ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 212E

MICROPROCESSOR SYSTEMS
TERM PROJECT

DATE	:	25.01.2021
GROUP NO	:	G15

GROUP MEMBERS:

150180082	:	TEVFIK OZGU
150170017	:	CEYHUN UGUR
150170021	:	OGUZHAN KARABACAK
150170005	:	SONER OZTURK
150180066	:	MUHAMMET AKCAN

FALL 2020

Contents

1 INTRODUCTION

In this project, it will be tried to create a linked list which orders incoming number in ascending order by using assembly. In this report, all operations will be explained briefly and assembly codes will be given.

2 MATERIALS AND METHODS

In this homework, assembly language is used to create linked list. The processor which was used is ARM Cortex M0+ and for this, Keil µVision 5 is used.

2.1 Reset Handler

In this part, it was tried to implement system tick handler by using own function. So for this purpose initially default system tick handler is taken in comment. After that in the reset handler new `SysTick_Handler` is imported. New reset handler is given at Listing 1.

Listing 1: Reset Handler

```
Reset_Handler PROC
           EXPORT  Reset_Handler          [WEAK]
           IMPORT  SystemInit
           IMPORT  __main
           IMPORT  SysTick_Handler ;This was imported

           LDR     R0, =SystemInit
           BLX     R0
           LDR     R0, =__main
           BX      R0
           ENDP
```

2.2 Clear Alloc

In this function it was tried to clear allocation table whether there is an redundant bit or not. The way to do it was creating a loop and writing zero to all bytes in the allocation table. These function's codes are given at Listing 2.

Listing 2: Clear Alloc Codes

```
           PUSH{R2,R3,R4,R5,LR}
```

```
            LDR R2, =AT_MEM    ;Load Allocation Table Addresses.
            MOVS R3, #0x00     ;Set Allocation Table to 0
            MOVS R4, #0        ;counter
            LDR R5, =AT_SIZE   ;Load Allocation Table Size
loopclear
            CMP R4, R5         ;if(r4 == r5)
            BEQ endclear       ;end
            STR R3, [R2,R4]    ;Store R3 To Allocation Table
            ADDS R4, #4        ;counter++
            B loopclear        ;back to loop
endclear
            POP{R2,R3,R4,R5,PC}
```

2.3 Write Error Log

In this Function we have loaded errors to an array called LOGMEM. The function works as it takes 4 parameters r0 as index of input dataset array, r1 as error code, r2 as operation and r3 as data. The algorithm is take the global variable index error and iterate a cursor 12 times the index (because each error takes 12 byte place). and then write the logs. following function does these operations.

```
        PUSH{r0,r1,r2,r3,r4,r5,r6,r7,lr}
            LDR r5,=LOG_MEM        ;load log_mem address to r5 register
            LDR r6,=INDEX_ERROR_LOG   ;r6 = where the new error will be stored
            LDR   r7,[r6]          ;take the value
            MOVS  r4,#12           ;r4 <- 12
            MULS  r7,r4,r7      ;r7 <- r4*r7
            ADDS  r5,r7,r5      ;r5 <- r7 + r5 find the start of error code
            STRH  r0,[r5]          ;write r0 as index of input dataset
            MOVS  r0, #0x02        ;r0 <- 2
            ADDS  r5,r0         ;increment the cursor
            STRB  r1,[r5]          ;write r1 as error code
            ADDS  r5,#1         ;increment cursor
            STRB  r2,[r5]          ;write r2 as operation
            ADDS  r5,#1         ;increment cursor
            STR   r3,[r5]          ;write r3 as data
            ADDS  r5,#4         ;increment cursor
            BL GetNow              ;get timestamp
            STR   r0,[r5]             ;write r0 as timstamp
```

```
LDR   r7,[r6]          ;take global index
ADDS  r7,#1          ;increment global index
STR      r7,[r6]          ;store global index
POP{r0,r1,r2,r3,r4,r5,r6,r7,pc} ;return
```

2.4 Clear ErrorLogs

This function is same with the clear alloc function. Only difference is that it clears not allocation table but error log array. This functions codes' are given at Listing 3.

Listing 3: Clear Error Logs Codes

```
          PUSH{R2,R3,R4,R5,LR}
          LDR R2, =LOG_MEM   ;Load Allocation Table Addresses.
          MOVS R3, #0x00     ;Set Allocation Table to 0
          MOVS R4, #0        ;counter
          LDR R5, =LOG_ARRAY_SIZE ;Load Allocation Table Size
loopclear2
          CMP R4, R5         ;if(r4 == r5)
          BEQ endclear2      ;end
          STR R3, [R2,R4]    ;Store R3 To Allocation Table
          ADDS R4, #4        ;counter++
          B loopclear2       ;back to loop
endclear2
          POP{R2,R3,R4,R5,PC}
```

2.5 Init GlobVars

In this function, in the global variables which are `TICK_COUNT`, `FIRST_ELEMENT`, `INDEX_INPUT_DS`, `INDEX_ERROR_LOG` and `PROGRAM_STATUS` stored 0. So this was done since there would be complicated situation because of initialization. These function codes' are given at Listing 4.

Listing 4: Init GlobVars

```
MOVS R2, #0        ;Set R3 to 0
LDR R3, =TICK_COUNT ;Load TICK_COUNT address to R2
STR R2, [R3]       ;Store R3 To Error Log Array
LDR R3, =FIRST_ELEMENT ;Load FIRST_ELEMENT address to R2
STR R2, [R3]       ;Store R3 To Error Log Array
LDR R3, =INDEX_INPUT_DS ;Load INDEX_INPUT_DS address to R2
```

```
STR R2, [R3]        ;Store R3 To Error Log Array
LDR R3, =INDEX_ERROR_LOG ;Load INDEX_ERROR_LOG address to R2
STR R2, [R3]        ;Store R3 To Error Log Array
LDR R3, =PROGRAM_STATUS ;Load PROGRAM_STATUS address to R2
STR R2, [R3]        ;Store R3 To Error Log Array
BX LR
```

2.6 SysTick Init

In this function reload value is stored onto reload value register which is at 0XE000E014. After that 0 value is stored to Current Value Register which is at 0XE000E018. After that, ENABLE, `TICKINT` and `CLKSOURCE` is made 1 which makes enable the counter, enable the SysTick exception request and selects tick timer as processor clock. Since these are last 3 bits of control register, 7 is stored into 0XE000E010 which is control register. And after that `PROGRAM_STATUS` is made 1.

2.6.1 Reload Value

The equation that gives period is:

$$\frac{1 + ReloadValue}{F_{CPU}} = Period$$

Period is given as 828 µs and F_{CPU} is given as 8 MHz. So when these values are used:

$$\frac{1 + ReloadValue}{8x10^6} = 828 * 10^{-6}$$

So when some simple operations are done, Reload Value can be found as (828 x 8) - 1 = 6623. As explained earlier this value is stored onto reload value register in this function.

2.7 System Tick Handler

In the handler, it was tried to implement a timer interrupt. As a brief explanation the way to achieve this function is making the current value register 0. So when current value register becomes 0, processor automatically interrupts and goes to systick handler. In this function delete, insert and finishing operations will be checked and managed to direct. In the first step, `EXPORT SysTick_Handler` is written to export this function as a handler for reset handler to import. Since Link Register is used to jump to other functions, it is pushed to stack initially. After that, `IN_DATA_FLAG`, `IN_DATA` and `TICK_COUNT` is read to decide which index to operate and what is the operation and data. From `TICK_COUNT`, it was read how many times system is interrupted. It is used as an index to identify which value to read from IN_DATA and IN_DATA_FLAG. These codes are given at Listing 5.

Listing 5: SysTick Handler Data Reading

```
EXPORT SysTick_Handler
PUSH {LR}
LDR R3, =IN_DATA_FLAG ;Load Input Data Flag Address to R32.
LDR  R0, =IN_DATA
LDR R1, =TICK_COUNT   ;Load TICK_COUNT Address
LDR R4, [R1]          ;R4 <- Index
LSLS R4, #2          ;R4 <- R4*4
LDR R2,[R3,R4]        ;R2 <- IN_DATA_FLAG[Index]
```

After this is read, it was decided which function to jump. If operation is delete which is coded as 0, it is jumped to remove function. If operation is 1, Insert function called and if operation is 2, SysTick_Stop function is called. These decision codes' are given at Listing 6.

Listing 6: SysTick Handler Decision Codes

```
              CMP  R2,#0              ;If Operation == Delete
              BNE  Add_Number         ;If Operation == Delete
              LDR  R0,[R0,R4]
              BL   Remove             ;Call Remove
              B    END_OF_TICK        ;If Removed, jump to END_OF_TICK
Add_Number
              CMP  R2,#1              ;If Operation == Add
              BNE  Finish_Program     ;If Operation != Add
              LDR  R0,[R0,R4]
              BL   Insert             ;Call Insert
              B    END_OF_TICK        ;If Inserted, jump to END_OF_TICK
Finish_Program
              CMP  R2,#2              ;If Operation == Finish
              BNE  NO_OP_FOUND        ;If Operation != Finish
              BL   LinkedList2Arr     ;Write values to linked list array
              BL   SysTick_Stop       ;Call Stop Timer
              B END_OF_TICK           ;finish handler
```

After all operations, it is jumped to END_OF_TICK label which checks whether any error is returned from functions or not if operation is delete, insert and stop. If operation is not found it is jumped to NO_OP_FOUND to make error register R0, 6. If there becomes an error, it is returned a number different from 0 in the R0 register. If there becomes an error, Index is stored into R0, Error code is stored in R1, Data is stored in R3 and it is jumped to `WriteErrorLog` function. After that, interrupt is finished by incrementing `TICK_COUNT` and `INDEX_INPUT_DS` accordingly and returned to main function by popping LR to PC. These codes are given at Listing 7.

Listing 7: SysTick Handler

```
NO_OP_FOUND
                MOVS R0,#6              ;If Operation is not found
END_OF_TICK
               CMP R0,#0               ;If there is no error
               BEQ   FINISH_INTERRUPT  ;then branch to FINISH_INTERRUPT
               MOV   R1,R0             ;R1 = Error Code
               MOV   R0,R4             ;R0 = Index of Input Data
               LDR   R3, =IN_DATA      ;R3 = Address of IN_DATA
               LDR   R3,[R3,R4]        ;R3 = Data
               BL WriteErrorLog        ;Branch to Error_Log
FINISH_INTERRUPT
               LSRS R4, #2             ;R4 <- R4/4 since it was shifted 2 bit
                  before
               ADDS R4,R4,#1;          ;Index+=1
               LDR R1, =TICK_COUNT     ;Load TICK_COUNT Address
               STR   R4,[R1]           ;TICK_COUNT += 1
               LDR R1, =INDEX_INPUT_DS ;Load INDEX_INPUT_DS Address
               STR   R4,[R1]           ;INDEX_INPUT_DS += 1
               POP   {PC}              ;Return to Main
```

2.8 Insert

We use the insert function to insert an element to the linkedlist. The function takes the new value as a parameter and returns an error code.

Listing 8: Insert

```
Insert         FUNCTION
            PUSH {r1,r2,r3,r4,LR} ;r0 is parameter for new value
            MOV   r1,r0          ;first r1<-r0, r0 will be used in malloc return
            LDR   r2,=FIRST_ELEMENT  ;Load FIRST_ELEMENT to r2
            LDR   r3,=DATA_MEM     ;Load DATA_MEM start address to r3
            LDR   r4,[r2]          ;Load FIRST_ELEMENT address to r4
```

```
CMP   r4,#0         ;First check linkedlist is empty, head == NULL
BNE   head_not_null    ;if head != NULL jump head_not_null
BL Malloc     ; Malloc function return free address with r0
CMP r0,#0           ;compare r0 is equal 0
BEQ list_full       ;if Malloc return 0, list is full jump list_full
STR   r1,[r0]    ;if not return 0,value be stored in return address
STR   r0,[r2]          ;The return address be stored in FIRST_ELEMENT
MOVS r5,#0          ;r5 <- 0 , to head point address is 0
STR   r5,[r0,#4]       ;head->next=0 ,
B not_error         ;then jump not_error
```

Since the R0 register will store the address returned from the malloc later, we assign the new value to the R1 register. FIRST_ELEMENT variable keeps the address of the head of the linkedlist. We load it into register R2. Since DATA_MEM is the address of the array, we load it into the R3 register. Since FIRST_ELEMENT keeps the address of the head, we load the head value into the R4 register.

Then we compare R4 with 0 to check if it is head 0 or not. If R4 is not 0, the linkedlist is not empty, so jump to head_not_null.

If Linkedlist is empty, we call Malloc to return an empty address. We compare the return value with 0. If there is no allocable area, jump to list_full.

If Malloc returns an address that is not 0, we store the new value at this address. Then we store this address in FIRST_ELEMENT for head to point to this address. Then we store the 4 index of the address, ie the address where the next node is kept, r5 value, 0. Then jump to not_error.

Listing 9: Insert head_not_null Block

```
head_not_null LDR r4,[r4]     ;if head is not null r4 <- head_value
          CMP   r1,r4      ;Compare new_value and head_value
          BGE   is_equal_head ;if new_value is not smaller jump is_equal_head
          BL Malloc        ;if smaller,malloc return free address with r0
          CMP r0,#0        ;compare r0 is equal 0
          BEQ list_full    ;if Malloc return 0,list is full jump list_full
          STR   r1,[r0]     ;if not return 0,new_value stored in return address
          ADDS r0,r0,#4    ; r0 <- r0 + 4 to access node address
          LDR r4,[r2]      ;Take FIRST_ELEMENT address, head address
          STR   r4,[r0]    ;Address of old header is pointed by new node
          SUBS r0,r0,#4    ;r0 <- r0 - 4 to access node address
          STR   r0,[r2]    ;store r0 value to FIRST_ELEMENT because it is head
```

```
          B not_error      ;then jump not_error
```

The head_not_null block specifies that head is not 0. First, we get the value stored by the address stored by FIRST_ELEMENT, that is, we get the value where the head points. And we compare this value with new_value. If new_value is not smaller than the value of head, jump to is_equal_head. But if it's small, then we first call malloc function to return empty address and its return value is kept at R0 register.
If the return value is 0, then there is no allocable area. In this case, jump to list_full. If R0 register is not 0, then we store the new_value in the return address. and we add 4 to the address value to get the address of new_node. Then we load the value kept by FIRST_ELEMENT, that is the address of head, to R4 register, then we load this R4 register value to the address part of the new node. So it is new_node - next = head.
Then we subtract 4 from the address value and reach the address of the node and load this value into FIRST_ELEMENT, so the value of R0 register becomes the new head. Then jump to not_error.

Listing 10: Insert is_equal_head Block

```
is_equal_head CMP r1,r4       ;compare head_value and new_value is equal
          BNE  bigger_head ;if is not equal jump bigger_head
          MOVS  r0,#2      ;if is equal this is error so r0 <- 2
          B return      ;then jump return
```

In the is_equal_head block, we check whether new_value is equal to the value of head. If not equal, jump to bigger_head, but if it's equal we assign error code 2 to R0 register and jump to return.

Listing 11: Insert bigger_head Block

```
bigger_head   LDR r5,[r2]        ;r5 = iter = head to traverse in linkedlist
loop      ADDS r5,r5,#4    ;r5 <- r5+4 to access address
          LDR r4,[r5]      ;take next node address
          CMP r4,#0        ;compare address and 0,
          ;if zero,the iter in the end of linkedlist
          BEQ not_equal    ;if it is equal, jump not_equal
          LDR r6,[r4]      ;if it is not equal,load r4 value to r6
          CMP r6,r1        ;compare new value and node value
          BGE end_of       ;if new value is smaller,jump to end_of
          MOV r5,r4        ;if not smaller, iter = iter->next
          B loop           ;then jump loop
```

We traverse on the linkedlist to find a suitable place for the new value to be inserted in the bigger_head block.
First of all, to traverse on the linkedlist, we assign the head value to the R5 register. This register will be used as Iter. Then we add 4 to the R5 register to access the address part of where the head points. Then we take the address of the next node that the iter points to and assign to R4 register.
If R4 register is 0, then Iter means at the end of the linkedlist. and getting jump to not_equal.
But if number is not 0, we get the value Node keeps and compare this value with new_value. If the value kept by the node is smaller, we jump to end_of. If not,the address of the next node be assigned to iter, R5 register and jump to the loop.

Listing 12: Insert end_of Block

```
end_of      LDR r6,[r5]   ;load current_node address to r6 register
            LDR r6,[r6]   ;and take current_node value
            CMP r6,r1     ;check iter->value == new_value
            BNE not_equal ;if not equal to new_value, jump not equal
            MOVS r0,#2    ;if is equal this is error so r0 <- 2
            B return   ;jump return
```

In the end_of block, we check whether the value of the node that iter points to and the value to be inserted are equal.
For this, first we load the address of iter into register r6. Then we load the value of iter's address point to the register R6 register again.
Then we compare the R6 and R1 registers. If these values are not equal,jump to not_equal. If it's equal we load the 2 error code into R0 register and jump to return.

Listing 13: Insert not_equal Block

```
not_equal     LDR r6,[r5]   ;load current_node address to r6 register r6 <- iter
            BL Malloc     ;call malloc
            CMP r0,#0     ;compare malloc return value with 0,
            BEQ list_full ;if malloc return 0 , list is full so jump list_full
            STR r1,[r0]   ;if malloc not return 0, new value stored in address
            ADDS r0,r0,#4  ;r0 <- r0+4 to access node point address
            STR r6,[r0]   ;new_node->address = iter->address
            SUBS r0,r0,#4  ;r0 <- r0-+ to access node address
            STR   r0,[r5]  ;iter->address = new_node
            B not_error   ;then jump not_error
```

If it comes to the not_equal block, it means that we find a suitable place for the value to be inserted and the value to be inserted is different from any value in the list.
First, we load the address pointed by iter into the R6 register. After We call the Malloc function to return the address to be inserted.
If Malloc function returns 0, there is no allocable area. So jump to list_full. If Malloc does not return 0, it returns a new address, we first store the new_value at this address. Then we add 4 to R0 register to access the address part of new_node. Then we store the address pointed to by iter in R0 address. We subtract 4 from R0 register to get the address of new_node. and we store this value of R0 register in the address of iter. Then we jump to not_error.

Listing 14: Insert End

```
list_full     MOVS r0,#1    ;if list_full this error code is 1
          B return    ;jump return

not_error     MOVS r0,#0    ;if not error error code is 0
return        POP {r1,r2,r3,r4,PC} ;return r0
```

In the list_full block, we assign 1 error code to R0 register then jump to return. In not_error block,we assign 0 no error code to R0 register. End of this function , in return block, the function returns R0 register.

2.9 Remove

We use the remove function to remove an element from the linkedlist. The function takes the new value as a parameter to delete and returns a success code or an error code if exists.

Listing 15: Delete from Head

```
          PUSH {r1,r2,r3,r4,LR}      ;data which will be removed
          LDR     r1, =FIRST_ELEMENT    ;determine head of the linked list
          LDR     r2,[r1]            ;load value of the FIRST_ELEMENT address
              to the r2(iter)
          LDR     r3, [r2]        ;load value of the r2 address to the r3
          CMP     r2, #0             ;control linked list is empty or not?
          BEQ     go_to_error_1      ;if r2 == NULL, then go to the
              go_to_error_1 label
          CMP     r0, r3             ;control (param ?= iter->data)----->
              BASTAN SIL
```

```
        BNE     while           ;if not equal, go to while label
        LDR     r3, [r2, #4]       ;load the next smallest element to the r3
        MOV     r0, r2             ;move r2 to the r0 to send the parameter
           to the free function
        STR     r3, [r1]        ;determine the new head element
        BL     Free            ;call free function
        MOVS  r0, #0           ;determine the success code as error code
        POP {r1,r2,r3,r4,PC}    ;return
```

If we want to remove element from head, we will work in these lines. `FIRST_ELEMENT` variable keeps the address of the head of the linked list. We loaded it into r1 register. Then, we loaded the value of the r1 register to the r2 register. Then, we loaded the value of the r2 register to the r3 register. This means that, r3 register keeps the value of the head of the linked list. Then, we are comparing the r2 register with '0' immediate value to determine is there any element in the linked list or not. If r2 register is '0', this means that the linked list empty and the head of the linked list is NULL. So, if it is '0', we are going to `'go_to_error_1'` label.

Listing 16: Error-1 of the Remove Function

```
go_to_error_1
        MOVS  r0, #3           ;ERROR (LINKED LIST IS EMPTY)
        POP {r1,r2,r3,r4,PC}
```

In this code block, we are loading '3' immediate value to the r0 register and return it. Error code '3' means that the linked list is empty.

Listing 17: Delete from Head

```
        CMP     r0, r3             ;control (param ?= iter->data)----->
           BASTAN SIL
        BNE     while           ;if not equal, go to while label
        LDR     r3, [r2, #4]       ;load the next smallest element to the r3
        MOV     r0, r2             ;move r2 to the r0 to send the parameter
           to the free function
        STR     r3, [r1]        ;determine the new head element
        BL     Free            ;call free function
        MOVS  r0, #0           ;determine the success code as error code
        POP {r1,r2,r3,r4,PC}    ;return
```

Then, we are comparing r0 register with r3 register. r0 means that the parameter of the remove function that takes the data to remove it. If these two registers are equal, then we are going on. We are loading the next smallest element after head element of the linked list to the r3 register. Then we moved the value of the r2 register to the r0 register to pass a parameter to the free function. Before calling the free function, we are determining the new head element using str function to store r3 register in the value of the r1 register. Then, we are calling the free function. After free function, we are moving the success code to the r0 register and returning it.

Listing 18: Delete from Middle

```
while
        LDR     r3, [r2, #4]        ;r3 = r2->next
        CMP     r3, #0              ;control r4 is NULL or not
        BEQ     end_while           ;if r4 is NULL, then go to the end_while
            label
        LDR     r4, [r3]         ;load the value of the address to the
            r4(iter->next->data)
        CMP     r4, r0              ;control (param =?
            iter->next->data)-----> ORTADAN SIL
        BNE     second_if           ;if r4 and r0 is different, go to the
            second_if label
        LDR     r4, [r3, #4]        ;r4 = iter->next->next
        MOV     r0, r3              ;r0 = iter->next
        STR     r4, [r2, #4]        ;iter->next = r4
        BL    Free              ;call free function
        MOVS  r0, #0               ;determine the success code as error code
        POP {r1,r2,r3,r4,PC}     ;return
```

If the r0 register which is parameter of the remove function, is not equals to the value of the head element of the linked list, then we are going to the while label. In while label, we are loading the next element of the r2 register ,which means that the iter pointer of the linked list, to the r3 register. After that, we are comparing the r3 register with the '0' immediate value. If they are equal, this means that finish the while loop and go to the last block of the function which is 'deleting from the end of the linked list'. If they are not equal, then load the value of the r3 register to the r4 register. Then controll r4 register with the parameter r0 register. If, they are not equal, this means that go to the `second_if` label. If they are equal, this means that we will remove the value of the r4 register. So, we are loading the next element of the next element of the iter pointer to the r4 register and we are moving the next element of the iter pointer to the r0 register. Then, we are making the r4 register as a next element of the iter pointer which is r2 register. Then, pass the r0 register to the free function as a parameter. After that, we moved '0' success code to the r0 register and returned it.

Listing 19: Delete from Middle('second_if' label)

```
second_if
        LDR     r3, [r3, #4]        ;r3 = iter->next->next
        CMP     r3, #0              ;control (iter-next->next =? NULL)
        BNE     end_second_if       ;if not, go to the end_second_if
        MOV     r1, r2              ;r1(keep) = iter
end_second_if
        LDR     r2, [r2, #4]        ;iter = iter->next
        B     while             ;go to the while label
```

In the 'second_if' label, we are loading the next element of the r3 register ,which is the next element of the next element of the iter pointer, to the r3 register again. Then, we are comparing r3 register with the '0' immediate value. If they are equal, then we are moving the r2 register to the r1 register. r1 register means that, the penultimate element in the linked list. If they are not equal, we are not going to do this, directly we sare loading the next element of the r2 register to the r2 register to make traverse operation in the while loop of the linked list. Then we are going to 'while' label again.

Listing 20: Delete from End

```
end_while
        LDR     r3, [r2]          ;r3 = iter->data
        CMP     r0, r3              ;control (param =? iter->data)----->
           SONDAN SIL
        BNE     go_to_error_2       ;if not, go to the go_to_error label
        MOV     r0, r2              ;r0 = iter
        MOVS  r4, #0              ;load 0 to r4 register
        STR     r4, [r1, #4]        ;r4 = NULL
        BL    Free              ;call free function
        MOVS  r0, #0              ;determine the success code as error code
        POP {r1,r2,r3,r4,PC}          ;return
go_to_error_2
        MOVS  r0, #4              ;ERROR (ELEMENT IS NOT IN THE LINKED LIST)
        POP {r1,r2,r3,r4,PC}
```

If we want to remove an element from the end of the linked list, we will work on these lines. In the `end_while` label, firstly, we are loading the value of the r2 register to the r3 register. Then we are comparing the r3 register with the r0 register with parameter of the remove function. If they are equal, this means that there is no problem we can remove the last element of the linked list. We are moving r2 register to the r0 register to pass as a parameter to the free function. Then we are moving the '0' immediate value to the r4 register and we are storing this value of the r4 register in the next element of the r1 register. After that, we are calling the free function and we are moving the '0' success code to the r0 register and we are returning it.

Listing 21: Error-2 of the Remove Function

```
go_to_error_2
          MOVS  r0, #4              ;ERROR (ELEMENT IS NOT IN THE LINKED LIST)
          POP {r1,r2,r3,r4,PC}
```

If the r3 register is not equals to the r0 register, this means that we couldn't find the element which will be removed from the linked list. Thus, we are moving the '4' immediate value which is the error code of the 'element is not in the linked list', and we return it as an error code.

2.10 Malloc

Malloc function is used for finding the unused memory node and allocating it using the allocation table. Allocation table has NUMBER_OF_AT words and each word has 32 bit. Each bit in the table points to one memory node in DATA_MEM array and each node has 2 words which equals to 8 bytes. If a bit in the allocation table is 0, that means pointed node by that bit is not allocated until now(empty). If a bit in the allocation table is 1, that means pointed node by that bit is already allocated. The function returns the memory address of the first possible memory node which is in DATA_MEM array. If it returns 0 as the address, that indicates an error which means all bits in the allocation table is 1, and that means the function cannot allocate any area, so the linked list is full. Malloc function consists of 3 parts: finding the index of the first 0-bit in the allocation table, calculating the address of the first possible empty place in the DATA_MEM array and changing the 0-value of the bit to 1 since the pointed memory is allocated after that now.

Listing 22: Finding Index

```
          PUSH{r1,r2,r3,r4,r5,r6,r7,lr}
          LDR r3, =AT_MEM           ;load allocation table array addresses
          LDR r6, =AT_SIZE       ;load allocation table size
          MOVS r4, #0            ;index for looping in at_mem array
          MOVS r1, #0            ;counter1 for number of bytes controlled in
              at_mem array
```

```
LOOP1       CMP r6, r1              ;if number of bytes controlled = at_size
            BEQ end_of_array        ;  end of at_mem array is reached, which means
               linkedlist is full
            LDRB r5, [r3, r4]         ;load AT_MEM[r4] byte by byte
            ADDS r4, r4, #1           ;index++
            MOVS r2, #0             ;counter2 for number of bits controlled in 1
               byte
            ADDS r1, r1, #1           ;counter1++
LOOP2       CMP r2, #8              ;if number of bits controlled = 8
            BEQ LOOP1               ;  all bits controlled in loaded 1 byte,
               branch to loop1 to load next bye
            MOVS r7, #0xFE          ;1111 1110, 2nd operand of OR
            ORRS r7, r5, r7           ;R7 = R7 OR R5
                                  ;except LSB all bits in loaded byte will be 1,
                                     and LSB dont change
            LSRS r5, r5, #1           ;shift right R5 to load the bit, on left
               side of the controlled bit, to LSB
            ADDS r2, r2, #1           ;counter2++
            CMP r7, #0xFF           ;compare r7 and FF, r7 = 1111 111x where x is
               the LSB of R5
            BEQ  LOOP2              ;if x=1 then branch to loop2 again to check
               remained bits
                                  ;else(if x=0) that means we found an empty place

            ;calculations for finding the index number of the first 0 bit in
               allocation table with using counter1 and counter2
            SUBS r1, r1, #1           ;counter1-- since it was increased one more
               time in the last loop1
            MOVS r7, r1             ;in order not to lose the value of r1, r7=r1
               is done, r7 will be used later
            MOVS r6, #8             ;r6 = 8
            MULS r1, r6, r1           ;r1 = r1*8, now r1 holds the num of bits
               controlled except the last controlled byte
            SUBS r2, r2, #1           ;r2-- since we want to start indexing bits
               from 0, not 1
            ADDS r1, r1, r2           ;r2 hold the bits controlled in the last
               taken byte
                                  ;add it to the r2 to calculate all num of bits
                                     controlled
            ;if the rightmost bit of allocation table is index0, r1 holds now
```

```
            first 0 bit from rightside
```

In the code above, finding the index number of the 0-bit in the allocation table is done. r1 counts the number of bytes controlled in the at_mem array. In the loop1, we are traversing in the at_mem array byte by byte. It is done with LDRB for loading just one byte. Loop1 continues until the end of the of at_mem array or finding a valid place. R2 holds the number of bits controlled in the current byte. If it reaches to 8, that means end of the byte so it branches to loop1 in order to get the next byte. If not, OR operation is done with the operands current byte and 1111 1110 value. After this r7 now holds a value like 1111 111x where x is the least significant bit of the r5. Then r5 which holds the current byte, is shifted right. If r7 = 1111 1111 that means current bit is allocated before and it branches to loop2 to get the next bit. Shift operation is done in loop2 for moving the next bit to the least significant bit in order to be ORed and checked again. If r7 = 1111 1110 that means the current bit is 0 so it is not allocated before. Since the first 0-bit found, it goes out of loop. In order to calculate the index of the found bit, the values in the r1 and r2 is used since r1 holds the number of controlled bytes and r2 holds the number of the controlled bits in the last byte. First, r1 is decreased by one since in the last loop1 it was increased one more time. So r1 holds the number of bytes including the last controlled byte but controlling last byte is not finished yet. That is way it is decreased. And in order not to lose the value in the r1, it is loaded to r7 because later it will be needed. R1 is multiplied with 8 since each byte has 8 bits. R2 is decreased by one as the same reason for r1. And it was added to r2 to calculate the index of the found 0-bit in the allocation table.

Listing 23: Address in Data_Mem array

```
;calculations for finding the corresponding address in DATA_MEM array
MULS r1, r6, r1          ;r1=r1*8 since each bit in allocation table
   corresponds 8 byte in DATA_MEM array
LDR r0, =DATA_MEM     ;load address of DATA_MEM
ADDS r0, r1, r0          ;r0=r0+r1 in order to find the corresponding addres
   in DATA_MEM array
```

In order to calculate the corresponding address in the data_mem array, the index of the found 0-bit is multiplied with 8 since each bit in the allocation table corresponds to 8 byte in the Data_Mem array and r1 now holds the index for the data_mem array. Index value is added to the address of the Data_Mem array and loaded to r0 since r0 returns the address of the first possible memory area.

It is time for changing 0-bit to the 1 since it is allocated after now.

Listing 24: Changing 0 to 1

```
;in the allocation table, found bit which holds 0, should be changed to 1
    since it will not be empty afterwards
MOVS r6, #0x01         ;r6 = 0000 0001
LSLS r6, r6, r2           ;shift left r6, r2 times(r2 holds num of bits
    controlled in the last byte)
LDRB r5, [r3,r7]       ;load the value of corresponding address in allocation
    table
ADDS r5, r5, r6           ;change 0 to 1;for example, if taken byte is 1100
    1111 then 0001 0000 is added to it
STRB r5, [r3,r7]       ;store the changed value to corresponding address
POP{r1,r2,r3,r4,r5,r6,r7,pc} ;return
```

0x01 is moved to r6 and r6 is shifted left r2 times where r2 holds the controlled bits in the last taken byte which also means 0-bit's index in the last taken byte. Then r7th byte is loaded to r5 where r7 holds the number of bytes called which also means the index of the byte that holds the first 0-bit. Shifted value of 0x01 is added to taken byte and stored again the same memory area. For example if taken byte is 1011 0111 then r2 holds 3 since 0 bit in the 3rd place of the byte (started with 0 index) and 3 bits controlled until 0 (which are 111). So 0000 0001 value is shifted left 3 times and it generates 0000 1000 value. If this value added to taken byte it results with 1010 1111 value. As seen the first 0-bit is changed with 1. For the clarification, if the first 4 byte of the allocation table is FF FF F7 F0 then in this case r7 is 2 and r1 is 4.

In the loop1, if r1 = r6 which means end of at_mem reached, also means there is no 0-bit in the allocation table so the linked list is full, program branches to the part below, and 0 value is returned with in r0 as an address indicating there is no empty memory area in the data_mem array.

Listing 25: No empty place

```
end_of_array  MOVS r0,#0              ;r0 = 0, no empty place, linked list full
            POP{r1,r2,r3,r4,r5,r6,r7,pc} ;return
```

2.11 Free

In this function we have taken r0 as the index of the freed value. This function takes address of the deallocate value and then clears the bit of that value from ATMEM array. Then Clears the value from DATAMEM array. Firstly we find in at table which register should be changed. To do this we firstly find the index of the bit to be freed then convert it into byte + bits form. and store those values at different registers. then take the correct index byte then change it and store back to ATMEM. Following code does these operations.

```
;r0,r1,r2,r3,r4,r5 were used
PUSH {R0,R1,R2,R3,R4,R5,r6,LR}
LDR r1,=AT_MEM;r1 <- AT_MEM
MOVS r6,#0;r6 <- 0
LDR r2,=DATA_MEM;r2<-DATA_MEM
STR  r6,[r0];store to r0
STR  r6,[r0,#4];store
SUBS r0,r0,r2 ;calculate the distance to the begining of the data mem
    array
LSRS r0,r0,#3 ;divide to find index
LSRS r3,r0,#3 ;find appropiate byte
LSLS r4,r3,#3 ;temp to subtract
SUBS r0,r0,r4 ;find which bit will be freed
LDRB r4,[r1,r3] ;take at mem register to change
MOVS r5,#1;r5<-1
LSLS r5,r5,r0;find which bit will be change
EORS r4,r4,r5;change the bit
STRB r4,[r1,r3];store back to at mem
POP {R0,R1,R2,R3,R4,R5,r6,PC};return
```

2.12 LinkedList2Arr

In this function we have converted the given linked list to normal array. To do this we iteratively traversed the linked list and store each value to array. then removed it and its next pointer from the linked list. before we started to loop we loaded 5 value to error register and in loop we changed it to 0 because if the algorithm never get into the loop that means the linked list is empty. So we should give linked list is empty error. And when it get into the loop then there is no error so we say there is no error and load 0 to error register. Following code does these operations.

```
PUSH{r1,r2,r3,r4,lr}
LDR     r1,=FIRST_ELEMENT  ;r1 = head
LDR     r2,=ARRAY_MEM      ;r2 = array
LDR     r1,[r1]            ;r1 = linked list element value
MOVS  r3,#0                ;index
MOVS  r4,#5                ;error code
```

```
loopa       CMP      r1,#0               ;compare r1 with 0
            BEQ      finish              ;if equal
            MOVS  r4,#0                  ;error flag
            LDR      r0,[r1]             ;r0 is data of the node
            LDR      r1,[r1,#4]          ;r1 is the next node's address
            STR      r0,[r2,r3]          ;store the value to the array
            ADDS  r3,#4                  ;r3 = r3 + 4
            BL     Remove                ;call remove function
            B      loopa                 ;jump to loopa
finish      MOVS     r0,r4               ;r0 = r4
            POP{r1,r2,r3,r4,PC}          ;return with r0
```

2.13 GetNow

In this function it was asked to find the error time occurred in microseconds. For this purpose initially TICK_COUNT is multiplied by the period to how many times passed until the last timer interrupt. So this was kept in one register. After that, as given before period function is given below. But remember that in this equation Reload Value is changed with the difference between Reload Value Register and the current value register to see how many clock is executed in the last interrupt.

$$\frac{1 + ReloadValue - CurrentValue}{F_{CPU}} = Time(us)$$

So for our system Time is found as:

$$\frac{1 + 6623 - CurrentValue}{8} = Time(us)$$

After that this last time is added to initial found number which was calculated by multiplying period and TICK_COUNT. These function codes are given at Listing .

Listing 26: GetNow Function

```
PUSH{R1,R2,R3,LR}      ;Push Registers and LR to SP
LDR R1, =TICK_COUNT    ;Load TICK_COUNT Address
LDR R2, [R1]           ;R2 <- Index
LDR R3, =828           ;R3<-Period
MULS R2, R3, R2        ;R2 <- Period * Index
LDR R1,=0XE000E010     ;Load SysTick Control and Status Register Address
LDR R3,[R1,#4]         ;Load Reload Value to R3.
LDR   R0,[R1,#8]       ;Load Current Value Register to R0
SUBS R0, R3, R0        ;R0 <- Reload Value Register - Current Value Register
```

```
ADDS R0, R0, #1       ;R0 <- R0 + 1
LSRS R0, #3           ;R0 <- R0/8
ADDS R0, R0, R2       ;R0 <- R0 + Period * Index
POP{R1,R2,R3,PC}      ;Return By LR
```

3 RESULTS

In the Experiment we were expected to generate a linked list which has insertion, removing and converting linked list to array. To test our algorithm we tried different edge cases to prove that our algorithm works true. To achieve this we tried our algorithm with 3 different data set. For the first data set we have tried non defined operation, deletion from empty linked list, converting empty linked list to array. Data set 1 is as follows;

```
;@brief   This data will be used for insertion and deletion operation.
;@note    The input dataset will be changed at the grading.
;      Therefore, you shouldn't use the constant number size for this
    dataset in your code.
          AREA    IN_DATA_AREA, DATA, READONLY
IN_DATA      DCD     0x10, 0x20, 0x15, 0x65, 0x25, 0x01, 0x01, 0x12, 0x65,
    0x01, 0x18, 0x18, 0x15, 0x25, 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     0x03, 0x00, 0x01, 0x01, 0x01, 0x01, 0x01, 0x00, 0x00,
    0x00, 0x01, 0x00, 0x00, 0x00, 0x02
END_IN_DATA_FLAG
```

And the operation we have done for data set 1 has been illustrated on the figure 1.

Data	DataFlag	ErrorCode	LinkedList	Explanation
0x10	0x03	6	-	Operation not defined
0x20	0x00	3	-	Deletion on empty list
0x15	0x01	0	0x15	Insertion of first element
0x65	0x01	0	0x15 – 0x65	Inserting to end of the list
0x25	0x01	0	0x15 – 0x25 – 0x65	Insertion into the list
0x01	0x01	0	0x01 – 0x15 – 0x25 – 0x65	Inserting to head
0x01	0x01	2	0x01 – 0x15 – 0x25 – 0x65	Insertion of same data
0x12	0x00	4	0x01 – 0x15 – 0x25 – 0x65	Deletion of non-existing data
0x65	0x00	0	0x01 – 0x15 – 0x25	Deletion from end of list
0x01	0x00	0	0x15 – 0x25	Deletion of head
0x18	0x01	0	0x15 – 0x18 – 0x25	Insertion into the list
0x18	0x00	0	0x15 – 0x25	Deletion
0x15	0x00	0	0x25	Deletion of head
0x25	0x00	0	-	Deleting the last element in the list
0x00	0x02	5	-	Linked list could not be transformed

Figure 1: data set 1 operations

So we were expecting to see the errors shown in Figure 1.0. And our log mem array can be seen in Figure 2.

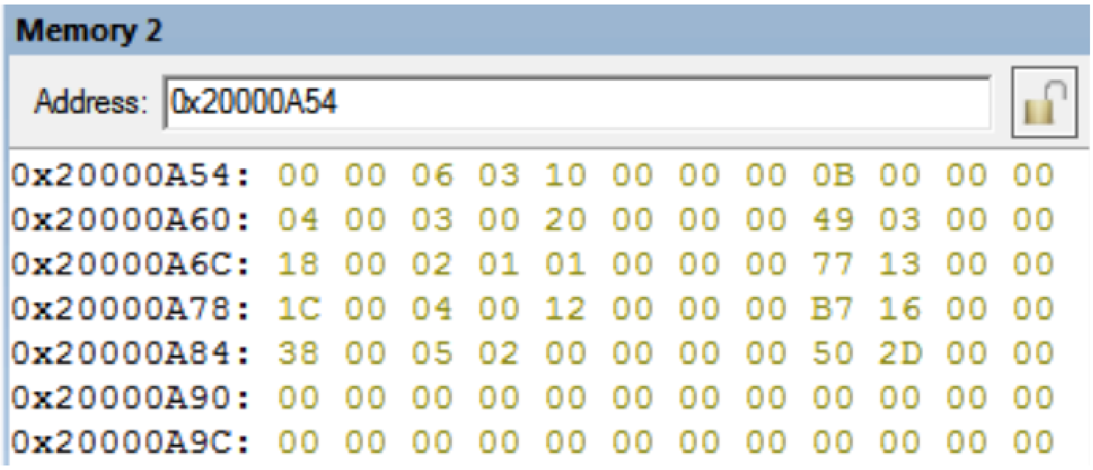

Figure 2: Log_mem array of data set 1

So it can be seen that we had true errors. So our algorithm works fine for data set 1 edge cases.

In second data set we tried to add element to a fulled linked list. So all of our operations are insertion and for the simplification NUMBER_OF_AT is changed as 1. That means at most 32 data can be added to linked list. Data set 2 is as follows.

```
;@brief   This data will be used for insertion and deletion operation.
;@note    The input dataset will be changed at the grading.
;         Therefore, you shouldn't use the constant number size for this
    dataset in your code.
          AREA    IN_DATA_AREA, DATA, READONLY
IN_DATA      DCD      0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18, 0x19,
    0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x21, 0x22, 0x23,
    0x24, 0x25, 0x26, 0x27, 0x28, 0x29, 0x30, 0x41, 0x40, 0x39, 0x38, 0x37,
    0x36, 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, 0x01, 0x01, 0x01,
    0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01,
    0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01, 0x01,
    0x01, 0x02
END_IN_DATA_FLAG
```

So we should have logged insertion to fulled linked list. And our result can be seen from the figure 3 so our algorithm works fine for this case too.

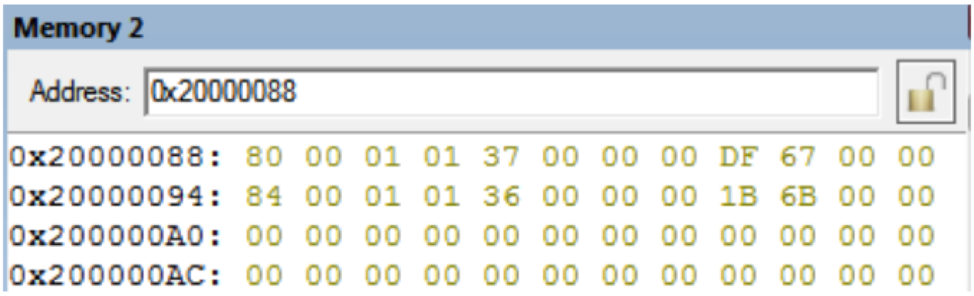

Figure 3: Log_mem array of data set 2

And after these operation we converted linked list to array. So we stored these values on data array. And our result can be seen from the figure 4. So our system work correctly.

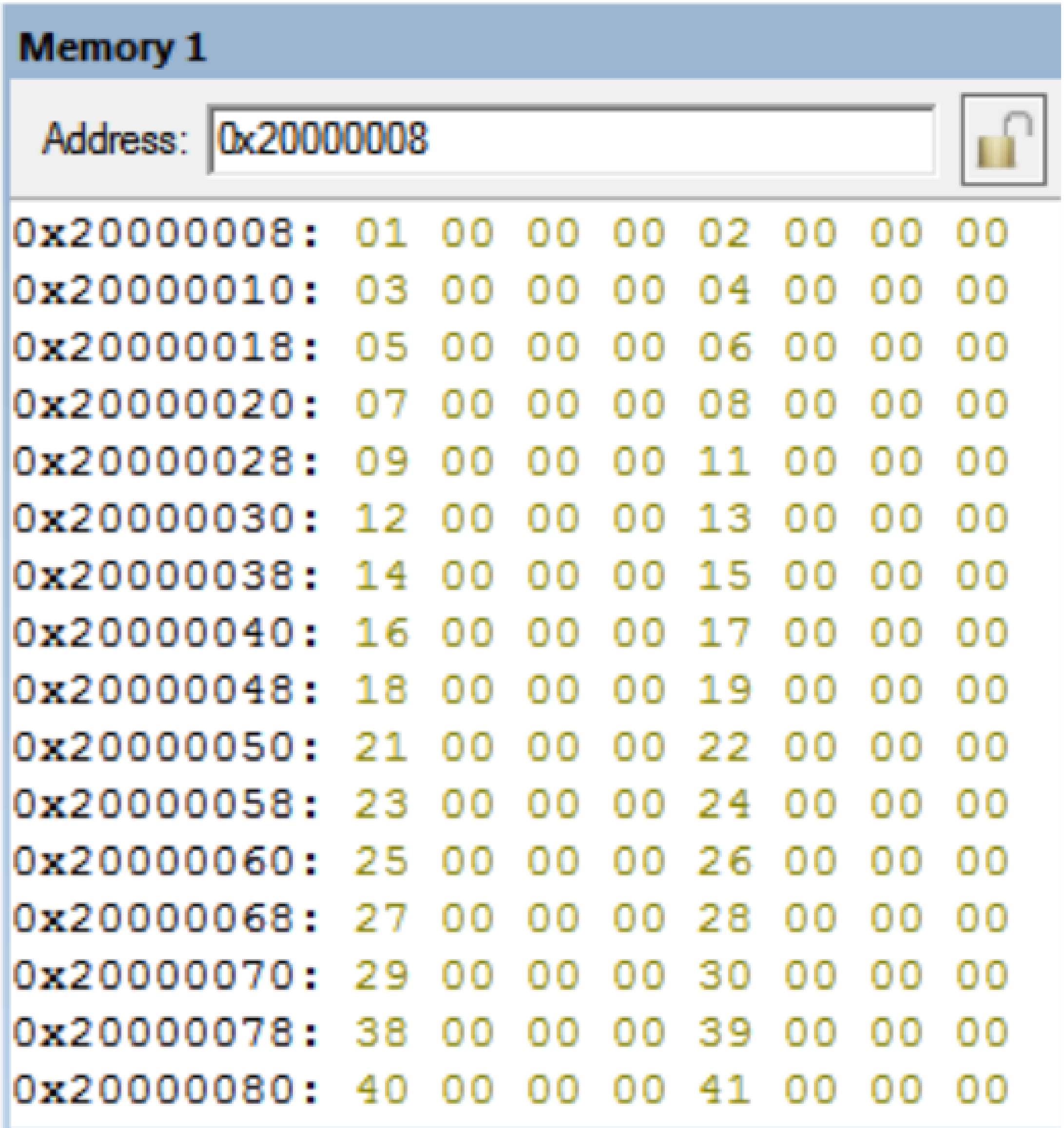

Figure 4: Array_mem array of data set 2

And lastly we tried the given data set on the homework file. Data set 3 is as follows:

```
;@brief   This data will be used for insertion and deletion operation.
;@note    The input dataset will be changed at the grading.
;      Therefore, you shouldn't use the constant number size for this
    dataset in your code.
          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
```

The operations on data set 3 can be seen in the figure 5.

Data	DataFlag	ErrorCode	LinkedList	Explanation
0x10	0x01	0	0x10	Insertion of first element
0x20	0x01	0	0x10-0x20	Inserting to end of the list
0x15	0x01	0	0x10 – 0x15 – 0x20	Insertion into the list
0x65	0x01	0	0x10 – 0x15 – 0x20 – 0x65	Insertion into the list
0x25	0x01	0	0x10 – 0x15 – 0x20 – 0x25 – 0x65	Insertion into the list
0x01	0x01	0	0x01 – 0x10 – 0x15 – 0x20 – 0x25 – 0x65	Inserting to head
0x01	0x00	0	0x10 – 0x15 – 0x20 – 0x25 – 0x65	Deletion of head
0x12	0x00	4	0x10 – 0x15 – 0x20 – 0x25 – 0x65	Deletion of non-existing data
0x65	0x00	0	0x10 – 0x15 – 0x20 – 0x25	Deletion from end of list
0x25	0x00	0	0x10 – 0x15 – 0x20	Deletion from end of list
0x85	0x01	0	0x10 – 0x15 – 0x20 – 0x85	Inserting to end of the list
0x46	0x01	0	0x10 – 0x15 – 0x20 – 0x46 – 0x85	Insertion into the list
0x10	0x00	0	0x15 – 0x20 – 0x46 – 0x85	Deletion of head
0x00	0x02	0	0x15 – 0x20 – 0x46 – 0x85	Transforming to linked list

Figure 5: operations on data set 3

So we expected to see deletion non existing data in log mem(figure 6) array and the inserted values on Array data(figure 7). And here are our results.

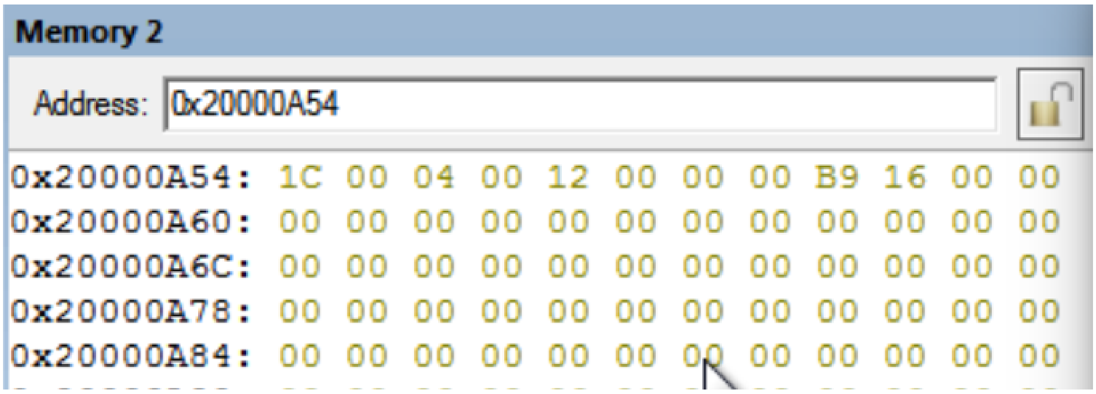

Figure 6: Log_mem array of dataset 3

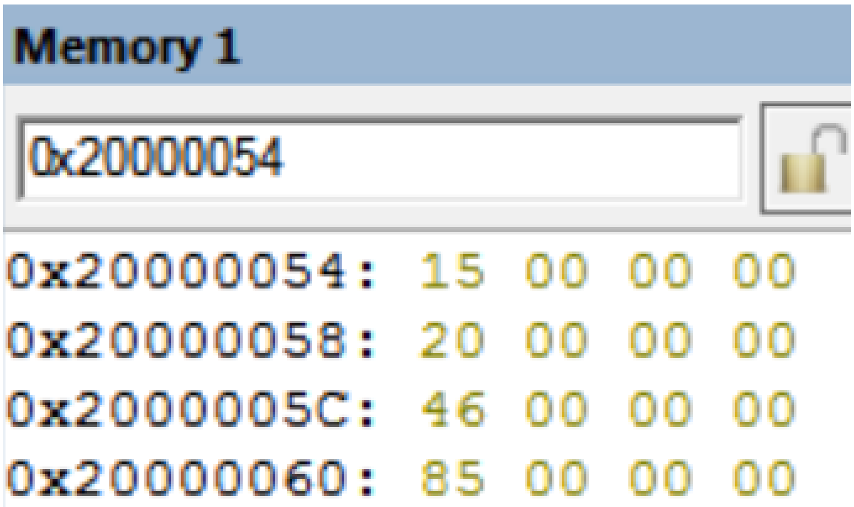

Figure 7: Array_mem array of dataset 3

4 DISCUSSION

In this experiment, we made a Linkedlist project using Assembly Arm Cortex M0 Plus. Linkedlist had insert, remote and transform to Arraylist operations. Except for Linkedlist, we used System Timers to set the timing of the system. Also, since Assembly is a primitive language, when we insert and remove elements, we wrote Malloc and Free functions to delete or allocate space on memory.
Apart from these, we used Errors to understand what the results of the operations we do on the Linkedlist. If a operation is successfully executed, we assigned 0 to the Error code. However, we used Error codes other than 0 in cases such as inserting the element in the list again, the list is full, removing an element from the empty list or missing the element we are looking for when removing. We have returned these error codes in our functions as a result of our operations.

And finally, we transformed the values in this Linkedlist respectively in the Array list. As we said before, we wrote everything manually because Assembly is a primitive language. For this, it is not enough to use only DATA_MEM in the Linkedlist. We used the Allocation table to check the free space and whether the list is full.
As a group, we did not have much knowledge of Assembly Language, so we did research on some subjects before the project. System Timer was the subject we did most research on. Since System Handler manages all processes, we did a research to avoid any mistakes. The Free and Malloc functions were functions that we can do with the existing information, but these parts were challenging because we managed 2 separate memory parties such as Allocation Table and DATA_MEM with these functions.
The Insert and Remove functions were a little challenging, but confusing because there was too much control. In addition, since the Insert and Remove functions work synchronously with the Malloc and Free functions, respectively, the errors in the Malloc and Free parts also affected these functions.
That's why we had to go over the whole project for errors in these parts. We distributed these functions equally among the group members and tried to do the project in an organized and synchronized way because errors in one function affect the others.
When we finished these functions, we finished most of the project and moved to the testing stages. We used our own data to test the Errors in the Error code table and made the necessary changes in the project according to the results of these tests. In case of any error, we tried to write the functions in a simple and appropriate way in order to detect the error quickly and not to make changes in the whole project.
As a result, there were 2 important points in this project in our opinion: The ability of group members to work synchronously with each other and to detect and fix our mistakes quickly.

5 CONCLUSION

In this term project of the microprocessor systems lecture, we have learned so many things. Firstly, we handled the timer interrupt and we didn't face any difficulties. After we found the reload value, assigning the values to the addresses was not hard for us. We got the main idea of the working principles. We understood the timer interrupt was working like we want according to results that we got the information from the view tab. In the Insert function, firstly we wrote the pseuducode and then we passed it to the assembly language. In the Remove function we also did the same thing. Before writing the malloc and free functions, we tried to understand what will we do about that and what should we understand about how is these functions work. Then we wrote the functions and there was no any problem. LinkedList2Arr and GetNow functions also relatively easy implemented functions after we got the main idea. Before all these things, we gave so many times to understand the concept of the project. We lots of talked about the calculations of arrays and the logic of the pointer. After all these things, we started to implement it. So, in the implementation part, we didn't have any difficulties. At the end of the experiment, everything was working correctly.