

[APPENDIXES I](#_Toc458532500)

[Initialization for Tcb with Virtual Stack: I](#_Toc458532501)

[Initialization of Task’s Stack: II](#_Toc458532502)

[Tasks and Task Switching: III](#_Toc458532503)

[LinkedList Data Structure: IV](#_Toc458532504)

# LIST OF FIGURE

[Figure 1 Assembly program after first step 3](#_Toc458532505)

[Figure 2 Assembly program with loaded register and SP changed 4](#_Toc458532506)

[Figure 3 Assembly program with stack pushed 5](#_Toc458532507)

[Figure 4 Assembly program pop out value from stack 6](#_Toc458532508)

[Figure 5 Header file of ASM function for C program to call 6](#_Toc458532509)

[Figure 6 Variable 'taskSp' declared and exported in the Assemble Program 7](#_Toc458532510)

[Figure 7 Variable 'taskSp' from the Assembly Program was referenced 7](#_Toc458532511)

[Figure 8 Calling C Function in Assembly Program 7](#_Toc458532512)

[Figure 9 C Function return 0x00C0 FFEE to the Assembly program 8](#_Toc458532513)

[Figure 10 R0 brought value that return from a C Function 8](#_Toc458532514)

[Figure 11 Before Interrupt Occur, SP pointing to 0x2000 16D8 in Thread Mode 9](#_Toc458532515)

[Figure 12 After Interrupt occur, SP pointed to 0x2000 16B8 in Handler Mode 10](#_Toc458532516)

[Figure 13 R4 and LR were pushed into the stack 10](#_Toc458532517)

[Figure 14 Return from Interrupt Handler, SP point back to 0x2000 16D8 and Thread Mode 11](#_Toc458532518)

[Figure 15 The function of initTcb() that add the Tcb into the LinkedList 13](#_Toc458532519)

[Figure 16 A sample of task (task3 on top) and the function to initialize the stack (taskInit at the bottom) 13](#_Toc458532520)

[Figure 17 switchSp() and querySp() that is used for task switching 13](#_Toc458532521)

[Figure 18 Values in stackList after initialization 14](#_Toc458532522)

[Figure 19 Values of each Tcb block after initialization 14](#_Toc458532523)

[Figure 20 Program stuck in a forever while loop in main 17](#_Toc458532524)

[Figure 21 First entry to Interrupt, main stack pushed 18](#_Toc458532525)

[Figure 22 Two extra value pushed into main stack, SP changed to task1 virtual stack 19](#_Toc458532526)

[Figure 23 'sp' field in mainTcb was updated and no longer 0 19](#_Toc458532527)

[Figure 24 The head of stackList now point to task1 and the main moved to the tail 19](#_Toc458532528)

[Figure 25 Task switched to task1 20](#_Toc458532529)

[Figure 26 Stack changed to task2 virtual stack again 21](#_Toc458532530)

[Figure 27 'sp' field of task1Tcb updated 21](#_Toc458532531)

[Figure 28 stackList updated to make head poiting to task 2. 21](#_Toc458532532)

[Figure 29 Task switched to task 2 upon exit Interrupt 21](#_Toc458532533)

[Figure 30 Upon exit of Interrupt, SP pointed to CPU stack 0x2000 16B0 22](#_Toc458532534)

[Figure 31 Task switch back to main again 22](#_Toc458532535)

[Figure 32 Registers R4 to R11 were pushed into the stack 23](#_Toc458532536)

[Figure 33 Registers R4 to R11 were pop-ed after task switch but before exiting the Handler 24](#_Toc458532537)

[Figure 34 Header file of TCB for Tcb Initialization I](#_Toc458532538)

[Figure 35 Source file of TCB for Tcb Initialization I](#_Toc458532539)

[Figure 36 Header file of InitTask that initialize the virtual stack II](#_Toc458532540)

[Figure 37 Header file of InitTask that initialize the virtual stack II](#_Toc458532541)

[Figure 38 Main Function, taskInit() that initialize stack and all the tasks III](#_Toc458532542)

[Figure 39 Interrupt Handler that perform task switching III](#_Toc458532543)

[Figure 40 Header File of LinkedList, only the red boxed function will be used. IV](#_Toc458532544)

[Figure 41 Source File of Linked List for the red boxed functions in Figure 38 IV](#_Toc458532545)

[Figure 42 Extension program pop and push V](#_Toc458532546)

# LIST OF TABLE

[Table 1 Data Structure of mainTcb after initialization 15](#_Toc458532547)

[Table 2 Data Structure of task1Tcb after initialization 15](#_Toc458532548)

[Table 3 Data Structure of task2Tcb after initialization 15](#_Toc458532549)

[Table 4 Data Structure of task3Tcb after initialization 15](#_Toc458532550)

# CHAPTER 1: INTRODUCTION

## Objective:

1.1 To learn programming Cortex M3 microprocessor in C and Assembly language.

1.2 To implement a Real Time Operating System (RTOS) with tasks switching mechanism.

1.3 To investigate the behavior of Cortex M3 microprocessor when Exception occurs.

## Brief Background:

Every response to a signal have a time delay and the maximum delay time frame define the response to be real-time or not. If the response time fall within maximum allowable time frame, the response is said to be real-time respond. The same definition also apply to operating system, a real-time operating system (RTOS) serve application processes with the respond time within the time frame.

Implementing a real-time operating system will not be difficult if the system is meant for a single task only. However, multi-tasking are always needed in practical which escalated the RTOS implementation complexity with the need of a scheduler. Generally, there are two types of scheduler for RTOS which include Cooperative Multi-tasking and Preemptive Multi-tasking.

Cooperative Multi-tasking used ‘yield’ to task switch from one to another and therefore, ‘yield’ must be implemented by every task that will be run. Starvation problem may occurs in Cooperative Multi-tasking when one or more tasks does not implement ‘yield’ and take full control of the processing power. However, starvation may not occur if all the tasks implemented ‘yield’. Preemptive Multi-tasking was introduced to solve the starvation problem as if the task implemented by other user does not implement ‘yield’.

Preemptive Multi-tasking solved the problem of starvation by taking the processor away from the task after a certain period of time or event controlled by the scheduler. Preemptive Multi-tasking required careful implementation as many other problem may occur such as dead lock and race condition. However, Preemptive Multi-tasking allow other user to implement task to the system as the task switching was handled by the scheduler and no rules of ‘yield’ needed.

In this practical, a RTOS with Preemptive Multi-tasking scheduler shall be implemented. The operating system preempt for task switch every 100ms and task switch will be arranged in a round robin order. The task switching shall be done be changing the stack pointer (SP) value to point to another stack. All the task switching shall be done within the interrupt handler so that the return from the handler will not go back to the entry task but the next task that switched in the handler. Exception behavior such as mode change, stack pushing and pop-ing activity shall be investigated first. The practical was accomplished with mix of C language and Assembly language.

The report consist of four chapters where the first chapter state the objective and brief background about RTOS and multi-tasking. Chapter 2 shows the methodology to accomplish the objective in analyzing, interpreting and writing the RTOS program. Whereas, Chapter 3 shows the result and verification of the RTOS program written. Finally, a conclusion will be concluded in Chapter 4 about the practical.

# CHAPTER 2: METHODOLOGY

This chapter divided into two parts that describe the process on conducting the experiment. First part is the *Behavior Investigation* that describe the method used to study the behavior of ARM Cortex M3 on handling exception and function call that consist of C, Assembly and mixed languages. Second part is *RTOS Implementation* that describe the logic planned to accomplish the RTOS and the method to verify the product of this practical.

## Behavior Investigation:

In this practical, software “Keil uVision” were used to study and develop the RTOS. Three major feature of the software were used to study the behavior of the ARM Cortex M3 which include the debugger, the memory watch window and the registers window. In general, debugger are used to study the program step by step and the two window was used to visualize the process that occurred to all these memory elements, the register and the stack.

### Behavior of Assembly Function

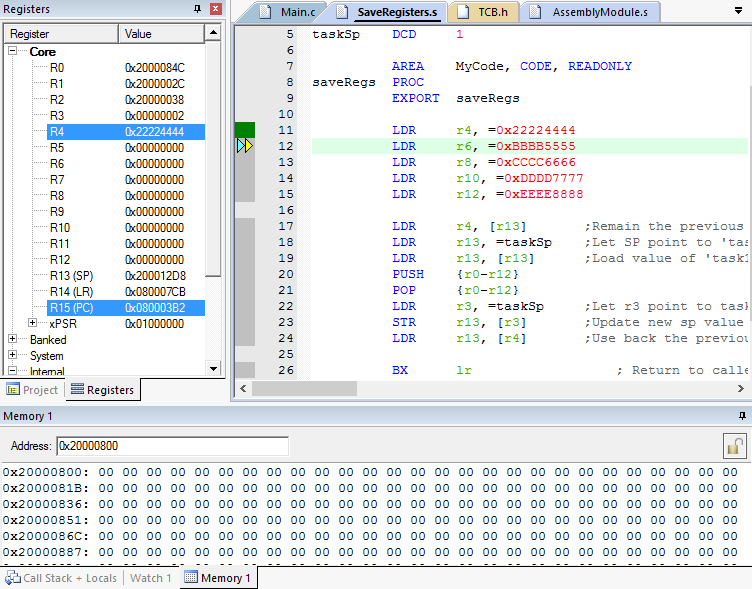


Figure 1 Assembly program after first step

The left side in Figure 1 was the Registers Window that shows all the working registers from R0 to R12, the Stack Pointer Register (SP) followed by the Link Register (LR) and the Program Counter (PC). The xPSR register at the most bottom have a few function which will be discussed in the later part. The memory stack shows in the bottom of Figure 1 display all the value stored in the location whereas the yellow and light blue arrow (debugger pointer) shows the next step to be execute in the program.

Figure 1 shows the initial state of the assembly program that was used to study the behavior. As the first instruction (LDR R4, =0x22224444) executed, the register R4 in the Registers window was highlighted and the value changed to 0x22224444. However, nothing happen to the memory space yet at this point.

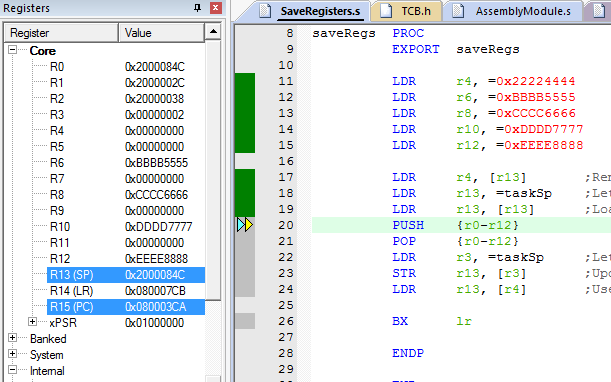


Figure 2 Assembly program with loaded register and SP changed

As shown in Figure 2, register r6, r8, r10 and r12 were loaded with the data that was set. The next step investigated is about changing the Stack Pointer (SP) as shown in Figure 2. The SP in Figure 1 was pointing to 0x2000 12D8 but the SP in Figure 2 was pointing to 0x2000 084C which is a virtual stack initialized in the C program. The SP (R13) was modified with two command but not directly hardcoding because the system provide a few rules such that the PC, SP and xPSR registers are not allowed to be changed when the program are in non-privileged mode which will be discussed later. When the PUSH command was executed the memory stack window display the register value that was stored in the location as shown in Figure 3. As expected, the value was of R0 to R12 was stored in the stack as shown in the gold color value in the Memory Window in Figure 3.

A few behaviors to be noted here where the SP was originally pointing to 0x2000 084C before executing the PUSH command. When the PUSH command was executed, the registers value was stored from 0x2000 084B and backward to 0x2000 0818. The memory does not store that value starting from 0x2000 084C but starting from one location before the SP. This mean that the data pushed when the SP point to 8 will be stored at 7 and move toward a lower addressed location. R0 to R12 have 13 register and each register take 4 addressed location in the memory stack. In total, the PUSH command should occupied 52 location in the stack or 0x34 in hex. 0x2000 084C minus 0x2000 0818 which is exactly the value of 0x34.

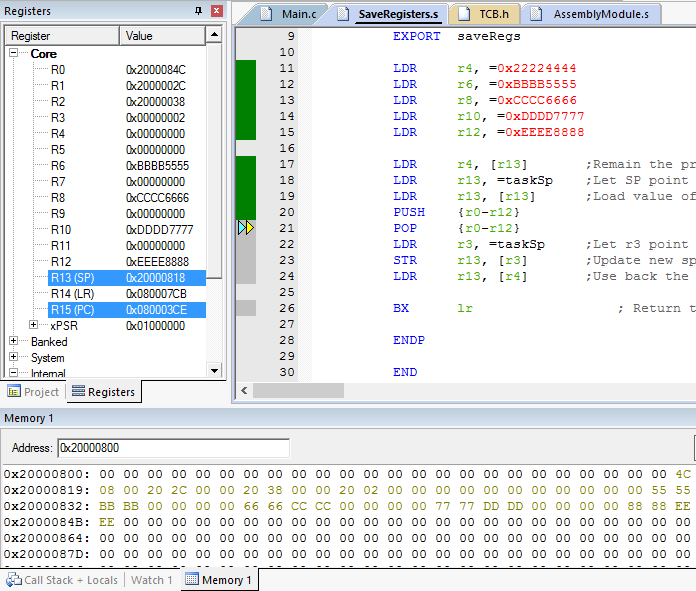


Figure 3 Assembly program with stack pushed

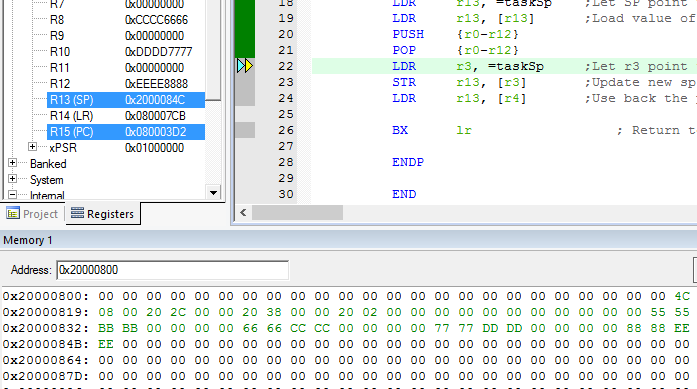


Figure 4 Assembly program pop out value from stack

As soon as the POP command was executed, the value in the memory stack that was originally gold in color changed to green indicating that the data has been pop-ed out as shown in Figure 4 above. The SP was also changed back to 0x2000 084C as all the value previously pushed was pop-ed and therefore, the SP point back to the original SP pointed location. At the end of this assembly program, the command “BX LR” was executed which branch back to the address stored in the LR register which is pointing to the function caller address when the this function was called.

### Investigation on Mixing Assembly with C

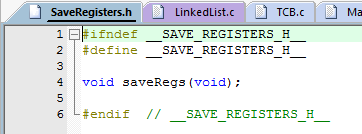


Figure 5 Header file of ASM function for C program to call

In order to call an Assembly function in a C program, a header file (.h) must be created with the same name as the assembly file (.s) as shown in Figure 5 above. Figure 5 show the assembly function discussed in previous section where the name of the function is saveRegs(). This function return a void as the variable used in the assembly program was exported which mean it can be referenced in the C program as shown in Figure 6 and 7.

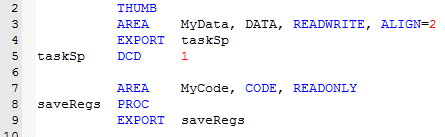


Figure 6 Variable 'taskSp' declared and exported in the Assemble Program

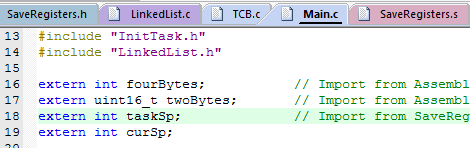


Figure 7 Variable 'taskSp' from the Assembly Program was referenced

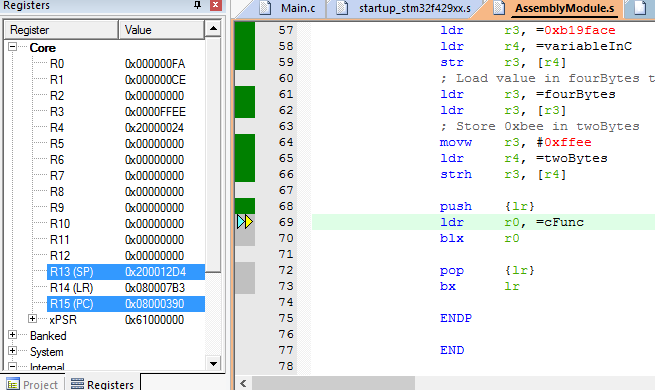


Figure 8 Calling C Function in Assembly Program

Assembly function can be called by a C program by exporting the function and creating a header file. Similarly, an Assembly program can call a C function also as shown in Figure 8 above. Where the C function address was loaded into a register and use a branch command to enter the function. As the function return a value, the value will be placed in register R0 as shown in Figure 9 and Figure 10 below.

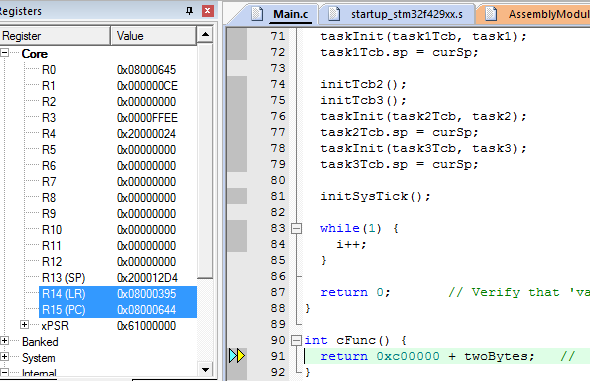


Figure 9 C Function return 0x00C0 FFEE to the Assembly program

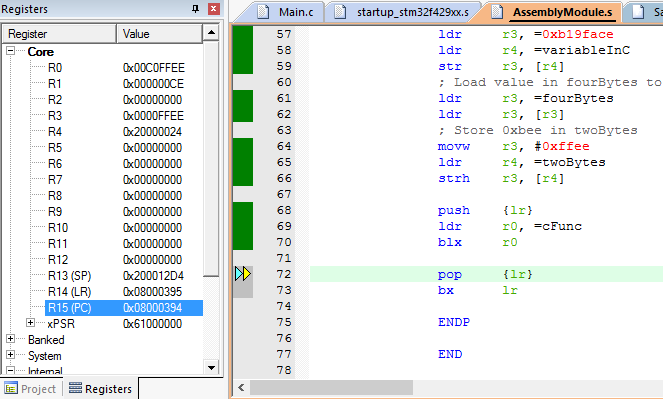


Figure 10 R0 brought value that return from a C Function

### Behavior of Interrupt Handler

One of the most important process for implementing RTOS with Preemptive Multi-tasking is the interrupt which is used for task switching. Figure 11 below shows the SP pointing to 0x2000 16D8 before any interrupt occur. It was working in Thread mode as shown in the Internal section in the Registers Window. Note that the xPSR register are storing the value of 0x0100 0000.

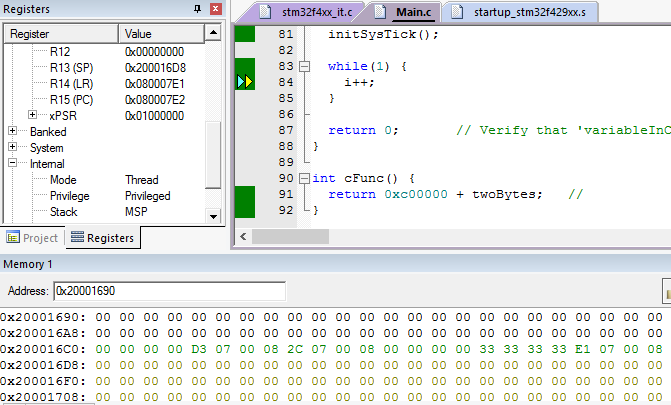


Figure 11 Before Interrupt Occur, SP pointing to 0x2000 16D8 in Thread Mode

When Interrupt occurred, the SP pointed to 0x2000 16B8 which mean 0x20 of byte were pushed into the stack. As shown in Figure 12 below, the 32 byte were pushed into the stack occupied location between 0x2000 16D8 to 0x2000 16B8. The data pushed into the stack follow the sequence of {xPSR, PC, LR, R12, R3, R2, R1, R0} and let this sequence be **Stacking Sequence**. As stated earlier, the xPSR register was 0x0100 0000 and it was pushed into the stack as 0x01000000. However, the xPSR was storing 0x0100 000F after entered the Interrupt Handler and the mode in the Internal section changed to Handler mode from previous Thread mode. This shows that the last nible of xPSR was indicating the mode.

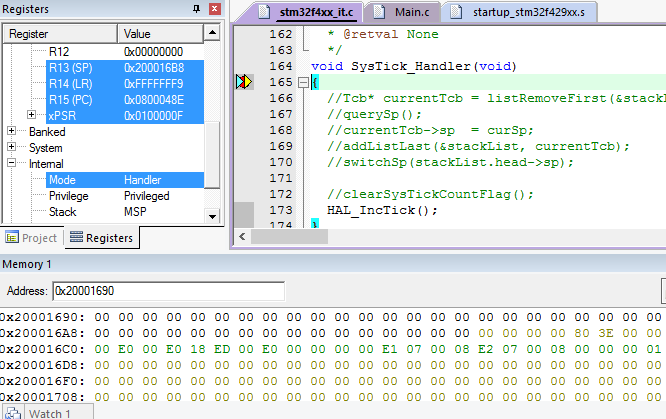


Figure 12 After Interrupt occur, SP pointed to 0x2000 16B8 in Handler Mode

After the first step in the Interrupt Handler, another two byte of data were pushed into the stack which is {R4, LR} and the LR at this time was 0xFFFF FFF9 as shown in Figure 13 below. This value must be noted such that only this value and 0xFFFF FFFD can return the mode back to Thread mode. Upon exit of the handler, the value 0xFFFF FFF9 was pop into the PC and this instruct the processor to pop the value pushed at interrupt entrance as shown in Figure 14.

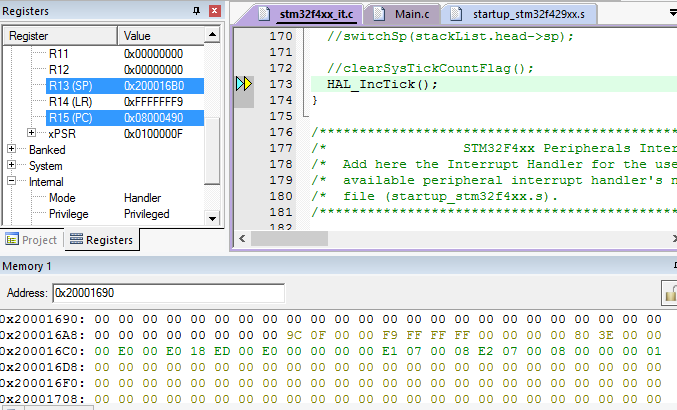


Figure 13 R4 and LR were pushed into the stack

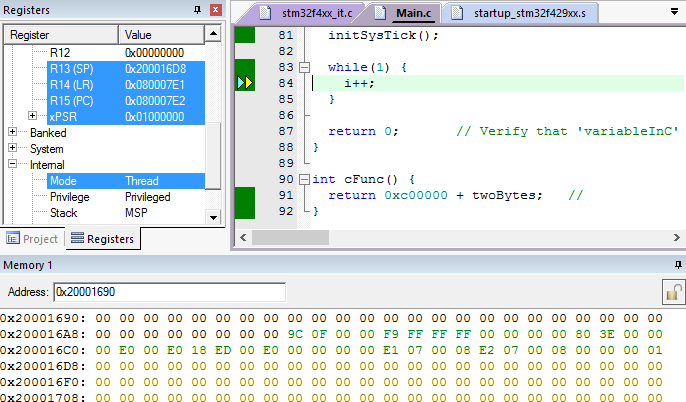


Figure 14 Return from Interrupt Handler, SP point back to 0x2000 16D8 and Thread Mode

The pop out result can be verify at the Registers Window, such that the xPSR value was pop and back to 0x0100 0000 and the PC value back to 0x0800 07E2 as following the sequence in the green color data in the stack memory as shown in Figure 14 above.

## RTOS Implementation:

The implementation of the RTOS Preemptive multi-tasking was divided into two part, the initialization and the task switching. The initialization part initialize three task function with a permanent while loop. This make the task function run forever before any interrupt occur. Therefore, the task switching part will be done in the interrupt.

### A. Initialization

First of all, data structure with a virtual stack, a stack pointer, a name and a next pointer was created with the type name called Tcb. Each task function will need a Tcb and therefore, four Tcb will be needed which include the three tasks and the main function. Function ***initTcb()*** shall be created to create an empty Tcb object. However, the main function will not use the virtual stack as it started using the CPU stack and therefore only the ‘sp’ and ‘next’ fields will be used.

As stated earlier, the interrupt will pop {R4, LR} and follow by the Stack Sequence upon interrupt exit but nothing was pushed in the virtual stack for the first attempt of task switching in the interrupt. Therefore, the Tcb’s virtual stack must be initialized with the Stack Sequence and the two extra register in it. This process shall be handled by a function called ***taskInit()***.

### B. Task Switching

The process of task switching are to be done in the interrupt handler. SysTick is pre-configured to generate an interrupt every 100ms which mean task switching activity will be carried out every 100ms. A LinkedList data structure are chosen to perform the task switching where all the 4 Tcb blocks are placed into the LinkedList. The head in the LinkedList will always be the running task before the interrupt occur. When an interrupt occurred, the head Tcb are switched to the tail of the LinkedList and the second Tcb element in the LinkedList will become the new head. The SP of the previous head Tcb will be updated to the new SP value in the register by calling ***querySp()*** function before loading the new SP in the new head Tcb into the running SP register. After saving the SP to the previous head, the SP register are switched to the new Tcb block’s SP by calling the function ***switchSp()***.

As a LinkedList is used to store all the Tcb blocks, the job of adding Tcb element into the LinkedList are added to the function ***initTcb()*** so that each Tcb block will be added to the LinkedList when initialized.

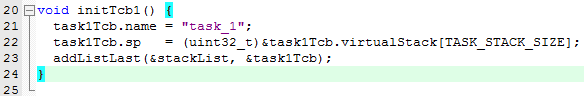


Figure 15 The function of initTcb() that add the Tcb into the LinkedList

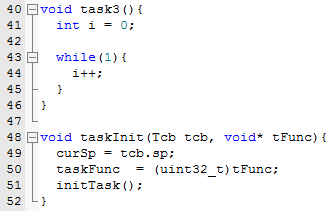


Figure 16 A sample of task (task3 on top) and the function to initialize the stack (taskInit at the bottom)

**Note:** initTask() is an assembly function that initialize the stack with ‘curSp’ and ‘taskFunc’

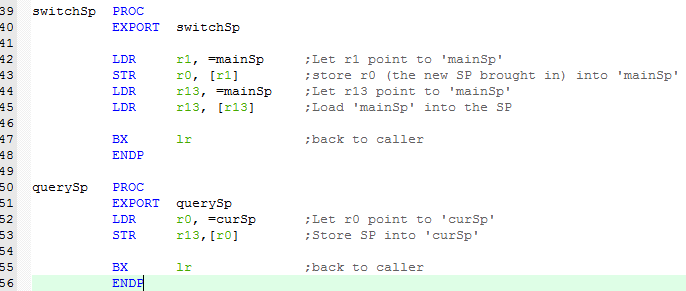


Figure 17 switchSp() and querySp() that is used for task switching

# CHAPTER 3: RESULT

This chapter express and discuss the results of verification on the RTOS implementation from the previous chapter. By using debugger, memory window and registers window, the result are verified. This chapter first verify the initialized result and check if it was as expected, the later part verify the task switching process.

## Result & Discussion:

Initialization play an important role in this practical, therefore the first step of verification is to verify the initialized items. After calling all the initialization function but before any interrupt occur, the stackList that used to hold the Tcb blocks are verified as shown in Figure 18. The head is pointing to 0x2000 0038 which is actually the ‘mainTcb’ block. Whereas, the tail is pointing to 0x2000 0C5C which represent ‘task3Tcb’ and the length shows that stackList are storing four Tcb blocks in it.



Figure 18 Values in stackList after initialization

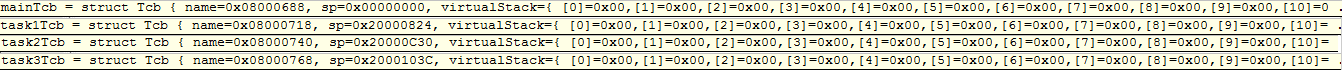


Figure 19 Values of each Tcb block after initialization

Figure 19 above shows that all the Tcb block have the ‘sp’ field initialized with some value except the mainTcb. This is because the main function will be the first task that enter the interrupt and therefore, the stack will be automatically pushed when it enters the Interrupt Handler. Unlike other tasks, they will first execute pop instead of push as the task switch in the Interrupt Handler. They go through Interrupt Handler exit first before going through Interrupt Handler entrance. However, the main Function is the only function that will go through Interrupt Handler entrance first after the initialization. Therefore, the sp field in the mainTcb will not need to be initialized in init function as it will be updated in the Interrupt Handler.

Table 1 Data Structure of mainTcb after initialization

|  |  |  |
| --- | --- | --- |
| **Field** | **Location Address** | **Value** |
| Name | 0x 2000 003C | “main” |
| Stack Pointer | 0x 2000 0040 | 0x0000 0000 |
| Virtual Stack  (Not Used) | 0x 2000 0044  …  0x 2000 0443 |  |
| Next | 0x 2000 0440 | 0x2000 0444 |

Table 2 Data Structure of task1Tcb after initialization

|  |  |  |
| --- | --- | --- |
| **Field** | **Location Address** | **Value** |
| Name | 0x 2000 0444 | “task1” |
| Stack Pointer | 0x 2000 0448 | 0x2000 0824 |
| Virtual Stack | 0x 2000 044C  …  0x 2000 084B |  |
| Next | 0x 2000 084C | 0x2000 0850 |



Table 3 Data Structure of task2Tcb after initialization

|  |  |  |
| --- | --- | --- |
| **Field** | **Location Address** | **Value** |
| Name | 0x 2000 0850 | “task2” |
| Stack Pointer | 0x 2000 0854 | 0x2000 0C30 |
| Virtual Stack | 0x 2000 0858  …  0x 2000 0C57 |  |
| Next | 0x 2000 0C58 | 0x2000 0C5C |



Table 4 Data Structure of task3Tcb after initialization

|  |  |  |
| --- | --- | --- |
| **Field** | **Location Address** | **Value** |
| Name | 0x 2000 0C5C | “task3” |
| Stack Pointer | 0x 2000 0C60 | 0x2000 103C |
| Virtual Stack | 0x 2000 0C64  …  0x 2000 1063 |  |
| Next | 0x 2000 1064 | 0x0000 0000 |



Table 1, Table 2, Table 3 and Table 4 tabulated the values in mainTcb, task1Tcb, task2Tcb and task3Tcb respectively. Noted that the ‘next’ field in Table n will always store the value of ‘name’ field of Table n+1. Table 4 shows that the next field was 0 as there is no more Tcb block to be pointed to. This verified that stackList stored all the Tcb blocks and linked them in a chain.

Next, the mainTcb have a sp of zero which is explained previously such that it does not need to be initialized in the main function. Since the virtual stack of the mainTcb was not used, the value in it will be ignored as the main concern of this practical is preemptive task switching. Table 2, 3 and 4 shows that the all the respective ‘sp’ field are 0x28 position above their respective ‘next’ field. This is because each register take up 0x04 location. Stack Sequence together with two exta register total taken 10 times of 4 location which is 0x28 location in hex.

Masking the mainTcb block, notice that the value in the virtual stack for each table are about the same. It always have 0x0100 0000 for the xPSR, 0x0000 0012 for the R12, 0x3333 3333 for R3, 0x2222 2222 for R2, 0x1111 1111 for R1 and 0x0000 0000 for R0. The value of the two top most pushed location will always store 0xFFFF FFF9 and 0x2000 0000. The different between each of table is the stack location for the PC and LR. This is because the PC that will pop by the processor have to store their respective task function address as to switch back to the respective task.

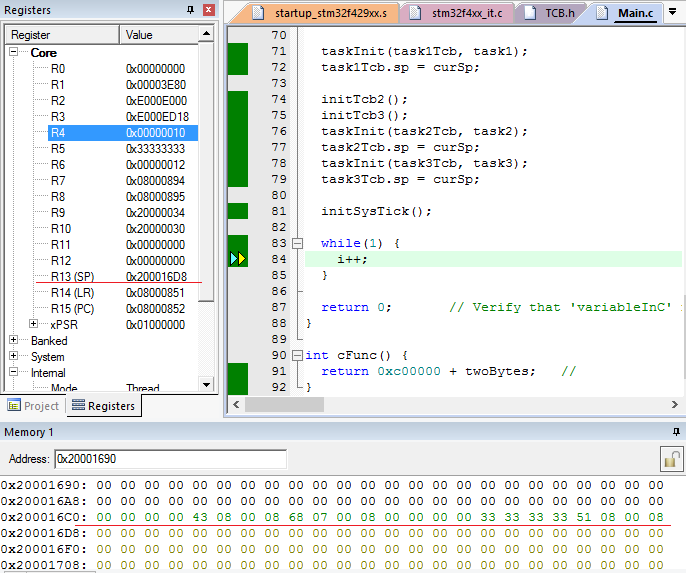


Figure 20 Program stuck in a forever while loop in main

Figure 20 above shows that the program have got stuck in a forever while loop in the main function. This is to prove that the task switch is not due to function call but a real task switch. Notice that the SP at this time was 0x2000 16D8 which is the CPU stack and mainTcb value are as Table 1. When the interrupt occur at this point, the stack memory value changed as shown in Figure 21 below where the Stack Sequence was pushed. The value of the SP also changed from 0x2000 16D8 to 0x2000 16B8. Not only that, the LR was also changed to 0xFFFF FFF9 which indicate the instruction to pop Stack Sequence when PC receive this value and the last nibble of xPSR changed to F which indicate Handler mode.

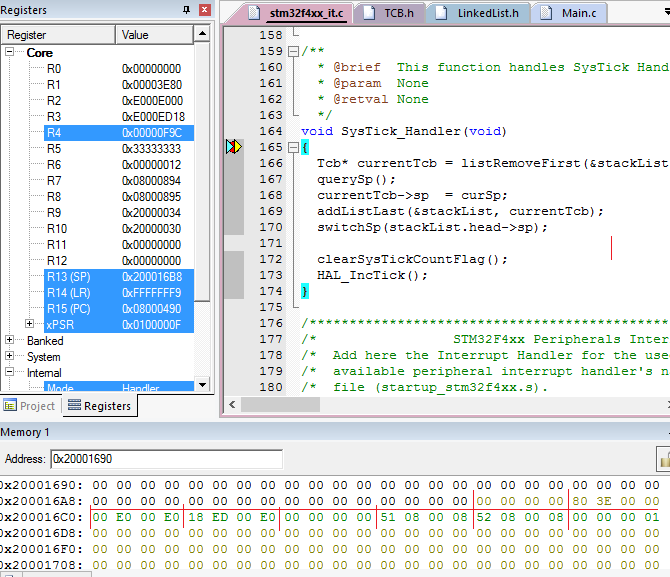


Figure 21 First entry to Interrupt, main stack pushed

After entering the Interrupt, two extra register were pushed into the CPU stack as shown in Figure 22 below which have the value of {0x0000 0F9C, 0xFFFF FFF9}. The ‘sp’ in the mainTcb then updated to the new value of 0x200016B0 and no longer equal to 0 as shown in Figure 23. Refer back to Figure 22 below, the SP value changed to 0x2000 0824 before exit the Handler and the format of the value in stack memory for both CPU stack and task1 virtual stack can be compared. The PC location and LR location for the two stacks were different and point to different function. Also, notice that the stackList value have changed where the head now point to task1Tcb and the tail point to the mainTcb as shown in Figure 24 below and comparable with Figure 18 above that state the initial stackList value.

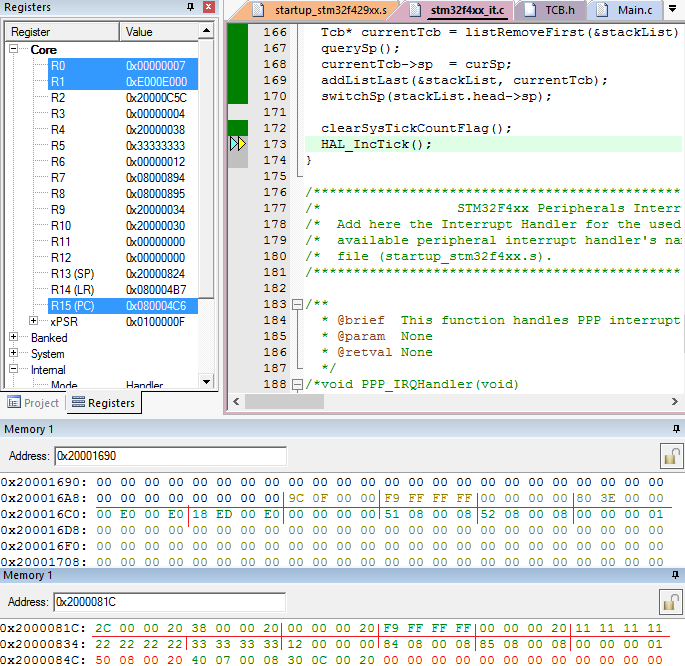


Figure 22 Two extra value pushed into main stack, SP changed to task1 virtual stack



Figure 23 'sp' field in mainTcb was updated and no longer 0



Figure 24 The head of stackList now point to task1 and the main moved to the tail

Upon exit of the Interrupt, the program jumped to the task1 instead of going back to main and the last nibble of the xPSR register turn back to 0 again indicating Thread mode as shown in Figure 25 below. The SP now point to 0x2000 084C as the value in 0x2000 0824 to 0x2000 084C was pop-ed into their respective location by the processor. As shown in Figure 25 below, the value in the stack still remain but the SP is now pointing to 0x2000 084C which indicated by the red line in the memory window.

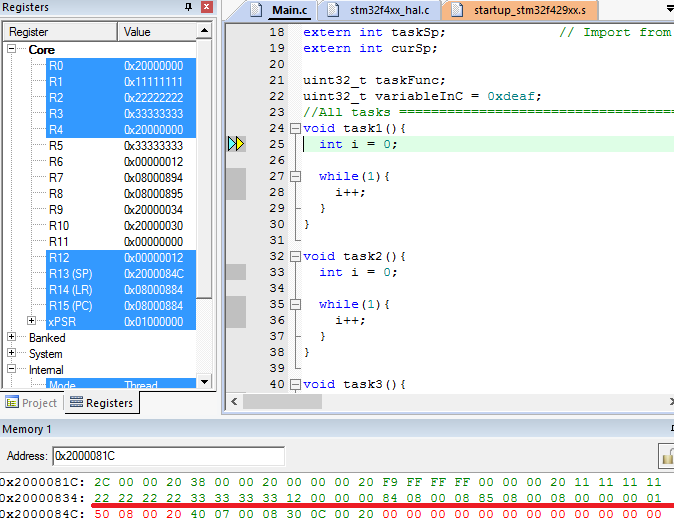


Figure 25 Task switched to task1

After stuck in the task1 forever while loop, an interrupt occur again and the process restart. As shown in Figure 26 and Figure 27 below, the SP changed to 0x2000 0C30 and the previous SP was updated to the task1Tcb sp field. Similarly, the stackList moved task1Tcp to the tail and the head now point to task2Tcb as in Figure 28. As compare with Figure 24, the head 0x2000 0444 (task1) was now at the tail in Figure 28. The process then repeat again and switch to task 3. When the entering the Interrupt from the task 3 forever while loop, the process occurred again and switch to main again as shown in Figure 30 and Figure 31 below.

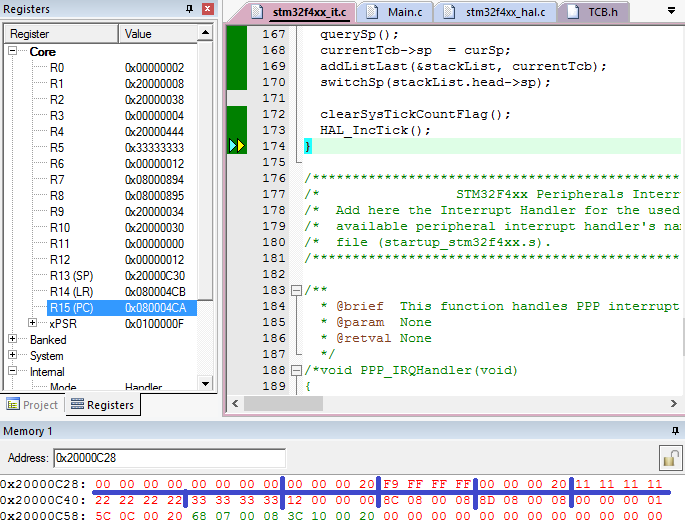


Figure 26 Stack changed to task2 virtual stack again



Figure 27 'sp' field of task1Tcb updated



Figure 28 stackList updated to make head poiting to task 2.

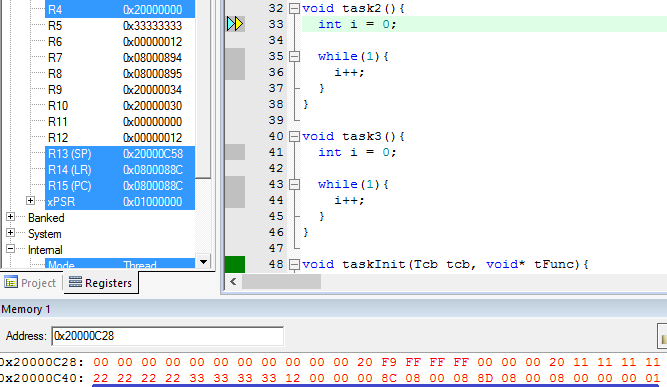


Figure 29 Task switched to task 2 upon exit Interrupt

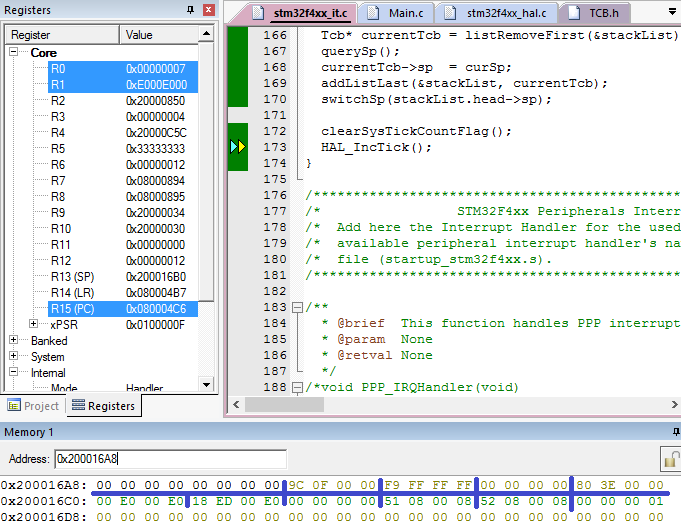


Figure 30 Upon exit of Interrupt, SP pointed to CPU stack 0x2000 16B0

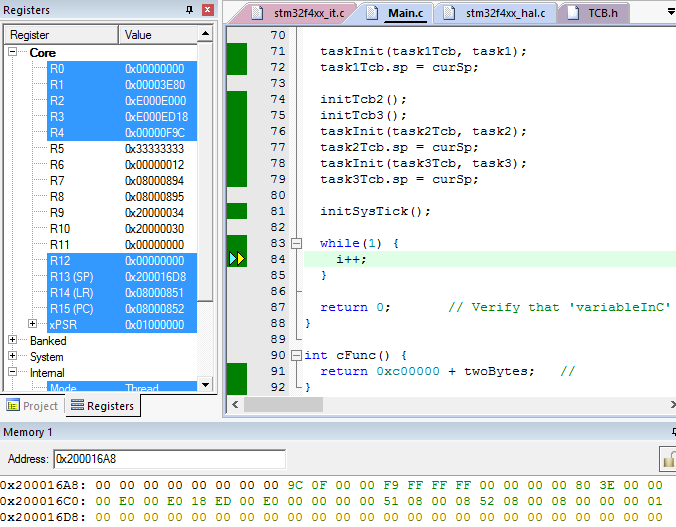


Figure 31 Task switch back to main again

The process then restart again and again forever by switching the task according to the Tcb blocks in the LinkedList. As the LinkedList is not becoming shorter, it’s rotating the blocks, the task switch process will never end, repeating the sequence of main > task1 > task2 > task3 and back to main again.

# CHAPTER 4: EXTENSION

This chapter is an extension on improving the working program implemented previously. An issue rise up in the previous design such that the task switching does not handle the extra register that is used. Let’s discuss with an example. The main function used registers R0 to R3 but task1 used R0 to R6 in the respective program. When interrupt occur, the Interrupt Handler automatically pushed R0 to R3 and task switched to task1 in the Handler. Upon exit the Handler, R0 to R3 will be pop out to task1’s virtual stack but R4 to R6 that is being used will not be pop. Therefore, the safest way to solve this issue is to manually push all R4 to R11 in the Handler before task switching and pop R4 to R11 after task switched.

Two function were introduced which is popRegs() and pushRegs() that will pop R4 to R11 and push R4 to R11 respectively. Also, initTask() was modified to push R4 to R11 also. As shown in Figure 32, the Memory Window shows that the value of R4 to R11 are pushed after the {R4, LR} extra push as indicated between the two red line. The data between the two blue lines are the data that was pushed previously before this extension which had no changes.

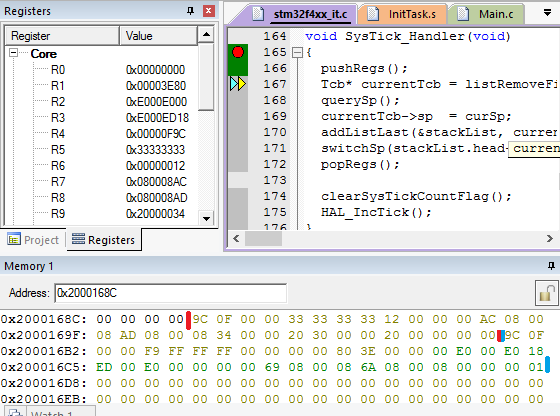


Figure 32 Registers R4 to R11 were pushed into the stack

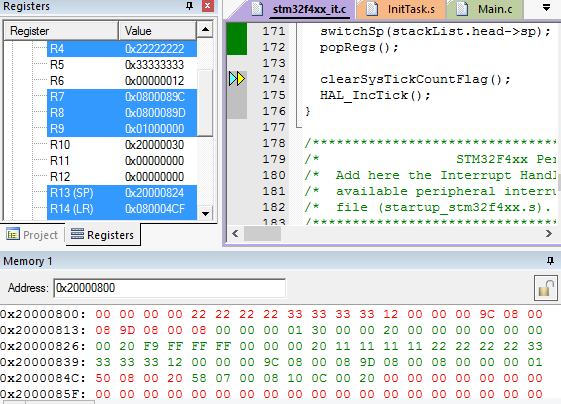


Figure 33 Registers R4 to R11 were pop-ed after task switch but before exiting the Handler

Figure 33 above shows the value of R4 to R11 being pop from task1’s virtual stack after task switched from main function to task1. Pop-ing of R4 to R11 are done before exit of the Handler because exiting the Handler will auto pop the stack sequence and this may violate the sequence as R4 to R11 are pre-pushed. Therefore, R4 to R11 must be pop before exit the Handler and of course after task switched.

# CHAPTER 5: CONCLUSION

Generally, Real Time Operating System with Preemptive Scheduler can be accomplished by using virtual stacks for tasks and a LinkedList to store these virtual stacks. By implementing a SysTick Interrupt Handler, task switch can be done by changing the SP value to point to the respective switching virtual stack. The Stack Sequence of {xPSR, PC, LR, R12, R3, R2, R1 and R0} will always be pushed and pop-ed automatically when the interrupt occurred. Two extra register R4 and LR will then be pushed in the Interrupt Handler where the LR at this point is 0xFFFF FFF9. This value then pop into the PC before exiting the Handler and the all the previously pushed Stack Sequence will be pop back to their respective Register. By initializing the virtual stack with the value of Stack Sequence and the R4 and LR, task switching can be easily done by switching the stack. There are a few thing required to take concern, the initialized Stack Sequence in the virtual stack must have the PC location pointed to the task that wanted to be switched. Another core is the value LR that was pushed after pushing the Stack Sequence, it must be 0xFFFF FFFX where the last nibble X can only be 0x9, 0xD or 0x1. Other than that may cause a Hard Fault error. Placing the address of the function into the LR also allow task switch but only once. As the processor stay in Handler mode and the SysTick Interrupt that have the same priority can never take over the program. Therefore, the task switch will not happen again if the LR was not 0xFFFF FFFX but a valid function address.

# REFERENCE

[1] Joseph, Y 2014, ‘Exception Handling in Detail’, *The definitive guide to ARM Cortex –M3 and –M4 Processor*, vol 3. no. 3, pp. 273 – 286.

[2] Texas Instrument 2010*, Mode (handler, thread) handling on cortex M3*, viewed 30th July 2016, < <https://e2e.ti.com/support/microcontrollers/stellaris_arm/f/471/t/84362> >

[3] Joseph, Y 2014, ‘Architecture’, *The definitive guide to ARM Cortex –M3 and –M4 Processor*, vol 3. no. 3, pp. 76 – 113.

# APPENDIXES

### Initialization for Tcb with Virtual Stack:

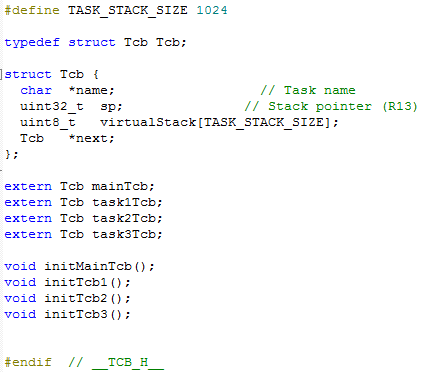


Figure 34 Header file of TCB for Tcb Initialization

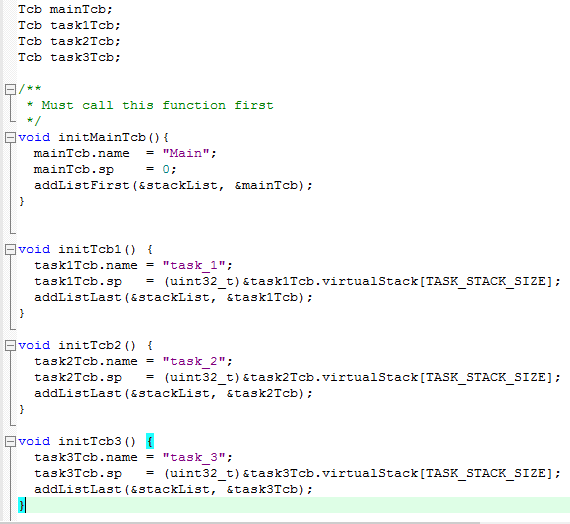


Figure 35 Source file of TCB for Tcb Initialization

### Initialization of Task’s Stack:

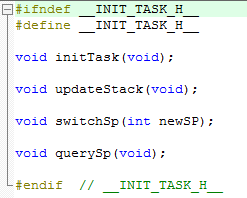


Figure 36 Header file of InitTask that initialize the virtual stack

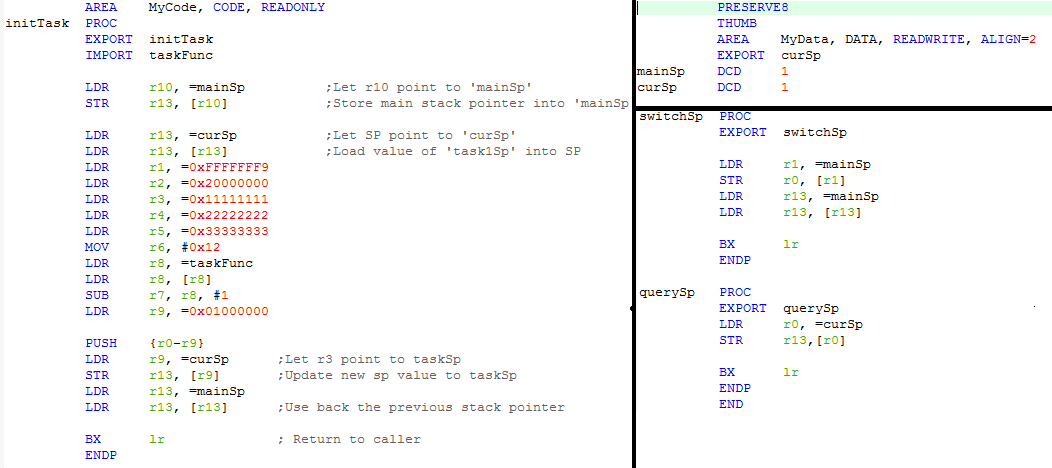


Figure 37 Header file of InitTask that initialize the virtual stack

### Tasks and Task Switching:

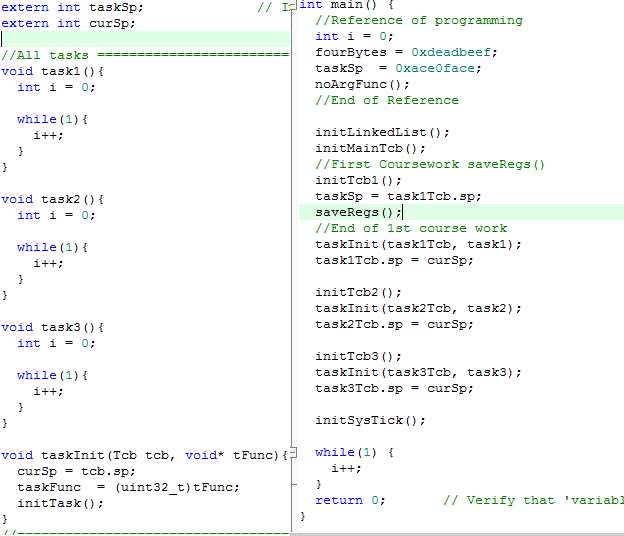


Figure 38 Main Function, taskInit() that initialize stack and all the tasks

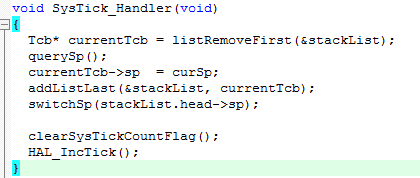


Figure 39 Interrupt Handler that perform task switching

### LinkedList Data Structure:

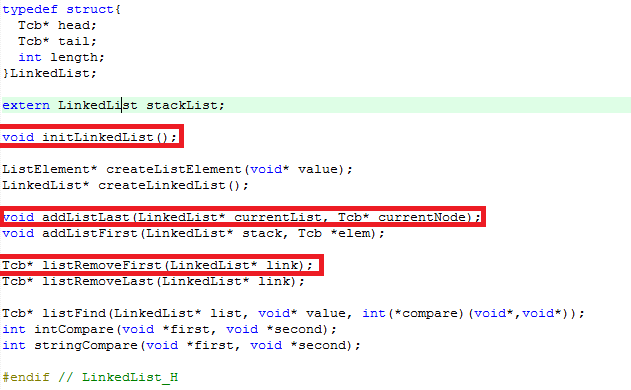


Figure 40 Header File of LinkedList, only the red boxed function will be used.

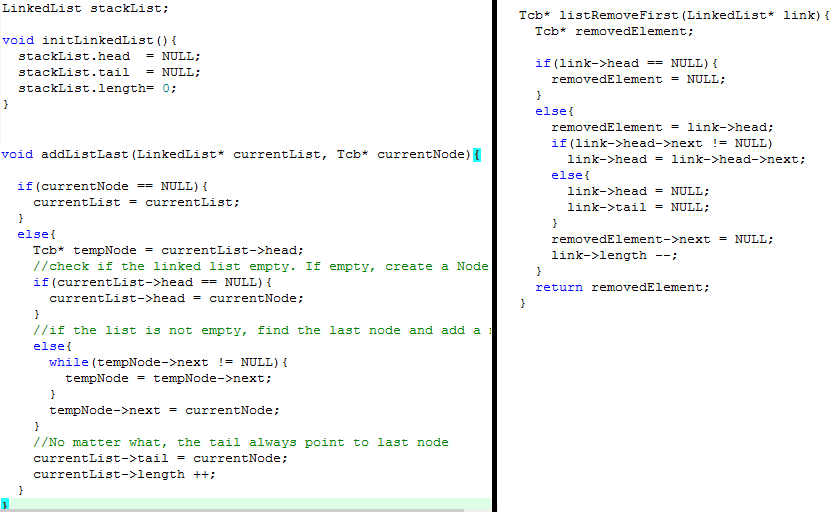


Figure 41 Source File of Linked List for the red boxed functions in Figure 38

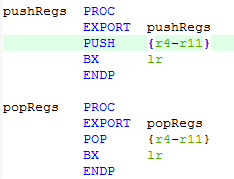


Figure 42 Extension program pop and push