ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 212E
MICROPROCESSOR SYSTEMS
TERM PROJECT

DATE	:	31.01.2021
GROUP NO	:	G42

GROUP MEMBERS:

150190719	:	Oben Özgür
150180058	:	Faruk Orak
150190739	:	Yasin Abdulkadir Yokuş
150200733	:	Yasin Enes Polat
150200732	:	Ziya Kağan Zeydan

FALL 2020

Contents

1 INTRODUCTION

In this project we created a sorted set linked list structure using assembly language. Our program reads the data and operation flags from the input data set arrays and performs an operation in each System Tick ISR. Stopping condition for The System Tick Timer is whether the program read all data in the input datasets or not. We have written functions such as SysTick_Init(), SysTick_Stop(), Clear_ErrorLogs() in order to initiate necessary components. We also have written System Tick Handler function which increases the tick count, reads the input data and its flag, and does operation of the input.In order to organize the memory, we have written our own malloc() and free() functions. Another examples like Insert(), Remove() and LinkedList2Arr() was needed operations we implemented. When these components came together we have reached our goal and completed this project with success.

2 MATERIALS AND METHODS[1]

2.1 Variables and Memory Areas

```
            AREA        IN_DATA_AREA, DATA, READONLY
IN_DATA        DCD        0x10, 0x05, 0x03, 0x10, 0x00
END_IN_DATA

            AREA        IN_DATA_FLAG_AREA, DATA, READONLY
IN_DATA_FLAG     DCD        0x01, 0x01, 0x01, 0x00, 0x02
END_IN_DATA_FLAG
```

IN_DATA area keeps inputs. The program takes indexes from that area.
IN_DATA_FLAG area keeps operation codes. The program carries out operations according to this flag values. 0x01 means adding, 0x00 means deleting, and 0x02 means turning linked list to an array. For example, the program adds 0x10, 0x05 and 0x03 to the list, deletes 0x10 from the list and turns it to an array.

```
NUMBER_OF_AT            EQU            20
AT_SIZE                 EQU            NUMBER_OF_AT*4
DATA_AREA_SIZE          EQU            AT_SIZE*32*2
ARRAY_SIZE              EQU            AT_SIZE*32
LOG_ARRAY_SIZE          EQU            AT_SIZE*32*3
```

NUMBER_OF_AT keeps the word count of the allocation table.
AT_SIZE keeps the total size of the allocation table. It equals NUMBER_OF_AT*4, because each cell has 4 bytes.
DATA_AREA_SIZE is the size of the Data Area or the maximum area that can be managed with our Allocation Table.
LOG_ARRAY_SIZE is an area for the log errors.

```
AREA        GLOBAL_VARIABLES, DATA, READWRITE
                      ALIGN
TICK_COUNT            SPACE      4
FIRST_ELEMENT         SPACE      4
INDEX_INPUT_DS        SPACE      4
INDEX_ERROR_LOG       SPACE      4
PROGRAM_STATUS        SPACE      4
```

TICK_COUNT stores the number of execution of system tick.
FIRST_ELEMENT stores the head address of the linked list
INDEX_INPUT_DS stores the current data's index.(unused)
INDEX_ERROR_LOG stores the count of error. PROGRAM_STATUS keeps current status of the program
For the status, you can have a look at Figure 2.

```
AREA        ALLOCATION_TABLE, DATA, READWRITE
                      ALIGN
__AT_Start
AT_MEM                SPACE      AT_SIZE
__AT_END

AREA        DATA_AREA, DATA, READWRITE
                      ALIGN
__DATA_Start
DATA_MEM              SPACE      DATA_AREA_SIZE
__DATA_END

AREA        ARRAY_AREA, DATA, READWRITE
                      ALIGN
__ARRAY_Start
ARRAY_MEM             SPACE      ARRAY_SIZE
__ARRAY_END
```

```
AREA        ARRAY_AREA, DATA, READWRITE
                  ALIGN
__LOG_Start
LOG_MEM           SPACE      LOG_ARRAY_SIZE
__LOG_END
```

ALLOCATION_TABLE area is used for allocation table.
DATA_AREA area is used for linked list variables.
First ARRAY_AREA is used for convertion. The program converts linked list to an array into this area.
Second ARRAY_AREA is used to store error logs.

2.2 __main Function

This function was already given to us. We did not change anything in the main function. The main function clears allocation table, clears error log table, initiates the global variables, and initializes the System Tick Timer using relevant functions. Then, it checks the program's status. The program goes to the STOP label if all data operations are completed. Figure 2 shows the flag codes of the program status variable.

2.3 SysTick_Handler Function

This function is responsible for:

- Performing the given operations on the data by calling the appropriate functions.
- Logging any errors that may occur during any of the operations (through `WriteErrorLog`)
- Stopping the System Timer when all operations are completed (through `SysTick_Stop`)

Function starts out by loading:

- Address of the input data array (`IN_DATA`) to `R0`
- 4 times the value of `TICK_COUNT` to `R1`, the offset that will be combined with `IN_DATA` and `IN_DATA_FLAG` to get input data and operation flag for current operation.
- Operation flag for the current operation to `R2`

```
PUSH {R4, LR}
LDR r0, =IN_DATA
LDR r1, =TICK_COUNT
LDR r2, =IN_DATA_FLAG
LDR r1,[r1]
LSLS r1, #2
LDR r2, [r2,r1]
```

It then checks the validity of the Operation Flag. If the flag is found to be invalid, it branches to `STH_error` which is responsible for calling `WriteErrorLog` within this function.

```
MOVS R4, #6
CMP R2, #2
BHI STH_error
```

If the Operation Flag is valid, then operation flag is evaluated and the proper function is called. Input data is loaded into `R0` before the evaluation of the flag, since all functions either have it as their one and only argument (in cases of `Insert` an `Remove`) or do not take arguments at all (in the case of `LinkedList2Arr`).

```
                PUSH {R1}
                CMP R2, #1
                BHI STH_case2
                BEQ STH_case1
STH_case0       BL Remove
                POP {R1}
                B STH_checkerror
STH_case1       BL Insert
                POP {R1}
                B STH_checkerror
STH_case2       BL LinkedList2Arr
                POP {R1}
                B STH_checkerror
```

Please note that `STH_case0` label exists only for the sake of completeness and nothing actually branches to it.

Return value of the called function is then checked for any errors by `STH_checkerror` which, if no errors are detected, branches to `STH_return` thus avoiding the execution of `STH_error` that comes right after it.

```
STH_checkerror  CMP R0,#0
```

```
                BEQ STH_return
                MOV R4, R0
                LDR R0, =IN_DATA
```

`STH_error` Simply shuffles the registers around to fit `WriteErrorLog`'s argument order, as well as deriving Index of the current operation by dividing the offset used internally by 4.

```
STH_error       MOVS R3, R0
                LSRS R1, #2
                MOVS R0, R1
                MOVS R1, R4
                BL WriteErrorLog
```

`STH_return` is the "closing statement" of this function, no matter how the function branches, it always ends here. It increments `TICK_COUNT` by one, and checks if `TICK_COUNT` equals the size of the input array which would signify end of operation. If this is true, it calls `SysTick_Stop`.

```
STH_return      LDR R0, =TICK_COUNT
                LDR R1, [R0]
                ADDS R1,#1
                STR R1, [R0]

                LDR R0, =IN_DATA
                LDR R2, =END_IN_DATA
                SUBS R2,R2,R0
                LSRS R2,#2
                CMP R1, R2
                BNE STH_notlasttick
                BL SysTick_Stop
```

And finally, `STH_notlasttick` returns the function.

```
    STH_notlasttick POP {R4, PC}
```

2.4 SysTickInit Function

In this function, we are expected to initialize System Tick Timer registers, start the timer and update the corresponding bits of program status register. We use clock frequency of the CPU and the period of the System Tick Timer given us after project announcement.

```
LDR     R0, =0xE000E010
LDR     R1, =11695
```

Function first loads the address of control and status register which is "0xE000E010" to register R0, after that function loads calculated reload value into R1. Reload Value = Period x CPU Frequency - 1 = 731µs x 16MHz - 1 = 11695

```
STR     R1,[R0, #4]
```

The statement above simply writes reload value into corresponding bits of control and status register which is on "0xE000E014".

```
MOVS    R1,#0
STR     R1,[R0, #8]
```

These two lines clear current value register which is stored in "0xE000E018".

```
MOVS    R1,#7
STR     R1,[R0]
```

Writing 7 (1112) to SysTickCST register to set enable, control and interrupt flags.

```
LDR     R0, =PROGRAM_STATUS
MOVS    R1,#1
STR     R1,[R0]
```

Lastly, function starts timer by changing PROGRAM_STATUS from 0 to 1.

2.5 SysTickStop Function

This function simply stops the System Tick Timer, clears the interrupt flag of it and updates the PROGRAM_STATUS variable from 1 to 0. Code is similar to some part of SYSTICK_INIT function, only difference is loading 2 to PROGRAM_STATUS instead of 0.

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

Function loads "0xE000E010" to register R0 which is the address of control and status register. Then changes first bit of corresponding memory area to 0. After that, loads address of PROGRAM_STATUS variable to R0 again, changes its first bit to 2 that indicates program is over.

2.6 Clear Allocation Table Function

In this function we are expected to clear all bits in the allocation table. In the beginning of the program, all bits of the allocation table is cleared. In order to achieve this purpose, we load start and end address of allocation table to registers. We load 0 value to a register since we will use it in clear operation. Then we use for loop and assign 0 to all addresses. From the code it can be seen that we compared _AT_Start and _AT_END to decide whether we reached the end of the allocation table as address.

```
Clear_Alloc                     FUNCTION
    ;load start addres of allocation table to r0
        LDR R0, =__AT_Start
        ;load end addres of allocation table to r1
        LDR R1, =__AT_END
        ;value to be loaded
        LDR R2, =0
        ;branch to for_start
        B ca_FOR_START
        ;load 0 to adress of r0
ca_LOOP                         STR R2,[R0]
    ;r0++ for iteration
        ADDS R0,R0,#4
    ;cmp adrress of r0 and end adres
ca_FOR_START        CMP R0, R1
    ;if ro<end_address loop over
        BLT ca_LOOP
        ;return
        BX LR
        ENDFUNC
```

If we would not implement and call this function, some unused areas of the memory may appeared to be used.

2.7 Clear_ErrorLogs Function

In this function we are expected to clear all cells in the Error Log Array because some memory addresses contain garbage values. In order to achieve this purpose, we load start and end address of error log table to registers. We load 0 value to a register since we will use it in clear operation. Then we use for loop and assign 0 to all addresses. From the code it can be seen that we compared _LOG_Start and _LOG_END to decide whether we reached the end of the error log table as address.

```
Clear_ErrorLogs FUNCTION
        ;load start addres of error log table to r0
        LDR R0, =__LOG_Start
        ;load end addres of error log table to r1
        LDR R1, =__LOG_END
        ;value to be loaded
        LDR R2, =0
        ;branch to for_start
        B ce_FOR_START
        ;load 0 to adress of r0
ce_LOOP                 STR R2,[R0]
        ;r0++ for iteration
        ADDS R0,R0,#4
        ;cmp adrress of r0 and end adres
ce_FOR_START    CMP R0, R1
        ; if ro<end_address loop over
        BLT ce_LOOP
        ;return
        BX LR
        ENDFUNC
```

If we would not implement and call this function, there might remain garbage values in some memory addresses.

2.8 Init_GlobVars Function

In this function we are expected to initialize global variables. In order to achieve this purpose, we load start and end address of global variables to registers. We load 0 value to a register since we will use it in clear operation. Then we use for loop and assign 0 to all addresses. From the code it can be seen that we compared _TICK_COUNT and _MAL_firstemptyoffset to decide whether we reached the end of the address of the global variables.

```
Init_GlobVars   FUNCTION
    ;load start addres of global variables to r0
        LDR R0, =TICK_COUNT
        ;load end addres of global variables to r1
        LDR R1, =MAL_firstemptyo
        ;value to be loadedffset
        LDR R2, =0
        ;branch to for_start
        B ig_FOR_START
        ;load 0 to adress of r0
ig_LOOP     STR R2,[R0]
    ;r0++ for iteration
        ADDS R0,R0,#4
        ;cmp adrress of r0 and end adres
ig_FOR_START     CMP R0, R1
    ;if ro<end_address loop over
        BLE ig_LOOP
        ;return
        BX LR
        ENDFUNC
```

Figure 3 shows the global variables and their usage purposes.

2.9 Malloc Function

This function is used to allocate memory inside `DATA_MEM` and, alongside `Free`, is responsible for managing the Allocation Table as well as `MAL_firstemptyoffset`. `MAL_firstemptyoffset` is a global variable holding the offset value (aka address relative to `AT_MEM`) for the first word in the Allocation Table that contains an empty spot.

Function starts out by loading:

- Address of `AT_MEM` to `R0`
- Value of `MAL_firstemptyoffset` to `R1`
- Offset for the last word of the Allocation Table (`AT_SIZE-4`) to `R2`

```
        LDR R0, =AT_MEM
        LDR R2, =AT_SIZE-4
        LDR R1, =MAL_firstemptyoffset
        LDR R1, [R1]
```

It then checks if `MAL_firstemptyoffset` is within the boundaries of `AT_MEM`. If it isn't, the function returns immediately with a return value of 0: there is no free space left in memory.

```
CMP R1, R2
BLS MAL_noerrors
MOVS R0, #0
POP {R4,R5,R6,R7}
BX LR
```

Now that it has confirmed that `MAL_firstemptyoffset` points to a valid offset, it makes preparations to find the first 0 bit inside `MAL_firstemptyoffset`. For this purpose:

- `R5`, which will store the index for the first 0 bit, is cleared
- Value corresponding to `MAL_firstemptyoffset` is loaded onto `R3` and inverted, since finding the first 1 bit is easier than finding the first 0 bit.
- A backup of the original value of `MAL_firstemptyoffset` is made onto `R6` for future use.

The search for the 0 bit happens in 5 steps. Each step:

- takes half the amount of bits as the previous step as its input
- checks whether the first 1 bit falls within the left or right half of the input
- If the bit falls within the left half, it increases `R5` by half the size of its input and moves the left half to the right half, aligning it properly for the step after it.

```
                LDR R4, =0x0000FFFF
                TST R3, R4
                BNE MAL_skip1
                ADDS R5, R5, #16
                LSRS R3, #16

MAL_skip1       LDR R4, =0x000000FF
                TST R3, R4
                BNE MAL_skip2
                ADDS R5, R5, #8
                LSRS R3, #8

MAL_skip2       LDR R4, =0x0000000F
                TST R3, R4
                BNE MAL_skip3
                ADDS R5, R5, #4
```

```
                        LSRS R3, #4

MAL_skip3               LDR R4, =0x00000003
                        TST R3, R4
                        BNE MAL_skip4
                        ADDS R5, R5, #2
                        LSRS R3, #2

MAL_skip4               LDR R4, =0x00000001
                        TST R3, R4
                        BNE MAL_skip5
                        ADDS R5, R5, #1
                        LSRS R3, #1
```

Using the bit index now stored at `R5` combined with `MAL_firstemptyoffset` we can now find the address of the allocated data in `DATA_MEM`. Function proceeds to set the bit corresponding to the chosen address and store the changes on the Allocation Table as well as calculating the effective address of the now allocated space.

```
MAL_skip5
                        MOVS R3, R6
                        MOVS R6,#0x1
                        LSLS R6,R5
                        ORRS R3, R6
                        STR R3, [R0,R1]

                        MOVS R7, R1
                        LSLS R1, #6
                        LSLS R5,#3
                        LDR R0, =DATA_MEM
                        ADDS R0, R5, R0
                        ADDS R0, R1, R0
```

Function lastly checks if the word at `MAL_firstemptyoffset` is now full and searches for a new one if it is. If there are no valid candidates, `MAL_firstemptyoffset` is set to an out-of-bounds value.

```
                        LDR R6, =0xFFFFFFFF
                        CMP R3, R6
                        BEQ MAL_findnextblock
```

```
                    POP {R4,R5,R6,R7}
                    BX LR


MAL_findnextblock
                    MOVS R1, R7
                    LDR R7, =AT_MEM

                    B MAL_l1_start
MAL_l1_loop         LDR R3, [R7,R1]
                    MVNS R3, R3
                    BNE MAL_found

                    ADDS R1,R1,#4

MAL_l1_start        CMP R1,R2
                    BLE MAL_l1_loop

MAL_found           LDR R3, =MAL_firstemptyoffset
                    STR R1, [R3]
                    POP {R4,R5,R6,R7}
                    BX LR
```

2.10 Free Function

This function is used to free previously allocated space and, alongside `Malloc`, is responsible for maintaining the Allocation Table and `MAL_firstemptyoffset`.

Function starts by loading the address of `DATA_MEM` to `R1`.

```
LDR R1, =DATA_MEM
```

It then proceeds to seperate the address given to AT block offset (offset for the word containing this address in the Allocation Table) and AT bit index (index of the bit corresponding to the address), storing at `R0` and `R1`. Numerically, these values equal to $\lfloor Address/256 \rfloor * 4$ and $\lfloor Address/8 \rfloor \% 32$ respectively.

```
LDR R1, =DATA_MEM
SUBS R0, R0, R1
LSRS R0, #3
```

```
        MOVS R1, R0
        LSRS R0, #5
        MOVS R2, #0x1F
        ANDS R1, R2
        LSLS R0, #2
```

It then checks if the AT block offset comes before `MAL_firstemptyoffset`, and updates `MAL_firstemptyoffset` if so.

```
        LDR R3, =MAL_firstemptyoffset
        LDR R2, [R3]
        CMP R0, R2
        BGE     FRE_noupdate

        STR R0, [R3]
```

Finally it clears the bit at AT bit index of the word at AT block offset and returns.

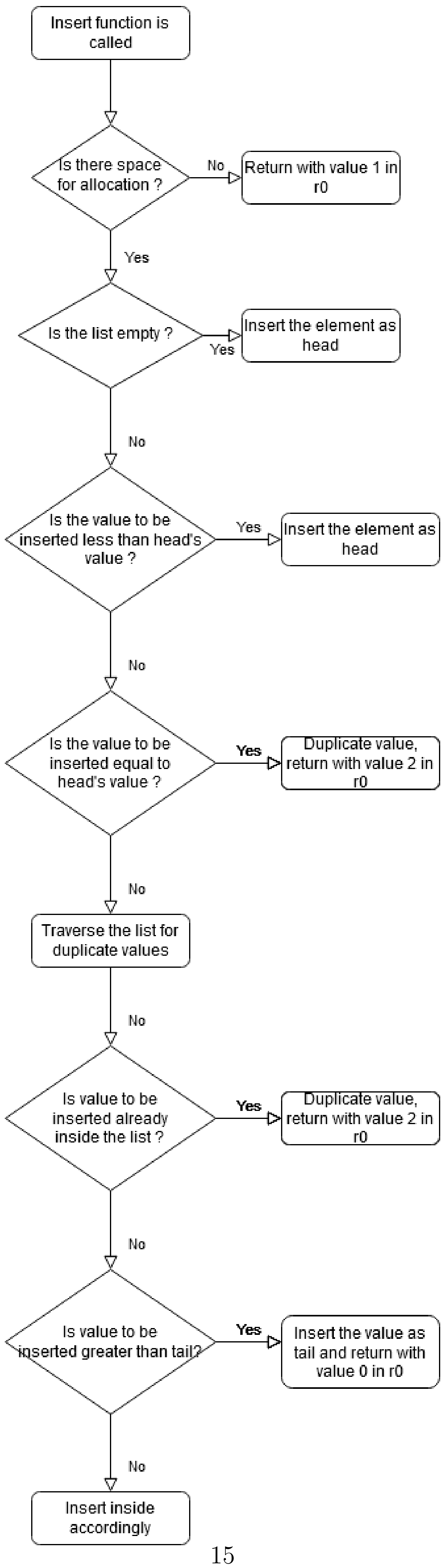
```
FRE_noupdate    LDR R3, =AT_MEM
                ADDS R0, R3
                LDR R3, [R0]

                MOVS R4, #1
                LSLS R4, R1
                MVNS R4, R4
                ANDS R3, R4
                STR R3, [R0]

                POP {R4}
                BX LR
```

2.11 Insert Function

Insert function takes only a single value as argument and inserts that value into the linked list according to the following rules:

- The list must be sorted in increasing order.
- The list can not contain duplicate elements.

The function returns “0” with R0 register if the insertion operation is done successfully. Otherwise, it produces some specific return values with invalid insertions. If there is no memory left to insert the element provided by the malloc function, insert function returns “1”” with R0 register. If the value to be inserted is already inside the linked list, insert function does not insert it and returns “2” with R0 register. Since insertion in linked lists has many edge cases. Each edge case’s control is done by some flow control inside the insertion function. The control flow of the insert function is given in the next page and it will be explained in code later on.



Insert function is called
Is there space for allocation ?
No
Return with value 1 in r0
Yes
Is the list empty ?
Yes
Insert the element as head
No
Is the value to be inserted less than head's value ?
Yes
Insert the element as head
No
Is the value to be inserted equal to head's value ?
Yes
Duplicate value, return with value 2 in r0
No
Traverse the list for duplicate values
No
Is value to be inserted already inside the list ?
Yes
Duplicate value, return with value 2 in r0
No
Is value to be inserted greater than tail?
Yes
Insert the value as tail and return with value 0 in r0
No
Insert inside accordingly

Figure 1: Flowchart of the insert function[2]

At the first part of the insertion function, link register and value to be inserted is pushed to stack since other functions such as “Malloc” must be called inside the insertion function. After pushing operation, Malloc function is called and it returns its value with R0 as all the other functions. If Malloc returns “0” it means that there is no allocable area. After calling Malloc, its return value is checked and program decides to go to next phase if it is not “0””, if it returns “0”” however, program returns with the value of “1” as mentioned in the description paragraph of the insertion function.

```
;this part is for checking if there is enough space to insert or not
PUSH {LR}    ;push lr to stack
PUSH {R0}         ;push value to stack
BL Malloc         ;allocate memory
LDR R1,=0         ;load 0 to r1 for comparison
CMP R0,R1         ;compare r0 and r1
BNE not_full      ;if there is allocable area go to not_full
POP {R0}     ;pop r0 value
LDR R0,=1         ;load r0 "0" as return value
POP {PC}          ;return with value of "0"
```

After making sure that there is allocable area, program goes to the next part. Firstly, since R0 holds the Malloc function’s return value(adress of allocated memory area), that same area is freed because it will be reallocated later in the program if needed. After that, program just checks whether the linked list is empty or not. If, it is empty program performs the insertion as a head value and updates the first elements adress accordingly before returning. Otherwise, program continues with the next part for the control of edge cases.

```
;this part is for insertion to an empty list
not_full BL Free;free the allocated memory it wil be reallocated later
LDR R1, =FIRST_ELEMENT  ;load the adress of the first element to r1
LDR R3,[R1];load head adress
LDR R2, =0;load 0 to r2 for comparison
CMP R3, R2;check if the list is empty
BNE     not_empty ;if not empty continue
BL Malloc;allocate memory
LDR R1, =FIRST_ELEMENT  ;load the adress of the first element to r1
STR R0,[R1];set first element adress as adress of allocated area
POP {R0};get the value to be inserted
LDR R2,[R1];load the head address to r2
STR R0,[R2];store the given value as head
```

```
LDR R0, =0;load r0 to "0" as null pointer
STR R0, [R2,#4];load null address as next element
POP {PC};return with value of "0"
```

In the third phase of the insertion function, edge case of insertion as head to a non-empty is performed if needed. If a linked list is not empty and the value to be inserted is less than the value of the head, then the the new element to be inserted to must be the head. That control flow performs that edge case. This part also checks for the duplicate value case of the head. If the head value is duplicate, it branches to duplicate returning part of the program. Otherwise, program starts to traverse the list.

```
;this part is for insertion as head to a non empty list
not_empty POP {R0};pop value to be inserted
LDR R1, =FIRST_ELEMENT ;load the adress of the first element to r1
LDR R3,[R1];load head adress
LDR R2, [R3];store r2 head value of the list
CMP R0, R2;compare r0 and r2
BEQ duplicate;if equal go to duplicate
BGT traverse;if value to be insterted is greater than head value
;branch to larger than head
PUSH {R3};push head adress
PUSH {R0};push value to be inserted
BL Malloc;allocate memory
LDR R1, =FIRST_ELEMENT ;load the adress of the first element to r1
;(in case it has changed)
STR R0, [R1];store allocated adress as head adress
POP {R2};pop value to be inserted to r2
POP {R3};pop previous head's adress to r3
STR R2, [R0];store value to be inserted to new head
STR R3, [R0,#4];store previous head adress as next adress to new head
LDR R0, =0;load r0 "0" as return value
POP {PC};return with value of "0"
```

The next part of the program just traverses the whole list for duplicate values until it reaches the null pointer meaning the end. If it finds any duplicate values, program branches to duplicate value returning part.

```
;this part is for traversing the list for duplicate values
traverse LDR R3, =FIRST_ELEMENT;load the adress of the first element to r3
LDR R1,[R3];load head adress
LDR R2,[R1,#0];r2 stores the current element's value
LDR R3,[R1,#4];r3 stores the next adress
next_element CMP R0, R2;compare value to be inserted and current value
BEQ duplicate;if equal branch to duplicate
CMP R3, #0;compare r3 and 0
BEQ end_of_list;if equal (NULL) branch to end_of_list
MOVS R1, R3;get r1 current element's adress
LDR R2,[R3,#0];get next address' value to r2
LDR R3,[R3,#4];get next adress's next adress to r3
B next_element;iterate
```

This small piece of branching code is for returning with the duplicate error code.

```
;this is for returning with duplicate error code
duplicate LDR R0, =2;load 2 to r0 as error code
POP {PC};return with value of "2"
```

After the traversal of the whole linked list, traversing pointer is at the end of the list. After the direct comparison between the value to be inserted and the last value of the linked list, edge case of insertion to tail can be performed. If value to be inserted is larger than the last element of linked list, then the new cell must the last element of the linked list. Following code piece performs that operation.

```
;this is for checking if insertion is going to be at the end
end_of_list CMP R0, R2;compare the last element of the list
;with value to be inserted
BGT     insert_to_end;if greater branch to insert to end
B insert_inside;if not branch to inser_inside

;this the part for insertion to end
insert_to_end   PUSH {R0};push value to stack
PUSH {R1};push last element's adress to stack
BL Malloc;allocate cell
POP {R1};pop last element's adress to r1
STR R0,[R1,#4];load last element's next adress as to allocated area
MOVS R1, R0;move allocated area's adress to r1
POP {R0};pop value to be inserted from stack
STR R0,[R1,#0];load value to be inserted to the allocated area's value
LDR R0, =0;;load r0 "0" as NULL pointer
STR R0,[R1,#4];load NULL adress to next adress
```

```
;since it is the last element
LDR R0, =0;;load r0 "0" as return value
POP {PC};return
```

Final edge case of the linked list insertion is the insertion in betweeen the elements of the linked list. In the last section of the insertion function, linked list is traversed once again and when the value that is larger than the value to be inserted is found, the function insert the value in between with setting the adresses to connect the list after insertion. Finally, it returns with success code of “0”.

```
;this is for insertion to inside
insert_inside   LDR R3, =FIRST_ELEMENT  ;load the adress of
;the first element to r3
LDR R3,[R3];load head adress to r3
next_one PUSH {R3};push current adress
LDR R3,[R3,#4];get next adress's next adress to r3
LDR R2,[R3,#0];get next address' value to r2
CMP R0, R2;compare value to be inserted and current value
BLT insert_it;if equal branch to duplicate
POP {R1};pop current current adress to r1,
;it is redundant and won't be used
B next_one;iterate


insert_it POP {R3} ;pop element's adress before the inserted one
PUSH {R0} ;push the value to be inserted to stack
PUSH {R3} ;push element's adress before the inserted one
BL Malloc ;allocate memory
POP {R3};pop element's adress before the inserted one
LDR R1,[R3,#4];get the element's adress that comes after the inserted node
STR R0,[R3,#4];put current node's next adress
;the allocated memory's adress
POP {R2};pop value to be inserted to r2
STR R2,[R0,#0];put value to be inserted to allocated cell
STR R1,[R0,#4];put allocated cell's next adress the correspoing adress(r1)
LDR R0, =0;;load r0 "0" as return value
POP {PC};return
```

2.12 Remove Function

In this function, we are expected to remove the given number from linked list. Function takes a number as parameter and starts to search that number on linked list from start to end.

```
LDR     r1 , =FIRST_ELEMENT
LDR     r1 , [r1 , #0]
CMP     r1 , #0
BEQ     EMPTY_LST_ERR
```

These 4 lines of code first loads address of FIRST_ELEMENT variable into r1 register, then checks that memory area, if there is no memory address in that memory area, it means the linked list is empty and function returns error code 3 to indicate that the list is empty.

```
LDR     r3 , [r1 , #0]
CMP     r3 , r0
BEQ     DELETE_FIRST
```

After empty checking, first element of the linked list is loaded into r3 by LDR instructin, then it checks whether the first element's data value is equal to given value or not, if it is equal to given number, it jumps to DELETE_FIRST branch which deletes/removes first element and changes the address pointed by FIRST_ELEMENT variable that is the address of next element of the first element.

```
MOVS    r2 , r1
LDR     r1 , [r1 , #4]
LDR     r3 , [r1 , #0]
```

If given value is not equal to first elements data value, then r2 holds current (which will be the previous element) element's starting address to connect to next element of removed node. After that, value of current element's next pointer is loaded into r1 and from new r1, data value loaded into r3.

```
EQUALITY_Check  CMP     r3 , r0
                BEQ     DELETE_ELEMENT
                MOVS    r2 , r1
                LDR     r1 , [r1 , #4]
                CMP     r1 , #0
                BEQ     NOT_FOUND_ERR
                LDR     r3 , [r1 , #0]
                B       EQUALITY_Check
```

These EQUALITY_CHECK loop traces along the linked list to find if given data value exists on it. It keeps previous elements starting address on r2 each iteration. If it reaches to end of linked list without finding given value, function returns error code 4 to indicate that given value doesn't exist on linked list. If it finds given value on linked list, it jumps to DELETE_ELEMENT branch to remove it and connect previous element to next element of removed element.

```
DELETE_FIRST    LDR     r4 , =FIRST_ELEMENT
                LDR     r5 , [r1 , #4]
                STR     r5 , [r4 , #0]
                MOVS    r0 , r1
                PUSH    {LR}
                BL      Free
                MOVS    r0 , #0
                POP {PC}
```

These lines of statements preserves FIRST_ELEMENT for always showing first index of linked list. To do that, next pointer value of current first element is loaded into r5, then that address written into FIRST_ELEMENT so FIRST_ELEMENT shows first index of linked list constantly .

```
DELETE_ELEMENT  LDR     r4 , [r1 , #4]
                STR     r4 , [r2 , #4]
                MOVS    r0 , r1
                PUSH    {LR}
                BL      Free
                MOVS    r0 , #0
                POP     {PC}
```

This part of function simply removes the element with data equal to the given value from linked list. Also previous element's starting address is already on r2 and next pointer value of element which will be deleted is loaded into r4, then address on r4 is written into previous element's next pointer (r2 + 4) variable so that connection is remained.

On both deleting branch (DELETE_FIRST, DELETE_ELEMENT), removed elements starting address is passed to Free funtion to deallocate the existing area. Before calling Free funtion, LR is pushed to stack and popped after returning to maintain the Link Register.

2.13 Linked List to Array Function

In this function we have converted linked list to array. Basically, the function clears array, starts from head and goes to the end of list and writes data part of variable to the array one by one.

```
                    PUSH {LR}
                    LDR r0 , =FIRST_ELEMENT
                    LDR r0 ,  [ r0 ]
                    CMP r0 ,#0
                    BEQ ERROR_LL2A

ERROR_LL2A          MOVS r0 ,  #5
                    POP {PC}
```

In this part, code checks whether linked list is empty. First pushes LR to stack, takes FIRST_ELEMENT variables address and checks its inside. If it is null(0x00), then it means linked list is empty and the function branches to error part. At the error part, the function returns related error code.

```
                    LDR r0 , =__ARRAY_Start
                    LDR r1 , =__ARRAY_END
                    MOVS r2 ,  #0
                    B LL2A_CLEAR_START
CLEAR_LL2ASTR       r2 , [ r0 ]
                    ADDS r0 , r0 ,#4
LL2A_CLEAR_STARTCMP r0 ,  r1
                    BLT CLEAR_LL2A
```

In this part, functiont takes start and end address of the array and fill with 0's all of the array with a loop. The loop condition is satisfied by comparison with current address [r0] and array end address [r1].

```
                    LDR r0 , =FIRST_ELEMENT
                    LDR r1 , =__ARRAY_Start
LOOP_LL2ALDR        r0 , [ r0 ]       ; take the address of the variable
                    LDR r2 , [ r0 ]       ; take the data from LL
                    STR r2 ,  [ r1 ]      ; store data to array
                    ADDS r0 ,#4           ; take the address of the
                    ;memory cell that keeps next variable 's address
                    ADDS r1 ,#4           ; next array memory cell
                    LDR r2 , [ r0 ]       ; take the next address
                    CMP r2,#0            ;if the next address is not null
                    BNE LOOP_LL2A        ;Go LOOP
                    MOVS r0, #0
                    POP {PC}
```

In this part, function turns the linked list to an array. It takes first element(head) and ARRAY_START address. It starts from head, takes the data, writes it to array and takes next variable's address. If it is not null, repeats this process until the end of the list.

2.14 WriteErrorLog Function

In this function we are expected to store the error log of the input dataset operations. This function takes Index, ErrorCode, Operation, and Data variables as the arguments via r0 - r3 registers as specified in contraints section of the project. Then, it stores these data to the current available area of the LOG_MEM if the LOG_MEM array is not full.

In order to achieve this purpose we follow the structures explained below.

Firstly, we use r4 and r5 so we push these register into stack with the LR then we load INDEX_ERROR_LOG address and take its value. We will use this index value to multiply with 32 and 3 to get the memory location inside error log array. Think it as 0*32*3 gives 0th index, 1*32*3 gives 0th index etc. The reason we do this multiplication is one error log array element consists of 3 words. Then we make a comparison to see whether error log array is full or not. If it is full we exit from function else we start writing to the specified location.

Error log structure can be seen in the Figure 4 in a more clear and structured manner.

```
WriteErrorLog   FUNCTION
        ; we will use r4, r5
        PUSH {r4, r5, lr}
        ; get address of the index of the error log array.
        LDR   r4, =INDEX_ERROR_LOG
        LDR r4, [r4,#0]
        ; get address of the index of the error log array.
        LSLS r4, #2
        LDR R5, =3
        MULS r4, R5, R4
        LDR R5, =__LOG_Start
```

```
; r4 = (load start addres + 32*3*index)
ADDS r4, r5
; load end addres of log table to r5
LDR r5, =__LOG_END
; cmp adrress of r4 and end adres
CMP r4, r5
; if r4 >= end_address return
BGE we_exit
; else do writing error log
; each word has 32 bits
```

We will write to the specified location of 3 words. First we write "Index" since it is half word, after the write operation we will increment the our special memory location register by half word. In this case it needs 2 with ADDS operation. We will use this pattern in the following write operations by multiplaying or dividing according to word size.

```
; writing Index          -> 16bit -> half word
; halfword store the address of the index of the dataset
; to the index of the error log array.
STRH r0, [r4]
; incr mem for a half word
ADDS r4, #2
; writing ErrorCode      -> 8bit  -> 1/4 word
; byte store the error code to index of the error log array.
STRB r1, [r4]
; incr mem for a byte
ADDS r4, #1
; writing Operation      -> 8bit  -> 1/4 word
; byte store the op code to index of the error log array.
STRB r2, [r4]
; incr mem for a byte
ADDS r4, #1
; writing Data           -> 32bit -> 1 word
; word store the data to the index of the error log array.
STR  r3, [r4]
; incr mem for a  word
ADDS r4, #4
```

Here as you can see we call "GetNow" function. We know that return value will be on r0 because of the project constrains. Then we continue writing these values in same manner as previous writing operations.

```
        ; writing TimeStamp      -> 32bit -> 1 word
        ; call GetNow, timestamp is stored in r0
        BL      GetNow
        ; word store the timestamp to the index of the error log array.
        STR   r0, [r4]
        ; incr mem for a  word
        ADDS r4, #4
```

We finished all writing operations. Since writing operations was succesfull we increment the INDEX_ERROR_LOG by one, then continue with the "we_exit". If we could not have written into error log array, we would directly jump to "we_exit" and pop the r4, r5 and pc registers.

```
    ; index error log update for next record
        LDR r4, =INDEX_ERROR_LOG
        LDR r4, [r4,#0]
        ADDS r4, #1
    LDR r5, =INDEX_ERROR_LOG
        STR r4, [r5,#0]
we_exit
        ; pop vals from stack
        POP {r4, r5, pc}
        ENDFUNC
```

2.15 GetNow Function

In this function we are expected to return the working time of the System Tick Timer in microseconds. This function takes no arguments. We are able to calculate the actual working time using the TICK_COUNT variable and System Tick Timer Current Value Register and did not need to use the Calibration Register. Also we assumed that the program does all operations in less than 70 minutes.

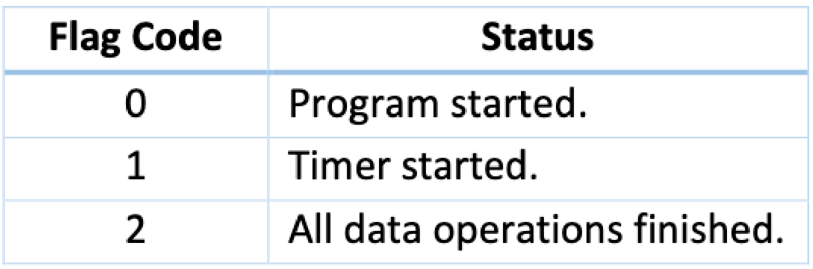
In order to achieve this purpose we follow the instructions explained below.

Firstly we load TICK_COUNT address and takes its value, then we multiply it with the reload value which we calculated and explained in the SysTick_Init function. In order to get the working time we also need to add the value of SYST_CVR, SysTick Current

Value Register. As the last step we divide this sum by 16. As an abstract explanation we can think such when we divide the sum by 16MHz in the actual formula it returns the value in microseconds.

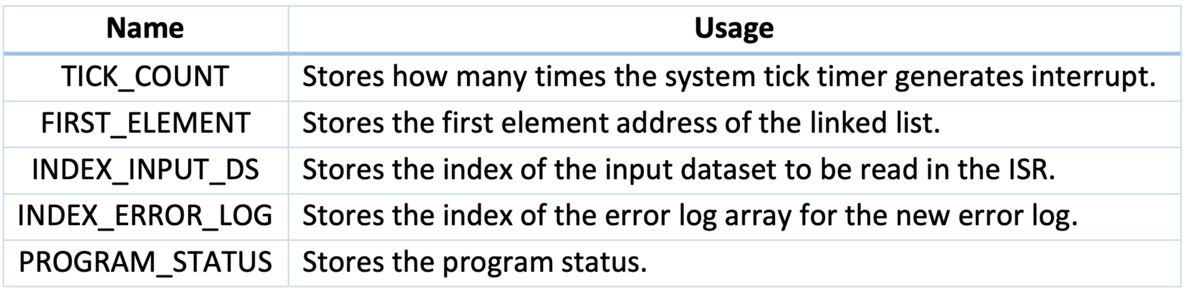
```
GetNow                      FUNCTION
          ; get tick count adress to r0
          LDR  r0, =TICK_COUNT
          ; get value of tick count to r0
          LDR  r0, [r0]
          ; load reload value to r1
          LDR  r1, =11695
          ; TICK_COUNT * reload value
          MULS r0, r1, r0
          ; get the address for SYST_CVR, SysTick Current Value Register
          LDR  r2, =0xE000E018
          ; get the value for SYST_CVR, SysTick Current Value Register
          LDR  r2, [r2]
          ; r0 = (TICK_COUNT * reload value) + SYST_CVR, in microseconds
          ADDS r0, r2
          ;r0 /= 16
          LSRS r0, #4
          ; return
          BX   LR
          ENDFUNC
```

3 RESULTS



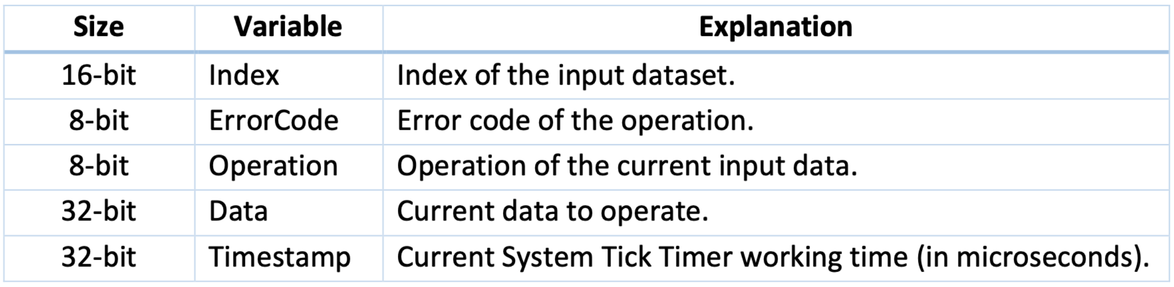
Flag Code	Status
0	Program started.
1	Timer started.
2	All data operations finished.

Figure 2: Flag Codes of the Program Status



Name	Usage
TICK_COUNT	Stores how many times the system tick timer generates interrupt.
FIRST_ELEMENT	Stores the first element address of the linked list.
INDEX_INPUT_DS	Stores the index of the input dataset to be read in the ISR.
INDEX_ERROR_LOG	Stores the index of the error log array for the new error log.
PROGRAM_STATUS	Stores the program status.

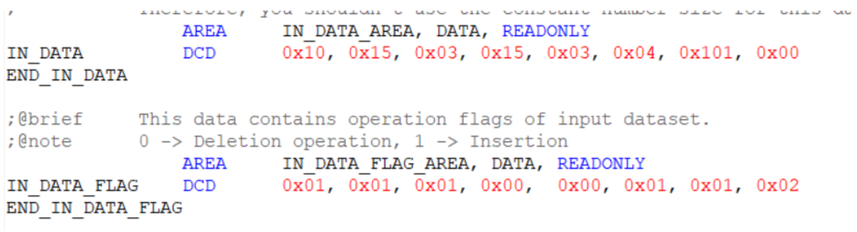
Figure 3: Global Variables



Size	Variable	Explanation
16-bit	Index	Index of the input dataset.
8-bit	ErrorCode	Error code of the operation.
8-bit	Operation	Operation of the current input data.
32-bit	Data	Current data to operate.
32-bit	Timestamp	Current System Tick Timer working time (in microseconds).

Figure 4: Error Log Struct

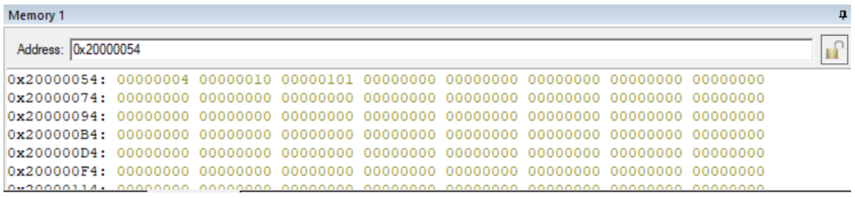
3.1 Case 1



```
                AREA     IN_DATA_AREA, DATA, READONLY
IN_DATA         DCD      0x10, 0x15, 0x03, 0x15, 0x03, 0x04, 0x101, 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, 0x00,  0x00, 0x01, 0x01, 0x02
END_IN_DATA_FLAG
```

Figure 5: case 1 inputs

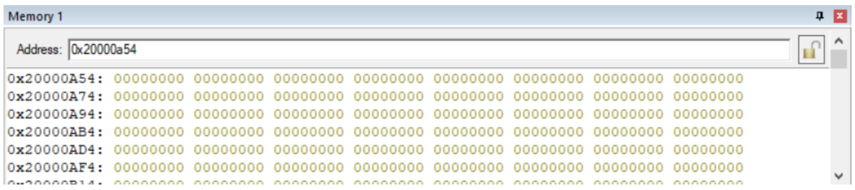
With these inputs, we expect that 0x10, 0x15, 0x03 will be inserted. After that 0x15 and 0x03 will be removed, 0x04 and 0x101 will be added. Finally, the list will be converted to the array. We expect no error.



```
Address: 0x20000054
0x20000054: 00000004 00000010 00000101 00000000 00000000 00000000 00000000 00000000
0x20000074: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000094: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x200000B4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x200000D4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x200000F4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

Figure 6: case 1 array

As it is expected, we have just 0x04, 0x010 and 0x101 in the array as increasing order.

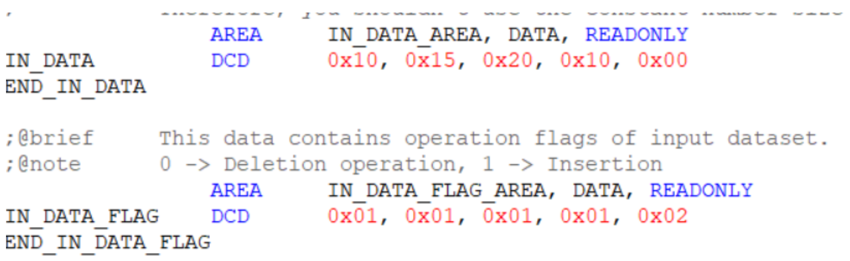


```
Address: 0x20000a54
0x20000A54: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000A74: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000A94: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000AB4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000AD4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000AF4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

Figure 7: case 1 errors

As it is expected, we have no errors.

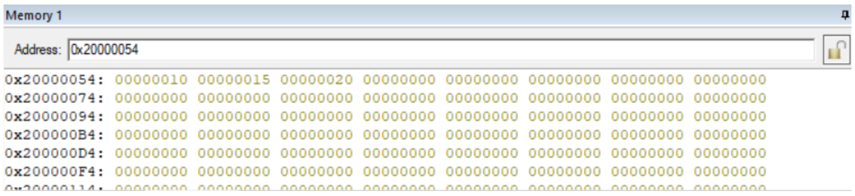
3.2 Case 2



```
                AREA     IN_DATA_AREA, DATA, READONLY
IN_DATA         DCD      0x10, 0x15, 0x20, 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, 0x02
END_IN_DATA_FLAG
```

Figure 8: case 2 inputs

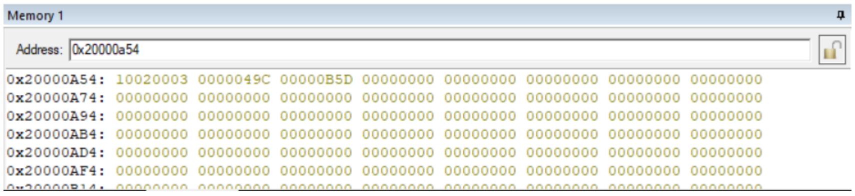
With these inputs, we expect that 0x10, 0x15, 0x20 will be inserted. After that 0x10 will be inserted again. Program will not insert duplicate values and gives an error.



```
Address: 0x20000054
0x20000054: 00000010 00000015 00000020 00000000 00000000 00000000 00000000 00000000
0x20000074: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000094: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x200000B4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x200000D4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x200000F4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

Figure 9: case 2 array

As it is expected, we have just 0x10, 0x015 and 0x20 in the array as increasing order.

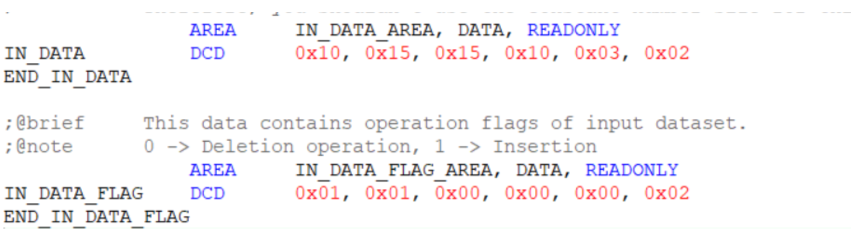


```
Address: 0x20000a54
0x20000A54: 10020003 0000049C 00000B5D 00000000 00000000 00000000 00000000 00000000
0x20000A74: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000A94: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000AB4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000AD4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000AF4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

Figure 10: case 2 errors

As it is expected, we have a duplicate error.

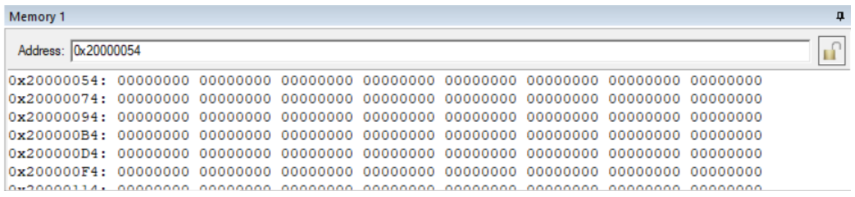
3.3 Case 3



```
                AREA     IN_DATA_AREA, DATA, READONLY
IN_DATA         DCD      0x10, 0x15, 0x15, 0x10, 0x03, 0x02
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, 0x00, 0x00, 0x00, 0x02
END_IN_DATA_FLAG
```

Figure 11: case 3 inputs

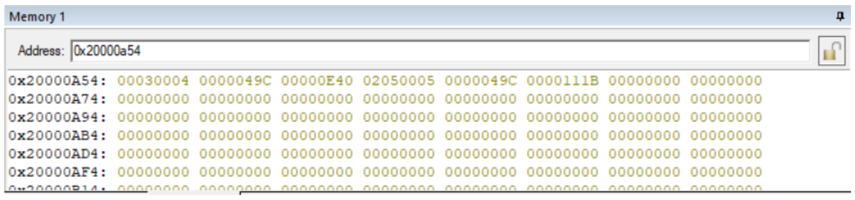
With these inputs, we expect that 0x10, 0x15 will be inserted. After that 0x10 and 0x15 will be deleted. Finally 0x03 will be removed but linked list is empty. Then the program gives an error. In addition, the program tries to convert linked list to array. List is empty. So the program gives another error.



```
0x20000054: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000074: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000094: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x200000B4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x200000D4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x200000F4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

Figure 12: case 3 array

As it is expected, we have an empty array.

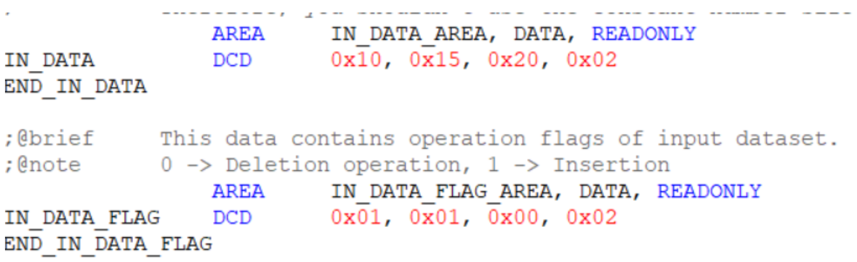


```
0x20000A54: 00030004 0000049C 00000E40 02050005 0000049C 0000111B 00000000 00000000
0x20000A74: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000A94: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000AB4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000AD4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000AF4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

Figure 13: case 3 errors

As it is expected, we have two errors.

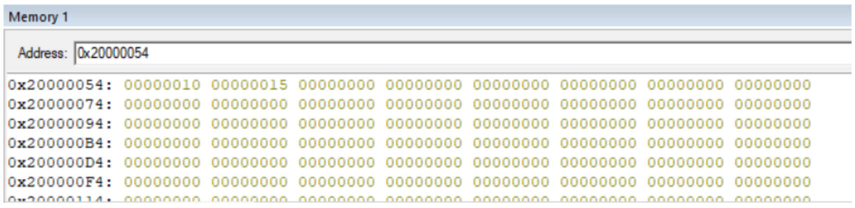
3.4 Case 4



```
                AREA     IN_DATA_AREA, DATA, READONLY
IN_DATA         DCD      0x10, 0x15, 0x20, 0x02
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, 0x00, 0x02
END_IN_DATA_FLAG
```

Figure 14: case 4 inputs

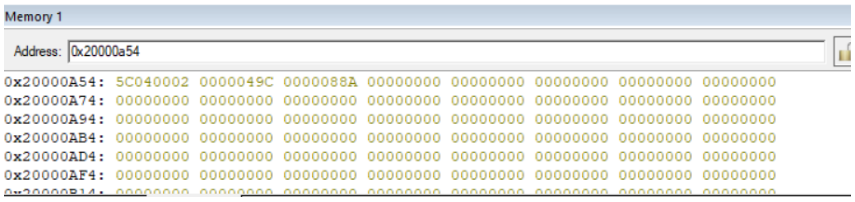
With these inputs, we expect that 0x10, 0x15 will be inserted. After that 0x20 will be deleted. But there is no element such that. The program gives and error and converts linked list to an array.



```
Memory 1
Address: 0x20000054
0x20000054: 00000010 00000015 00000000 00000000 00000000 00000000 00000000 00000000
0x20000074: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000094: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x200000B4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x200000D4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x200000F4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

Figure 15: case 4 array

As it is expected, we have 0x10 and 0x15 in the array in increasing order.



```
Memory 1
Address: 0x20000a54
0x20000A54: 5C040002 0000049C 0000088A 00000000 00000000 00000000 00000000 00000000
0x20000A74: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000A94: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000AB4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000AD4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000AF4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

Figure 16: case 4 errors

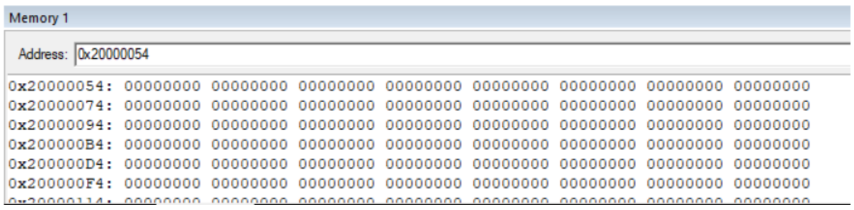
As it is expected, we have an error.

3.5 Case 5

```
                AREA     IN_DATA_AREA, DATA, READONLY
IN_DATA         DCD      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      0x02
END_IN_DATA_FLAG
```

Figure 17: case 5 inputs

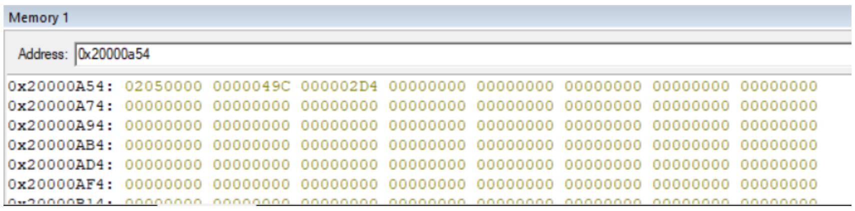
With these inputs, we expect an error because the program tries to convert and empty list to an array.



```
Memory 1
Address: 0x20000054
0x20000054: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000074: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000094: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x200000B4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x200000D4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x200000F4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

Figure 18: case 5 array

As it is expected, we have an empty array.



```
Memory 1
Address: 0x20000a54
0x20000A54: 02050000 0000049C 000002D4 00000000 00000000 00000000 00000000 00000000
0x20000A74: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000A94: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000AB4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000AD4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000AF4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

Figure 19: case 5 errors

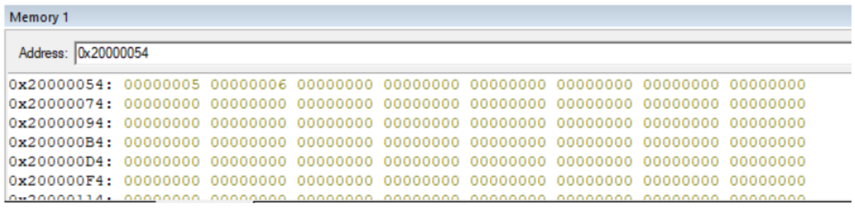
As it is expected, we have an error.

3.6 Case 6

```
                AREA     IN_DATA_AREA, DATA, READONLY
IN_DATA         DCD      0x05, 0x06, 0x89, 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, 0x03, 0x02
END_IN_DATA_FLAG
```

Figure 20: case 6 inputs

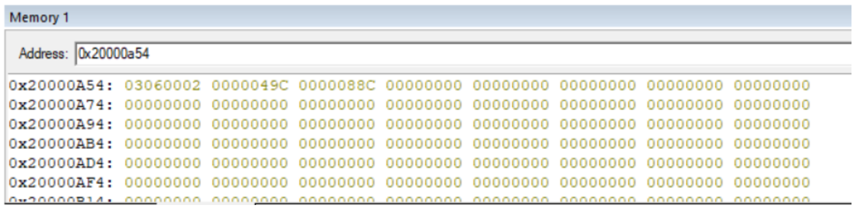
With these inputs, we expect that program will insert 0x05 and 0x06. After that it gives and error. Because there is no operation code such 0x03. Finally it converts linked list to an array.



```
Memory 1
Address: 0x20000054
0x20000054: 00000005 00000006 00000000 00000000 00000000 00000000 00000000 00000000
0x20000074: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000094: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x200000B4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x200000D4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x200000F4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

Figure 21: case 6 array

As it is expected, we have 0x05 and 0x06 in the array in increasing order.



```
Memory 1
Address: 0x20000a54
0x20000A54: 03060002 0000049C 0000088C 00000000 00000000 00000000 00000000 00000000
0x20000A74: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000A94: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000AB4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000AD4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0x20000AF4: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

Figure 22: case 6 errors

As it is expected, we have an error.

4 DISCUSSION

In this experiment, we created a sorted set linked list structure by building wanted functions. As a common design principle or problem solving approach, we start with simple building blocks and create something functional and more complex. This was also the case in this experiment. In the first part of the experiment, we implemented modules with purposes such as Malloc, Free, Insertion, Removing, Initilizations, Current Time in order to use them in the next parts.
In the Handler function we simply called other functions, depending on the operations and any errors that occurred during them. It was interesting to use the shortcuts created by having full control over the specific registers, such as being able to load arguments before deciding which function we are going to call and using the same `CMP` statement for different conditions. In SysTick_Init function we write calculated reload value into corresponding field of control and status register. We calculated reload value with this equation: Reload Value = Period x CPU Frequency - 1 = 731µs x 16MHz - 1 = 11695 Period and CPU frequency are given to us by our instructors. We calculated it as 11695. After that, we clear the current value field of it by loading 0 to a register and storing it into that memory area which is "0xE000E018". Lastly, 1 is written to PROGRAM_STATUS variable, 1 indicates that our program is started, it is done by first loading variable's memory address to 'A' register, loading 1 into 'B' register and storing 'B's value into memory area where 'A' register pointed. SysTick_Stop function simply stops the System Tick Timer, clears the interrupt flag of it and updates the program status. Function first loads control and status register address into R0 and 0 into R1, then stores R1 into the memory area shown by address in R0. After that, function loads address of PROGRAM_STATUS variable into R0 again and 2 into R1. Finally R1 is stored into memory area which is pointed by R0 and function returns to the caller. The value 2 is indicating that program is over. `Malloc` was created and used to do allocate memory space within the program. Using bit masks was fun and having to substitute different operations for functionality Thumb instruction set was missing was interesting. Two substitutions especially stand out:

- Divisions by powers of two had to be substituted with shifts.
- Modulos by powers of two had to be substituted with `AND` operations.

Free was created and used to free space allocated by `Malloc`. Bit masks were used once again, however there is not much to discuss here since we already discussed `Malloc`, and `Free` is very similar to it in composition. Insert function basically inserts the value with the given rules to the linked list. In abstract thinking, it is not a complicated function the

implement thanks to our experince from our *Data Structures* course, but implementing that peculiar insertion function using Assembly language was enlightening and complicated at times with the edge cases. After weeks of our development in our *Microprocessor Systems* course we were able to implement it even though it was challenging. In Remove function, we delete the node with value equal to passed value to the function. Function first checks if current linked list is null or not, if it is null, returns error code 3, means linked list is empty. Then it checks wether given value equal to the first element of the linked list or not. If it is equal, that element will be deleted and after that FIRST_ELEMENT variable shows the element which is next node of removed element. If given element is not equal to first element of the list, function traces along the linked list to find if any element with value equal to given value exists. Whenever it finds the node with equal value, it jumps to deleting branch which deletes current node by connecting previous element to the element next to current element. Function passes the address of removed element to Free function in both scenario to deallocate corresponding memory area. If given value doesn't exist on the linked list, whenever it reaches to end of the list, function returns error code 4. The linked list to array function basically converts linked list to an array. It checks for empty list. If it is empty, then gives an error. If it is not, convert whole list to an array. It starts from head and iterates to the end of the list. It writes data part of all variables one by one.
In WriteErrorLog function we store the error log of the input dataset operations. This function takes parameters such as Index, ErrorCode, Operation, and Data variables. Then, it stores these data to the current available area of the LOG_MEM if the LOG_MEM array is not full. We use INDEX_ERROR_LOG index value to multiply with 32 and 3 to get the memory location inside error log array. Then we check whether error log array is full or not. According to this check we start writing or jump to we_exit. We write to the specified location of 3 words. Depending the size of the variable we are writing such as "Index" as half word, after the write operation we will increment the our special memory location register by necessary word correspondant like 2 for half word and 4 for a word. We also make a function call with "B GetNow". Once we finish all writing operations. We increment the INDEX_ERROR_LOG by one, then continue with the "we_exit". If we did not write into error log array, we would directly jump to "we_exit".
In GetNow function we return the working time of the System Tick Timer in microseconds. This function takes no parameters. We calculate the actual working time using the TICK_COUNT variable and System Tick Timer Current Value Register. We take TICK_COUNT value, then we multiply it with the reload value. In order to get the working time we also add the value of SYST_CVR, SysTick Current Value Register. As the last step we make a division by 16 so we could obtain the value of the working time

in microseconds.

5 CONCLUSION

To sum up, we created a sorted set linked list structure using assembly language. It was really good to work on an important data structure subject. It was also challenging and interesting since we had to work with a low level language where we have control over basic components such as registers. It was challenging because we have to be careful for every detail. It was interesting because this Assembly language gave us the thoughts like we have all the power to control the computers. It felt like the one language to rule them all. Overall, as a team of 5 each and every one of us really enjoyed studying with, working on and achieving this project's goal. We successfully implemented the whole part of the code and tried many test cases. There were few test cases where our code did not work. We reacted fast and as a team fixed all possible corner cases. It was a fulfilling experience for us.

REFERENCES

[1] *Cortex™-M0+ DevicesGeneric User Guide*. 2012.

[2] Kadir Ozlem. *BLG 351E – Microprocessor Systems Recitation Slides*. 2020.