ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 212E
MICROPROCESSOR SYSTEMS
TERM PROJECT

DATE	:	31.01.2021
GROUP NO	:	G38

GROUP MEMBERS:

150170716	:	Ali Osman Kılıç
150180034	:	Emine Darı
150180905	:	Fatima Rahimova
150170100	:	İrem Öztürk
150180043	:	Yusuf Alptigin Gün

FALL 2020

Contents

1 INTRODUCTION

In this project, we built a simple sorted linked list using assembly language. The main idea of the project was to use a SysTick Timer to be able to do one operation for each cycle of the timer. Since this was an assembly project; in addition to the normal linked list operations such as insert, remove, etc.; we built extra operations that were unique to assembly coding and Systick Timer. These operations included the transformation of the list to an array, SysTick Timer calculations, memory manipulations, memory allocations, and many more. All these basic and unique functionalities combined achieved us the sorted linked list idea on assembly language which was the goal of the project.

2 MATERIALS AND METHODS

- Arm Cortex M0+ processor was used in the making of this project with default configurations.
- A total of 13 functions were built in the making of this project. All of these functions are thoroughly explained one by one with the addition of code snippets throughout the explanation. Since the __main() function was already given, it is not explained.

2.1 SysTick_Handler()

The Systick_Handler() function is the main function of the program that does all the operations. Whenever an interrupt is called on the Systick Timer, this function activates. For each interrupt, we are able to do one operation read from the input data and do everything accordingly. Just to note that throughout the handler, some push operations will be done to save the values needed for the WriteErrorLog() function since the values loaded to registers might change after some operations. These values then will be popped just before branching to WriteErrorLog().

To begin with, we have to check if we have reached the end of the input data set and then depending on this, read the data input and do the instruction. To do this, we first get have many operations have been done previously by loading the TICK_COUNT variable to a register. After that, we multiply it by 4 to get have many words in the memory we have passed. After this, we get the start and end addresses to two different registers. We add the $TICK\_COUNT * 4$ value to the start address and compare it with the end address. This way, we can compare the two addresses and if they are equal; this would mean that we have reached the input data set so we can branch to the Systick_Stop() function which will stop the interrupts happening on the timer and end the simulation. Systick_Stop() is another function created for the project which will be explained in further discussion.

```
LDR     R5,=TICK_COUNT
LDR     R1,[R5]
PUSH    {R1}
LSLS    R1,R1,#2
LDR     R2,=IN_DATA
LDR     R6,=END_IN_DATA
ADDS    R7,R2,R1
CMP     R7,R6
BEQ     SysTick_Stop
```

If we are not at the end of the program, we first get load the data at the index to R0 (since R0 is used to pass data to other functions) and also the operation code given at the same index.

```
LDR     R0,[R2,R1]
PUSH    {R0}
LDR     R2,=IN_DATA_FLAG
LDR     R4,[R2,R1]
PUSH    {R4}
```

After this, we make comparison operations to check which operation we have to do. In total we do 3 comparison operations and depending on the outcomes, we may branch to different parts of the handler. We check each comparison one by one and if the operation value is equal to that operations predetermined values, we branch there. If the said operation is not found, we load R0 with the error code and branch to the write_error label which will be discussed in a bit. In each branch of the operation, we branch with the link register to the said operations function. After the operations is done, the link register traces back to the handler and then each operation branch, goes to write_error.

```
LDR     R0,[R2,R1]
PUSH    {R0}
LDR     R2,=IN_DATA_FLAG
PUSH    {R4}
CMP     R4,#0
BEQ     remove
CMP     R4,#1
BEQ     insert
CMP     R4,#2
BEQ     transform
MOVS    R0,#6
```

```
B               write_error
remove          BL Remove
B               write_error
insert          BL Insert
B               write_error
transform       BL LinkedList2Arr
B               write_error
```

In the write_error label, we compare R0 value with 0. Since if R0 value is not 0 after coming to the handler with the link register after any operation, it means an error has occurred and we have to write an error. After this comparison, we then pop the values discussed previously to get ready if an error occurs. Now if comparison set, this mean so error has occurred and we go to the increment_tick label. If the comparison is false, we branch to the WriteErrorLog() function with the link register to write the error to the memory.

```
write_error     CMP     R0,#0
                MOVS    R1,R0
                POP     {R2}
                POP     {R3}
                POP     {R0}
                BEQ     increment_tick
                BL      WriteErrorLog
```

In the increment_tick label, we basically just add 1 to both TICK_COUNT and INDEX_INPUT_DS variables by loading their address, getting the value at the address and incrementing it by 1 before loading the incremented value to the said addresses. This is done because after we think of each cycle on the handler as one interrupt done on the timer. This is also one operation from the input data set.

```
increment_tick  LDR     R5,=TICK_COUNT
                LDR     R1,[R5]
                ADDS    R1,R1,#1
                STR     R1,[R5]
                LDR     R1,=INDEX\_INPUT_DS
                LDR     R2,[R1]
                ADDS    R2,R2,#1
                STR     R2,[R1]
```

2.2 SysTick_Init()

SysTick_Init() is used to basically start the timer of the Systick Timer and the program. With this function, we initilaze the System Tick Timer registers and start the timer. We also set the program status to 1 since we start the timer. To do this, we first load the systick control and status register address to a register. We also load the reload value to another register. This reload value is calculated from the values given to us in the start of the project using the $Clockperiod * (Reloadvalue + 1) = Interruptperiod$. The value calculated was 17568 and this is what was loaded to the register. Then we go word by word to set values. We firstly store reload value to the reload value register. Then we clear the register to then clear the current value register. Then we set enable, clock and interrupt flags. Also at the end we load the program status address to a register and store the value 1 to that address since the timer has started.

```
LDR     R0,=0xE000E010
LDR     R1,=17567
STR     R1, [R0,#4]
MOVS    R1,#0
STR     R1, [R0,#8]
MOVS    R1,#7
STR     R1, [R0]
LDR     r0,=PROGRAM_STATUS
MOVS    r1,#1
STR     r1 , [ r0 ]
```

2.3 SysTick_Stop()

SysTick_Stop() is used to stop the timer and set the program status to 2 so the loop in main function of the program can stop which then would stop the program. We do this in a similar way in SysTick_Init() where we load the systick control and status register address to a register first. Then move a 0 value to another register then clear the interrupt, tickInt, clock flags along with count flag and update the program status. This done again by loading the address to a variable and then storing the value 2 to that loaded address.

```
LDR     R0,=0xE000E010
MOVS    R1,#0
STR     R1, [R0]
STR     R1, [R0,#8]
LDR     r0,=PROGRAM_STATUS
MOVS    r1,#2
```

```
        STR     r1 ,[ r0 ]
```

2.4 Clear_Alloc()

Clear_Alloc() is used to clear the whole space given for the allocation table since it will be changed in the future. To do this, we load the start address of the allocation table and the size of it to two registers. Then we add them to get the end address of the allocation table. Then we create a loop. In this loop, we first compare if the address is equal to to end address. If this is the case, we don't loop. If this is not the case, then we clear the address by storing a 0 to the address then add an offset to the address value to go to the next word. Then we branch back to the start of the loop and repeat this process until we reach the end where the whole allocation table is cleared.

```
        LDR     R4,=AT_MEM
        LDR     R5,=AT_SIZE
        ADDS    R5,R5,R4
loop    CMP     R4,R5
        BEQ     out
        MOVS    R6,#0
        STR     R6,[R4]
        ADDS    R4,R4,#4
        B       loop
out     BX      LR
```

2.5 Clear_ErrorLogs()

Clear_ErrorLogs() function basically does the same thing Clear_Alloc() function does as it clears the whole space given for the error log in the same manner Clear_Alloc() clears the space given for the allocation table. The same process is repeated where we load the error logs start address and the size of it to two registers. Then we add them and use the exact same loop which was explained in the Clear_Alloc() function to clear the whole space for the error logs.

```
        LDR     R4,=LOG_MEM
        LDR     R5,=LOG_ARRAY_SIZE
        ADDS    R5,R5,R4
logloop CMP     R4,R5
        BEQ     log_out
        MOVS    R6,#0
```

```
         STR     R6,[R4]
         ADDS    R4,R4,#4
         B       logloop
log_out  BX      LR
```

2.6 Init_GlobVars()

Init_GlobVars() initializes all the global variables we alter throughout the operations to zero. This is done by loading the variables address to a register, then moving a 0 value to another register and then storing the 0 value to that address. This operation is done 5 times for each global variable so we can initialize all of them.

```
LDR     r0,=TICK_COUNT
MOVS    r1,#0
STR     r1,[r0]
LDR     r0,=PROGRAM_STATUS
MOVS    r1,#0
STR     r1,[r0]
LDR     r0,=FIRST_ELEMENT
MOVS    r1,#0
STR     r1,[r0]
LDR     r0,=INDEX_INPUT_DS
MOVS    r1,#0
STR     r1,[r0]
LDR     r0,=INDEX_ERROR_LOG
MOVS    r1,#0
STR     r1,[r0]
BX      LR
```

2.7 Malloc()

As a note before starting, the code snippets doesn't have to be in the same order they are given in. The relevant branches/labels are given back to back to make the explanation easier. Malloc() function is used to allocate the first corresponding bit on the allocation table where a value can be written. To do this, we use a loop inside a loop. The outer loop searches the lines the of allocation table and the inner loop searches its bits. We first load the start address and the size of the allocation table to 2 registers. Then we add them to get the end address of the allocation table. After that we enter the loop. Since the first operation on the loop is an addition operation, we first use a subtraction so the start address doesn't change. Then we compare the address value to the end address. If we have reached the end of the allocation table without finding a 0 bit on the allocation table, this means that the list is full and we should print an error. When this happens, we branch to list_is_full label and set the error code to R0. Then we branch back to where malloc() was called with the link register. If the list is not full and we keep going through the loop, we continue with the inner loop. We move the value 0 to a register and compare it with 31 each iteration of the inner loop. If this value is grater than 33, that means we have reached the end of the line without finding a free space on the line so we branch back to the start of the outer loop. To check if we have a free space on the allocation table, we move 1 to a register and then use the ANDS operation between 1 and the word value the allocation table has on the address. If the ANDS result is 0, this means that there is at least one bit on the line where that bit is 0 so we branch to the malloc_out label to find that first bit and make it 1.

```
                LDR     R4,=AT_MEM
                LDR     R5,=AT_SIZE
                ADDS    R5,R5,R4
                SUBS    R4,R4,#4
line            ADDS    R4,R4,#4
                CMP     R4,R5
                BEQ     list_is_full
                MOVS    R7,#0
                LDR     R0,[R4]
                LDR     R3,[R4]
bit_loop        CMP     R7,#31
                BGT     line
                MOVS    R2, #1
                MOVS    R0,R3
                ANDS    R0,R0,R2
                CMP     R0,#0
                BEQ     malloc_out
                LSRS    R3,R3,#1
                ADDS    R7,R7,#1
                B       bit_loop
list_is_full    MOVS    R0,#1
                BX      LR
```

In the malloc_out label, we try and find the first 0 bit in the allocation table line and make it 1. To do this, we move 1 to a register and shift by the amount of times we have iterated through the inner loop. This gives us a value where the 1 bit is in the index where the first 0 bit is in the allocation table line. Then we load the line value to another register. We then add these to together to get the final allocation table line value. This makes the first 0 bit of the line 1 and we can store it back to the address. After that we how to find the address corresponding to this specific bit on the allocation table. We load the start address of the allocation table space and subtract it from the line address. This gives us how many lines we have passed on the allocation table. Then we divide this value by 4 and multiply it by 32. Then to add how many bits we have passed on the specific line to the value, we load how many times we have shifted to a register and add that value to the value we had found to find how many nodes of memory space we have already allocated. Then we multiply this value by 8 since each node of the linked list in the memory 8 bytes. Then we load the start address of the linked list space to a register and add it to the value multiplied by 8. We return this value in the register R0 and it is the exact address of the first free available space on the linked list space.

```
malloc_out      MOVS    R5,#1
                LSLS    R5,R5,R7
                LDR     R6,[R4]
                ADDS    R6,R6,R5
                STR     R6,[R4]
                LDR     R3,=AT_MEM
                SUBS    R4,R4,R3
                LSRS    R4,#2
                LSLS    R4,#5
                MOVS    R0,R7
                ADDS    R0,R0,R4
                LSLS    R0,#3
                LDR     R1,=DATA_MEM
                ADDS    R0,R1,R0
                BX      LR
```

2.8 Free(address)

Free() function clears the bit on the allocation table that corresponds to the index on the linked list space. This function takes the address of the linked list node to be deleted as a parameter on R0 so some operations will be done on that register. Firstly, we load the start address of the linked list to a register. Then we subtract the that address from the address of the node be deleted. We lastly divide the value by 8 so we can get the index of the node to be deleted. After that, we check at which line of the allocation table we have to deal with. Since each line of the allocation table is 32 bits, we compare the index with 32. if we find the value to be smaller than 32, that means we are dealing with the first line of the allocation table and we continue to the clear_bit label. If we don't find the value to be smaller than 32, we basically mod it with 32 using a loop and get which gives us the specific bit corresponding to the address to be deleted. Also how many times we have divided it by 32, gives us the line of the allocation table which the address to be deleted is in. We do this by subtracting 32 each from the index and adding 1 to a registers that we had moved a 0 value to initially each iteration of the loop. Whenever the index becomes smaller than 32 after the loop, we branch to clear_bit. At the clear_bit label, we first load the allocation table start address to a register. Than we multiply the value we found of how many 32's the index has by 4. This was, we get how many words in the allocation table has passed. Now to free the value, we load the allocation table word at the line we found, to a register. Then we move 1 to another register. We multiply this register by the index value we found which is smaller than 32. Then we subtract this value from the word value we get from the allocation table address which frees (writes 0) that specific bit of the allocation tabla that corresponds to the deleted address. Than we store the final value to the allocation table address than branch back to where this function was called from.

```
                    LDR     R1,=DATA_MEM
                    SUBS    R1,R0,R1
                    LSRS    R1,#3
                    MOVS    R2,#0
find_line           CMP     R1,#32
                    BLT     clear_bit
                    SUBS    R1,R1,#32
                    ADDS    R2,R2,#1
                    B       find_line
clear_bit           LDR     R3,=AT_MEM
                    LSLS    R2,R2,#2
                    ADDS    R2,R2,R3
```

```
                LDR     R4,[R2]
                MOVS    R5,#1
                LSLS    R5,R5,R1
                SUBS    R4,R4,R5
                STR     R4,[R2]
                BX      LR
```

2.9 Insert(value)

As a note before starting, the code snippets doesn't have to be in the same order they are given in. The relevant branches/labels are given back to back to make the explanation easier. Insert() function is used to insert a value to the linked list in the memory. There are a lot cases and all of them including the error cases are discussed one by one. To start with, We load the FIRST_ELEMENT variable to a register and then load its value. We compare this value with 0 and if we find this to be true, then we go to the first_insert label. Here, we push the necessary registers that might change in the malloc() function and then call the malloc() function. This way, we get the address of the first free allocation space on the linked list to do the insertion on. Next, we compare R0 with 1. The reason for doing this is to spot if we there are is no allocatable space on the allocation table. Since we have already designed our malloc() function to return error code 1, we can just compare the R0 value with 1 and if they are equal, we will branch to the check_error label. In this label, we will branch back to the handler and write the error. If R0 is not 1, it will have returned an address for the free space on the linked list and we can continue the insertion. After getting the address, we load the FIRST_ELEMENT variable address to a register. Then since we are inserting the first element of the linked list, we store the found address from the malloc() to the address of the FIRST_ELEMENT variable address. The we store the value to be inserted to the found address from the malloc() and store its next pointer as by storing 0 at the next word of the found address from the malloc() which is the next pointer. Just as another note, stop label snippet will be used on other successful operations as well but its snippet will not be shown. Whenever a "stop" label is discussed from this point on, it means the stop label shown below. This is also the same with the check_error label.

```
                LDR     R1,=FIRST_ELEMENT
                LDR     R1,[R1]
                CMP     R1,#0
                BEQ     first_insert
first_insert    PUSH    {R0}
```

```
                BL          Malloc
                LDR         R6,=DATA_MEM
                POP         {R4}
                CMP         R0,#1
                BEQ         check_error
                STR         R4,[R0]
                LDR         R2,=FIRST_ELEMENT
                STR         R0,[R2]
                MOVS        R3,#0
                STR         R3,[R0,#4]
                B           stop
stop            MOVS        R0,#0
                POP         {PC}
check_error     POP         {PC}
```

If we do not find the first element of the list to be 0, that means we have already inserted something to the list. With this idea, we now check if the inserted element had already been inserted or not. If we are trying to insert an element that already exists, we have to give an error. We do this by going through the whole linked list with a loop and checking each value if they are equal to the data trying to be inserted. We already had the address of the first element from the FIRST_ELEMENT variable. We load the value from this address to a register and compare it with the data trying to be inserted. If we find this comparison to be true; that means we are trying to insert a data that already exists so we branch to the duplicate label, assign the error code to R0 and branch back to the handler to write the error. If we the comparison is false, we go to the next word of the address and get the value in it. This gives us the address of the next nodes value'. We compare this value with 0, and if this comparison is true, that means we have reached the end of the list without finding a duplicate value and we can branch to the can_insert label to do the insertion operation. Else, we move the next nodes value' address to a register and branch back to the start of the loop to go through the whole linked list.

```
traverse_list   LDR         R2,[R1]
                CMP         R2,R0
                BEQ         duplicate
                ADDS        R1,R1,#4
                LDR         R3,[R1]
                CMP         R3,#0
                BEQ         can_insert
                MOVS        R1,R3
                B       traverse_list
duplicate       MOVS    R0,#2
                POP     {PC}
```

If we have reached the end of the linked list without finding a duplicate value, we come to the can_insert label. We here get the FIRST_ELEMENT variables value to get the head address of the linked list. Then we get the value at the head address to start comparing to find out where we should insert the new data. We do one comparison to find out if we should add the new data as head of the linked list. If the data is smaller than the already existing head data, we branch to insert_head label to insert the head. In that label, we push the register values we need that might change in the malloc() function and call the malloc() function. We get the address to write the node in the memory in R0 and pop the registers we had pushed previously. We then do the error check operation discussed in the previous insertion case to see if there is an error. If there is no error, we store the data to be stored at the address that came from malloc(). We also store the newly inserted head nodes next pointer as the previous heads value address. Then we store the FIRST_ELEMENT variables as the new heads value address since we are inserting to the start of the linked list.

```
can_insert      LDR     R1,=FIRST_ELEMENT
                LDR     R1,[R1]
                LDR     R2,[R1]
                CMP     R0,R2
                BLT     insert_head
insert_head     PUSH    {R0}
                PUSH    {R5}
                BL      Malloc
                POP     {R5}
                POP     {R4}
                CMP     R0,#1
                BEQ     check_error
                STR     R4,[R0]
                LDR     R5,=FIRST_ELEMENT
                MOVS    R7,R5
                LDR     R5,[R5]
                STR     R5,[R0,#4]
                STR     R0,[R7]
                B       stop
```

If in the can_insert label, we find the data to be stored to be bigger than the head nodes value, then we search through the list to find where we should insert that data (not as in an array fashion but regarding the next pointers). Since we know that if we are here in this search loop, the value to be stored is bigger than some nodes value. We just have to compare it with the next nodes value to see if we are going to keep searching. We do this by loading the next word address value (which is the address of the next node) to a register and comparing it with 0. If the comparison is true, that means we have reached the end of list and we branch to the insert_to_end label. If this is not true meaning the we are not at the end of the list, we continue and check if the data to be inserted is smaller than the next nodes value. If this is true, we branch to the insert_between label to insert a node somewhere between two nodes in the list. If all of these are false, we just repeat the same cycle for the next node until we find the correct place for the node to be stored. Both insert_to_end and insert_between labels work in the same way. We both push the registers that might be used before the malloc() function to pop them later. Then we call the malloc() function to get address of where the node should be on the register R0. Then we do our usual error check to the if an error has returned from R0. If not, the store the data to be stored on the address that came from malloc() and store its next pointer (next word of the address that came from malloc()) as the node value address of which node the data to be stored was smaller from. If we are at the end of the list, since this next pointer will be showing 0 and we want the last elements next pointer to show 0; the same process doesn't have any negative effects. We also assign the next pointer value of the earlier node as the address value that came from malloc().

```
search          ADDS    R1,R1,#4
                LDR     R5,[R1]
                CMP     R5,#0
                BEQ     insert_to_end
                LDR     R6,[R5]
                CMP     R0,R6
                BLT     insert_between
                MOVS    R1,R5
                B       search
insert_between  PUSH    {R0}
                PUSH    {R1}
                PUSH    {R5}
                BL      Malloc
                POP     {R5}
                POP     {R1}
                POP     {R4}
                CMP     R0,#1
                BEQ     check_error
                STR     R4,[R0]
                STR     R0,[R1]
                STR     R5,[R0,#4]
                B       stop
insert_to_end   PUSH    {R0}
                PUSH    {R1}
                PUSH    {R6}
                BL      Malloc
                POP     {R6}
                POP     {R1}
                POP     {R4}
                CMP     R0,#1
                BEQ     check_error
                STR     R0,[R1]
                STR     R4,[R0]
                MOVS    R6,#0
                STR     R6,[R0,#4]
                B       stop
```

2.10 Remove(value)

As a note before starting, the code snippets doesn't have to be in the same order they are given in. The relevant branches are given back to back to make the explanation easier. Remove() function is used to delete a value in the linked list. There are a lot cases and all of them including the error cases are discussed one by one. The first thing we check on the delete operation is an error. We load FIRST_ELEMENT variable address and its value to a register and compare it with 0. If this comparison is correct, it means that the head of the list is null (0) meaning the list is empty. We then branch to the empty_list label. There we assign the error code to the R0 register and go back to the handler to write the error.

```
                PUSH    {LR}
                LDR     R1,=FIRST_ELEMENT
                LDR     R1,[R1]
                CMP     R1,#0
                BEQ     empty_list
empty_list      MOVS    R0,#3
                POP     {PC}
```

If the first element of the list is not 0, that means we have already added some values to the linked list. Another error we have to check here is if the value we are trying to delete exists or not. To check this, we traverse the already existing linked list. This process is exactly the same process done on the insertion method so not going into to much detail, we basically compare the value of the linked list to the value to be deleted. Then if the comparison is correct, we branch to delete label to delete that value. If comparison is not correct, we keep moving by getting the next pointers value and going to the start of the loop to check pointers value. We also check if the next pointers address is 0 in each iteration. If this is correct, this means that we have reached the end of the linked list and have not found the value to be deleted. If this is the case, we branch to the not_found label. There we assign the error code for the error to R0 and pop back to the handler to write the error.

```
traverse2       LDR     R2,[R1]
                CMP     R2,R0
                BEQ     delete
                ADDS    R1,R1,#4
                LDR     R3,[R1]
                CMP     R3,#0
                BEQ     not_found
                MOVS    R1,R3
                B       traverse2
not_found       MOVS    R0,#4
                POP     {PC}
```

If we have found the data to be deleted on the linked list, we are at the delete label. In this label, we have to again traverse list again to get to where the address of node with the data to be deleted. But first, we check another case. We get the FIRST_ELEMENT variable value to load the address of the head node. Then we load the head value of the linked list and compare it with the data to be deleted. If they are equal, that means this is a special case where we delete the head so we branch to the delete_head label. There we also compare the next pointer of the head node with 0. Because if this comparison is true, that means that there are no other elements in the list and that we have to assign the FIRST_ELEMENT accordingly as 0. If this comparison is false, we keep continuing with the normal delete operation. Both these operations work in a similar fashion where the next pointer value of the head to be deleted is assigned FIRST_ELEMENT value and the deleted nodes both value and next pointers are assigned as 0 so they are cleared. We also call the free() function with the already assigned address to R0 to clear the allocation table bit corresponding to the linked list memory address. Then since these operations will be successful, we go to the finish label and assign RO as the success operation value. Then we pop back to the handler.

```
delete          LDR     R1,=FIRST_ELEMENT
                LDR     R1,[R1]
                LDR     R2,[R1]
                CMP     R2,R0
                BEQ     delete_head
delete_head     LDR     R4,[R1,#4]
                MOVS    R0,R1
                CMP     R4,#0
                BEQ     clear_list
                MOVS    R5,#0
                STR     R5,[R1]
                ADDS    R1,R1,#4
                LDR     R2,[R1]
                STR     R5,[R1]
                LDR     R6,=FIRST_ELEMENT
                STR     R2,[R6]
                BL      Free
                B       finish
clear_list      MOVS    R5,#0
                STR     R5,[R1]
                LDR     R1,=FIRST_ELEMENT
                STR     R5,[R1]
                BL      Free
                B       finish
finish          MOVS    R0,#0
                POP     {PC}
```

If the value to be deleted is not the head node, then we do not go to the delete_head label but rather traverse through the list to find the address of the value to be deleted. We use a simple traverse method we have used many times to go through each node and in every iteration, we compare the nodes value to the value to be deleted. If they are equal, we branch to the delete_node label to delete the node. There, to delete the node, we firstly assign the previous nodes next pointer value as the to be deleted nodes next pointer value so the linked list connection stays in tact. After that, we clear the node to be deleted by clearing (storing 0) to the value and next pointer addresses of the node. After that, we move the node to be deleted' address to R0 and call the free function to free the corresponding bit on the allocation table for that address. Then we branch to the finish label to move 0 to R0 as the operation was a success and then to pop back to the handler.

```
search2 ADDS     R1,R1,#4
                 LDR       R5,[R1]
                 LDR       R6,[R5]
                 CMP       R6,R0
                 BEQ       delete_node
                 MOVS      R1,R5
                 B         search2
delete_node      LDR       R6,[R5,#4]
                 STR       R6,[R1]
                 MOVS      R7,#0
                 STR       R7,[R5]
                 STR       R7,[R5,#4]
                 MOVS      R0,R5
                 BL        Free
                 B         finish
finish           MOVS      R0,#0
                 POP       {PC}
```

2.11 LinkedList2Arr()

LinkedList2Arr() function is used to turn the linked list in the memory to an array an write it into the memory. Firstly before doing this, we load the FIRST_ELEMENT variable address to a register along with the start address and array size to two other registers. Then we add the start address and the size to get the end address of the array space. After this, we clear the array. We do this by using a basic loop. We start by comparing the address to the end address of the space and if they are equal, we then go to the continue label. If they are not equal, we move the clear value 0 to a register and store that value to the address. Then we increment the address by 4 bytes to go to the next word and go to the start of the loop. Doing this until we reach the end of the array space, we clear the whole space.

```
                LDR     R1,=FIRST_ELEMENT
                LDR     R4,=ARRAY_MEM
                LDR     R6,=ARRAY_SIZE
                ADDS    R6,R4,R6
clear_array     CMP     R4,R6
                BEQ     continue
                MOVS    R5,#0
                STR     R5,[R4]
                ADDS    R4,R4,#4
                B       clear_array
```

In the continue label, we check if the array is empty or not. We do this by getting the FIRST_ELEMENT variable value at the address. Then we compare it to 0. If it is 0 (meaning head of the linked list is null), we go to the empty_error then set the error code to R0 and return back to the handler. We also reset the array start address since we have traversed through it. As a reminder that the branch/label snippets are shown for clear demonstration. The branch/label position may not be in the order as shown below.

```
continue        LDR     R4,=ARRAY_MEM
                LDR     R1,[R1]
                CMP     R1,#0
                BEQ     empty_error
empty_error     MOVS    R0,#5
                BX      LR
```

If we manage to get through the continue label without jumping to the empty_error label, then we can write the linked list as array in the memory. This is done one by one traversing through the linked list. Since we already have the address of the head of the linked list in a register (this operation was done on the continue label to check if list was empty), we can load the value at that address and store it to the array address. Then we go to the next word at the linked list by incrementing the address by 4 bytes. Then we load the value at that address to a register. This way, we have the next elements value address. If this address is 0, this means that we have reached the end of the linked list so we branch to the end_of_list label, set R0 as 0 since the operation was a success and branch back to the handler. If we have not reached the end of the linked list, we move that address next elements value address to the R1 register so we can do the same things we did again on the next element of the linked list. We also increment the address at the array space by a word so we can write the new value to there. Then we go back to the start of the loop to go through the whole linked list. Again, as a reminder that the branch/label snippets are shown for clear demonstration. The branch/label positions may not be in the order as shown below.

```
traverse3       LDR     R2,[R1]
                STR     R2,[R4]
                LDR     R2,[R1,#4]
                CMP     R2,#0
                BEQ     end_of_list
                MOVS    R1,R2
                ADDS    R4,R4,#4
                B       traverse3
end_of_list     MOVS    R0,#0
                BX      LR
```

2.12 WriteErrorLog(Index, ErrorCode, Operation, Data)

WriteErrorLog() function is used to write an error with the given parameters to the memory. Since all the variables are already assigned before the calling of the function, we only have to make operations on those variables instead of manipulating them. In this function, we first have to go to the address where we are going to write the error to. To do this, we load the start address of the error log and the INDEX_ERROR_LOG variable address to registers. By doing this, we can get the INDEX_ERROR_LOG value from it's address, multiply it by 12 and add it to the start address of the error log to find the address to write the error to. This is doable because each error is as long as 12 bytes any by multiplying with INDEX_ERROR_LOG value, we can add the total memory already used for error logs to the starts address. After finding the address to write the error to, we just start writing the already set variables. We use STRH for index of the input (since it is 16 bits), STRB for error code and operation (since 8 bits) and normal STR for data and timestamp (since 32 bits). To get the timestamp, we call the getNow() function before the actual store operation. We also Push the link register so we can trace back to the function after getNow() is finished. Then lastly we increment the INDEX_ERROR_LOG variable by 1 since we have written an error to the memory. This done by the usual way of loading the address, then loading the value at address, incrementing it by 1 and storing it to the oroginal address.

```
        LDR     R4,=LOG_MEM
        LDR     R6,=INDEX_ERROR_LOG
```

```
LDR     R6, [R6]
MOVS    R7,#12
MULS    R6, R7, R6
ADDS    R4, R4, R6
STRH    R0, [R4]
STRB    R1, [R4,#2]
STRB    R2, [R4,#3]
STR     R3, [R4,#4]
PUSH    {LR}
BL      GetNow
STR     R0, [R4,#8]
LDR     R6,=INDEX_ERROR_LOG
LDR     R7, [R6]
ADDS    R7, R7,#1
STR     R7, [R6]
POP     {PC}
```

2.13 GetNow()

GetNow() function is used to return how much time has passed on the SysTick Timer. This function is called in the WriteErrorLog() while writing an error. We calculate the working time of the SysTick Timer with and return it on the R0 register. To calculate, we first get the TICK_COUNT and SysTick Timer current value register addresses. Then we load the values at those addresses to registers. We then also load the reload value and the timer interrupt value to another 2 registers. After loading these values into registers, we basically do the calculation $PERIOD*(TICK\_COUNT+1)+(reload-currentvalue)/32$ and return it in the register R0. Here the $PERIOD * (TICK\_COUNT + 1)$ finds how much time on previous interrupts has passed on the SysTick Timer. The +1 is because before even starting the operations, one interrupt passes to get to the handler. $(reload - currentvalue)/32$ is to find how much time has passed since the last interrupt of the SysTick Timer where 32 is the CPU clock frequeny given to us in the project.

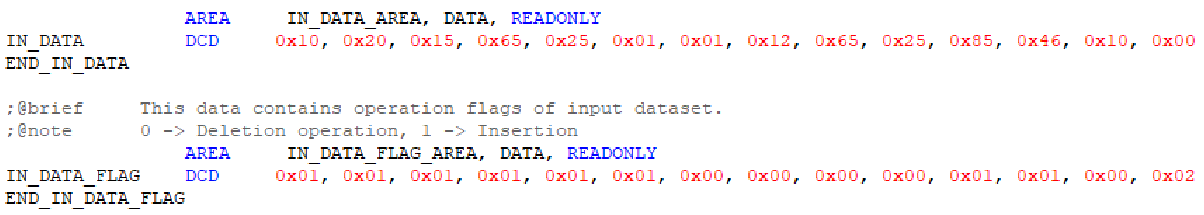
```
LDR     R0,=0xE000E018
LDR     R0,  [R0]
LDR     R1,=TICK_COUNT
LDR     R1,  [R1]
ADDS    R1, R1,#1
LDR     R2,=17567
```

```
SUBS      R0,  R2,  R0
LDR       R6,=549
MULS      R1,R6,R1
LSRS      R0,#5
ADDS      R0,R0,R1
BX        LR
```

3 RESULTS

The project was tested with many different data sets to cover all possibilities that we thought may arise for the testing part. Since there were many cases, not all the test are shown but 2 of them are given as examples. The first input data set is the default input data set from project template and the second one is an input data set we have created. There also is another test shown for the *getNow*() function.

The default input data set and the listed operations for it is shown below.



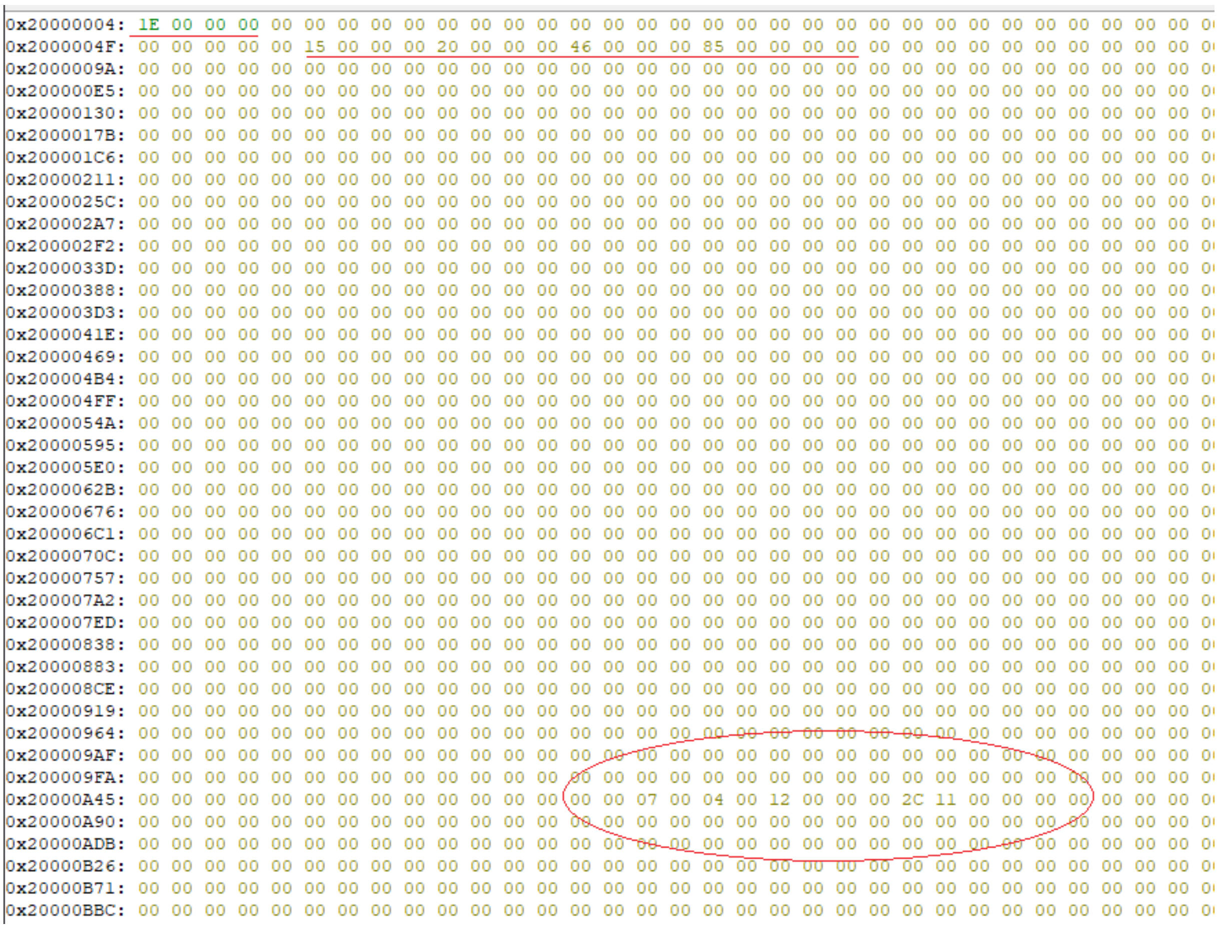
```
                AREA     IN_DATA_AREA, DATA, READONLY
IN_DATA         DCD      0x10, 0x20, 0x15, 0x65, 0x25, 0x01, 0x01, 0x12, 0x65, 0x25, 0x85, 0x46, 0x10, 0x00
END_IN_DATA

;@brief     This data contains operation flags of input dataset.
;@note      0 -> Deletion operation, 1 -> Insertion
                AREA     IN_DATA_FLAG_AREA, DATA, READONLY
IN_DATA_FLAG    DCD      0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x00, 0x00, 0x00, 0x00, 0x01, 0x01, 0x00, 0x02
END_IN_DATA_FLAG
```

Figure 1: Default Input Data Set for Test 1

1. Operation:Insertion / Data:0x10
2. Operation:Insertion / Data:0x20
3. Operation:Insertion / Data:0x15
4. Operation:Insertion / Data:0x65
5. Operation:Insertion / Data:0x25
6. Operation:Insertion / Data:0x01
7. Operation:Deletion / Data:0x01
8. Operation:Deletion / Data:0x12
9. Operation:Deletion / Data:0x65
10. Operation:Deletion / Data:0x25

11. Operation:Insertion / Data:0x85
12. Operation:Insertion / Data:0x46
13. Operation:Deletion / Data:0x10
14. Operation:Linked List to Array / Data:0x00

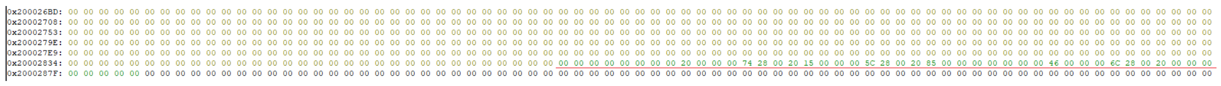
There are a total of 14 operations done for the default input data set. The only error that should be appearing here would be the operation with index 7(eight operation) which would be the error on deleting a value that doesn't exist. Other than that, every operations should run smoothly and the remaining values 15, 46 and 85 should be turned into an array and printed to memory. We let our code run until the end of the program and here our the results we got.



```
0x20000004: 1E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x2000004F: 00 00 00 00 00 15 00 00 00 20 00 00 00 46 00 00 00 85 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x2000009A: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x200000E5: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x20000130: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x2000017B: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x200001C6: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x20000211: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x2000025C: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x200002A7: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x200002F2: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x2000033D: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x20000388: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x200003D3: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x2000041E: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x20000469: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x200004B4: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x200004FF: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x2000054A: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x20000595: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x200005E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x2000062B: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x20000676: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x200006C1: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x2000070C: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x20000757: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x200007A2: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x200007ED: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x20000838: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x20000883: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x200008CE: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x20000919: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x20000964: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x200009AF: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x200009FA: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x20000A45: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 07 00 04 00 12 00 00 00 2C 11 00 00 00 00 00 00 00 00
0x20000A90: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x20000ADB: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x20000B26: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x20000B71: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x20000BBC: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 2: Allocation Table, Array and Error Log

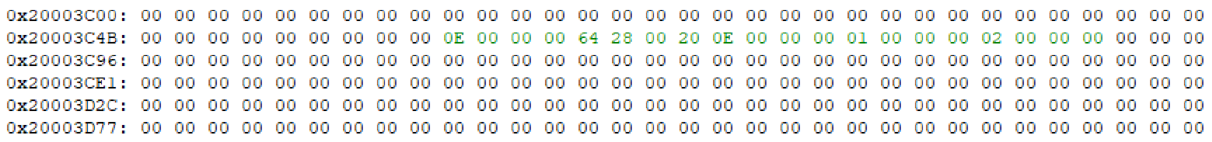
On the operation set; there were 6 back to back insertions and 3 deletions that wouldn't result in an error. These deletions were also done on the lastly added three elements. Then there were 2 more insertions than a deletions of the first added element. This would make the allocation table have 4 zeroes starting from the second LSB since the first element inserted was deleted. This would give us 0001 1110 which is 1E and is in compliance with what we got. Also the array part can be seen in the memory as well with 15, 46 and 85 printed consecutively. The error log is also shown on the bottom with one error on the index 7 for data 12 which also complies.



```
0x200026BD: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x20002708: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x20002753: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x2000279E: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x200027E9: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x20002834: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 00 74 28 00 20 15 00 00 00 5C 28 00 20 85 00 00 00 00 00 00 00 46 00 00 00 6C 28 00 20 00 00 00
0x2000287F: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 3: Linked List After All Operations

The linked list after all the operations are done can be seen here. The linked list next pointers and the data comply with what was expected at the start of the test.

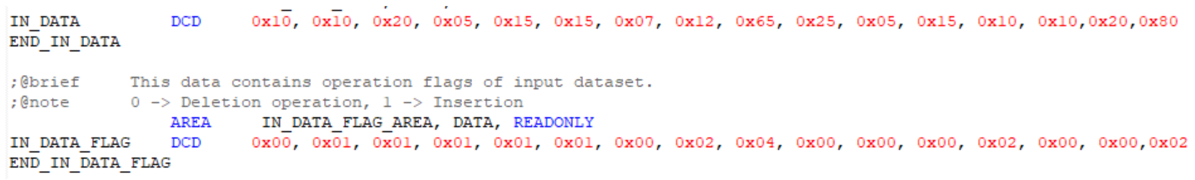


```
0x20003C00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x20003C4B: 00 00 00 00 00 00 00 00 00 0E 00 00 00 64 28 00 20 0E 00 00 00 01 00 00 00 02 00 00 00 00 00 00 00 00 00 00 00
0x20003C96: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x20003CE1: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x20003D2C: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x20003D77: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 4: Global Variables After All Operations

All global variables can be seen here. Program status can be seen as 2 as the program has stopped. Both INDEX_INPUT_DS and TICK_COUNT variables are showing the last operation index values and the INDEX_ERROR_LOG is 1 since there is only one error so it has passed index 0. Also the FIRST_ELEMENT value complies with where the first element of the linked list is stored. This can also be checked by looking at the previous picture.

The input data set we created can and the listed operations for it can be seen below.



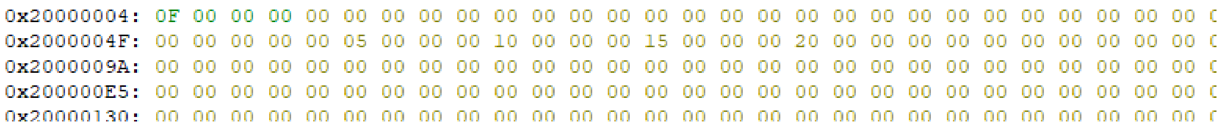
```
IN_DATA         DCD     0x10, 0x10, 0x20, 0x05, 0x15, 0x15, 0x07, 0x12, 0x65, 0x25, 0x05, 0x15, 0x10, 0x10,0x20,0x80
END_IN_DATA

;@brief     This data contains operation flags of input dataset.
;@note      0 -> Deletion operation, 1 -> Insertion
                AREA     IN_DATA_FLAG_AREA, DATA, READONLY
IN_DATA_FLAG    DCD     0x00, 0x01, 0x01, 0x01, 0x01, 0x01, 0x00, 0x02, 0x04, 0x00, 0x00, 0x00, 0x02, 0x00, 0x00,0x02
END_IN_DATA_FLAG
```

Figure 5: Allocation Table, Array and Error Log

1. Operation:Deletion / Data:0x10
2. Operation:Insertion / Data:0x10
3. Operation:Insertion / Data:0x20
4. Operation:Insertion / Data:0x05
5. Operation:Insertion / Data:0x15

6. Operation:Insertion / Data:0x15
7. Operation:Deletion / Data:0x07
8. Operation:Linked List to Array / Data:0x12
9. Operation:Wrong Operation / Data:0x65
10. Operation:Deletion / Data:0x25
11. Operation:Deletion / Data:0x05
12. Operation:Deletion / Data:0x15
13. Operation:Linked List to Array / Data:0x10
14. Operation:Deletion / Data:0x10
15. Operation:Deletion / Data:0x20
16. Operation:Linked List to Array / Data:0x80

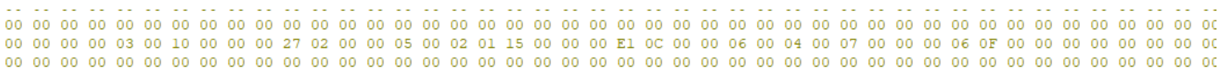
We put 3 linked list to array functions in the input data set and thought of them as checkpoints of the program. We took the three type of photos for the first input data set for each of the checkpoints to clearly show the output of the program. The only error that was not checked through this data set was the "linked list is full" error. This is because this required a lot of operations and the minimization of the allocation table which we couldn't do at the same time with this data set. That error was also tested separately but is not shown on this data set. Until the first checkpoint, we expect to see an error on the first operation since the list would be empty on the deletion operation. Then there would be 4 successful insertion operations followed by 2 errors. There are inserting the same value and deleting a non-existing value errors consecutively. Then we expect the list to turn into an array and be printed on the memory.



```
0x20000004: 0F 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x2000004F: 00 00 00 00 00 05 00 00 00 10 00 00 00 15 00 00 00 20 00 00 00 00 00 00 00 00 00 00
0x2000009A: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x200000E5: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x20000130: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 6: Allocation Table and Array For Checkpoint 1

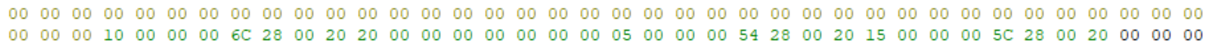
We can see that the allocation table is 0F. This is correct since we expect it to be 0000 1111 after 4 successful insertion operations. Also we can see that the linked list is printed as array for the 4 successful insertion operations as 5, 10, 15, 20.



```
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
00 00 00 00 03 00 10 00 00 00 27 02 00 00 05 00 02 01 15 00 00 00 E1 0C 00 00 06 00 04 00 07 00 00 00 06 0F 00 00 00 00 00 00 00 0
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
```

Figure 7: Error Log For Checkpoint 1

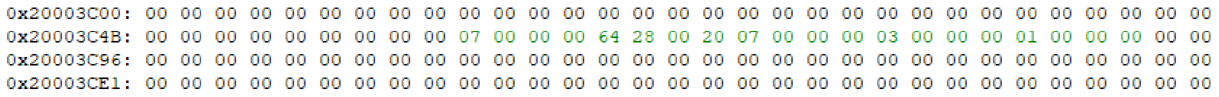
We can see that 3 errors have been encountered like expected. The error indexes can be found as 0, 5 and 6 and the other values can be checked for each error to see that they all comply with the expected error log outputs.



```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 10 00 00 00 6C 28 00 20 20 00 00 00 00 00 00 00 05 00 00 00 54 28 00 20 15 00 00 00 5C 28 00 20 00 00 00
```

Figure 8: Linked List For Checkpoint 1

Linked list for first checkpoint can be seen here. All 4 inserted values and their next pointer addresses can be seen to comply with what we expect.

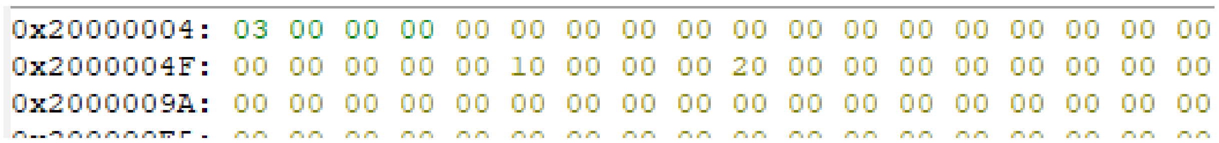


```
0x20003C00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x20003C4B: 00 00 00 00 00 00 00 00 00 07 00 00 00 64 28 00 20 07 00 00 00 03 00 00 00 01 00 00 00 00 00
0x20003C96: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x20003CE1: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 9: Global Variables For Checkpoint 1

All global variables for checkpoint 1 can be seen here. Program status can be seen as 1 since the program has not yet stopped and the timer is running. Both INDEX_INPUT_DS and TICK_COUNT variables are showing the last operation index values and the INDEX_ERROR_LOG is 3 since there are 3 errors we have passed. Also the FIRST_ELEMENT value complies with where the first element of the linked list is stored.

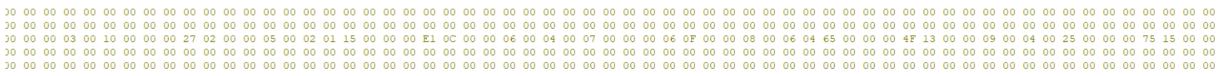
Now for the second checkpoint, we firstly expect an error for a wrong operation input followed by another deletion error where we try to delete a non-existing value. Then we make 2 successful deletion operations from the end of the allocation table and print the linked list as array.



```
0x20000004: 03 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x2000004F: 00 00 00 00 00 10 00 00 00 20 00 00 00 00 00 00 00 00
0x2000009A: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 10: Allocation Table and Array For Checkpoint 2

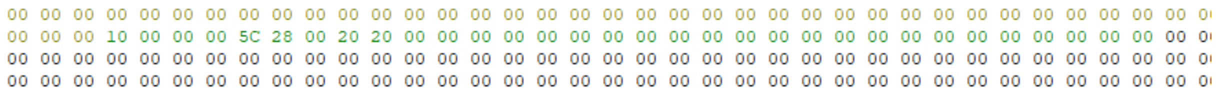
We can see that the allocation table is 03. This is expected since we make the deletion operations from the end of the allocation table so it is something like 0000 0011 which is 03. Also we can see that the linked list is printed as array for the remaining two values on the linked list, 10 and 20.



```
0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0 00 00 03 00 10 00 00 00 27 02 00 00 05 00 02 01 15 00 00 00 E1 0C 00 00 06 00 04 00 07 00 00 00 06 0F 00 00 08 00 06 04 65 00 00 00 4F 13 00 00 09 00 04 00 25 00 00 00 75 15 00 00
0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 11: Error Log For Checkpoint 2

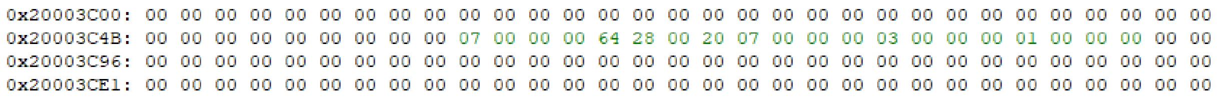
We can see the error log for the second checkpoint here. Two more errors have been added as expected for the indexes 8 and 9. Their other values also comply with the errors we expected.



```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
00 00 00 10 00 00 00 5C 28 00 20 20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
```

Figure 12: Linked List For Checkpoint 2

Linked list for checkpoint 2 can be seen here. The remaining two values are still in the list while the other values have been turned to null(0 in memory).

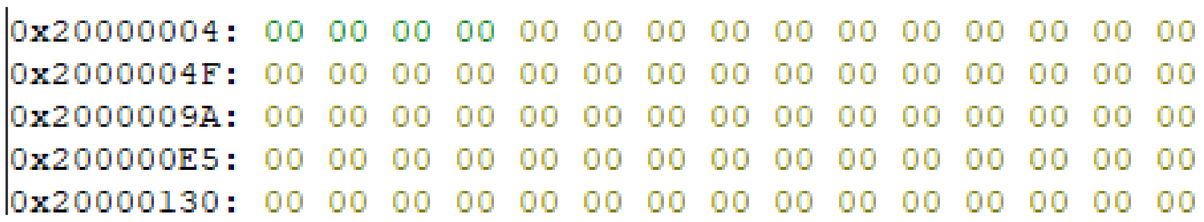


```
0x20003C00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x20003C4B: 00 00 00 00 00 00 00 00 00 07 00 00 00 64 28 00 20 07 00 00 00 03 00 00 00 01 00 00 00 00 00
0x20003C96: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x20003CE1: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 13: Global Variables For Checkpoint 2

All global variables for checkpoint 2 can be seen here. Program status is still 1. Both INDEX_INPUT_DS and TICK_COUNT variables are showing the last operation index values and the INDEX_ERROR_LOG is 5 since we have added 2 more errors. Also the FIRST_ELEMENT value can be seen to be pointing to the first data address of the linked list.

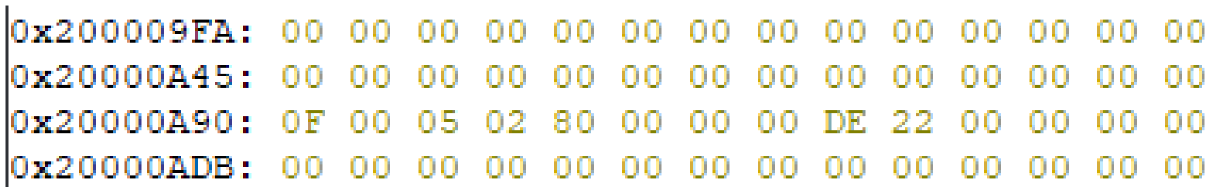
Now for the last checkpoint, we expect two more successful deletion operations followed by an error on the linked list to array function since we will have cleared the list fully.



```
0x20000004: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x2000004F: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x2000009A: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x200000E5: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x20000130: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 14: Allocation Table and Array For Checkpoint 3

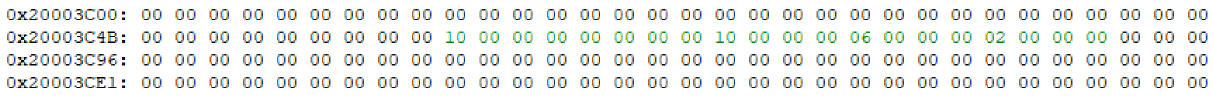
We can see that the allocation table has been cleared fully and the linked list has also been cleared but nothing was written since the list was empty and an error occurred.



```
0x200009FA: 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x20000A45: 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x20000A90: 0F 00 05 02 80 00 00 00 DE 22 00 00 00 00
0x20000ADB: 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 15: Error Log For Checkpoint 3

We can see that 1 more error has been added. Note that only the added error is shown. But it can be seen that it is in the next line of the error log for seen in checkpoint 2. This was done for the sake of image simplicity.



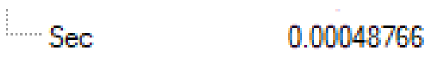
```
0x20003C00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x20003C4B: 00 00 00 00 00 00 00 00 00 10 00 00 00 00 00 00 00 10 00 00 00 06 00 00 00 02 00 00 00 00 00 00
0x20003C96: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x20003CE1: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 16: Global Variables For Checkpoint 3

All global variables for checkpoint 3 can be seen here. Program status can be seen as 2 since the program has ended. Both INDEX_INPUT_DS and TICK_COUNT variables are showing the last operation index values and the INDEX_ERROR_LOG is 6 since we have encountered a total of 6 errors throughout the whole program. Also the FIRST_ELEMENT value is 0 since the whole list is deleted.

The last thing we did for test was to test the *getNow*() function. We took the time passed until the timer started and gave an error on the second input operation. Then we took the time *getNow*() gave us for the error and compared the time passed from the start of the timer to the value we got.

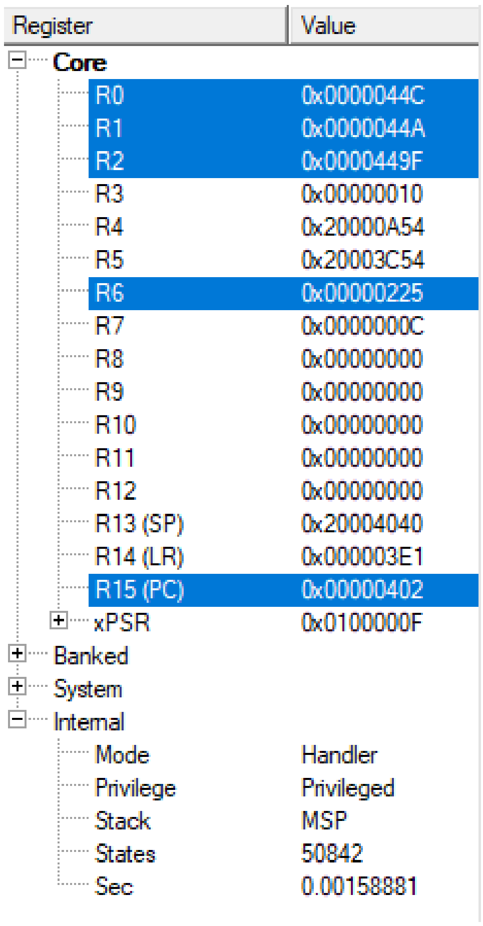
The total time passed before the timer stars can be seen below. This time is passed for the cleat operation at the beginning of the main loop.



```
Sec                 0.00048766
```

Figure 17: Time Passed Before Timer Starts

The time *getNow*() function gives us can be seen on the R0 register. Also the total time passed from the start of the program including the clear operations in the main function is shown as second in "Internal".



Register	Value
Core	
R0	0x0000044C
R1	0x0000044A
R2	0x0000449F
R3	0x00000010
R4	0x20000A54
R5	0x20003C54
R6	0x00000225
R7	0x0000000C
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20004040
R14 (LR)	0x000003E1
R15 (PC)	0x00000402
xPSR	0x0100000F
Banked	
System	
Internal	
Mode	Handler
Privilege	Privileged
Stack	MSP
States	50842
Sec	0.00158881

Figure 18: Total Time and Time Returned From $getNow()$

The value shown on R0 is 44C which is equal to 1100 in decimals. Since this is in micro second, changing it to seconds gives us 0,0011. This is the total time passed on the timer. By adding this to the total time passed on the initial clear operations, we should be getting the total time. $0,0011 + 0,00048766 = 0,00158766 \approx 0,00158881$ so we can say the calculation is correct.

4 DISCUSSION

Starting the project was probably the most challenging part for us to do. When a project with a lot of functions and a lot of cases to be handled is presented, it generally is not easy to understand what you are supposed to do at first to put all the things together. We first tried to get a feel of what we were supposed to do to integrate everything together and what we would have in the end result. This proved to be very useful and actually pretty confusing at the same time since throughout the project; there were times where we were doing stuff we had already thought about but there were also times were we had to reconsider a lot of the things we had discussed. This happened because of various reasons like not understanding what was needed properly, or finding out a better solution to what we had discussed etc. Overall the project was a combination of we knowing what to do generally and adding or changing stuff along the way. The first thing we wanted to do was write some of the basic function that we can easily test and then go on to the more complex ones. We started by writing the very basic Clear_Alloc(), Clear_ErrorLogs(), Init_GlobVars() functions. Since these functions were easily testable and wouldn't take much time to test/write; we finished them firstly. As expected, these testing of the functions were pretty easy and we were able to go on to the more complex ones. At this point in the project is where we probably should not have had the approach we initially finished the project with. We thought that writing the SysTick Timer codes at the end and writing/testing the general linked list functions before getting them in the handler would be easier. In fact this process was a lot easier in the coding part since we wrote the functions one by one instead of writing all of them in the timer and testing once but it served us with some more problems that we probably could have avoided if we had written the timer functions a lot earlier and tested all the linked list functions in the timer. First of all since we were writing all these functions manually, we would have to put these functions in the handler and call the handler in the main function each time for testing them. Instead of doing this if we had just written the normal timer functions, we wouldn't have had do to all this time-consuming effort to test every function one by one. Also since we had to change the main function to call these functions, the branch methods we initially did were a lot different than what we would have to do when we were writing the timer functions. This did cause some more problems when we got to that point of the project. One way or the other though, we were able to write all the linked list functions one by one and test them. The functions talked about here are Malloc(), Free(address), Insert(value), Remove(value) and LinkedList2Arr() function. Also as another note here, while writing all these functions since we weren't using a timer to call them; instead of writing one WriteErrorLog(Index, ErrorCode, Operation, Data) and calling it in the handler; we had initially called this function for every operation in each case which made the code harder to read/write/debug etc. This was another problem we had to deal with while writing the timer functions. After doing all of this, we finally started writing the SysTick timer functions. Writing these functions in itself wasn't that hard but dealing with the problems we have caused on ourselves was the main problem. Especially the remaining branches from the tests we had to deal with. We had to change all those branches to act according to the handler instead of the main function. Other than these, writing the handler was no problem and the fact that we had the safety of knowing that everything we called in the worked created a leeway for us. We finished the SysTick_Handler(), SysTick_Init() and SysTick_Stop() functions and were able to get the timer working properly. Last thing left for us to do was to write the GetNow() function. We had left this function initially since at the time of writing the WriteErrorLog() function, we hadn't yet started using the timer. We had many ideas on how to get the timer value but at the end decided to use the formula $PERIOD * (TICK\_COUNT + 1) + (reload - currentvalue)/32$. There was a lot of discussion between us on how the timer worked and how we should have translated the way the timer worked to an actual value. Two key points here were the fact that we used $TICK\_COUNT + 1$ and $(reload - currentvalue)$. We were debating between adding an +1 to the tick count and through tests, we saw that the timer actually uses an interrupt before even doing a cycle on the handler so we decided to add the +1. Also, the fact that the timer worked backward from the reload value to 0, we had to find the difference between the current value and the reload value to see how much time had passed since the last interrupt. After figuring this out and integrating the GetNow() function to the WriteErrorLog() function, we started testing our project. We tested the project with many input data sets which tried to cover all the cases and errors we thought could arise (the default input data was included in these tests). We also tested the GetNow() function by making an error at random points and comparing how much time has passed from the start of the timer to now with the value we were getting while writing the error. Examples of all of these are also shown in the results part of the report one can refer to there for further information. After finishing our test, we cleaned up our code, wrote the necessary comments, and finished our project.

5 CONCLUSION

In this project, we learned the general idea of how to use a SysTick Timer and also some ideas on how to make more complex memory and assembly operations. Rather than the complexity of the actual operations, the fact that there were a lot of cases to be handled both on memory and the written code; was the real difficulty of the project. We had already known how to build a sorted linked-list. In fact, this type of linked list is one of the easier types of linked lists to be built. Building this on assembly language though faces us with the difficulty that for each operation or function done; there are multiple memory places and code branches that needed to be checked for the debugging of the code which makes it a time-consuming effort. Also, the fact that the whole project is done on a SysTick Timer and in quite a large memory space, further increases the effort we have to put into it. Besides this though, it is quite a learning experience since the complexity of it makes you understand how all these operations work on both code and memory a lot more clearer. Basically, the hardening of the effort put into the code gives a lot more understanding of what you are doing instead of just writing the code to make it work. This was quite important since, at times, we would have problems understanding what was needed from us in the project but would be able to solve it by going step by step on what we had to do. In general, we can say that despite the time-consuming effort we had to put in, the project was a good learning experience on general assembly coding, SysTick Timer, and all the operations on memory.