ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 212E

MICROPROCESSOR SYSTEMS
TERM PROJECT

DATE : 31.01.2021

GROUP NO : G32

GROUP MEMBERS:

150180116 : Ömür Fatmanur Erzurumluoğlu

150180053 : Mehmet Karaaslan

150180734 : Sinan Şar

150180901 : Ramal Seyidli

150170016 : Ümit Başak

FALL 2020

Contents

1 INTRODUCTION

In this project we were required to create a sorted set linked list structure using ARM assembly language. The program has to read the data and operation flags from the input data set arrays and perform an operation in each System Tick ISR. The System Tick Timer will stop if the program reads all the data in the input datasets.

2 MATERIALS AND METHODS

All the functions and methods used in our implementation are explained in a deeply detailed fashion in the subparts below.

2.1 Clear_Alloc()

This function changes all bits of the allocation table to 0.

Listing 1: Clear_Alloc

```
1  Clear_Alloc     FUNCTION
2  ;//-------- <<< USER CODE BEGIN Clear Allocation Table Function >>>
       ----------------------

3                  LDR     R2, =AT_MEM          ;loads start address of
                       allocation table to r2
4                  LDR     R0, =AT_SIZE         ;loads size of
                      allocation table to r0
5                  MOVS    R1, #0               ;loads 0 to r1
6  Clear_loop      SUBS    R0, R0, #4           ;decreases the index by
       4 to clear from last element of allocation table to first
      element
7                  STR     R1, [R2,R0]          ;assigns 0
8                  CMP     R0, R1               ;compares 0 and R0 (
                      index)
9                  BNE     Clear_loop           ;if the first element
                      is not reached, branches to Clear_loop
10                 BX      LR                   ;branches to main

11 ;//-------- <<< USER CODE END Clear Allocation Table Function >>>
       ------------------------
12                 ENDFUNC
```

2.2 Clear_ErrorLogs()

This function clears all cells in the Error Log Array in case the memory addresses contain unwanted values.

Listing 2: Clear_ErrorLogs

```
1  Clear_ErrorLogs FUNCTION
2  ;//-------- <<< USER CODE BEGIN Clear Error Logs Function >>> ↩
       ----------------------                                    ↩

3                       LDR     R2, =LOG_MEM            ;loads start ↩
                            address of error log array to r2
4                       LDR     R0, =LOG_ARRAY_SIZE     ;loads size of ↩
                            error log array to r0
5                       MOVS    R1, #0                  ;loads 0 to r1
6  Clear_error_loop     SUBS    R0, R0, #4              ;decreases the ↩
       index by 4 to clear from last element of error log array to ↩
       first element
7                       STR     R1, [R2,R0]             ;assigns 0
8                       CMP     R0, R1                  ;compares 0 and↩
                             R0 (index)
9                       BNE     Clear_error_loop        ;if the first ↩
                            element is not reached, branches to ↩
                            Clear_error_loop
10                      BX      LR                      ;branches to ↩
                            main
11 ;//-------- <<< USER CODE END Clear Error Logs Function >>> ↩
       ------------------------
12                  ENDFUNC
```

2.3 Init_GlobVars()

This function sets the global variables that are used in the program which are TICK_COUNT, FIRST_ELEMENT, INDEX_INPUT_DS, INDEX_ERROR_LOG and PROGRAM_STATUS to 0.

Listing 3: Init_GlobVars

```
1  Init_GlobVars   FUNCTION
2  ;//-------- <<< USER CODE BEGIN Initialize Global Variables >>> ↩
       ----------------------
3                  MOVS     R1, #0                      ;assigns 0 to ↩
                      R1
4                  LDR      R0, =TICK_COUNT             ;loads the ↩
                      address of TICK_COUNT global value
5                  STR      R1, [R0]                    ;assigns 0 to ↩
                      TICK_COUNT
6                  LDR      R0, =FIRST_ELEMENT          ;loads the ↩
                      address of FIRST_ELEMENT global value
7                  STR      R1, [R0]                    ;assigns 0 to ↩
                      FIRST_ELEMENT
8                  LDR      R0, =INDEX_INPUT_DS         ;loads the ↩
                      address of INDEX_INPUT_DS global value
9                  STR      R1, [R0]                    ;assigns 0 to ↩
                      INDEX_INPUT_DS
10                 LDR      R0, =INDEX_ERROR_LOG        ;loads the ↩
                      address of INDEX_ERROR_LOG global value
11                 STR      R1, [R0]                    ;assigns 0 to ↩
                      INDEX_ERROR_LOG
12                 LDR      R0, =PROGRAM_STATUS         ;loads the ↩
                      address of PROGRAM_STATUS global value
13                 STR      R1, [R0]                    ;assigns 0 to ↩
                      PROGRAM_STATUS (Program started)
14                 BX       LR                          ;branches to ↩
                      main
15 ;//-------- <<< USER CODE END Initialize Global Variables >>> ↩
       ------------------------
16                 ENDFUNC
```

2.4 SysTick_Init()

In the SysTick_Init function, the SysTick Registers are set to interrupt the program in the determined period. As stated in the assignment, period of the System Tick Timer interrupt is 964 us and CPU clock frequency is 16 MHz. The reload value is calculated as 15423 by using equation (1). The reload value is loaded to the SysTick Reload Value Register. The SysTick Current Value Register that counts down from the reload value to zero is set to 0. The SysTick Control and Status Register is assigned as 7 in order to set CLKSOURCE, TICKINT and ENABLE as 1. Since the timer will start after this function, the value of PROGRAM_STATUS becomes 1.

$$Period\ Of\ the\ System\ Tick\ Timer\ Interrupt = (Reload\ Value + 1)/Clock\ Frequency \tag{1}$$

Listing 4: SysTick_Init Function

```
1  SysTick_Init    FUNCTION
2  ;//-------- <<< USER CODE BEGIN System Tick Timer Initialize >>> ↩
       ----------------------
3                  LDR  R0, =0xE000E014            ;loads the address ↩
                       of SysTick Reload Value Register
4                  LDR  R1, =15423                 ;assigns computed ↩
                       reload value (15423) to R1
5                  STR  R1,[R0]                    ;assigns 15423 to ↩
                       SysTick Reload Value Register
6
7                  LDR  R0, =0xE000E018            ;loads the address ↩
                       of SysTick Current Value Register
8                  MOVS R1, #0                     ;assigns 0 to R1
9                  STR  R1,[R0]                    ;assigns 0 to ↩
                       SysTick Current Value Register
10
11                 LDR  R0, =0xE000E010            ;loads the address ↩
                       of SysTick Control and Status Register
12                 MOVS R1, #7                     ;assigns 7 to R1
13                 STR  R1, [R0]                   ;assigns 1 to ↩
                       CLKSOURCE, TICKINT and ENABLE
14
15                 LDR  R0, =PROGRAM_STATUS        ;loads the address ↩
                       of PROGRAM_STATUS global value
16                 MOVS R1, #1                     ;assigns 1 to R1
```

```
17                  STR   R1, [R0]                    ;assigns 1 to ↩
                        PROGRAM_STATUS (Timer started)
18
19                  BX    LR                          ;branches to main
20  ;//-------- <<< USER CODE END System Tick Timer Initialize >>> ↩
        ------------------------
21                  ENDFUNC
```

2.5 SysTick_Handler()

In the SysTick_Handler function, the input data and its flag are read from IN_DATA and IN_DATA_FLAG areas. When the flag value is 0, the input value is found in the linked list and removed from the list. If the linked list is empty, error code becomes 3, and if the input data cannot be found in the list, it becomes 4 during remove operation. When the flag value is 1, Insert function is called to insert the input value to linked list. If no allocable area is found in the insert operation, the error code is set to 1 and if the input is an existing value in the linked list, error code becomes 2. When flag is 2, the linked list is transformed to the array. Error code becomes 5 if the linked list is empty and the error code becomes 6 for other flag values. If the operation is performed properly, the error code is set to success code.

Listing 5: SysTick_Handler Function

```
1   SysTick_Handler FUNCTION
2   ;//-------- <<< USER CODE BEGIN System Tick Handler >>> ↩
        ----------------------                                    ↩
3                   EXPORT SysTick_Handler
4
5                   PUSH    {LR}                        ;pushes LR to stack
6
7                   LDR     R7, =TICK_COUNT             ;loads address of ↩
                        TICK_COUNT to r7
8                   LDR     R1, [R7]                    ;loads the value of↩
                         TICK_COUNT to r1
9                   MOVS    R6, #4                      ;assigns 4 to R6
10                  MULS    R6, R1, R6                  ;multiplies ↩
                        TICK_COUNT value and 4
11                  LDR     R5, =IN_DATA_FLAG           ;loads start ↩
                        address of flags of input dataset to r5
```

```
12                  LDR     R2, [R5, R6]            ;R2 stores the flag↩
                        (operation)
13                  LDR     R5, =IN_DATA            ;loads start ↩
                        address of input dataset to r5
14                  LDR     R3, [R5, R6]            ;R3 stores the ↩
                        input data
15                  MOVS    R0, R3                  ;R0 also stores the↩
                         input data
16                  PUSH    {R7,R1,R2,R3}           ;pushes the ↩
                        registers to stack
17
18                  CMP     R2, #0                  ;if the flag is 0
19                  BEQ     remove                  ;branches to remove
20                  CMP     R2, #1                  ;if operation flag ↩
                        is 1, call insert.
21                  BEQ     call_insert             ;branches to ↩
                        call_insert
22                  CMP     R2, #2                  ;if operation flag ↩
                        is 2
23                  BEQ     transform               ;branches to ↩
                        transform
24                  BNE     error                   ;if not equal, ↩
                        branches to error
25  call_insert     BL      Insert                  ;branches with link↩
        to Insert function
26                  B       pp                      ;after insert ↩
                        operation, branches to pp
27  remove          BL      Remove                  ;branches with link↩
        to Remove function
28                  B       pp                      ;after remove ↩
                        operation, branches to pp
29  transform       BL      LinkedList2Arr          ;branches with link↩
        to LinkedList2Arr function
30                  B       pp                      ;after transform ↩
                        operation, branches to pp
31
32  error           MOVS    R0, #6                  ;R0 stores the ↩
       error code as 6
```

After the operation, WriteErrorLog function is called to write the index, error code, flag and input data in LOG_MEM area. At the end of this function, it checks that all data is read. If all data is read, the SysTick_Stop function is called and the PROGRAM_STATUS is set to 2.

Listing 6: SysTick_Handler Function

```
1  pp              POP     {R7,R1,R2,R3}           ;pops from stack to the registers
2                  PUSH    {R7,R1}                 ;pushes the registers to stack
3                  MOVS    R7, R1                  ;to interchange R1 and R0
4                  MOVS    R1, R0                  ;R1 stores the error code
5                  MOVS    R0, R7                  ;R0 stores the index
6                  BL      WriteErrorLog           ;branches with link to WriteErrorLog function
7                  POP     {R7,R1}                 ;pops from stack to the registers
8                  ADDS    R1, #1                  ;increases R1 value by 1
9                  STR     R1, [R7]                ;loads R1 to TICK_COUNT
10
11                 LDR     R5, =IN_DATA_FLAG       ;loads start address of flags of input dataset to r5
12                 LDR     R3, =END_IN_DATA_FLAG   ;loads end address of flags of input dataset to R3
13                 MOVS    R6, #4                  ;assigns 4 to R6
14                 MULS    R6, R1, R6              ;multiplies TICK_COUNT value and 4
15                 ADDS    R6, R5, R6              ;sums the start adress of flags array and TICK_COUNT
16                 CMP     R3, R6                  ;compares the sum and the end adress of the array
17                 BEQ     SysTick_Stop            ;if equal, branches to SysTick_Stop
18                 LDR     R0, =PROGRAM_STATUS     ;Load Program Status Variable Addresses.
19                 POP     {PC}                    ;returns to where the SysTick_Handler function was called
20
21 ;//-------- <<< USER CODE END System Tick Handler >>> ----------------------
22                 ENDFUNC
```

2.6 Insert(value)

The Insert function takes the input value that is coming from systick handler function with the R0 register then, iterates the linked list, compares the values that are in the linked list and inserts the new value to the correct place. While iterating, if a duplicate value is encountered, the function returns to the systick handler with an error code. There are four major cases that the program may encounter while the insertion:

Listing 7: Insert Function

```
1  Insert          FUNCTION
2  ;//-------- <<< USER CODE BEGIN Insert Function >>> ↩
       ----------------------
3                  PUSH    {LR}                ;Push LR, will pop pc ↩
                       in order to return to systick_handler
4                  LDR     r5,=FIRST_ELEMENT   ;store head pointer at ↩
                       r5
5                  LDR     r1,=DATA_MEM        ;store value of head at↩
                        r1
6                  LDR     r1,[r1]     ;load r1 with first nodes data↩

7                  LDR     r2,[r5]     ;load r2 with heads address
8                  ;LDR        r2,[r2]
9
10             ;THis should be done for R0 in handler ;;LDR        r2↩
                   ,=[IN_DATA,TICK_COUNT]    ;;!!!!!input == 0 and list↩
                    is empty, there will be some issues..
11
12                 CMP     r2,#0               ;Check if list is empty
13                 BEQ     Empty_list          ;if it is, branch to ↩
                       empty list case
14
15                 CMP     r0,r1               ;Check if input is ↩
                       smaller than head
16                 BLT     New_Head        ;if it is, branch to add ↩
                       new head
17                 MOVS    r1,r5               ;move head pointer to ↩
                       r1
18                 LDR     r1,[r1]         ;load r1 with heads address
19                 B       Check           ;branch to case checking ↩
                       loop
```

- Inserting to an empty list: While inserting to an empty list, first, the function puts the input to the first space in the DATA_MEM and then changes the last bit of the Allocation Table as 1 and returns to the systick handler with a success code.

Listing 8: Inserting to an empty list

```
1   Empty_list      LDR     r1,=DATA_MEM        ;set r1 as ↩
        Data_MEM's start address
2               STR     r1,[r5]
3               STR     r0,[r1]             ;Set head of the list↩
                    as input
4               LDR     r4,=(__AT_END-4)    ;load end address of ↩
                allocation table to r4
5               LDR     r5,=0x00000001      ;prepare value to ↩
                store in allocation table.(32 bits 000...001)
6               STR     r5, [r4]            ;Set allocation table
7               MOVS    r5,#1               ;Load 1 to r5
8               MOVS    r0,#0               ;Set r0 with success ↩
                code
9               ;LDR        r2,=IS_EMPTY        ;load  IS_EMPTY '↩
                s address to r2
10              ;STR        r5,[r2]             ;Set IS_EMPTY ↩
                variable as 1, since list is not empty anymore
11
12              ;;This doesn't require malloc, we don't think...
13              POP     {PC}                ;return to ↩
                systick_handler
```

- Inserting to the head of the list: While inserting to the head of the list, first, the function calls the malloc function, the malloc function returns an appropriate address from the DATA_MEM and changes the Allocation Table's corresponding bit to 1. Then, the insert function stores the new input value to the DATA_MEM and changes the DATA_MEM's second word as previous head's address. Finally it makes the new inserted element the FIRST_ELEMENT. While this function is working, if the malloc function returns an error, it branches to Mal_Ins_Error and assigns 1 to R0 to indicate that a malloc error occured, and returns to systick handler.

Listing 9: Inserting before the head of the list

```
1   New_Head        PUSH    {r0,r1,r5}          ;push values ↩
        before calling malloc
2               BL      Malloc              ;call malloc
3               MOVS    r2,r0               ;Save address coming ↩
                    from malloc to R2
4               POP     {r5,r1,r0}          ;pop
5               CMP     r2,#0               ;if malloc raised an ↩
                    error
6               BEQ     Mal_Ins_Error           ;branch to error ↩
                    label
7               STR     r0,[r2]             ;store input value at↩
                     allocated memory address
8               LDR     r1,[r5]             ;load current heads ↩
                    address to r1
9               ;LDR    r1,[r1]
10              STR     r1,[r2,#4]          ;store next pointer ↩
                    at allocated memory address.
11              STR     r2, [r5]            ;make the new head ↩
                    first_element
12              MOVS    r0,#0               ;Set r0 with success ↩
                    code
13              POP     {PC}                ;return to ↩
                    systick_handler
```

- Inserting in between the nodes: While inserting in between the nodes, similar to the previous case, the function calls the malloc function, stores the input in the first word of DATA_MEM, it changes the next elements address as next address for the new element and changes new elements address as current elements next address. Similar to the previous case, it returns to systick handler if an error occurs.

Listing 10: Inserting in between the list

```
1   Check
2               MOVS    r3,r1               ;Load r3 with current↩
                     nodes next address
3               LDR     r6,[r3,#4]          ;Lad r6 with next ↩
                    nodes next value
4               LDR     r4,[r3]             ;load r4 with next ↩
                    nodes data
5               CMP     r0,r4               ;check if input is ↩
```

```
                    duplicate
6                   BEQ     Duplicate_Error         ;branch to error ↩
                    table
7                   CMP     r0,r4                   ;compare input with ↩
                    next node,if its lesser then we found where to↩
                     insert
8                   BLT     found_insert            ;branch to insert↩
                     new node
9                   CMP     r6,#0                   ;Check if we are at ↩
                    the tail
10                  BEQ     add_tail                ;if we are, branch to↩
                     add new tail
11                  ;MOVS   r1,r4                   ;else, load r1 with ↩
                    next nodes data
12                  MOVS    r7,r1                   ;load r1 with current↩
                     nodes data
13                  MOVS    r1,r6                   ;else, load r1 with ↩
                    next nodes next
14                  B       Check
15
16  found_insert    PUSH    {r0,r1,r3,r5,r7}    ;push values before ↩
    calling malloc
17                  BL      Malloc              ;call malloc
18                  MOVS    r2,r0               ;Save address coming ↩
                    from malloc to R2
19                  POP     {r7,r5,r3,r1,r0}    ;pop
20                  CMP     r2,#0               ;if malloc raised an ↩
                    error
21                  BEQ     Mal_Ins_Error           ;branch to error ↩
                    label
22                  STR     r0,[r2]             ;store input value at↩
                     allocated memory address
23                  STR     r3,[r2,#4]          ;store next elements ↩
                    data address as next for new element
24                  STR     r2,[r7,#4]          ;store new elements ↩
                    data address as current elements next.
25                  MOVS    r0,#0               ;Set r0 with success ↩
                    code
26                  POP     {PC}                ;return to ↩
                    systick_handler
```

- Inserting after the last element: While inserting after the last element, similar to the previous cases, the function calls the malloc function, stores the input in the first word of DATA_MEM, changes the old tails next address as the new elements address and sets new elements next address as null. Similar to the previous case, it returns to systick handler if an error occurs in malloc.

Listing 11: Inserting after the last element of the list

```
1   add_tail        PUSH    {r0,r1,r3,r4}       ;push values ↩
        before calling malloc
2               BL      Malloc              ;call malloc
3               MOVS    r2,r0               ;Save address coming ↩
                    from malloc to R2
4               POP     {r4,r3,r1,r0}       ;pop
5               CMP     r2,#0               ;if malloc raised an ↩
                    error
6               BEQ     Mal_Ins_Error           ;branch to error ↩
                    label
7               STR     r0,[r2]             ;store input value at↩
                     allocated memory address
8               STR     r2,[r3,#4]          ;Load old tails next ↩
                    with new tails data address
9               MOVS    r4,#0               ;Load r4 with next ↩
                    nodes data
10              STR     r4,[r2,#4]          ;Set new tails next ↩
                    as null.
11              MOVS    r0,#0               ;Set r0 with success ↩
                    code
12              POP     {PC}
```

2.7 Malloc()

The Malloc function takes the input value that is coming from Insert function with the Register 0 then, iterates the allocation table to check for empty space in the allocation table. If there is space for a new node, a memory address is returned to Insert function correspondingly. If there is not any empty space in the allocation table, Malloc will return an error code to the Insert function via Register 0.

- Allocation Table: Malloc function finds the free memory spaces and allocates them using the allocation table. The allocation table consists of 20 words and each word has 32 bits. Each bit corresponds to one memory node. If the bit is 0, the memory node is free and a new input can be allocated there. We used the allocation table like the figure present in the assignment pdf. Nodes are allocated in each row starting from the least significant bit of the 32 bits. We also selected the last word of the allocation table memory area to be the first row, just to make it easier to visualise. The whole table is allocated from right to left visually.

- Connection between Allocation Table and Data Memory: Each bit in allocation table refers to a node in the Data Memory area. This relation is set in a very straightforward manner. The first bit in the Allocation Table is related with the first 2 words (1 word for data, 1 word for next pointer) of the Data Memory. All other bits and the space is similarly related to each other.

- Iteration: Malloc function starts iterating on the first row of the allocation table. This iteration is done by a loop, throughout the loop each bit is checked for their value with a mask. If a free space is found, the program branches to the found label. If not, the word gets shifted and masked again to check the next bit. If all bits in the row are checked and no free space is found the program branches to the newline label, which advances to a new row in the allocation table.

Listing 12: Malloc iteration

```
1 Malloc          FUNCTION
2 ;//-------- <<< USER CODE BEGIN System Tick Handler >>> ↩
    ----------------------
3                 LDR     r1,=DATA_MEM        ;load start address ↩
                      of linked list to r1
4                 LDR     r2,=AT_MEM          ;load start address ↩
                      of allocation table to r2
5                 LDR     r4,=(__AT_END-4)    ;load end address of ↩
                      allocation table to r4
```

```
6                   LDR     r5,=0xFFFFFFFE      ;mask for each line ↩
                        on allocation table.(32 bits 111...110)
7                   MOVS    r7,#0               ;keep an index for ↩
                        iterations at r7
8                   LDR     r6,[r4]             ;load 32 bits of ↩
                        allocation table to r6. The 32 bits will ↩
                        change as we iterate on the table.
9   iterate_mal     CMP     r7,#32              ;check if we looked ↩
        at all 32 bits in allocation table line, if true go to next ↩
        line
10                  BEQ     new_line
11                  PUSH    {r7}                ;push r7 to preserve ↩
                        its value
12                  MOVS    r7,r6               ;move r6 to r7
13                  ORRS    r7,r5,r7            ;mask r6 value to ↩
                        check if lsb is 0
14                  CMP     r7,r5               ;if it is 0, we found↩
                         a free memory space for a new node.
15                  BEQ     found
16                  POP     {r7}                ;if not, pop r6 and ↩
                        keep iterating on the allocation table line.
17                  LSRS    r6,#1               ;shift r6 value to ↩
                        right to check the next bit
18                  ADDS    r7,#1               ;increment loop index↩
                         by one
19                  ADDS    r1,#8               ;increment linked ↩
                        list node address by 8.(4 for data,4 for next)
20                  B       iterate_mal         ;return to start of ↩
                        the loop
```

- Newline label: This label advances to the next row on the allocation table. If the whole table has been checked and no free space is found, it branches to the error label.

Listing 13: Malloc newline label

```
1   new_line        SUBS    r4,#4               ;iterate to a new ↩
        line in the allocation table
2                   CMP     r4,r2               ;if we checked the ↩
                        whole table and there is no empty space
3                   BLT     mal_error           ;go to error label
4                   MOVS    r7,#0               ;keep an index for ↩
                        iterations at r7
```

```
5                   LDR     r6,[r4]             ;load value at r4 to ↩
                        r6
6                   B       iterate_mal         ;if not at the end of↩
                         table, check a new line at the loop
```

- Malloc Error: Malloc returns an error if there is no empty space in the allocation table.

Listing 14: Malloc error label

```
1   mal_error       MOVS    r0,#0               ;set r0 as allocation↩
        error flag
2                   BX      LR                  ;return to Insert
```

- Found label: If a free space in the allocation table is found, the Malloc returns its corresponding memory area to Insert. It also sets the found bit in the allocation table to 1.

Listing 15: Malloc found label

```
1   found           POP     {r7}                ;if found, pop r7.
2                   MOVS    r0,r1               ;load found node ↩
                        address to r0
3                   MOVS    r3,#1               ;set r3 as 1
4                   LSLS    r3,r7               ;shift r3 left by ↩
                        amount of iterations in loop
5                   LDR     r6,[r4]             ;load value at r4 to ↩
                        r6
6                   ORRS    r6,r3,r6            ;set the used bit to ↩
                        1
7                   STR     r6,[r4]             ;store new allocation↩
                         table value
8                   BX      LR                  ;return to Insert
9   ;//-------- <<< USER CODE END System Tick Handler >>> ↩
        -------------------------
10                  ENDFUNC
```

2.8 Remove(value)

This function is used to find and remove the node from the linked list. It searches for the node and removes it from the linked list. If there is not any node which has the same data with the argument this function returns the error code "4" or linked list is empty it returns the error code "3" otherwise it removes the node from the linked list and returns the success code "0".

- In the beginning, remove function loads necessary addresses and checks if the linked is empty. If the linked list is empty function returns error code "3".

Listing 16: Beginning

```
1  Remove          FUNCTION
2  ;//-------- <<< USER CODE BEGIN Remove Function >>> ↩
       ----------------------                                    ↩

3                  PUSH    {LR}            ;Preserves the address of↩
                       the Sys_handler
4                  LDR     R1,=DATA_MEM    ;Load the address of Data↩
                       memory in R1
5                  LDR     R2,=FIRST_ELEMENT   ;Load the address of ↩
                       the First_Element in R2
6                  LDR     R3,[R2]     ;Load the address of the ↩
                       first element of linked list
7                  CMP     R3, #0      ;Compare R3 with 0
8                  BEQ     empty       ;If linked list is empty ↩
                       returns error code 3
9                  .
10                 .
11                 .
12 empty           MOVS    R0, #3  ;If linked list is empty returns ↩
       error code 3
13                 POP     {PC}        ;Back to Sys_Handler
14 ;//-------- <<< USER CODE END Remove Function >>> ↩
       ------------------------
15                 ENDFUNC
```

- In the iteration part, it searches for the node to delete. If there is not any node with the given argument it returns error code "4". If it finds the node to remove branches to delete part.

Listing 17: Iteration

```
1                   MOVS    R5, #0      ;counter for loop (i)
2   iteration       LDR     R4, [R3]        ;Load the data of the ↩
        first element to R4
3                   CMP     R4, R0          ;Compares R4 with input (↩
                        R0)
4                   BEQ     delete      ;If R4 is what user looks for↩
                         program goes for delete
5                   MOVS    R6, R3      ;Copy the current element ↩
                        address to R6
6                   LDR     R3, [R3,#4] ;Assigns the address of the ↩
                        next element to R3
7                   CMP     R3, #0      ;Looks if Array is finished ↩
                        yet or not
8                   BEQ     not_found   ;If yes, program goes for ↩
                        not_found
9                   ADDS    R5, #4      ;increasement of i
10                  B       iteration   ;branching loop
11                  .
12                  .
13                  .
14  not_found       MOVS    R0, #4  ;If linked list is empty returns ↩
        error code 4
15                  POP     {PC}        ;Back to Sys_Handler
```

- In the delete part, it first controls if the node is the first node and branches to delete first part or continues if the node is not the first element of the linked list. To delete an element it first loads the address of the next node to the previous node than removes the element from the linked list. After deleting the node it calls the Free function with the address of the node that is deleted than returns success code "0".

Listing 18: Delete

```
1   delete          CMP     R5, #0              ;compare r5 with 0, ↩
        program removes the asked element
2                   BEQ     delete_first        ;if element asked for↩
                         removal is the first one, goes for ↩
                        delete_first
3
4                   LDR     R4, [R3, #4]        ;Load address of the ↩
                        next of the removal element to R4
```

```
5                   STR     R4, [R6, #4]        ;Store R4 in previous↩
                        element of linked list
6                   MOVS    R4, #0              ;Assings 0 to R4
7                   STR     R4, [R3]            ;Assigns 0 to data of ↩
                        removal element, which means deletion
8                   STR     R4, [R3,#4]         ;Assigns 0 to address of ↩
                        removal element, which means deletion
9                   MOVS    R0, R3              ;Copy R5(index of removal↩
                        element) to R0
10                  BL      Free                ;Branch Free
11                  MOVS    R0, #0          ;success code
12                  POP     {PC}            ;Back to Sys_Handler
```

- In the delete first part, it first loads the address of the next element to the FIRST ELEMENT. Then removes the element from the linked list, calls the Free function and returns success code "0".

Listing 19: Delete_first

```
1   delete_first    LDR     R4, [R3,#4]         ;Load address of the next↩
        element to R4
2                   STR     R4, [R2]            ;Change the address of ↩
                        the FIRST_ELEMENT
3
4                   MOVS    R4, #0              ;Assings 0 to R4
5                   STR     R4, [R3]            ;Assigns 0 to data of ↩
                        removal element, which means deletion
6                   STR     R4, [R3,#4]         ;Assigns 0 to address of ↩
                        removal element, which means deletion
7                   MOVS    R0, R3              ;Copy R5(index of removal↩
                        element) to R0
8                   BL      Free            ;Branch Free
9                   MOVS    R0, #0          ;success code
10                  POP     {PC}            ;Back to Sys_Handler
```

2.9 Free(address)

This function is used to clear the corresponding bit from the allocation table using the address value.

- In the beginning, it loads the address of the end of the allocation table to the register 1 and address of the DATA MEM to register 2 (R2) and finds a value that we call index using the following formula.

$$\text{index} = \frac{\text{address} - R2}{8}$$

This value(index) gives us the address of the bit from the allocation table that will be set to 0.

Listing 20: Beginning

```
1 Free              FUNCTION
2 ;//-------- <<< USER CODE BEGIN Free Function >>> ↩
      ----------------------
3                   LDR         R1, =(__AT_END-4)     ;load end address↩
                        of allocation table to R1
4                   LDR         R2, =DATA_MEM         ;load address of ↩
                    data memory to R2
5                   SUBS    R0, R0, R2              ;substract data ↩
                    memory address from remova element's address
6                   LSRS    R0, R0, #3              ;divide R0 by 8
```

- In the dec loop part, it compares index value with 32(4 bytes = 1 word), decrements it by 32 if bigger or equal and decrements the value at register 2 by 4(1 word) and continues until index value is smaller than 32.

Listing 21: dec_loop

```
1                   MOVS    R3, #32         ;Assign 32 to R3
2 dec_loop          CMP         R0, R3      ;Compare R0 with R3 (32)
3                   BLT         dec_end     ;Branch to end loop if ↩
                    smaller
4                   SUBS    R1, R1, #4      ;Substract R1 by 4
5                   SUBS    R0, R0, R3      ;Substract R0 by 32
6                   B        dec_loop       ;Branch to loop again
```

- After dec loop, loads 0xFFFFFFFE to register 4 that will be used to set the corresponding bit 0 using and operation.

Listing 22: After dec_loop

```
dec_end             LDR         R4, =0xFFFFFFFE    ;Load 111...1110 ↩
    (32 bits) to R4
```

- In the shift loop part, function compares index value with 0, decrements it by 1 if bigger, shifts the value in register 4 one bit to the left and since left shift makes the last bit 0, sets the last bit 1 again.

Listing 23: shift_loop

```
                    MOVS     R3, #1           ;Assign 1 to R3
shift_loop          CMP         R0, #0        ;Compare R0 with 0
                    BEQ         shift_end         ;Branch to end loop ↩
                        if equal
                    LSLS     R4, R4, #1       ;Shift left R4 one bits
                    ORRS     R4, R4, R3       ;Or operation R4 with R3 ↩
                        to set last bit 1 again, left shift makes last↩
                         bit 0
                    SUBS     R0, R0, #1       ;Substract R0 by 1
                    B         shift_loop      ;Branch to loop again
```

- After the shift loop part, the corresponding bit from the allocation table is set to 0 using the value at register 4 and "AND" operation.

Listing 24: End of Free

```
shift_end         LDR         R5, [R1]         ;Load value inside R1↩
    (allocation table) to R5
                    ANDS     R5, R5, R4       ;And operation to change ↩
                        value with new one
                    STR         R5, [R1]      ;Store new value in R1(↩
                        allocation table)
                    BX          LR             ;Branching back to Remove↩
                         function
;//-------- <<< USER CODE END Free Function >>> ↩
    ------------------------
```

```
6 ||                    ENDFUNC
```

2.10 LinkedList2Arr()

In the *LinkedList2Arr* function, The linked list that is stored in the DATA_MEM area is copied into the ARRAY_MEM area in order. It does not take any parameter. The function returns the success code if the linked list is properly transformed to the array. However, it returns error code 5 if linked list is empty. As shown in Listing ??, firstly the content of the ARRAY_MEM area is cleared. In order to assign 0 to each word of ARRAY_MEM area, clearll2a loop is used.

Listing 25: Clearing the ARRAY_MEM area

```
1 LinkedList2Arr  FUNCTION
2 ;//-------- <<< USER CODE BEGIN Linked List To Array >>> ↩
      ----------------------
3                 LDR     R7, =ARRAY_MEM      ;loads start address of↩
                      array to r7
4                 LDR     R5, =ARRAY_SIZE     ;loads size of array to↩
                      r5
5                 MOVS    R6, #0              ;loads 0 to r6
6 clearll2a       SUBS    R5, R5, #4          ;clears from last ↩
      element of array to first element
7                 STR     R6, [R7,R5]         ;assigns 0
8                 CMP     R5, R6              ;compares 0 and R5
9                 BNE     clearll2a           ;if the first element ↩
                     is not reached, branches to clearll2a
```

Then, Starting from the first element of the linked list, the value of each element is copied to the array. At the begining of this part of code, it checks that the linked list is empty. If FIRST_ELEMENT is 0, it is empty and the function returns error code 5 with register 0. Otherwise, the values of linked list are transferred to the array. The next element is reached by using the address value that is stored in the second word of the element. After transformation, the function returns the success code with register 0.

Listing 26: Transforming the linked list to the array

```
1                   LDR     R3,=FIRST_ELEMENT   ;loads the address of ↩
                        the FIRST_ELEMENT to R3
2                   LDR     R4, [R3]            ;loads the value of ↩
                        first element to R4
3                   CMP     R4, #0              ;compare R5 with 0
4                   BEQ     emptyll2a           ;If linked list is ↩
                        empty, branch to emptyll2a
5                   LDR     R5, [R4]            ;loads the value of ↩
                        first element to R5
6                   MOVS    R6, #0              ;assisgns 0 to R6 to ↩
                        reach the first element of the array
7   loopll2a        STR     R5, [R7, R6]        ;assigns the value of ↩
        linked list to the array
8                   ADDS    R6, #4              ;increases inex of the ↩
                        array (R6) by 4
9                   ADDS    R4, #4              ;to reach the address ↩
                        value of linked list element
10                  LDR     R4, [R4]            ;assigns the address ↩
                        value of linked list element to R3
11                  LDR     R5, [R4]            ;assigns the value of ↩
                        the next element to R5
12                  CMP     R4, #0              ;if the address value ↩
                        of linked list element is not empty
13                  BNE     loopll2a            ;branch to loop
14                  MOVS    R0, #0              ;success code
15                  BX      LR                  ;Back to Sys_Handler
16
17  emptyll2a       MOVS    R0, #5              ;if linked list is ↩
        empty returns error code 5
18                  BX      LR              ;Back to Sys_Handler
19  ;//-------- <<< USER CODE END Linked List To Array >>> ↩
        ------------------------
20                  ENDFUNC
```

2.11 WriteErrorLog(Index, ErrorCode, Operation, Data)

In the *WriteErrorLog* function, informations about operation that is performed in SysTick_Handler function are written in LOG_MEM area. The function takes Index, ErrorCode, Operation, and Data variables as parameters and 3 word is written in the area. The first 16 bits, 8 bits and other 8 bits of the first word belong to index, error code and operation. The second word belongs to the input data and the working time is stored in the third word. In order to get the working time of the System Tick Timer (in us), GetNow function is called.

Listing 27: WriteErrorLog Function

```
1  WriteErrorLog   FUNCTION
2  ;//-------- <<< USER CODE BEGIN Write Error Log >>> ↩
       ----------------------
3                  PUSH    {LR}                    ;pushes LR to stack
4
5                  LSLS    R4, R1, #8              ;shifts left error ↩
                       code 8 times
6                  ADDS    R2, R2, R4              ;sums operation ↩
                       value and shifted error code
7                  LSLS    R4, R0, #16             ;shifts left index ↩
                       16 times
8                  ADDS    R2, R2, R4              ;sums shifted index↩
                       , shifted error code and operation value
9                  LDR     R4, =INDEX_ERROR_LOG    ;loads the address ↩
                       of the INDEX_ERROR_LOG to R4
10                 LDR     R5, [R4]                ;loads the value of↩
                        INDEX_ERROR_LOG to R5
11                 MOVS    R0, #12
12                 MULS    R0, R5, R0
13                 LDR     R6, =LOG_MEM            ;loads start ↩
                       address of error log array to R6
14                 STR     R2, [R6, R0]            ;loads the first ↩
                       word to the array
15                 ADDS    R0, R0, #4              ;increases index ↩
                       value by 4
16                 STR     R3, [R6, R0]            ;loads the second ↩
                       word to the array (data)
17                 ADDS    R0, R0, #4              ;increases index ↩
                       value by 4
18                 MOVS    R1, R0                  ;loads R0 value to ↩
                       R1
```

```
19                  BL      GetNow                  ;branch to GetNow ↩
                        function
20                  STR     R0, [R6, R1]            ;loads the third ↩
                        word (timestamp)
21                  ADDS    R5, #1                  ;increases R5 value↩
                         by 1
22                  STR     R5, [R4]                ;loads R5 to ↩
                        INDEX_ERROR_LOG
23
24                  POP     {PC}                    ;returns to ↩
                        SysTick_Handler function
25  ;//-------- <<< USER CODE END Write Error Log >>> ↩
        ------------------------
26                  ENDFUNC
```

2.12 GetNow()

In the *GetNow* function, the working time of the System Tick Timer (in us) is computed by using Equations (2,3). As stated in the assignment, period of the System Tick Timer interrupt is 964 us and CPU clock frequency is 16 MHz. Reload value is 15423. SysTick_Handler function is called every 964 us. The number of times the function was called is kept in the TICK_COUNT. The result of the product of two values is the time until calling this SysTick_Handler function. The result of the product of two values is the time until calling this SysTick_Handler function. In order to calculate the elapsed time in this SysTick_Handler function, the current value is subtracted from the reload value and the result is divided by 16. Finally, two values are added.

$$TICK\_COUNT \times Period\ Of\ the\ System\ Tick\ Timer\ Interrupt \tag{2}$$

$$(Reload\ Value - System\ Tick\ Timer\ Current\ Value)/Clock\ Frequency \tag{3}$$

Listing 28: GetNow Function

```
1  GetNow          FUNCTION
2  ;//-------- <<< USER CODE BEGIN Get Now >>> ----------------------
3                  LDR     R2, =TICK_COUNT         ;loads the address of TICK_COUNT to R2
4                  LDR     R2, [R2]                ;loads the value of TICK_COUNT to R2
5                  ADDS    R2, R2, #1              ;increases TICK_COUNT value by one
6                  LDR     R0, =964                ;assigns the period of the System Tick Timer Interrupt to R0
7                  MULS    R0, R2, R0              ;multiplies TICK_COUNT value and 964 microseconds
8                  LDR     R3, =0xE000E018         ;loads the address of SysTick Current Value Register
9                  LDR     R3, [R3]                ;loads the value of SysTick Current Value Register to R3
10                 LDR     R7, =15423              ;assigns computed reload value (15423) to R7
11                 SUBS    R3, R7, R3              ;subtracts the current value from the reload value
12                 LSRS    R7, R3, #4              ;shifts right the result 4 times to divide by 16
13                 ADDS    R0, R0, R7              ;calculates the final the working time
14
15                 BX      LR                      ;returns to WriteErrorLog function
16 ;//-------- <<< USER CODE END Get Now >>> ------------------------
17                 ENDFUNC
```

2.13 SysTick_Stop()

This function stops the System Tick Timer, clears the interrupt flag of it and updates the program status.

Listing 29: SysTick_Stop

```
1  SysTick_Stop    FUNCTION
2  ;//-------- <<< USER CODE BEGIN System Tick Timer Stop >>> ↩
       ----------------------
3                  LDR  R0, =0xE000E010            ;loads the address ↩
                       of SysTick Control and Status Register
4                  MOVS R1, #0                     ;assigns 0 to R1
5                  STR  R1, [R0]                   ;assigns 0 to ↩
                       CLKSOURCE, TICKINT and ENABLE
6
7                  LDR  R0, =0xE000E014            ;loads the address ↩
                       of SysTick Reload Value Register
8                  MOVS R1, #0                     ;assigns 0 to R1
9                  STR  R1, [R0]                   ;assigns 0 to ↩
                       SysTick Reload Value Register
10
11                 LDR  R0, =PROGRAM_STATUS        ;loads the address ↩
                       of PROGRAM_STATUS global value
12                 MOVS R1, #2                     ;assigns 2 to R1
13                 STR  R1, [R0]                   ;assigns 2 to ↩
                       PROGRAM_STATUS (All data operation finished)
14
15                 POP  {PC}                       ;returns to where ↩
                       the SysTick_Handler function was called
16 ;//-------- <<< USER CODE END System Tick Timer Stop >>> ↩
       ------------------------
17                 ENDFUNC
```

3 RESULTS

The results for this experiment were as they were intended. In this section we will dive deeply into many different cases and have a look at their outcomes to analyze and critique our work.
We will check the state of the program using the test data set provided for us by our teaching assistants.

- Results after initial insertions: First few inputs are all different cases of "Insert" operations into the list. We will have a look at Data Memory, Allocation Table, Array Memory and Log Memory values.

  Looking at the Data Memory (Figure 1) at this point of the program, we can see that all of the insert operations have been done and pointers/data values are indeed correct.

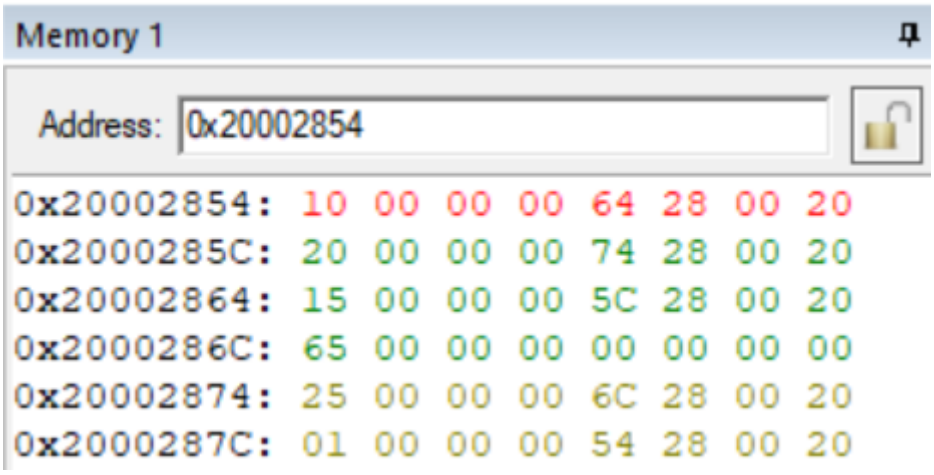

Figure 1: Data Memory at the middle of the program

After looking at the allocation table values (Figure 2)we can see that the necessary bits have been set to 1. These bits are some of the very last bits on the table and this result is right considering our design for the allocation table.

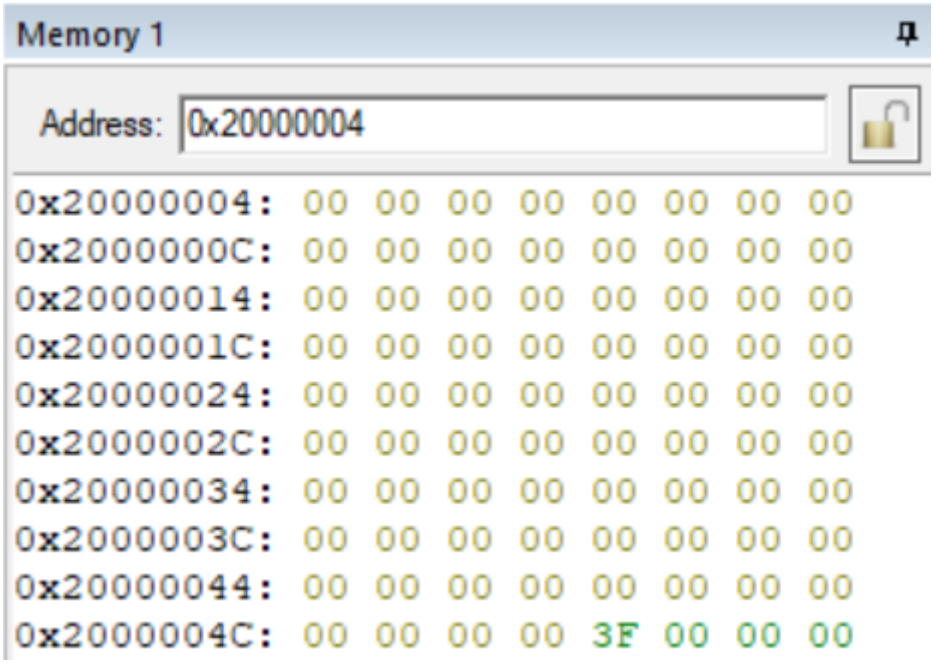

Figure 2: Allocation Table at the middle of the program

Log Memory will be discussed at the end of program, since early values don't change throughout the execution time of the program.

Observing the Array Memory would be quite pointless at this point of the program since Linked List To Array function has not been called yet.

- Results at the end of program: We will now observe and interpret the result our code produces at the end of the program execution.

  Looking at the Data Memory (Figure 3) at this point of the program, we can see that all of the insert operations have been done and pointers/data values are indeed correct.

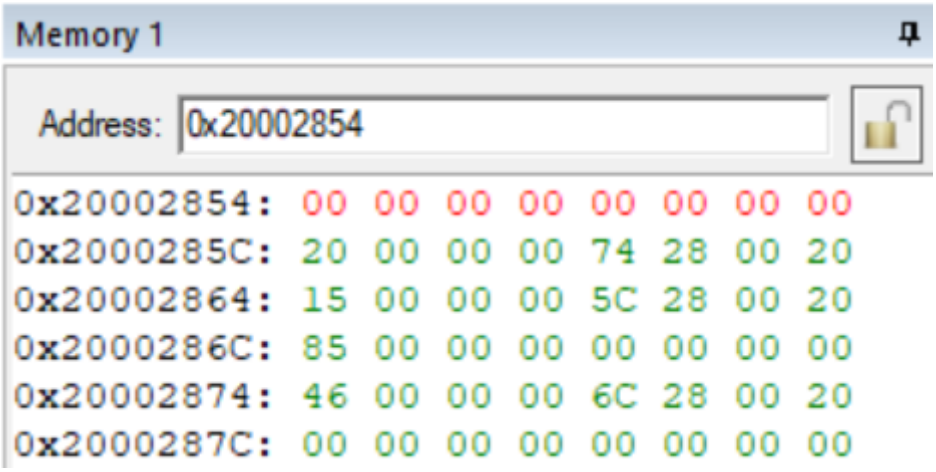

Figure 3: Data Memory after the program

After looking at the allocation table values (Figure 4)we can see that the necessary bits have been set to 1 and some bits have been set back to zero. This result is right considering our design for the allocation table and the input values.

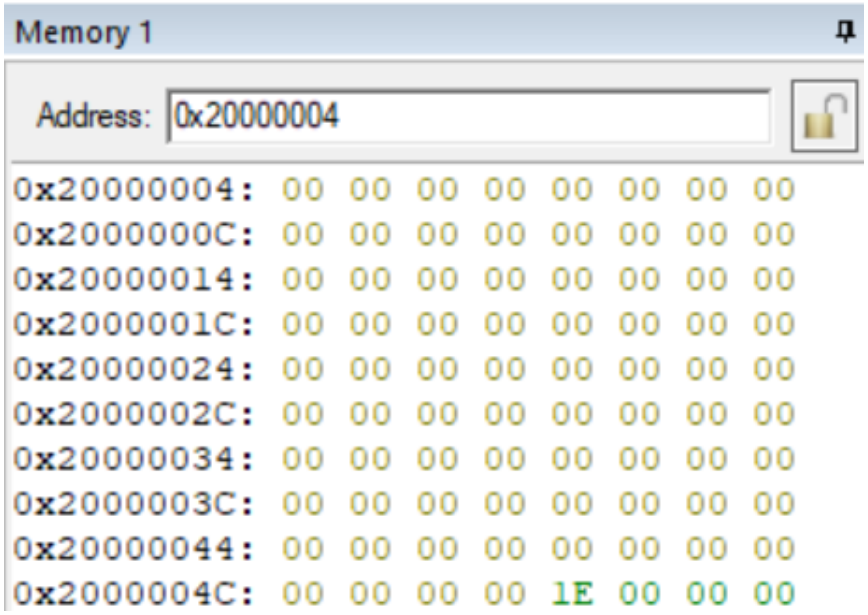

Figure 4: Allocation Table after the program

Looking at the Log Memory, we can clearly see that operation flags, error codes, tick count, input data and system time values have been stored successfully.

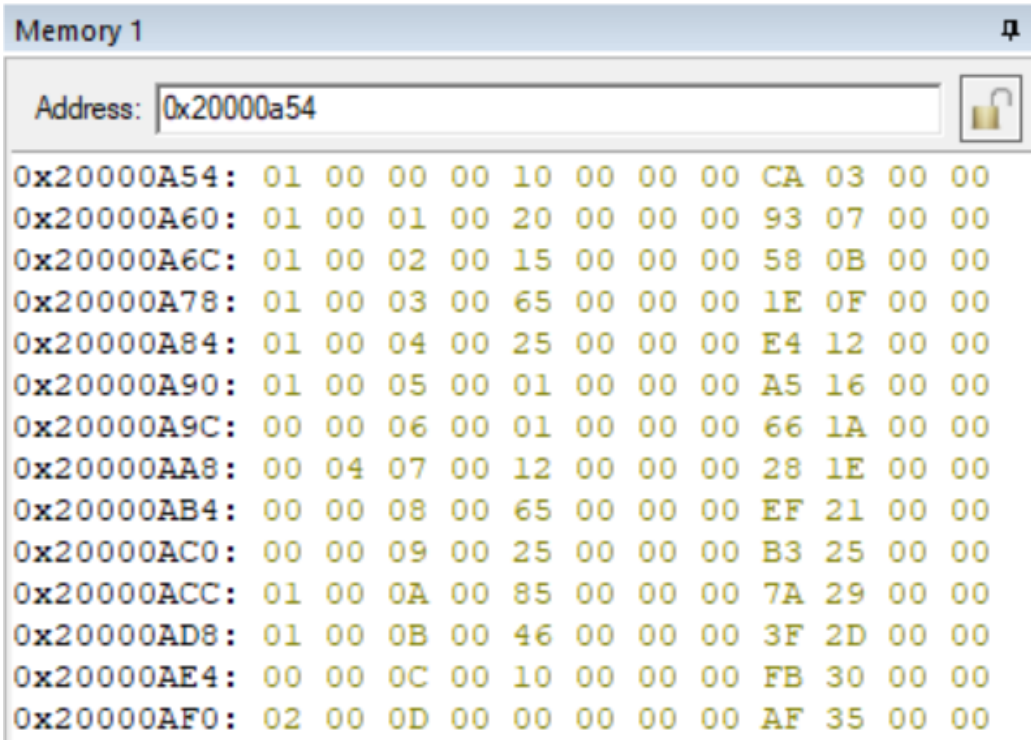

Figure 5: Log Memory after the program

Looking at the (Figure 6)we can see that the data that was once inside the linked list is now word by word located in the Array Memory.

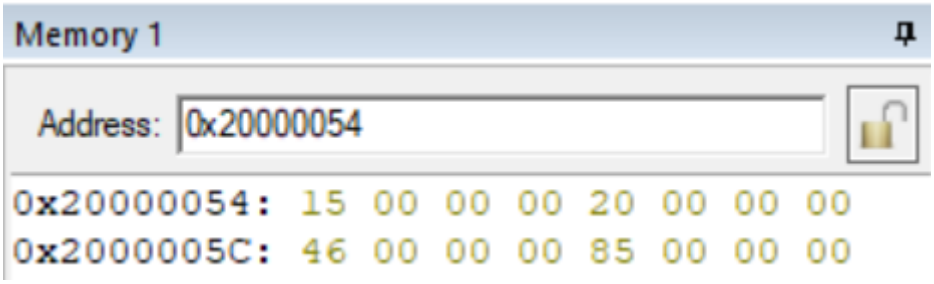

Figure 6: Array Memory after the program

4 DISCUSSION

Our noble team was able to produce the desired results without running into many issues. This can be attributed to our superb task management and teamwork. We divided the project into smaller tasks and planned the whole operation that way, which help us considerably throughout the project we must add. The main parts were: Configuring the systick and its related functions, inserting data to the list, removing data from the list and printing results. We implemented each of these parts in succession and handled their errors/log entries as we kept implementing them one by one. This workflow resulted in a smooth and favourable journey for our team. Each member has taken part in designing each of these different tasks. Hence, we were able to produce different ideas coming from different points of view. This situation was extremely fruitful when coming up with a design since it widened our options massively.
The results have been analysed in detail in part 3 of the report, but if we have to discuss them shortly here we can say that they were on par with our expectations. We were able to get the expected outcomes for different input datasets we created which is an end result that was quite satisfying four our team.

5 CONCLUSION

Our noble team strongly feels that, this was a quite eye opening implementation. Until this experiment, while we were writing code in c++ and other languages we were able to use the malloc() function by typing just one little keyword and we didn't really pay much attention to what was happening behind the scenes. But after doing this experiment, we acquired some inside information about these functions and understood what is happening inside them thoroughly. This newly gained experience will surely help not only our assembly implementations but will also improve our C/C++ coding abilities as well.

REFERENCES

https://developer.arm.com/documentation/dui0497/a/introduction/about-the-cortex-m0-processor-and-core-peripherals
https://www.keil.com/support/man/docs/armasm/
https://developer.arm.com/documentation/ddi0484/latest/
Assignment PDF found in NINOVA