ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 212E

MICROPROCESSOR SYSTEMS
TERM PROJECT

DATE : 31.01.2021

GROUP NO : G30

GROUP MEMBERS:

150160802 : MEHMET CAN GÜN

150170054 : ZAFER YILDIZ

150170063 : BURAK ŞEN

150180012 : MUHAMMED SALİH YILDIZ

150180080 : BAŞAR DEMİR

FALL 2020

Contents

1 INTRODUCTION

In this project, we have implemented a program that performs fundamental linked-list operations for different lengths of inputs based on system tick interrupt for ARM Cortex M0+ using the Assembly Language. The source code template of the term project includes initialization of necessary functions and we have filled them according to system needs with our theoretical microprocessor systems and data structures knowledge.

2 MATERIALS AND METHODS

- ARM Cortex M0+ Microprocessor
- Keil μVision IDE v5

2.1 Initialization

The assembly project that is created with Keil μVision IDE v5 needs configuration in startup file and microprocessor specifications to provide proper working mechanism.

Assembly handler functions are initialized in startup file automatically by IDE but for our project, System Tick Handler must be placed in main function. To handle this condition, we have commented out initialization of the handler from the startup file. Then, we have exported our handler from our code and imported it in reset handler that is placed in startup file as can be seen in Figure 1.

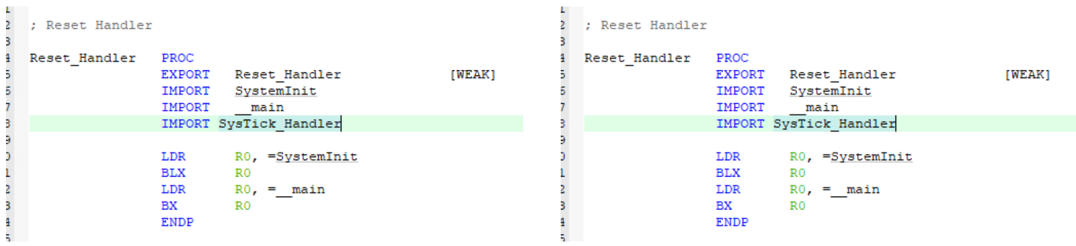

Figure 1: System Tick Handler Configuration

In the project requirements, our central processor unit working frequency given as **64MHz**. In our IDE, processor frequency can be set using Flash configurations menu as seen in Figure 2.

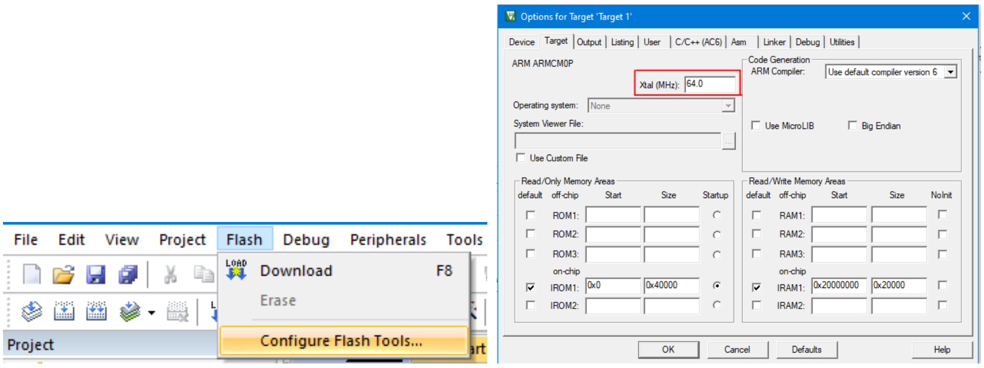

Figure 2: CPU Frequency Configuration

2.2 Main

Main is the base function of the program. Some setup and initialization operations are handled in this function and it checks program status for termination.

The function calls the Clear_Alloc function to reset the area that will be allocated and calls the Clear_ErrorLogs function to reset the area that will be used to write error logs. Also, it performs several initialization functions using Init_GlobeVars and SysTick_Init that are used to initialize global variables and system tick, respectively. The last and the most considerable task of this function is checking whether the program has finished or not. It controls the Program_Status variable in an infinite loop to get status of the program. If it is equal to 2, that means the program has finished then, the function goes to the Stop label to finishes the program.The implementation is shown below in Figure 3.

```
__main                          FUNCTION
                                  EXPORT __main
                                  BL          Clear_Alloc ; Call Clear Allocation Function.
                                  BL  Clear_ErrorLogs ; Call Clear ErrorLogs Function.
                                  BL          Init_GlobVars ; Call Initiate Global Variable Function.
                                  BL          SysTick_Init ; Call Initialize System Tick Timer Function.
                                  LDR R0, =PROGRAM_STATUS ; Load Program Status Variable Addresses.
LOOP                            LDR R1, [R0] ; Load Program Status Variable.
                                  CMP          R1, #2 ; Check If Program finished.
                                  BNE LOOP ; Go to loop If program do not finish.
STOP                            B        STOP ; Infinite loop.
                                  ENDFUNC
```

Figure 3: Implementation of Main Function

2.3 SysTick_Handler

This function is one of the most significant functions. Since it evaluates the given input data and its flag afterwards, leads the program according to the input and returns errors if there are any of them.

First of all, this function increases the Tick_Count by one since; the program calls this function in certain periods whose number is equal to Tick_Count. The second operation in this function is getting input data and input data flag from the input data area and input flag area, respectively. After these operations, the function starts to evaluate the input data and input flag. It checks the flag of the input, and according to the result, it calls Insert, Remove, or LinkedList2Array functions. Furthermore, according to the results of these functions, SysTick_Handler calls the WriteErrorLog function if necessary. After all these operations, the function increments the index of the input data. Finally, it checks if the last input data has been read already or not. If it has been read then, the function calls the SysTick_Stop function else, returns. Also, the implementation is demonstrated below in Figure 4 and Figure 5.

```
SysTick_Handler        FUNCTION
;//-------- <<< USER CODE BEGIN System Tick Handler >>> ----------------------
                                EXPORT SysTick_Handler
                                PUSH {LR} ; Push LR to stack to preserve
                                LDR  R3, =TICK_COUNT ; Address of Tick_Count -> R3
                                LDR  R4, [R3] ; Tick_Count -> R4
                                ADDS R4, R4, #1 ; Tick_Count += 1
                                STR  R4, [R3] ; Store new Tick_Count to address of the Tick_Count
                                LDR  R3, =INDEX_INPUT_DS ; Address of INDEX_INPUT_DS -> R3
                                LDR  R3, [R3] ; INDEX_INPUT_DS -> R3
                                LSLS R3, R3, #2 ; Multiply R3 with 4
                                LDR  R4, =IN_DATA ; Starting address of the Input Data -> R4
                                LDR  R4, [R4, R3] ; Move forward in input data area as index * 4 in bytes and get the next input data
                                LDR  R5, =IN_DATA_FLAG ; Starting address of the Input Data Flag -> R5
                                LDR  R5, [R5, R3] ; Move forward in input data flag area as index * 4 in bytes and get the next input data flag
                                MOVS R0, R4 ; Set input data address in R0 as send parameter
                                PUSH {R0} ; Push input data to stack
                                CMP  R5, #0 ; Check if operation is remove
                                BEQ  REMOVELABEL ; If yes go to REMOVELABEL
                                B           COMPARE1 ; Else go to next if statement
```

Figure 4: Implementation of SysTick_Handler Function

```
REMOVELABEL
                                BL   Remove ; Go to Remove operation
                                MOVS R1, R0 ; Fill R1 with input data to send error
                                MOVS R2, #0 ; Fill R2 with 0 which is operation flag of remove operation
                                B           CHECK_ERROR ; Check if there is any error
COMPARE1
                                CMP  R5, #1 ; Check if operation is insert
                                BEQ  INSERTLABEL ; If yes go to INSERTLABEL
                                B           COMPARE2 ; Else go to next if statement
INSERTLABEL
                                BL   Insert ; Go to Insert operation
                                MOVS R1, R0 ; Fill R1 with input data to send error
                                MOVS R2, #1 ; Fill R2 with 1 which is operation flag of insert operation
                                B           CHECK_ERROR ; Check if there is any error
COMPARE2
                                CMP  R5, #2 ; Check if operation is linked list to array
                                BEQ  LIST2ARR ; If yes go to LIST2ARR
                                B    OP_NOT_FOUND ; Else go to not found statement
LIST2ARR
                                BL          LinkedList2Arr ; Go to linked list to array operation
                                MOVS R1, R0 ; Fill R1 with input data to send error
                                MOVS R2, #2 ; Fill R2 with 2 which is operation flag of linked list to
                                ↪  array operation
CHECK_ERROR
                                POP {R3} ; Pop input data from stack and put it to R3
                                CMP R0, #0 ; Check if the operations return an error
                                BEQ INCINDEX_INPUT ; If not then go to increasing input data index
                                ↪  function
                                B          WRITE_ERROR ; Else go to error write function
OP_NOT_FOUND
                                MOVS R1, #6 ; Fill R1 with 6 which means operation not found in error
                                ↪  codes table
                                MOVS R2, R5 ; Fill R2 with operation code
                                POP {R3} ; Pop LR and Return
WRITE_ERROR
                                LDR R0, =INDEX_INPUT_DS ; Assign R0 to index input address
                                LDR R0, [R0] ; Assign R0 to index input value
                                BL         WriteErrorLog ; Go to error writing function
INCINDEX_INPUT
                                LDR  R3, =INDEX_INPUT_DS ; Get input index address in R3
                                LDR  R4, [R3] ; Get input index value in R4
                                ADDS R4, R4, #1 ; Incremenet input index value by one
                                STR  R4, [R3] ; Store updated input index value
                                LDR  R3, =END_IN_DATA ; Get the end point of the input data
                                LDR  R5, =IN_DATA ; Get the starting point of the input data
                                SUBS R3, R3, R5 ; Get their difference
                                LSRS R3, R3, #2 ; Divide it by 4 to get rid of address size
                                CMP         R3, R4 ; Check if result is equal to current index of input
                                ↪  data set
                                BNE         handler_stop ; If no, go to handler stop
                                BL  SysTick_Stop ; Else, go to systick stop
handler_stop
                                POP  {PC} ; Return
;//-------- <<< USER CODE END System Tick Handler >>>
↪  ------------------------
                                ENDFUNC
```


Figure 5: Implementation of SysTick_Handler Function

2.4 SysTick_Init

In this assignment, we are expected to handle all operations within the interrupt function and also period of the system tick timer interrupt is given as **946 µs**. In microprocessors, there is a memory area that is responsible for system tick interrupt properties such as enable, clock source. Programmers who use system tick interrupt must manipulate related bits according to system needs.

System timer was constructed on some fundamental principles that are directly dependent to reload value, processor frequency also interrupt system consist of counter, reload value, system clock and flags. In each system clock, mechanism decrements the counter value by 1 until it reaches to 0. If the counter value reaches to 0, mechanism sets the count flag as 1 and it triggers system interrupt. Then, it loads the counter with reload value and thus, the process repeats itself. Therefore, the period of the system tick interrupt directly dependent to CPU frequency that adjusts speed of the decrement operation and reload value that sets total time (how many clock required). For given CPU frequency and interrupt period, the formula for the **reload value** is given in Equation 1.

$$Reload\,Value = (T_{interrupt} * F_{CPU}) - 1$$

Equation 1: Formula for Reload Value

Our system works with **64 MHz** CPU clock frequency and required interrupt period is given us as **946 µs**. After plugging these values into formula, we found reload value as **60543**.

Table 1: ARM Cortex M0+ System Tick Registers and Their Addresses

Address	**Description**
0xE000E010	System Tick Control and Status Register
0xE000E014	System Tick Reload Value Register
0xE000E018	System Tick Current Value Register

In ARM Cortex M0+ microprocessors, current value, reload value and control registers can be manipulated using addresses that are given in Table 1. In SysTick_Init function, we have performed memory operations on these addresses using values that are calculated above. We store reload value as 60543 in 0xE000E014, current value as 0 in 0xE000E018 and control flag as 0x111 in 0xE000E010 respectively. The control flags determines system enable, clock source, tick interrupt as seen in Table 2 . In our program, we have to enable counter, interrupt and we have to use clock of the CPU. Thus, we have set all flags as 1.

At the end of the program, we have to update program status. We set it with 1, it means that timer is started.

Table 2: Control Flags

Flag	Description	Value
Clock Source	Chooses Clock Source	1
Tick Interrupt	Enables System Tick Interrupt	1
Enable	Enables Counting Mechanism	1

Our implementation is given below as Figure 6.

```
SysTick_Init        FUNCTION
;//-------- <<< USER CODE BEGIN System Tick Timer Initialize >>> ----------------------
                                LDR  R0, =0xE000E010 ; it takes system tick control memory address
                                LDR  R1, =60543 ; keeps reload value
                                STR  R1, [R0, #4] ; stores reload value into memory
                                MOVS R1, #0 ; keeps current value as 0
                                STR  R1, [R0, #8] ; stores current value into memory
                                MOVS R1, #7 ; sets flags as 1 (enable, tickint, clksource)
                                STR  R1, [R0] ; stores flags in proper memory address
                                MOVS R0, #1 ; keeps program status value as 1 (timer started)
                                LDR  R1, =PROGRAM_STATUS ; takes address of program status variable
                                STR  R0, [R1] ; writes 1 to program status
                                BX   LR ; branches with link register
;//-------- <<< USER CODE END System Tick Timer Initialize >>> ------------------------
                                ENDFUNC
```

Figure 6: Implementation of SysTick_Init Function

2.5 SysTick_Stop

The main purpose of this function is preparing the program for termination by manipulating program status and system tick interrupt mechanism.

We have already mentioned the system tick registers and their purposes in SysTick_Init subsection. To stop system tick interrupt, we have to set tickint and enable flags of the register as 0. Tickint flag disables interrupt trigger mechanism and enable flag prevents counting respect to CPU clock. Thus, we have stored **#4** (100) in system tick control and status register that's address given as 0xE000E010. After disabling system tick interrupt, we have read address of the PROGRAM_STATUS variable and we have stored #2 value that refers to "**All data operations finished.**" in this register.

At the end of the this function, program stops system tick interrupt and exits from loop that is located in main function. Finally, it branches to STOP label continuously and it waits for the program to be terminated by an user.

Our SysTick_Stop function implementation is given below as Figure 7.

```
SysTick_Stop        FUNCTION
                                LDR  R0, =0xE000E010 ; it takes system tick control memory address
                                MOVS R1, #4 ; sets flags as 100 (enable=0, tickint=0, clksource=1)
                                STR  R1, [R0] ; stores flags in proper memory address
                                MOVS R0, #2 ; keeps program status value as 2 (All data operations finished.)
                                LDR  R1, =PROGRAM_STATUS ; takes address of program status variable
                                STR  R0, [R1] ; writes 2 to program status
                                BX   LR ; branches with link register
                                ENDFUNC
```

Figure 7: Implementation of SysTick_Stop Function

2.6 Clear_Alloc

In this function, we clear the allocation table to get rid of undesirable values in the memory space. Firstly, we have created a loop to iterate all of the allocation table. For each step, we check index of current position. If we are not at the end, we set 0 to that table-space and increase our for loop index. If our index is equal to the allocation table size, we branch with link register. As can be seen in Figure 8, the implementation of the Clear_Alloc is given below.

```
Clear_Alloc                FUNCTION
                                MOVS R0, #0 ;Set R0: i = 0
                                MOVS R3, #0 ; Set R3 as 0
                                LDR  R1, =AT_SIZE ; get Allocation Table size
                                LDR  R2, =__AT_Start ; get Allocation Table's address
                                B         COMPARECLEAR ; branch to compareclear label
LOOPCLEAR
                                STR  R3, [R2,R0] ; clear the allocation table's R2th address's R0th index
                                ADDS R0, R0, #4 ; increase the index counter
COMPARECLEAR
                                CMP         R0, R1 ; compare index and allocation table size
                                BNE         LOOPCLEAR ; if not equal branch to LOOPCLEAR label
                                BX         LR ; returns
                                ENDFUNC
```

Figure 8: Implementation of Clear_Alloc Function

2.7 Clear_ErrorLogs

In this function, we are expected to clear error logs array that is defined globally.

Firstly, we have created a for loop to iterate all of the error logs area. For each step, we check index of current position. If we are not at the end, we set 0 to that error log space and increase our for loop index. If our index is equal to the error log area size, we branch with link register. The code of the Clear_ErrorLogs in Figure 9 is given below.

```
Clear_ErrorLogs       FUNCTION
                                MOVS R0, #0 ;Set R0: i = 0
                                MOVS R3, #0 ; Set R3 as 0
                                LDR  R1, =LOG_ARRAY_SIZE ; get log array size
                                LDR  R2, =__LOG_Start ; get log array's address
                                B          COMPAREERCLEAR ; branch to compareerclear label
LOOPERCLEAR
                                STR  R3, [R2,R0] ; clear the log array's R2th address's R0th index
                                ADDS R0, R0, #4 ; increase the index counter
COMPAREERCLEAR
                                CMP          R0, R1 ; compare index and log array size
                                BNE          LOOPERCLEAR ; if not equal branch to LOOPERCLEAR label
                                BX          LR ; returns
                                ENDFUNC
```

Figure 9: Implementation of Clear_ErrorLogs Function

2.8 Init_GlobVars

In this function, we are expected to initialize all global variables with **#0**. It was one of the simplest functions in the project. We simply loaded R1 register with the address of the global variables and then we stored **#0** to these addresses for each of TICK_COUNT, FIRST_ELEMENT, INDEX_INPUT_DS, INDEX_ERROR_LOG, PROGRAM_STATUS variables. As can be seen in Figure 10, the implementation of the Init_GlobVars is given below.

```
Init_GlobVars       FUNCTION
                                MOVS R0, #0
                                LDR R1, =TICK_COUNT ;TICK_COUNT is initialized with #0
                                STR R0, [R1]
                                LDR R1, =FIRST_ELEMENT ;FIRST_ELEMENT is initialized with #0
                                STR R0, [R1]
                                LDR R1, =INDEX_INPUT_DS ;INDEX_INPUT_DS is initialized with #0
                                STR R0, [R1]
                                LDR R1, =INDEX_ERROR_LOG ;INDEX_ERROR_LOG is initialized with #0
                                STR R0, [R1]
                                LDR R1, =PROGRAM_STATUS ;PROGRAM_STATUS is initialized with #0
                                STR R0, [R1]
                                BX  LR ;branch with link register
                                ENDFUNC
```

Figure 10: Implementation of Init_GlobVars Function

2.9 Malloc

In this function, we are expected to search allocation table that is given as global variable to find a place that points unused memory address in DATA_MEM and apply proper allocation procedures such as writing 1 to allocated bit. Each bit of the allocation table refers to a node that has 8 byte size and it means that each word of the allocation table can use for 32 different linked-list node allocation. Also the principles are visualized in Figure 11.

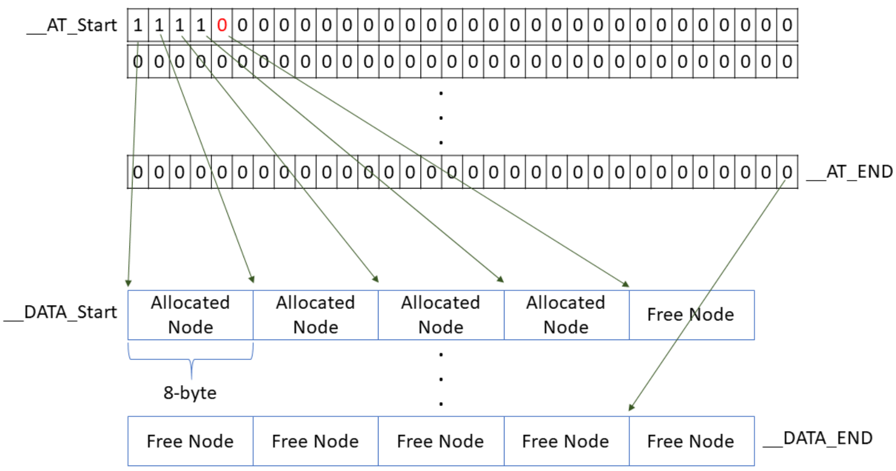

Figure 11: Visualization of Node Allocation Principles

Malloc function is constructed on two nested for loops that are used for search operation. These loops allow us to divide our problem into sub-problems. In the parent for loop, we have iterated on **each word** (4-byte) of allocation table that's size is given as AT_SIZE and in the inner loop we have iterated on **each bit** of the word. In the inner-loop, we have to check value of the each bit and we have solved this problem using mask value that is 1. In each iteration, we perform AND operation using our mask value and identify current bit shows allocated node or not according to Equation 2. At the end of the iteration, we shift our mask value to left by one to provide a code structure that checks each bit of the word separately. After the mask operation if the the result value is 0, it means that value of the current bit is 0 that indicates current bit is free. After finding the empty node, program allocates bit using OR operation that writes 1, its mechanism is given in Equation 3, to current bit and directly branches to FOUND label.

$$01010\textcolor{red}{0}01110 \cdot 00000\textcolor{red}{1}00000 = 00000\textcolor{red}{0}00000$$

$$01010\textcolor{red}{1}01110 \cdot 00000\textcolor{red}{1}00000 = 00000\textcolor{red}{1}00000$$

Equation 2: Mask Operation Using AND

$$01010\textcolor{red}{0}01110 + 00000\textcolor{red}{1}00000 = 01010\textcolor{red}{1}01110$$

Equation 3: Mask Operation Using OR

After the iteration on allocation table, two cases can be occurred that are finding or not finding a free space. To check these conditions, we have used labels and we have placed not found instructions between for loop and found label. By this way, we determine which condition appeared. If program exits for loop without branch operation, it means that there is not any free space and program returns 0 that indicates linked-list is full. If program finds a free node, then it must reach the corresponding node address. To perform these operation, firstly we have to determine index of the allocated node. As we stated in previous paragraph, we used two nested for loops. The parent for loop gives us index of the word that is called as i and inner loop gives us the position of the node in this word that is called as j. Therefore, we have performed $\mathbf{i} * \mathbf{32} + \mathbf{j}$ operation to find index of the related node. After this operation, we have added index value to __DATA_Start address by multiplying with 8 (each node contains 8 bytes) and we have returned this address value. Our assembly implementation is given below as Figure 12.

```
Malloc                              FUNCTION
;//-------- <<< USER CODE BEGIN System Tick Handler >>> ----------------------
                                MOVS R4, #0 ; initializes i with 0
                                LDR  R6, =AT_SIZE ; takes size of the allocation table
                                LDR  R7, =__AT_Start ; takes start address of the allocation table
                                B          COMPARESEARCH ; branches to compare label
SEARCH
                                LDR  R5, [R7,R4] ; takes current word (32-bit) to R5
                                MOVS R2, #0 ; initializes j with 0
                                MOVS R3, #1 ; creates mask value with 1
                                B          COMPAREWORD ; branches to compare word
LOOPWORD
                                MOVS R1, R3 ; movs mask value to R1
                                ANDS R1, R5, R1 ; maskes current word with 1 (checks first bit)
                                CMP  R1, #0 ; compares masked value with 0
                                BNE  NOTFOUND ; if it is not 0, it means that current bit is not empty
                                ORRS R5, R5, R3 ; if it free area, it allocates current bit with OR operation
                                STR  R5, [R7,R4] ; stores allocated word to memory
                                B          FOUND ; branches to found label
NOTFOUND
                                ADDS R2, R2, #1 ; increments j with 1
                                LSLS R3, R3, #1 ; shifts mask value with 1
COMPAREWORD
                                CMP          R2, #32 ; compares j with 32
                                BNE          LOOPWORD ; if not equal, it continues to iterating
                                ADDS R4, R4, #4 ; incements i with 4 byte
COMPARESEARCH
                                CMP          R4, R6 ; compares i with allocation table size
                                BNE          SEARCH ; if not equal, branches to search word
                                MOVS R0, #0 ; meaning that the linked list is full.
                                B end_Malloc ; bracnhes to end of the suction
FOUND
                                LSLS R4, R4, #3 ; it shifts index number with word number*32
                                ADDS R4, R4, R2 ; it adds j value to index number
                                LDR  R6, =__DATA_Start ; takes data start address
                                LSLS R4, R4, #3 ; it multiplies index value with 8 (each node is 8 byte)
                                ADDS R6, R4 ; it reaches to allocates node
                                MOVS R0, R6 ; returns allocated node address
end_Malloc
                                BX   LR ; branches with link register
;//-------- <<< USER CODE END System Tick Handler >>> ------------------------
                                ENDFUNC
```

Figure 12: Implementation of Malloc Function

2.10 Free

This function is expected to mark as unused the given location, allocated and marked as used by the Malloc function earlier, in the allocation table. It is known that the allocation table consists of lines that has 32-bit data. So, the location of a node can be defined with a line number and a bit number in the allocation table. The main idea of this function is to analyze the given address and finding its exact location in the allocation table as lines and bits.

First of all we need to find the exact order of the given node. Dividing the distance between __DATA_Start, and address of the given node in the memory by 8, the size of a node in terms of bytes, gives the order of that node. When the order is divided by 32 which is size of a line, exact line number of that node can be found. Also, when multiplication of line number and size of a line (32-bit) subtracted from the order, exact bit of the node can be found. For example, if the order of a node is 42, dividing 42 by 32 gives 1 which is line number of that node. Also, subtraction 32 from 42 gives 10 which is bit number of that node. The key point here is word numbers and bit numbers are 0 indexed.

Secondly, we need to change the exact bit to zero in the allocation table. To manage that, we get the address of that line by adding multiplication of the line number with 4 (4-byte) to __AT_Start. Then, we have the data of the line that should be changed. To set the exact bit as 0, we created a number with 32-bits that consists of 1's except the least significant bit. Afterward, we have placed the 0 bit to the required position through circular shifting and its example given in Example 4. Then, doing "AND" operation between the line and shifted data marked the right bit as 0 as expected as given in Equation 5. Later on, the result of this operation is stored in the allocation table as an updated line.

$$...11111111110 \ggg 32 - j(bitNumber) = ...11111011111$$

Equation 4: Circular Shift Operation Example

$$10100111010 \cdot 11111011111 = 10100011010$$

Equation 5: Mask Operation Using AND

At the end of all these operations, the bit that points to a given address location is marked as 0, which means it is an unused memory space. So in the next allocation operations that memory space will be shown as free. You can see the implementation below in Figure 13.

```
Free                            FUNCTION
;//-------- <<< USER CODE BEGIN Free Function >>> ----------------------
                                LDR  R6, =__DATA_Start ; Starting address of the data -> R6
                                MOVS R3, R0 ; Input data -> R3
                                SUBS R3, R3, R6 ; Get difference between given node and start address
                                LSRS R3, R3, #3 ; Divide difference by 8 to find order of the given
                                ↪  address
                                MOVS R4, R3 ; Copy order to R4
                                LSRS R5, R3, #5 ; Divide the order by 32 to get line number
                                LSLS R3, R5, #5 ; Multiply line number with 32
                                SUBS R4, R4, R3 ; Get bit number in the line by subtracting 32 times line
                                ↪  number from exact order number
                                LDR  R7, =__AT_Start ; Starting address of the Allocation Table -> R7
                                LSLS R5, R5, #2 ; Multiply line with 4
                                LDR         R2, =0xFFFFFFFE ; Create 1111 1111 1111 1110 number
                                MOVS R6, #32 ; Assign 32 to R6
                                SUBS R4, R6, R4 ; 32 - bit number
                                RORS R2, R2, R4 ; Do circular shift for 32 - bit number times
                                LDR  R3, [R7, R5] ; Store current line data to R3
                                ANDS R3, R3, R2 ; Current line data && updated lien data
                                STR  R3, [R7, R5]; Store new line data to allocation in allocation table
                                BX   LR ; Return
;//-------- <<< USER CODE END Free Function >>> ------------------------
                                ENDFUNC
```

Figure 13: Implementation of Free Function

2.11 Insert

Insert function is in charge of adding a new data to corresponding location in the linked list. To achieve that our general algorithm has 3 steps as in below:

1. Allocate a new node with the given data parameter
2. Search for the correct order for the new node
3. Connect new node in a suitable location in the linked list

Nevertheless, there are several cases we need to think about while inserting a new data to the linked list. Thus, we had to control each of these cases one by one throughout the code flow. These cases are insert into empty linked list, insert as first element, insert as intermediate element and insert as last node of the linked list.

2.11.1 Insert into Empty Linked List

To check that whether the current linked list empty or not, we get value of the global **FIRST_ELEMENT** variable. FIRST_ELEMENT keeps the address of the first node in the linked list like a head pointer and if the linked list is empty its value is 0 to represent NULL in high level programming languages. So, we loaded the value of head pointer to a register. Then, we compared the value of loaded register with 0 to check whether the linked list is empty or not. If it is not equal to 0, code branches to **is_not_empty** label which will be explained in below cases. If it is 0, that means the linked list is empty, so we should add the new node as the first node of the linked list and we should also change the value of FIRST_ELEMENT by the address of the new node.

Firstly we allocated memory to add the new node to the linked list by our **Malloc()** function. Malloc function returns the empty address value from the allocation table or 0 if the allocation table is full. Thus, we checked the return value of the Malloc function. If it is 0, code branches to **ERROR1** label to set the error code as 1. If there is an empty space in the allocation table, we used this space for the new node by using the address value that is returned from the Malloc() function. Firstly, we set the data of this node to given data parameter of the Insert() function. Then, we set the next address of the new node as 0, because it is the both first and last node in the linked list as it is the only node in the linked list. After all of these operations, code branches to **SUCCESS** label to represent that insertion is completed successfully. Here is the implementation of insert into empty linked list case:

```
Insert                          FUNCTION
                                  PUSH {LR} ;push LR to stack to protect its value
                                  MOV R3, R0 ;mov the given data parameter to R3
                                  LDR R1, =FIRST_ELEMENT ;R1 = address of the FIRST_ELEMENT global variable
                                  LDR R2, [R1] ;R2 = address of the first node
                                  CMP R2, #0 ;check if first node is null
                                  BNE is_not_empty ;if it is not null go to is_not_empty label
                                  PUSH {R1-R3} ;push {R1-R3} to stack to protect its value
                                  BL Malloc ;go to Malloc label and return the address of the new node in R0
                                  POP {R1-R3} ;pop {R1-R3} from stack with their old values
                                  CMP R0, #0 ;check if R0 is #0
                                  BEQ ERROR1 ;if it is 0 that means allocation table is full, so go to ERROR1 label
                                  STR R3, [R0] ;new_node->data = R3 (R3 is given data parameter)
                                  STR R0, [R1] ;FIRST_ELEMENT = address of the new node
                                  MOVS R2, #0 ;R2 = 0
                                  STR R2, [R0, #4] ;new_node->nextAddress = #0,because it is the only node in linked list
                                  B success ;go to success label
```

Figure 14: Insert into Empty Linked List

2.11.2 Insert as First Element

In this case, the linked list is not empty, but the new coming node will be the new smallest value, new first element, in the linked list. To check whether this case occurs or not, we compared the given data parameter with the data of the first node. If the new data is greater than the data of the first node, code branches to **greater** label which will be explained in next cases. If they are equal, code branches to **ERROR2** label to set the error code as 2 for the **WriteErrorLog** function because that means the new data is already in the linked list. After these controls, the last possibility is that the new data is smaller than the first data in the linked list and we should add new data as the new first node in the linked list. To perform that, we allocated new space by our **Malloc()** function. The value of the empty space is returned by R0 register from Malloc() function. If the value is 0, that means there is no empty space in allocation table, so code branches to **ERROR1** label to set error code as 1 and to end Insert() function. If there is a suitable location in memory, we assigned the data parameter to data of the allocated node. After that, we assigned the address of the current FIRST_ELEMENT to next address attribute of the new node, because the the new FIRST_ELEMENT will be the new node as it contains the new smallest data in the linked list. As a last step, we changed the value of FIRST_ELEMENT by the address of the new node. Finally, code branches to **success** label to represent that insertion operation is completed successfully. Here is the implementation of insert as first element case:

```
is_not_empty
                                LDR R5, [R1] ;R5 = address of the first node
                                LDR R5, [R5] ;R5 = first_node->data
                                CMP R3, R5 ;compare the given data parameter with data of first_node
                                BHI greater ;if the new data is greater than the data of first node go to greater label, else insert as first node
                                BEQ       ERROR2 ;if they are equal that means the given data is already in linked list, go to ERROR2 label
                                PUSH {R1-R3} ;push {R1-R3} to stack to protect its value
                                BL        Malloc ;go to Malloc label and return the address of the new node in R0
                                POP {R1-R3} ;pop {R1-R3} from stack with their old values
                                CMP R0, #0 ;check if R0 is #0
                                BEQ ERROR1 ;if it is 0 that means allocation table is full, so go to ERROR1 label
                                STR R3, [R0] ;new_node->data = R3 (R3 is given data parameter)
                                LDR R2, [R1] ;R2 = address of the first node
                                STR R2, [R0, #4] ;new_node's nextAddress is equal to address of first element
                                STR R0, [R1] ;FIRST_ELEMENT = address of the new node, because we inserted at start of the linked list
                                B         success ;go to success label
```

Figure 15: Insert as First Element

2.11.3 Insert as Intermediate Element

In this case, the new coming data is bigger than the smallest data in the linked list, but we have to determine the correct order for it. To implement this case, we keep 2 registers like linked list pointers as tail and traverse registers. Traverse register keeps the address of the current node while searching and tail register always shows the previous node from the node that is pointed by traverse register. To determine the correct order for the new coming data, we implemented a while loop. In this loop, we compared the new coming data with the current node's data. If the new data is greater than current node's data, tail register is loaded with the value of traverse register and traverse register is loaded by the address of the next node in the linked list to continue search operation for the correct location. If the new data is equal to current node's data code branches to **ERROR2** label to set the error code as 2. Last possibility is that new data is smaller than current node's data and that means we found the correct order for insertion operation. To insert the new data into the linked list, we allocated space by our Malloc() function and set the data of the allocated node as given data parameter of Insert() function. As a final step, we set the next address attribute of the new allocated node as the value of traverse register and the value of next address attribute of the node which is pointed by tail register as the address of the new node to connect the node between the nodes which are pointed by tail and traverse registers. After all of these operations, code branches to success label to represent that insertion operation is completed successfully. Here is the implementation of insert as intermediate element case:

```
greater
        LDR R5, [R1] ;R5 = tail
        LDR R6, [R5, #4] ;R6 = traverser
while_label
        CMP R6, #0 ;check whether it reaches end of the linked list or not
        BEQ        insert_to_end ;if it reaches go to insert_to_end label
        LDR R4, [R6] ;R4 = data of the node that is pointed by traverser
        CMP R3, R4 ;compare the given data parameter with data of the current node
        BEQ ERROR2 ;if they are equal that means the given data is already in linked list, go to ERROR2 label
        BHI while_end ;if R3>R4, we need to continue to our search operation
        PUSH {R3, R5, R6} ;push {R3, R5, R6} to stack to protect its value
        BL         Malloc ;go to Malloc label and return the address of the new node in R0
        POP {R3, R5, R6} ;pop {R3, R5, R6} from stack with their old values
        CMP R0, #0 ;check if R0 is #0
        BEQ ERROR1 ;if it is 0 that means allocation table is full, so go to ERROR1 label
        STR R3, [R0] ;new_node->data = new data (parameter)
        STR R0, [R5, #4] ;tail->next = address of the new node
        STR R6, [R0, #4] ;new_node->next = traverser
        B success ;branch to success label
while_end
        MOVS R5, R6 ;tail = traverser
        LDR R6, [R6, #4] ;traverser = traverser->next
        B while_label ;branch to start of the loop
```

Figure 16: Insert as Intermediate Element

2.11.4 Insert as Last Element

Insertion to the end of the linked list is very similar to insertion as intermediate element which we explained above. In search operation to determine the correct position for the new data, we used tail and traverse registers and we simply slided these register to the end of the linked list. If the new data is bigger than all of the datas in the linked list, our search operation continues until the end of the linked list inside our while loop. To check whether we reach the end of the linked list or not, we compared the value of traverse register by #0, because the next address attribute of the last node in the linked list is #0 to represent NULL value in high level programming languages. If the correct location for the new coming data is the end of the linked list, we loaded next address attribute of the node which is pointed by tail pointer as the address of the new node that keeps the data that will be inserted. We also loaded next adress attribute of the new node as #0, because it is the new last element in the linked list. Here is the implementation of insert as last element case:

```
insert_to_end
                                PUSH {R3, R5} ;push {R3, R5} to stack to protect its value
                                BL Malloc ;go to Malloc label and return the address of the new node in R0
                                POP {R3, R5} ;pop {R3, R5} from stack with their old values
                                CMP R0, #0 ;check if R0 is #0
                                BEQ ERROR1 ;if it is 0 that means allocation table is full, so go to ERROR1 label
                                STR R3, [R0] ;new_node->data = new data (parameter)
                                STR R0, [R5, #4] ;tail->next = address of the new node
                                MOVS R7, #0 ;R7 = 0
                                STR R7, [R0, #4] ;new_node->next = #0, because it is the new last node in linked list
                                B success ;branch to success label
ERROR1
                                MOVS R0, #1 ;R0 = 1 (error code)
                                B end_insert ;branch to end_insert label
ERROR2
                                MOVS R0, #2 ;R0 = 2 (error code)
                                B end_insert ;branch to end_insert label
success
                                MOVS R0, #0 ;R0 = 0 (success)
end_insert
                                POP {PC} ;pop LR to pc from stack
;//-------- <<< USER CODE END Insert Function >>> ------------------------
                                ENDFUNC
```

Figure 17: Insert as Last Element

2.12 Remove

Remove function is in charge of removing given data from the linked list. To achieve that our general algorithm has 3 steps as in below:

1. Search for the node the keeps the target data
2. Remove the node from the linked list
3. Deallocate the node from the allocation table

At the start of the remove function, firstly we checked whether the linked list is empty or not. To control this situation, we loaded a register by the value of the global

FIRST_ELEMENT variable. Then, we compared the value of this register by **#0**. If they are equal that means the current linked list is empty, so code branches to **ERROR3** label to set the error code as 3.

If the linked is not empty, we search for the target node whose data is equal to data that will be removed. We have 2 different case in remove operation which are remove first element and remove other elements from the linked list.

2.12.1 Remove First Element

If our target data is kept at the first node of the linked list, our code branches to **REMOVEFIRST** label. In this label, we change the value of the FIRST_ELEMENT global variable, because we removed the first element in this case and address of the first node is changed. After remove operation, our new first node is the second node in the linked list, so we stored the address of the second node in global FIRST_ELEMENT variable. Finally, we call our **FREE** function by address of the removed node as a parameter to it.

2.12.2 Remove Other Elements

To remove elements other than the first element, firstly we searched the target data in the linked list. For search operation method we implemented is similar to one we used in insertion operation. We kept again 2 register, one of them is traverse register and the other one is tail register in a while loop. At the start of the loop, we compared the value of the data that will be removed by the data of the current node that is pointed by traverse register. If they are equal that means we found the node that will be removed, so code branched to **REMOVENODE** label. If they are not equal, tail register is loaded with the value in traverse register and the traverse register is loaded by next address attribute of the node that is pointed by traverse register. Another situation we have to check, the given data is not in the linked list and we controlled this case by comparing the value of the traverse register by **#0**. If it is equal to #0, that means the target data is not in the linked list. Thus code branches **ERROR4** label to set the error code as 4. In **REMOVENODE** label, the connection of the node that keeps the target data is destroyed. We made this by simply setting the next address attribute of the node that is pointed by tail register as next address attribute of the node that is pointed by traverse register. After we destroyed the connection of the node, we also deallocated the space for the deleted node from the allocation table by our **FREE** label. After all of these operations RO register is loaded by **#0** to represent that remove operation is completed successfully.

```
Remove                          FUNCTION
;//-------- <<< USER CODE BEGIN Remove Function >>> ----------------------
                                PUSH {LR} ;push LR to stack to protect its value
                                MOVS R1, R0 ;mov the given data parameter to R1
                                LDR  R2, =FIRST_ELEMENT ;R2 is the address of FIRST_ELEMENT global variable
                                LDR  R3, =FIRST_ELEMENT ;R3 is the address of FIRST_ELEMENT global variable
                                LDR  R3, [R3] ;R3 = address of the first node
                                CMP  R3, #0 ;check if R3 = 0 for empty linked list check
                                BEQ  ERROR3 ;if linked list is empty branch to ERROR3 label
                                LDR  R5, [R3] ;R5 = address of the first node
                                CMP  R1, R5 ;compare the given data parameter with data of the first node
                                BEQ  REMOVEFIRST ;if they are equal branch to REMOVEFIRST label to remove the first node
FINDTARGET
                                CMP  R3, #0 ;check if it reaches end of the linked list
                                BEQ  ERROR4 ;if we reach end of the linked list that means the given is not in linked list, so branch to ERROR4 label
                                LDR  R4, [R3] ;R4 = data of the current node
                                CMP  R1, R4 ;check if it is the target data
                                BEQ  REMOVENODE ;if current data is equal to data that will be deleted, branch to REMOVENODE label
                                MOVS R2, R3 ;R2 = R3 (current node)
                                LDR  R3, [R3, #4] ;R3 = R3->next (R3 is the traverser to iterate over linked list)
                                B           FINDTARGET ;branch to FINDTARGET label to continue search operation
REMOVEFIRST
                                LDR  R6, [R3, #4] ;R6 is loaded with the address of the second node
                                LDR  R7, =FIRST_ELEMENT ;R7 is the address of FIRST_ELEMENT global variable
                                STR  R6, [R7] ;FIRST_ELEMENT = R6 (address of the second node)
                                MOVS R0, R3 ;R0 = address of the deleted node
                                B           CALLFREE ;branch to CALLFREE label
REMOVENODE
                                LDR  R5, [R3, #4] ;R5 = address of the next node of deleted node
                                LDR  R0, [R2, #4] ;R0 = address of the deleted node for Free operation
                                STR  R5, [R2, #4] ;previous_node->next = deletednode->next
CALLFREE
                                BL   Free ;branch to Free label
                                MOVS R0, #0 ;R0 = 3 (success)
                                B END_REMOVE ;branch to END_REMOVE label
ERROR3
                                MOVS R0, #3 ;R0 = 3 (error code)
                                B END_REMOVE ;branch to END_REMOVE label
ERROR4
                                MOVS R0, #4 ;R0 = 4 (error code)
END_REMOVE
                                POP {PC} ;pop LR to pc from stack
                                ENDFUNC
```

Figure 18: Implementation of Remove Function

2.13 LinkedList2Arr

In this function, we were expected to implement a function that can convert our linked list to an array.

Firstly, we had to check if our linked list is empty or not, for this we check if our linked list's head is null or not if it is null we branch to ERROR5 label. If our linked list is not empty, we continue with LOOPSTART label.

In the LOOPSTART label, we had to clear all space in the data array. By iterating each of the data addresses from the data array and storing 0 to each address, we have cleared the data array. If our data address is the end of the data array then we branch to L2ARR label to convert our linked list to an array.

In the L2ARR label, we firstly get the linked list's first element's address and we get data arrays address, after this we enter a for loop in this loop we first check if our node is null or not. If our node is null, we branch to L2ARR_SUCCESS label; if not we execute the loop. Inside the loop, we get the data from the linked list and store that data to data array, to iterate data array we increment array address, and to iterate linked list we get the next node's address and store that in a register that is also our comparison register for our for loop. If the value of this register is null, we branch to L2ARR_SUCCESS label. If we branch to L2ARR_SUCCESS label, we set error_code to 0.

In the ERROR5 label, we just set our error_code to 5 with loading 5 to R0 register. At the end of the function, we branch with link register to wherever we call the function. We can see our implementation at Figure 19.

```
LinkedList2Arr        FUNCTION
;//-------- <<< USER CODE BEGIN Linked List To Array >>> ----------------------
                                LDR  R0, =FIRST_ELEMENT ; get the first element's address's address
                                LDR  R0, [R0] ; get firs element's address
                                CMP  R0, #0 ; if it is compare if it is null
                                BEQ  ERROR5 ; if it is null linkedlist is empty then branch to error5 label
                                MOVS R0, #0 ; set R0: i = 0
                                LDR  R1, =ARRAY_MEM ; get start address of the array
                                LDR  R2, =__ARRAY_END ; get end address of the array
LOOPSTART
                                CMP  R1, R2 ; compare if our traversed address is end adress
                                BEQ  L2ARR ; if they are equal that means we cleared array succesfully
                                STR  R0, [R1] ; clear each addresses value
                                ADDS R1, R1, #4 ; traverse the addresses
                                B          LOOPSTART ; branch to LOOPSTART label and loop all the array
L2ARR
                                LDR  R0, =FIRST_ELEMENT ; get the head of the first element
                                LDR  R0, [R0] ; get address of the first element
                                LDR  R1, =ARRAY_MEM ; get the start address of the array
L2ARRLOOP
                                CMP  R0, #0 ; compare if nodes address is null
                                BEQ  L2ARR_SUCCESS ; branch to L2ARR_SUCCESS label
                                LDR  R2, [R0] ; get the data from R0
                                STR  R2, [R1] ; store the data to array
                                ADDS R1, R1, #4 ; increment array address
                                LDR  R3, [R0, #4] ; get the next node's address to R3
                                MOVS R0, R3 ; MOV R3 to R0 to make comparison
                                B          L2ARRLOOP ; branch to L2ARRLOOP
ERROR5
                                MOVS R0, #5 ; if linkedlist is empty set error code to 5
                                B         END_L2ARR ; end Link list to array with errorcode 5
L2ARR_SUCCESS
                                MOVS R0, #0 ; if all operations are success error code is 0 which means no error
END_L2ARR
                                BX LR ; branches with link register
;//-------- <<< USER CODE END Linked List To Array >>> ------------------------
                                ENDFUNC
```

Figure 19: Implementation of LinkedList2Arr Function

2.14 WriteErrorLog

In the WriteErrorLog function we just gather our necessary data and store them in the Error Log Array.

First of all, we take the log memory and error log index, we can observe how many errors occurred with error log index. With this information, we can easily find the last error log data and we can insert new error log to the end of the error log array. Before inserting any error log, we have to check if our error log array has any space or not because we use the memory sequential manner and we have a possibility that we can write an error log on top of our other data. We just control if we are at the end of the error log array and if we are at the end of it we just branch to end_writelog label. After these controls, we get the current time with GetNow function and store all the data and current time to memory with the given format. And finally, we increase the error log index by 1 at the end of the function because we add a new error log. We can see our implementation at Figure 20

```
WriteErrorLog       FUNCTION
                                PUSH {LR} ; Store LR to stack
                                LDR R4, =LOG_MEM ; Get Start of Log memory area
                                LDR R5, =INDEX_ERROR_LOG ; Get the address of the Index of error
                                LDR R5, [R5] ; Get the value of the index of error
                                MOVS R7, #12 ; Fill R7 with 12
                                MULS R5, R7, R5 ; Multiply Index and 12 to get address address distance
                                ADDS R5, R5, R4 ; Add distance and start address so, get the current address
                                LDR R7, =__LOG_END ; Get the address of end of the Log memory area
                                CMP R7, R5 ; Check if reached end of the memory of error log
                                BEQ        end_writelog ; If yes call end_writelog
                                ; Else
                                STRH R0, [R5] ; Store index which comes as parameter in the first 16-bit
                                STRB R1, [R5, #2] ; Store error code which comes as parameter in next 8-bit
                                STRB R2, [R5, #3] ; Store operation which comes as parameter in next 8-bit
                                STR  R3, [R5, #4] ; Store data which comes as parameter in next 32-bit
                                BL   GetNow ; Call GetNow and get Current System Tick Timer working time
                                STR  R0, [R5, #8] ; Store it in next 32-bit
                                LDR R5, =INDEX_ERROR_LOG ; Get the address of the Index of error
                                LDR  R6, [R5] ; Get the value of the index of error
                                ADDS R6, R6, #1 ; Increase index by 1
                                STR  R6, [R5] ; Update index data
end_writelog
                                POP  {PC} ; Return
                                ENDFUNC
```

Figure 20: Implementation of WriteErrorLog Function

2.15 GetNow

In the GetNow function, we have to calculate the time that passed until the function GetNow called.

Firstly, we have analyzed the working mechanism of the tick count. We increment tick count variable at the start of the interrupt handler function. Therefore to find the time that is passed until interrupt, we have to look for the period of system tick interrupt and tick count. Our system tick interrupt period is 946 microsecond, if we multiply this with tick count, we can find the elapsed time until the interrupt that we call the GetNow function.

$$Time_{Interrupt} = T_{sysTick} * TICK\_COUNT$$

Equation 6: Time Until Interrupt Call Formula

We also have to calculate the time passed from the beginning of the interrupt until the GetNow function executes. For this, we have to look for our reload value's and current value's relation. When we look at these variables, current value starts from reload value and decreases from that until it reaches to 0 and when the current value is 0, our tick count increases. Then we calculate the time that passes between beginning of the interrupt and GetNow execution based on this information. If we subtract current value from reload time + 1, we can find the number of clock cycles that we passed (reload + 1 - current). If we want to change this information to time, we can divide our calculated value by our clock frequency and find the time between beginning of interrupt and GetNow execution.

$$Time_{Interrupt-GetNow} = \frac{(Reload + 1 - Current)}{F\_CPU}$$

Equation 7: Time Between GetNow Function Call and Interrupt

Finally, to find the total time that passes between the system tick initialization and execution of GetNow function, we can add these two value and find our passed time.

$$Time = Time_{Interrupt} + Time_{Interrupt-GetNow}$$
$$Time = (T_{sysTick} * TICK\_COUNT) + \frac{(Reload + 1 - Current)}{F\_CPU}$$

Equation 8: GetNow Formula

In the implementation part, we get our tick count and our current value from their addresses. To calculate our equation, we set our reload value because our CPU frequency was 64MHz we can easily divide our values with 6 times right shifting. Therefore using these information, we have implemented our GetNow function easily. We can see our implementation at Figure 21

```
GetNow                              FUNCTION
;//-------- <<< USER CODE BEGIN Get Now >>> ----------------------

                                ; getnow function is get_now = interrupt_period * tick_count + (reload+1-current)/F_cpu
                                LDR  R0, =TICK_COUNT            ; takes address of TICK_COUNT variable to R0 register
                                LDR  R0, [R0]                            ; reads value of the tick count from address of the TICK_COUNT
                                LDR  R1, =60544                            ; set R1 with (reload+1)
                                LDR  R3, =0xE000E018            ; load address of the current value to R3 register
                                LDR  R3, [R3]                            ; load current value from current_value address to R3 registere
                                SUBS R4, R1, R3                            ; subtract (current_value) from (reload + 1) and assign to R4 register
                                LSRS R4, #6                                    ; divide (R4 register) (reload+1-current) by 64
                                LDR  R2, =946                            ; set R2 with the Period Of the System Tick Timer Interrupt
                                MULS R2, R0, R2                            ; Multiply interrupt_period with tick_count
                                ADDS R0, R2, R4                            ; and calculate time interrupt_period * tick_count + (reload+1-current)/F_cpu
                                BX          LR; branches with link register
;//-------- <<< USER CODE END Get Now >>> ------------------------
                                ENDFUNC
```

Figure 21: Implementation of GetNow Function

3 RESULTS

In this part of the project, we are trying out the given data by explaining in details and prepared general test case for special cases like error codes in Table 4, defined in problem description.

3.1 The Initial Data

First of all, the ARM code of the given test data shown in Figure 22 is refers to the data and its flag like insert, delete and LinkedList2Arr operations. Also, the NUMBER_OF_AT is initialized with 20. The whole operations are done with respect to NUMBER_OF_AT constant value so the other contants like AT_SIZE, DATA_AREA_SIZE, ARRAY_SIZE and LOG_ARRAY_SIZE are created memory locations for the data areas like AT_MEM, DATA_MEM, ARRAY_MEM, and LOG_MEM, respectively. Therefore, the values of the memory locations for the given data is special. If the NUMBER_OF_AT constant is changed at the beginning of the program, the memory range changes.

```
                                AREA     IN_DATA_AREA, DATA, READONLY
IN_DATA                 DCD        0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x06, 0x99, 0x07, 0x08,
↪  0x08, 0x09, 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0x20, 0x21, 0x22, 0x23, 0x24,
↪  0x25, 0x26, 0x27, 0x28, 0x29, 0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x35, 0x36, 0x37, 0x38, 0x39, 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, 0x00, 0x01, 0x01, 0x00, 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, 0x00, 0x01, 0x01, 0x01, 0x01, 0x02
END_IN_DATA_FLAG
```

Figure 22: In ARM code, input dataset is defined in Data Area as readonly, initially.

As can be illustrated in Table 5, the given data is run step by step with operation, data, data flag and status. The Insert, Remove and LinkedList2Arr flags shown in Table 3 are used for operations and the error codes are used in Table 4 to check the operations' status. The detailed information for the steps are shown **Related Figure** column of the Table for each operation. Then, the result is shown in Table 6 for LinkedList to Array statement, which runs at the end of the program.

Table 3: Operation Flags

Flag Code	Operation
0	Remove the data from the linked list.
1	Insert the data to the linked list.
2	Transform the linked list to the array (The data will be ignored).

Table 4: Error Codes

Error Code	Operation	Explanation
0	All	Operations No error.
1	Insertion	There is no allocable area. (The linked list is full.)
2	Insertion	Same data is in the array. (Duplicated insertion operation.)
3	Deletion	The linked list is empty.
4	Deletion	The element is not found.
5	Linked List to Array	The linked list could not be transformed. (The linked list is empty.)
6	-	Operation is not found.

Table 5: The simulation of the given data and its flag corresponding the operation

Related Figure	#	Operation	Data Flag	Data(hex)	Status	Error Code
Figure 28	1	Insert	0x01	0x0010	No Error, 0x0010 is inserted	0
Figure 29	2	Insert	0x01	0x0020	No Error, 0x0020 is inserted	0
Figure 30	3	Insert	0x01	0x0015	No Error, 0x0015 is inserted	0
Figure 31	4	Insert	0x01	0x0065	No Error, 0x0065 is inserted	0
Figure 32	5	Insert	0x01	0x0025	No Error, 0x0025 is inserted	0
Figure 33	6	Insert	0x01	0x0001	No Error, 0x0001 is inserted	0
Figure 34	7	Remove	0x00	0x0001	No Error, 0x0001 is deleted	0
Figure 35	8	Remove	0x00	0x0012	Element is not found.	4
Figure 36	9	Remove	0x00	0x0065	No Error, 0x0065 is deleted	0
Figure 37	10	Remove	0x00	0x0025	No Error, 0x0025 is deleted	0
Figure 38	11	Insert	0x01	0x0085	No Error, 0x0085 is inserted	0
Figure 39	12	Insert	0x01	0x0046	No Error, 0x0046 is inserted	0
Figure 40	13	Remove	0x00	0x0010	No Error, 0x0010 is deleted	0
Figure 41	14	Linkedlist to Array	0x02	0x0000	No Error, Linkedlist shown as array	0

Table 6: The result of the simulation with respect to LinkedList2Arr in step 14 as shown in Figure 22

Index	Result
1	0x15
2	0x20
3	0x46
4	0x85

The main program is started but the breakpoint are kept due to observe the first initializations by the program definitions before branching operations like Clear_Alloc.

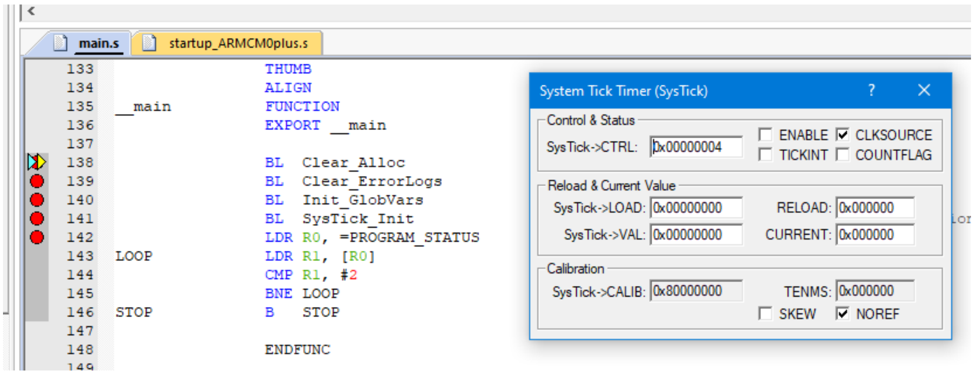

Figure 23: The System Tick Timer(SysTick) is not loaded and the main program is ready to branch **Clear_Alloc**

Based on the AT_SIZE constant with regarding to NUMBER_OF_AT, in Figure 24, the Clear_Alloc function is cleared memory location from 0x20000004 to 0x20000050, inclusively.

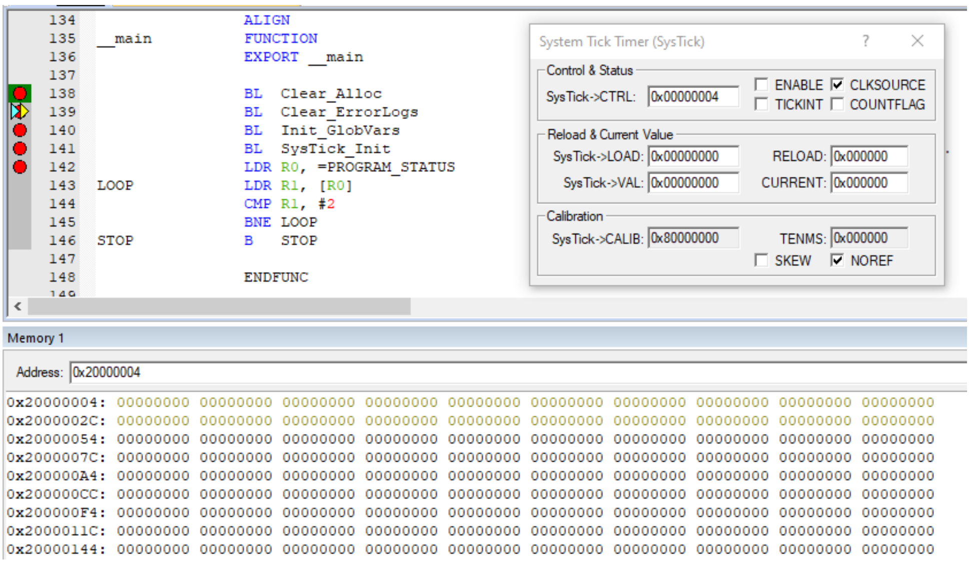

Figure 24: After returning the **Clear_Alloc** function, the allocation table is cleared shown in Memory 1 with 0x20000004 location.

After operating the Clear_ErrorLogs function, in Figure 25, the memory locations from 0x20000A54 to 0x20002850, inclusively, memory location is cleared due not to flow garbage values.

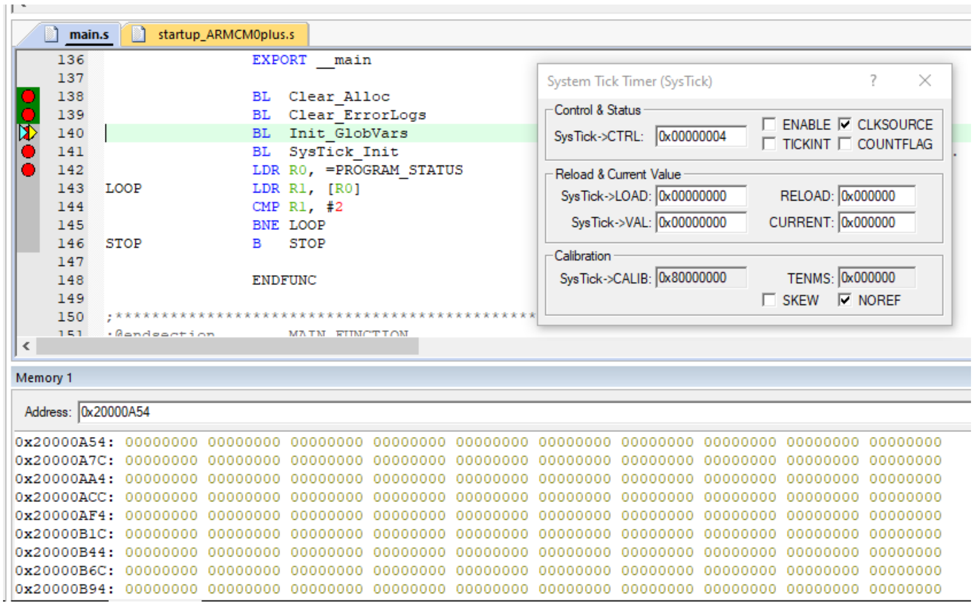

Figure 25: The error logs are cleared shown in Memory 1, starting address from 0x20000A54

As can be seen in Figure 26, memory locations 0x20003C54 for TICK_COUNT, 0x20003C58 for FIRST_ELEMENT, 0x20003C5C for INDEX_INPUT_DS, 0x20003C60 for INDEX_ERROR_LOG, 0x20003C64 for PROGRAM_STATUS is set to zero by using **Init_GlobVars** function.

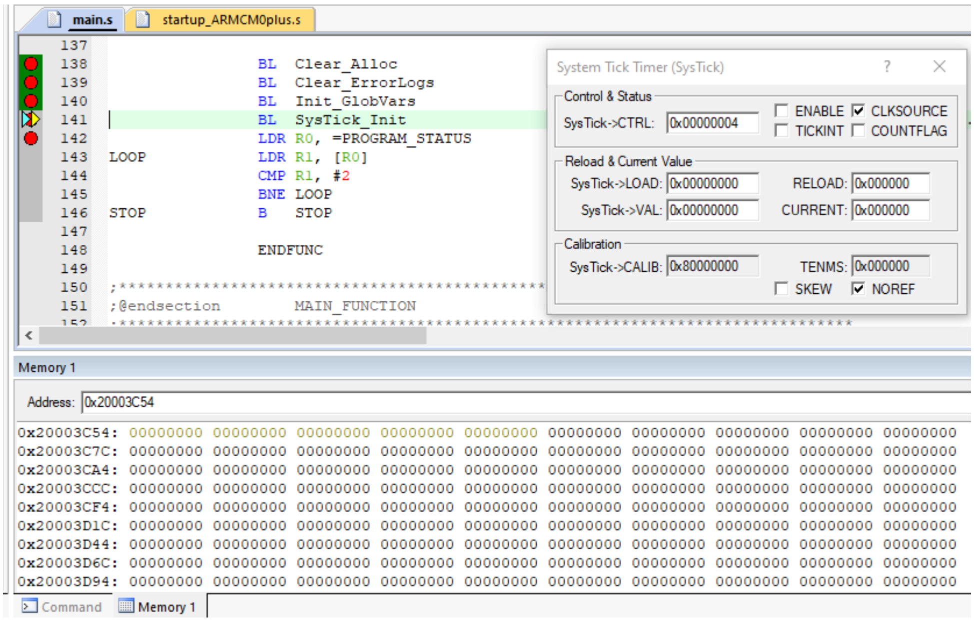

Figure 26: The global initializations are set to zero for TICK_COUNT, FIRST_ELEMENT, INDEX_INPUT_DS, INDEX_ERROR_LOG, PROGRAM_STATUS with memory locations like 0x20003C54, 0x20003C58, 0x20003C5C, 0x20003C60, 0x20003C64, respectively

Now, the **Clear_Alloc**, **Clear_ErrorLogs** and **Init_GlobVars** functions did not start the System Tick Interrupt Service Routine. As can be seen in Figure 27, therefore, the SysTick_Init function is called to start System Tick Timer(SysTick) with regard to the given 946 microseconds, which is the Period of System Tick Interrupt. Also, the PROGRAM_STATUS is updated as 1 to indicate the timer started.

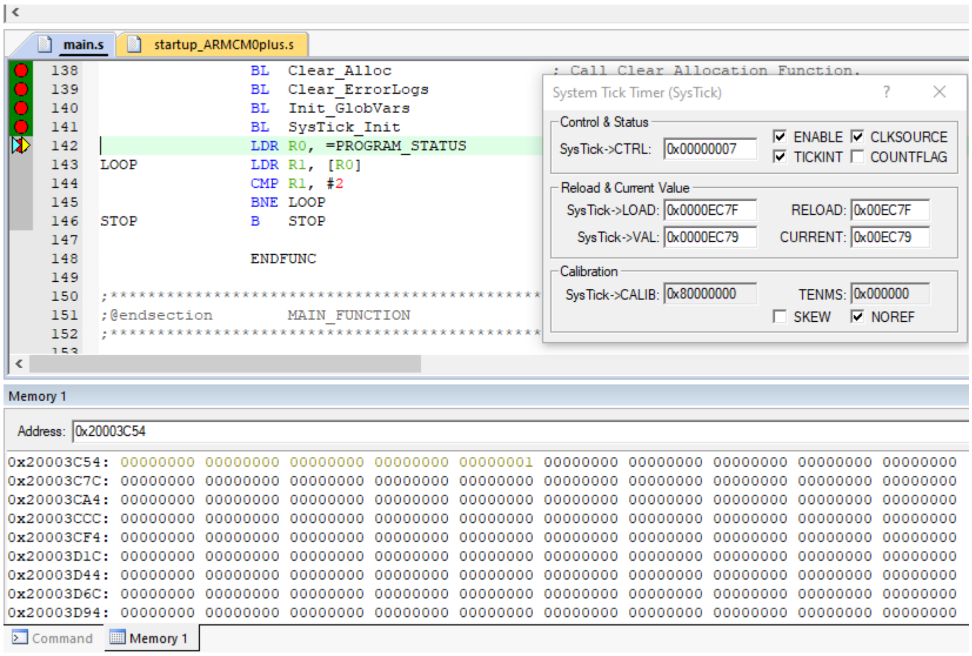

Figure 27: The SysTick is initialized ENABLE, TICKINT and CLKSOURCE flags and Reload Value with respect to the given period value as 946 microseconds

As can be seen in Figure 28-42, the 'main.s' file is run in debug mode and the breakpoints are dedicated for each operation, yellow color refers to next run is selected function. Also, the Memory1 is related with Allocation Table from 0x20000004 to 0x20000050 by tag in the upside as 'Allocation Table until 0x20000050', and the memory is read as hex like 0x0000001E. On the other hand, the memory range from 0x20000054 to 0x20000A50 shows the hex formatted result that run by LinkedList2Arr function. The Memory2 is related with Error Logs range from 0x20000A54 to 0x..., and the first two byte like 07 00 is read as 0x0007 for index value, next one byte like 0x04 refers to ErrorCode and the next one byte like 0x00 refers to Operation Flag. Also, the next 32 bits(4 byte) like 12 00 00 00, reading as 0x00000012 for observing to the current data. The last 32 bits(4 byte) like 92 1D 00 00, reading as 0x00001D92 for the current working time in microseconds. Moreover, the Memory3 on right side refers to the Linkedlist memory beginning from 0x020002854 to 0x..., and the first 4 byte of the given data for data and the second 4 byte of the given data for the next address. The last memory called as Memory 4 shows 0x20003C54 for TICK_COUNT, 0x20003C58 for FIRST_ELEMENT, 0x20003C5C for INDEX_INPUT_DS, 0x20003C60 for INDEX_ERROR_LOG and 0x20003C64 for PROGRAM_STATUS. Also, the System Tick Timer is shown in each iteration.

In Figure 28, the 0x10 data is inserted shown in LinkedList Memory. Also, the memory is allocated, there is no error, PROGRAM_STATUS is preserved the status, TICK_INT and INDEX_INPUT_DS increased by one and the head pointer as FIRST_ELEMENT is changed from 0(Null Pointer) to the address 0x20002854.

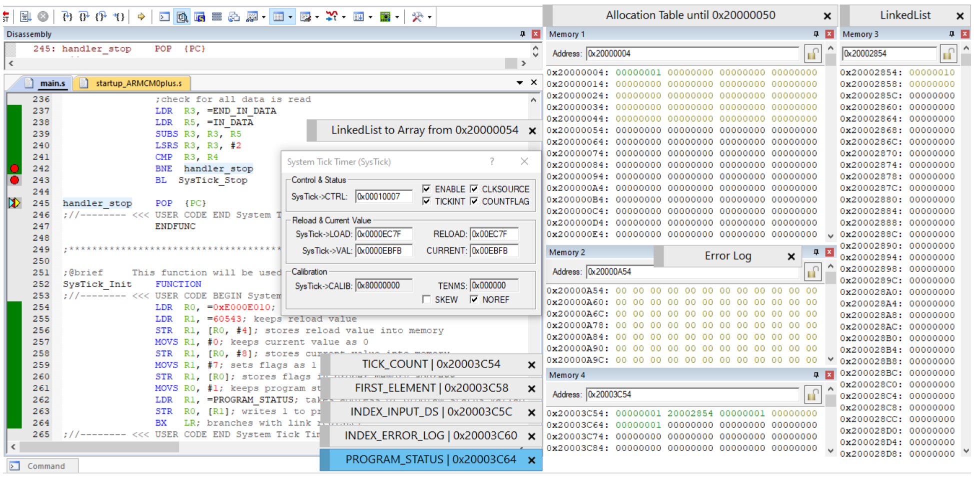

Figure 28: The memory allocated, and then the 0x10 is inserted by updating the FIRST_ELEMENT, TICK_INT and INDEX_INPUT_DS.

As can be seen in Figure 29, the 0x20 is inserted after allocating the data area and the next pointer of the first node is connected to the 0x20's address. Also, There is no error, PROGRAM_STATUS and FIRST_ELEMENT have preserved their status, TICK_INT and INDEX_INPUT_DS increased by one.

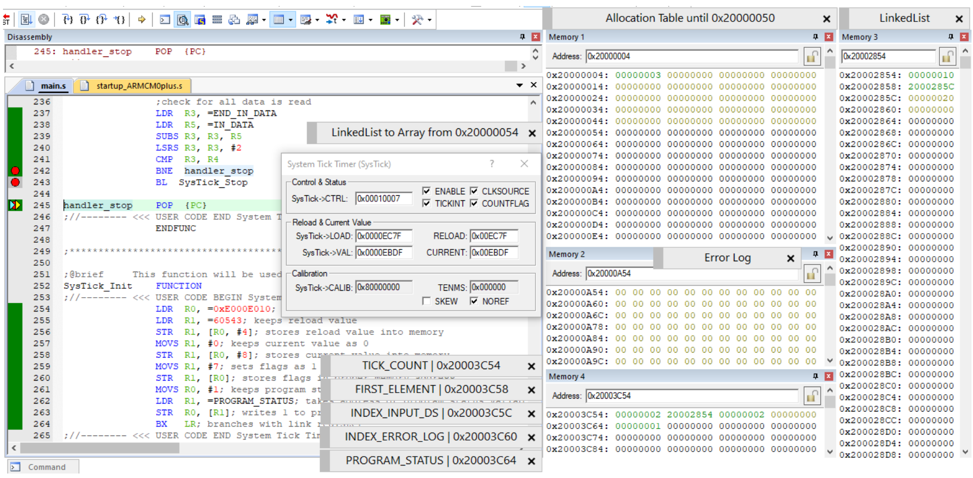

Figure 29: The 0x20 is inserted and the next pointer of the first data is connected to the 0x20's address

In Figure 30, the memory allocated for the insert operation and the 0x15 is inserted between 0x10 and 0x20 to make sortable. In the insert operation, the next node of the first node is connected to current node's address and the current node's next address shows the address of last node's having data 0x20. Also, There is no error, PROGRAM_STATUS and FIRST_ELEMENT have preserved their status, TICK_INT and INDEX_INPUT_DS increased by one.

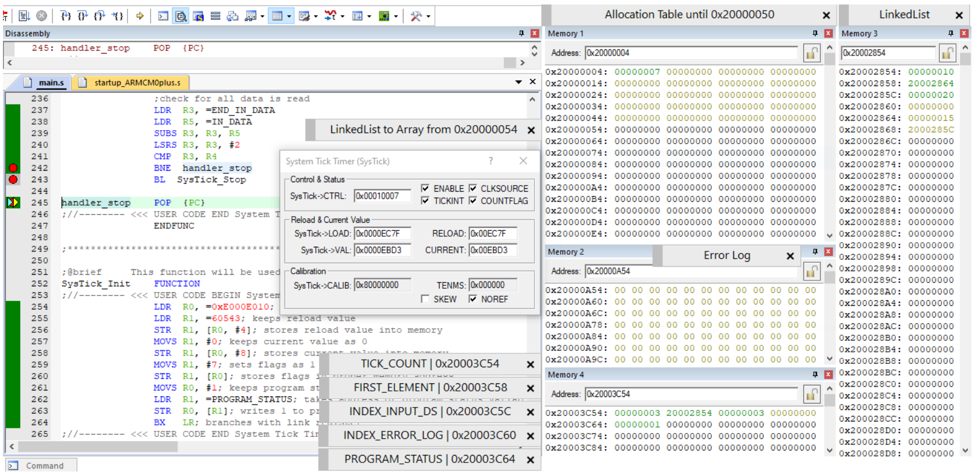

Figure 30: The 0x15 is inserted between 0x10 and 0x20.

As can be seen in Figure 31, the 0x65 data value is inserted at the end of the linked list after allocating the memory. Therefore, the end data was 0x20 to update as 0x65 and the next address of the 0x20 is connected to 0x65's address. Notice that the next address of the 0x65's address is NULL Pointer(0x0). Also, There is no error, PROGRAM_STATUS and FIRST_ELEMENT have preserved their status, TICK_INT and INDEX_INPUT_DS increased by one.

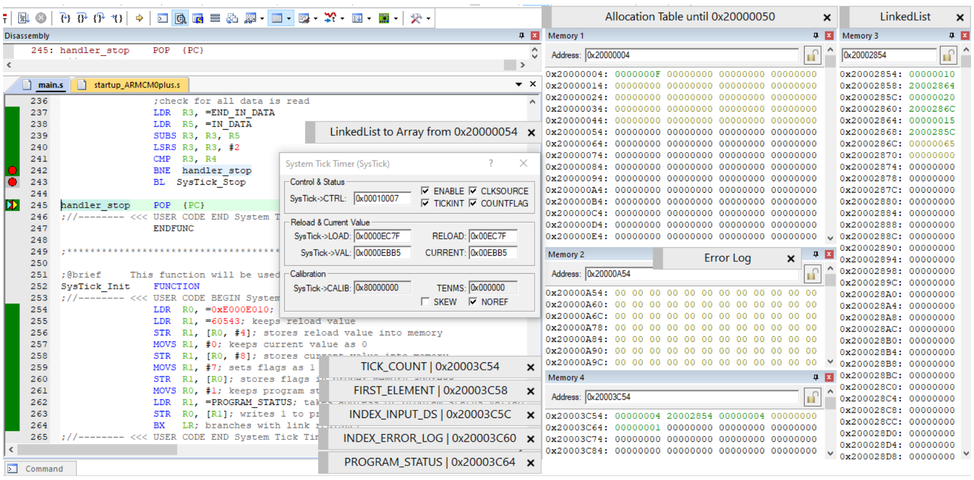

Figure 31: The data 0x65 is inserted at the end of the linked list

In Figure 32, The 0x25 is inserted between 0x20 and 0x65 after allocation memory space for 0x25. The operation is the next address of the 0x20 is set to the 0x25's address and the next address of the 0x25 shows the 0x65's address. Also, There is no error, PROGRAM_STATUS and FIRST_ELEMENT have preserved their status, TICK_INT and INDEX_INPUT_DS increased by one.

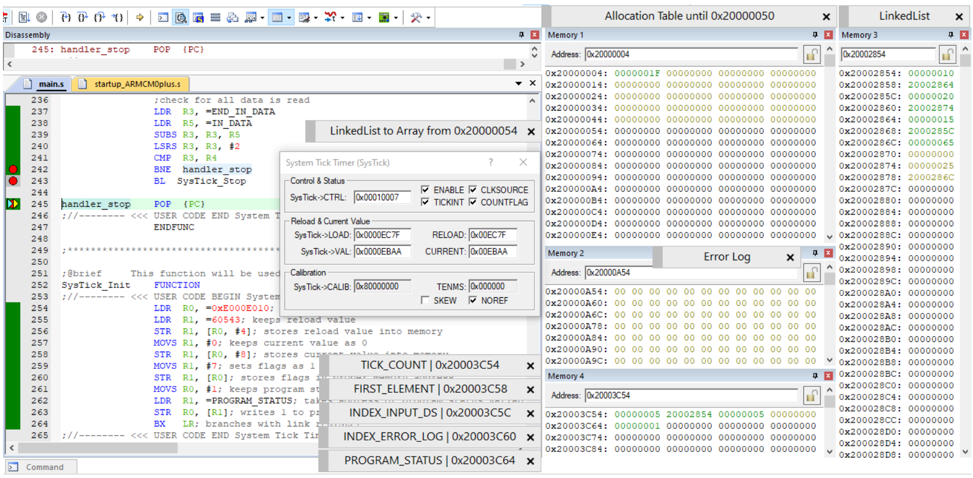

Figure 32: The 0x25 is inserted between 0x20 and 0x65

In Figure 33, the 0x01 is inserted as a first element of the linked list after allocating the space. The operation is changing the first node's address as 0x01's address so the FIRST_ELEMENT is updated. Also, the next address of the 0x01's address shows the 0x10's address. There is no error, PROGRAM_STATUS preserved its status and TICK_INT and INDEX_INPUT_DS increased by one.

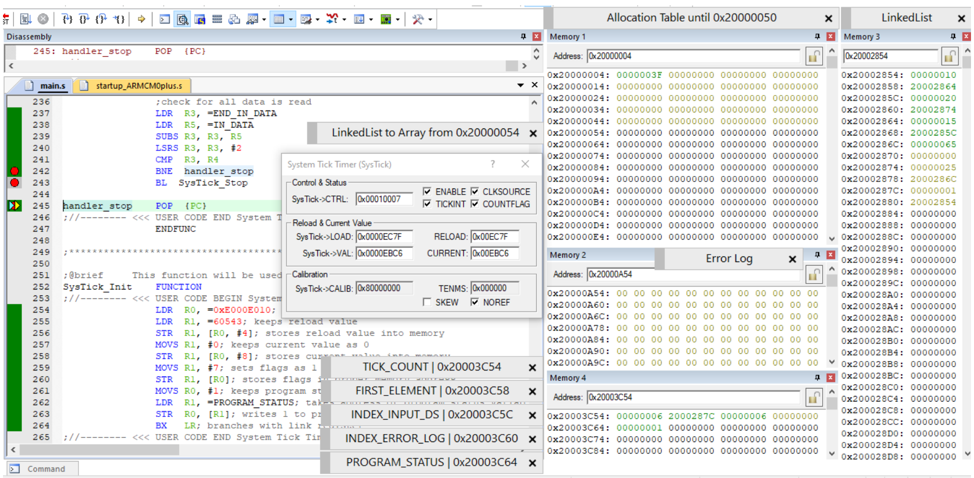

Figure 33: The head pointer is changed to 0x01 by inserting the new smallest element.

As can be illustrated in Figure 34, the remove operation for the 0x01 is done with free the space on the allocation table. Also, the head pointer is changed back to the 0x10's address. Therefore, the FIRST_ELEMENT is updated. There is no error, PROGRAM_STATUS preserved its status and TICK_INT and INDEX_INPUT_DS increased by one.

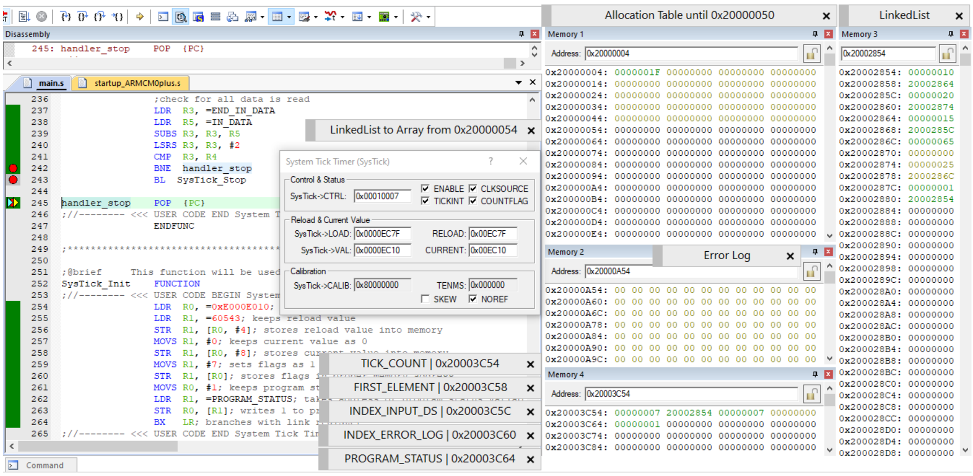

Figure 34: The 0x01 is removed from the linked list

The deletion of the 0x12 element, in Figure 35, is occurred as an error like 'The element is not found'. Therefore, error logs memory writes the status of the TICK_COUNT, Error Code as 4, Operation Flag as 0x00 for deletion, the Current Data and the current timestamp in microseconds. Also, the **INDEX_ERROR_LOG**, TICK_INT and INDEX_INPUT_DS is increased by one, and the PROGRAM_STATUS preserved its status.

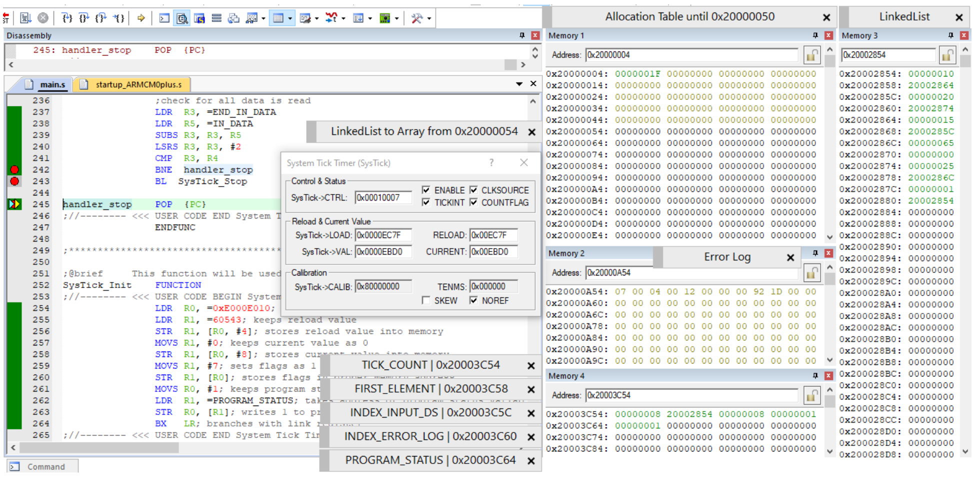

Figure 35: The 0x12 is not found on the linked list for delete operation so the error occurs with error code 4, meaning 'Element is not Found'

As can be seen in Figure 36, The 0x65 is deleted from the linked list and the allocated area for 0x65 is deallocated. Therefore, the 0x65 was the end value and the second last value's address is set to the null pointer(0x0). Also, the error does not occur, PROGRAM_STATUS and FIRST_ELEMENT have preserved their status, TICK_INT and INDEX_INPUT_DS increased by one.

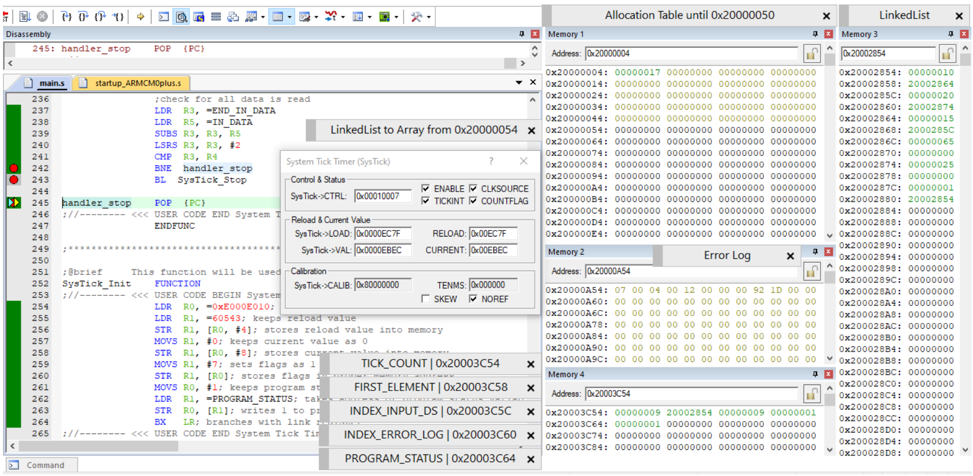

Figure 36: The 0x65 is deleted without error.

The 0x25 set as last address in the previous step, in Figure 37, is deleted from the linked list without getting error and the memory space is free with its address from the corresponding bit in the allocation table. Therefore, the next address of previous node of the 0x25 is set as null pointer(0x0). Also, the error does not occur, PROGRAM_STATUS and FIRST_ELEMENT have preserved their status, TICK_INT and INDEX_INPUT_DS increased by one.

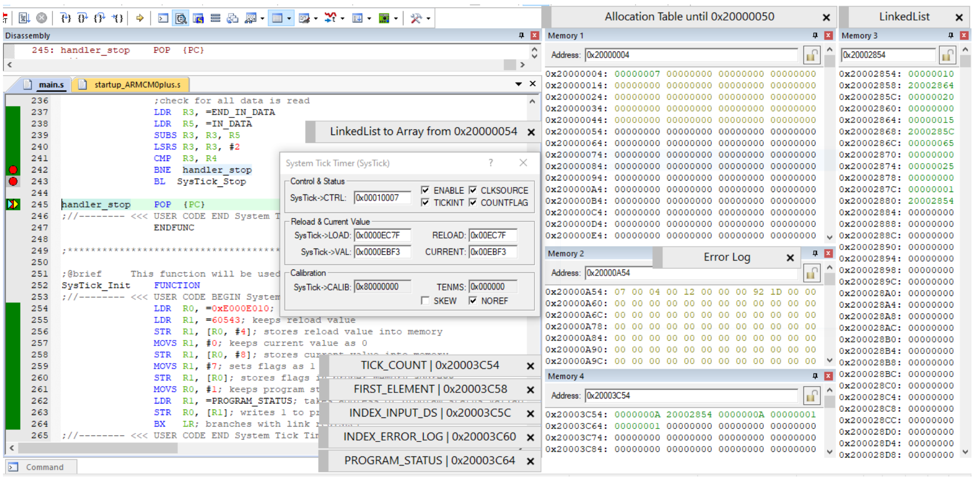

Figure 37: The 0x25 is deleted, accurately.

As can be seen in Figure 38, the 0x85 is correctly inserted at the end of the linked list after allocating an memory space for 0x85. Also, the last element was 0x25 and its next address is changed with the address of the recently created 0x85's address. Moreover, the next address is set to the null pointer(0x00). The error does not appear, PROGRAM_STATUS and FIRST_ELEMENT have preserved their status, TICK_INT and INDEX_INPUT_DS increased by one.

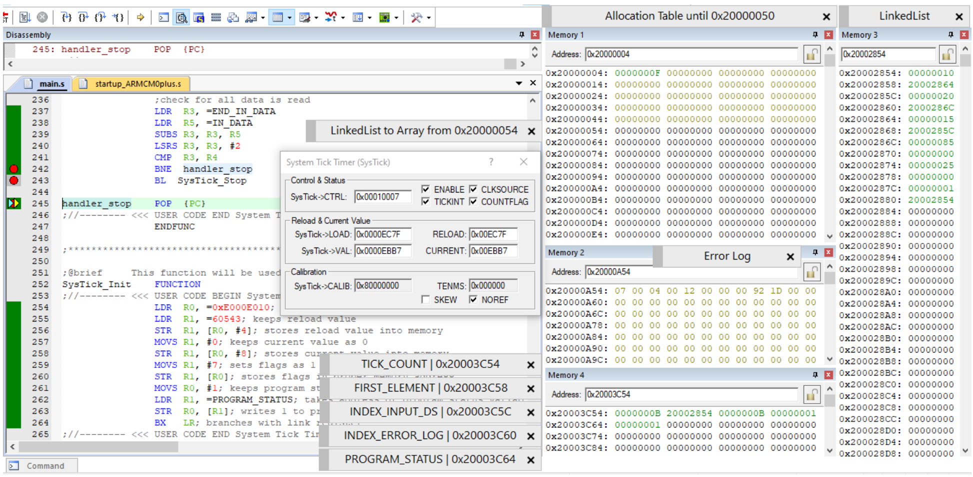

Figure 38: The 0x85 is inserted at the end of the linked list without error.

In Figure 39, the 0x46 is accurately inserted between 0x20 and the 0x86 after preserving the memory space from allocation table. Therefore, the 0x20's next address is shown with the address of the 0x46, and the next address of the 0x46 is put the address of the 0x85. Also, there is no error, PROGRAM_STATUS and FIRST_ELEMENT have preserved their status, TICK_INT and INDEX_INPUT_DS increased by one.

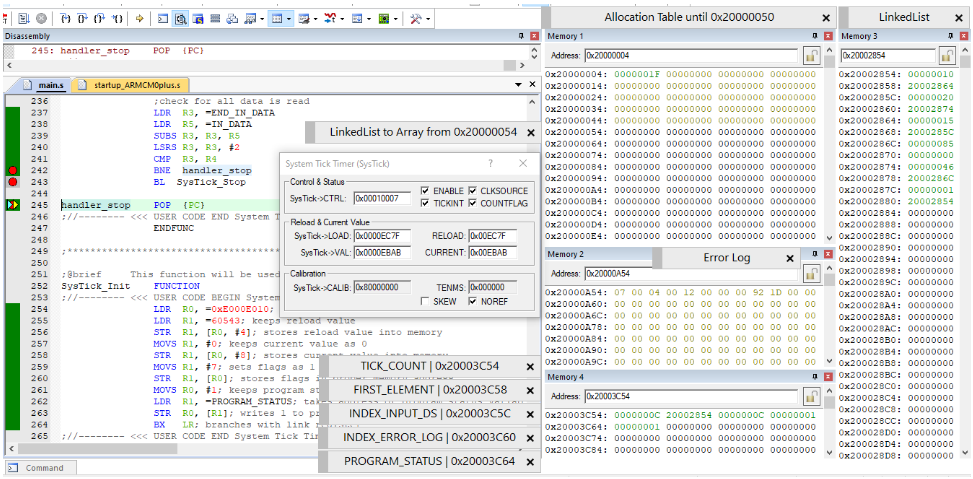

Figure 39: The 0x46 is correctly inserted between 0x20 and 0x85 in the linked list to make sortable

In Figure 40, the 0x10 is correctly deleted, and the memory space related to the 0x10 is deallocated. Therefore, the FIRST_ELEMENT's address is changed to the next address of the 0x10 node. Also, PROGRAM_STATUS is kept its status, TICK_INT and INDEX_INPUT_DS increased by one.

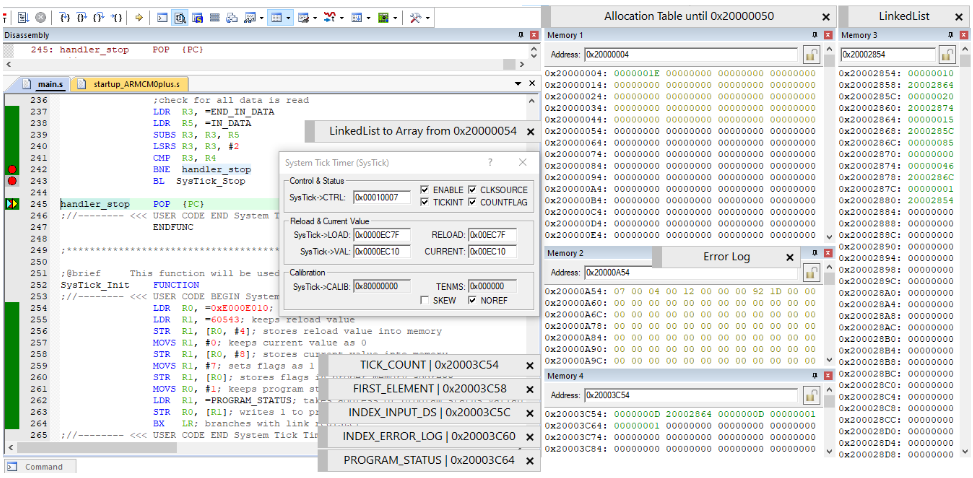

Figure 40: The 0x10 is deleted without getting an error.

The end operation is to observe the data on the linked list. As can be seen in Figure 41, the LinkedList2Arr function is called to clean the memory and write the data starting from 0x20000054 with hex formatted. Therefore, the error is not occurred, PROGRAM_STATUS is kept its status, TICK_INT and INDEX_INPUT_DS increased by one.

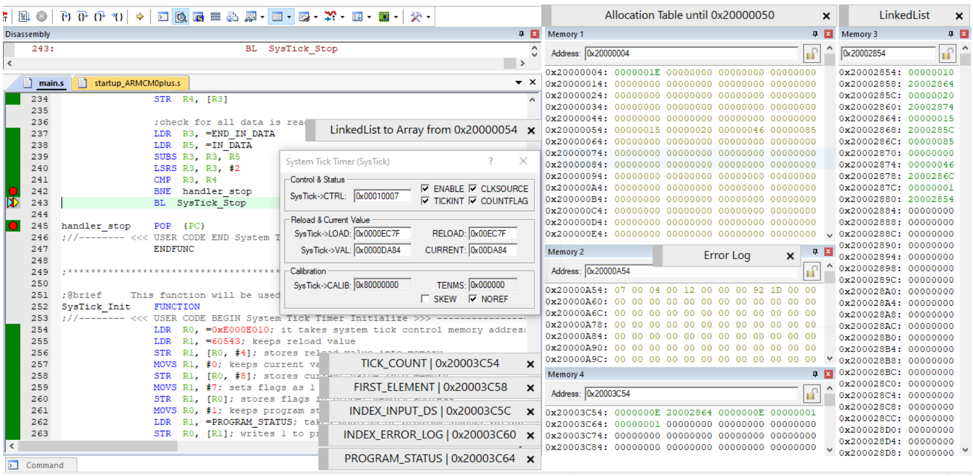

Figure 41: The linked list is cleared and printed as an array on the Memory1 from 0x20000054

At the end of the branch of the SysTick_Handler, in Figure 42, the SysTick_Stop is called. Therefore, the timer is stopped and the PROGRAM_STATUS is updated as 2, meaning that 'All data operations finished.'

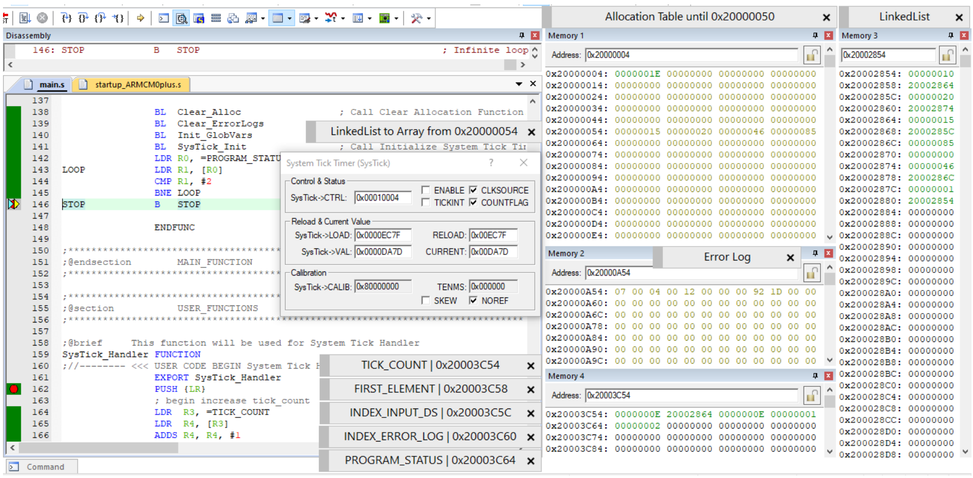

Figure 42: The program runs correctly and the status updated as 2, meaning that 'All data operations finished.'

3.2 The Prepared Data

The ARM codes about the prepared data in Figure 48, and its flag in Figure 49 at the Appendix section. The workflow of the prepared data is listed step by step in Table 7

Table 7: The simulation of the prepared data

#	Operation	Data Flag	Data(hex)	Status	Error Code
1	Remove	0x00	0x0001	The linked list is empty	3
2	LinkedList2Arr	0x02	0x0001	The Linked list could not be transformed.	5
3	No operation	0x25	0x0013	Operation is not found	6
4	Insertion	0x01	0x0003	No Error.	0
5	Remove	0x00	0x0002	Element is not found.	4
6	Insertion	0x01	0x0002	No Error.	0
7	Remove	0x00	0x0002	No Error.	0
8	Insertion	0x01	0x0003	Same data is in the array	2
9	No operation	0x03	0x0025	Operation is not found	6
10	Remove	0x00	0x0003	No Error.	0
11	Remove	0x00	0x0001	The linked list is empty	3
12	LinkedList2Arr	0x02	0x0000	The Linked list could not be transformed.	5
13	No operation	0x49	0x0036	Operation is not found	6
14	Insertion	0x01	0xFFFF	No Error.	0
15	Insertion	0x01	0xFFFE	No Error.	0
16	Insertion	0x01	0x0FFF	No Error.	0
17	Insertion	0x01	0x0FFE	No Error.	0
18	Insertion	0x01	0x00FF	No Error.	0
19	Insertion	0x01	0x00FE	No Error.	0
20	Insertion	0x01	0x000F	No Error.	0
21	Insertion	0x01	0x000E	No Error.	0
22-35	Insertion	0x01	0x0000 to 0x000D	No Error.	0
36	Insertion	0x01	0x000E	Same data is in the array	2
37	Insertion	0x01	0x000F	Same data is in the array	2
38-275	Insertion	0x01	0x0010 to 0x00FD	No Error.	0
276	Insertion	0x01	0x00FE	Same data is in the array	2
277	Insertion	0x01	0x00FF	Same data is in the array	2
278-657	Insertion	0x01	0x0100 to 0x027B	No Error.	0
658-722	Insertion	0x01	0x027C to 0x02BB	There is no allocable area.	1
723	LinkedList2Arr	0x02	0x0000	No Error.	0
724	Insertion	0x01	0xFFFF	Same data is in the array	2
725	LinkedList2Arr	0x02	0x0000	No Error.	0
726	Remove	0x00	0x0258	No Error.	0
727	Remove	0x00	0x01F4	No Error.	0
728	Remove	0x00	0x0190	No Error.	0
729	Remove	0x00	0x012C	No Error.	0
730	Remove	0x00	0x022B	No Error.	0
731	Remove	0x00	0x00C8	No Error.	0
732	Remove	0x00	0x0064	No Error.	0
733	Remove	0x00	0x0032	No Error.	0
734	Remove	0x00	0x1313	Element is not found.	4
735	Remove	0x00	0x027B	No Error.	0
736	LinkedList2Arr	0x02	0x0000	No Error.	0
737	Insertion	0x01	0xEFEF	No Error.	0
738	Insertion	0x01	0xABCD	No Error.	0
739	Insertion	0x01	0xBCDE	No Error.	0
740	Insertion	0x01	0xCDEF	No Error.	0
741	Insertion	0x01	0x4747	No Error.	0
742	Insertion	0x01	0x1111	No Error.	0
743	Insertion	0x01	0x1233	No Error.	0
744	Insertion	0x01	0x1233	Same data is in the array	2
745	Insertion	0x01	0x4747	Same data is in the array	2
746	LinkedList2Arr	0x02	0x0000	No Error.	0

As can be seen in Figure 43, the memory location between 0x20000004 and 0x20000050, inclusively is used to keep track of the allocation table. Then, the memory range between 0x20000054 and 0x20000A50, inclusively is used to observe the data filled by using the LinkedList2Arr function.

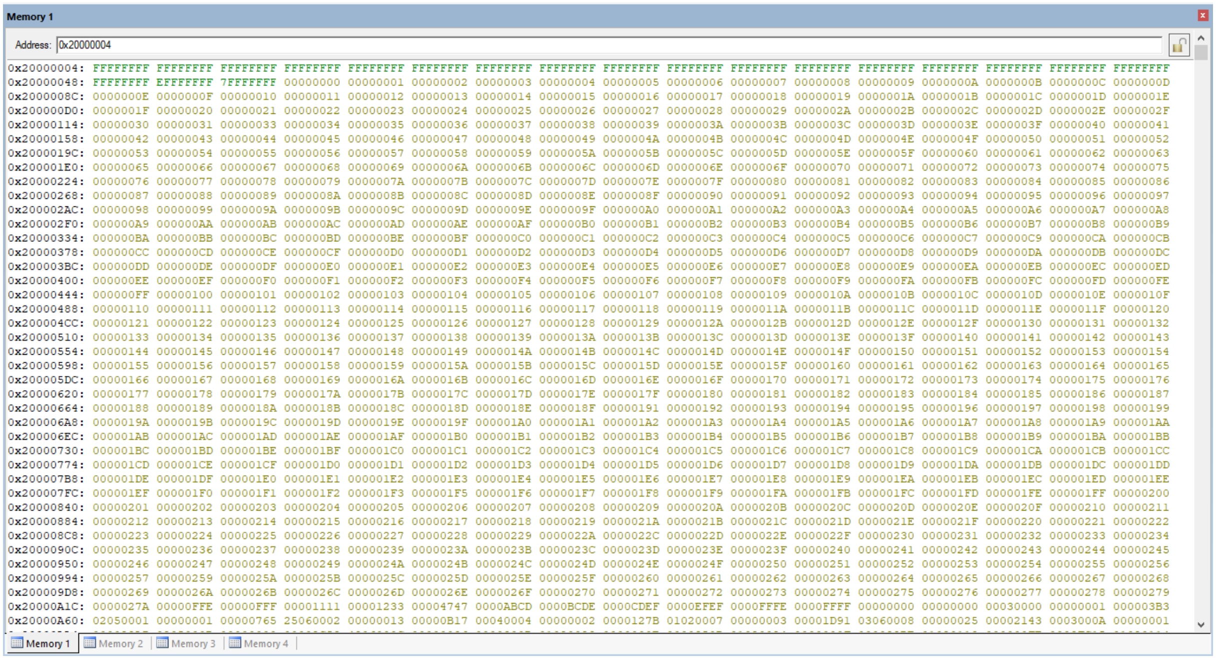

Figure 43: The allocation table and the data is shown for the result of the prepared workflow

In Figure 44, the Error Log Memory is shown in memory range between 0x20000A54 and 0x20002850, inclusively.

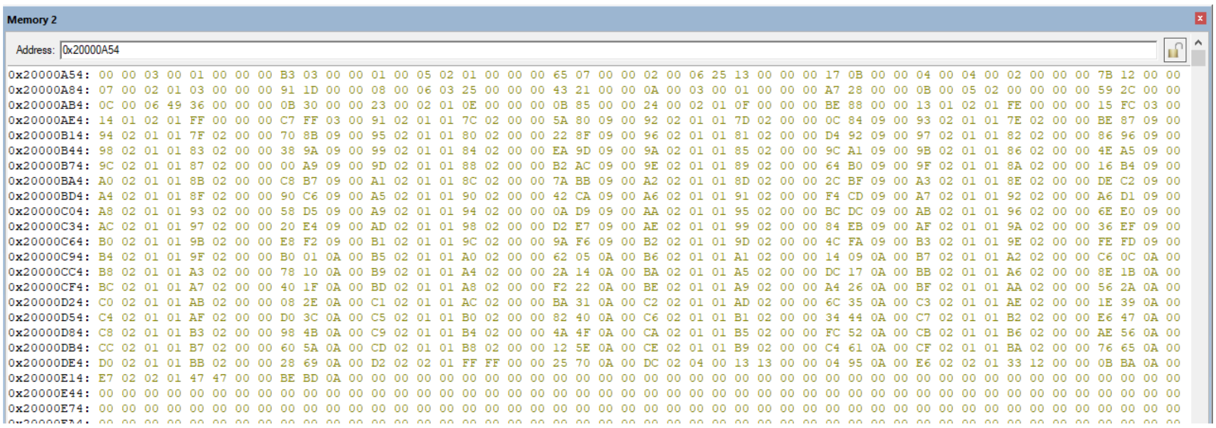

Figure 44: The Error Log Memory is shown for the end of the program

In Figure 45, 46, the Linked List Memory is shown at the end of the program.

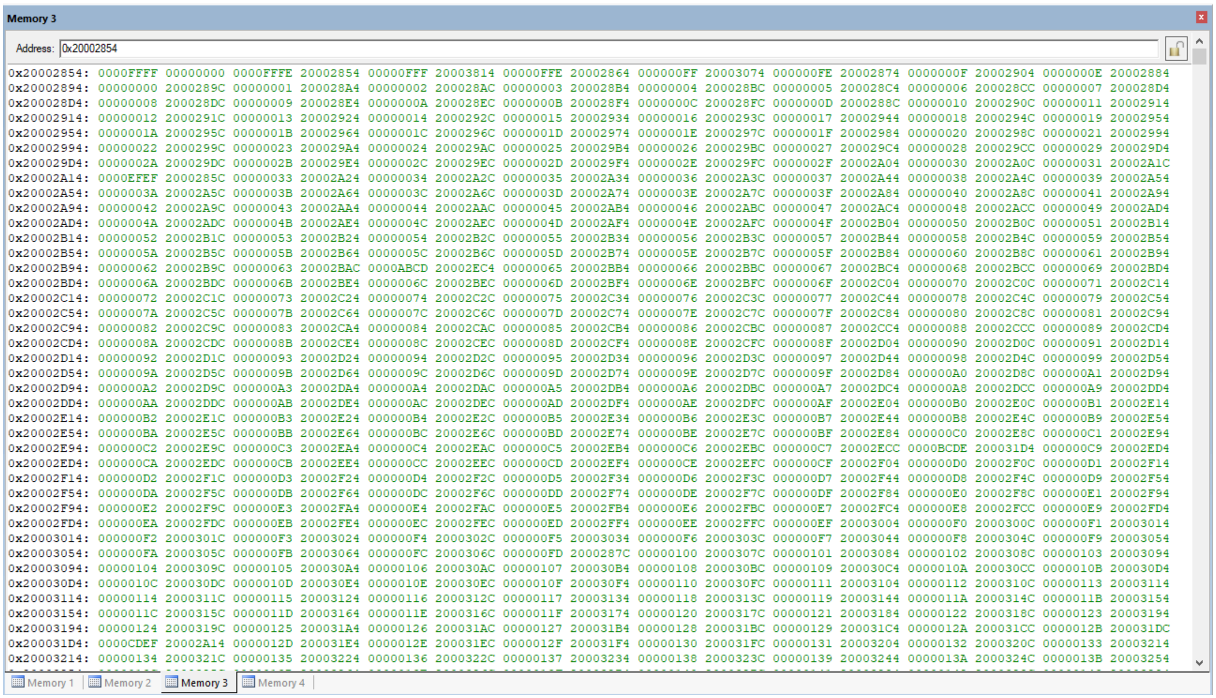

Figure 45: First section about Linked List Memory

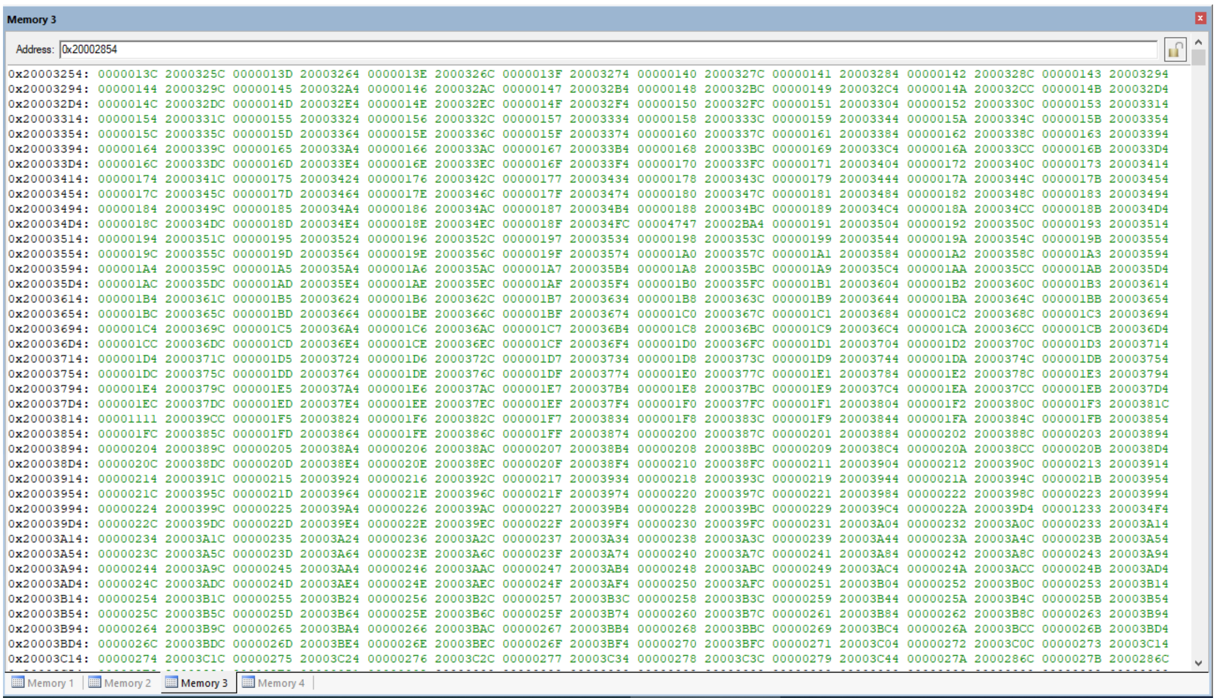

Figure 46: Second section about Linked List Memory

In Figure 47, the memory locations like 0x20003C54 for TICK_COUNT, 0x20003C58 for FIRST_ELEMENT, 0x20003C5C for INDEX_INPUT_DS, 0x20003C60 for INDEX_ERROR_LOG, 0x20003C64 for PROGRAM_STATUS is shown at the end of the program.

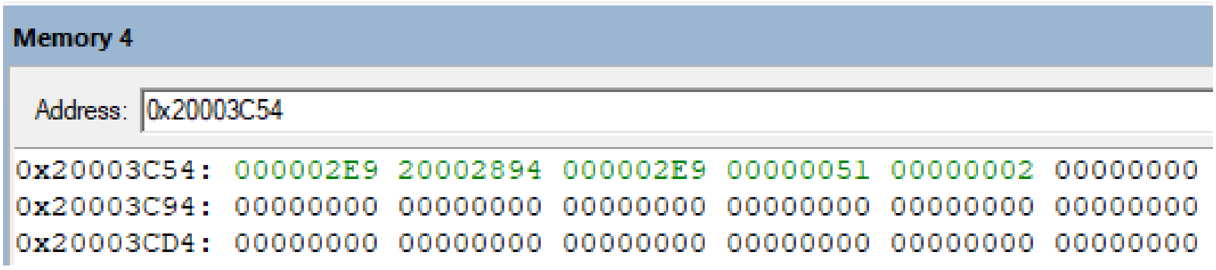

Figure 47: TICK_COUNT, FIRST_ELEMENT, INDEX_INPUT_DS , INDEX_ERROR_LOG and PROGRAM_STATUS is shown

4 DISCUSSION

4.1 SysTick_Handler

This function stands for evaluating input and leading the program according to the input. So, we had to create a decision mechanism and we decided to build this structure with if statements. However, this structure was strait for us because there were too many cases such as inserting, removing, list to an array, and errors. To control all of them we composed a structure that consists of many "if" and "else if" statements. These statements are formed with "CMP" instructions.

This function was a very instructive practice of decision mechanisms. The most significant acquisition of this implementation was the training of complex judgment statements.

4.2 SysTick_Init & SysTick_Stop

In these functions, we were expected to manipulate system tick interrupt of the our microprocessor. We thought that most critical point of the project is providing correct interrupt period to the circuit.

All of us know the mechanism behind of the System Tick but to be sure, we have listened the recitation that includes system tick interrupt subject again. Thanks to lesson, we have consolidated our knowledge, we have written the interrupt formula for our case and we have found our reload value. After this point, implementation phase of the functions was very simple and straightforward. Especially, implementing SysTick_Stop function only needs few changes on SysTick_Init.

In this part, the most difficult point was testing the correction of the interrupt period that is given by the system but we have handled this problem using GetNow function and we are happy to see that it works correctly. We thought that it was a crucial point of the microprocessor systems and we have learned it well.

4.3 Clear_Alloc & Clear_ErrorLogs

In this function there is not much to say. Because both Allocation Table and Error Logs are arrays, we just had to iterate each data space and clear them. Therefore, we easily come up with some loop mechanisms for each of these functions.

4.4 Init_GlobVars

Init_GlobVars function is responsible for initializing global variables. Thus, implementation of this function is not challenging for us when it is compared with other functions. We loaded addresses of every global variable to a register, then store **#0** to these addresses one by one.

4.5 Malloc

In this function, we were expected to write a malloc function that is similar to its use in the C programming language. Actually, this function consist of two separate parts that are searching&allocation and returning node address. The crucial point of implementing this function was understanding the structure and logic of the allocation table. At the point of memory optimization, using 1 bit to determine allocation condition of the node is most proper way but in implementation phase, it needs more effort that is required to make the program work. As an computer engineer, our priority is implementing most efficient algorithm. After our analyzes, we have determined that reaching each word for single time and performing mask operations on same register provides most optimal solution for our problem. Then, we have determined the principles of our searching mechanism that checks each bit using mask register. Based on the knowledge we have learned in the lesson, we have controlled each bit of the word using AND operation and while allocating space, we have used OR operation.

In this part of the project, we had the opportunity to use our theoretical knowledge that we learned in the classroom such as mask operations, memory addressing. Also, we have experienced implementation and key point of 'malloc' that is a function we use frequently in other programming languages. Therefore, it was a great opportunity to apply theoretical knowledge into real microprocessor system.

4.6 Free

This function has some complex calculations to find the exact location of the given address in the allocation table. Differences between node size and allocation table size caused this problem. We had to use various shifting like left shift, right shift, and circular shift. These operations are used instead of mode and dividing operations to figure out the mentioned problem. Also, to assign 0 to an exact bit of 32-bit data was not an easy operation. We solved that problem with an "and" operation. Moreover, data consist of full 1's except the least significant bit that is 0, is used in that operation.

4.7 Insert

Main responsibility of this function is inserting a new value into the linked list, but there were various situations we need to control to construct a robust subroutine. To provide that, we handled insertion operation in different ways according to current state of the linked list, correct order of the new data that will be inserted into the linked list. Our algorithm is mainly to search for the correct position for the new coming data. We implemented this idea by using tail-traverser registers like pointers in C programming language to manage insertion. The reason why we have used 2 register as tail and traverser is that traverser is for compare operations between the current node's data and the new coming data to determine the correct location and the tail is for connection operation of new coming node into the linked list. Tail register provides that retain the address of node which will be the previous node of the new coming node, because we do not have any chance to return the previous node in our linked list structure (it is not a doubly linked list). We performed insertion operation based on the order of the new coming node. We treated differently for insertion at first order, intermediate orders and the last order because of the nature of the linked list. We also fulfilled the error handling in this subroutine. We checked whether the given data is already in the linked list or not, because our linked list has set property and we also set specific error code for this error type.

4.8 Remove

Remove function is expected to delete the given data from the linked list. To implement it, our algorithm is based on search for target data in the linked list, destroy the connections of the target from the linked list and deallocate the memory of the deleted node. After implementation of insertion operation, it is easier to implement remove function, because we used the same tail-traverser logic for search operation in this function. After we found the target node, we destroyed the connections of the deleted node and construct a new connection between the previous and next nodes of the deleted node to protect the chain structure of the linked list. After destruction of the node, we deallocate the space of deleted node from the allocation table to use it for further allocations. We also carried out error handling in case of linked list is empty or the given data is not in the linked list and set corresponding error codes for each of these error types.

4.9 LinkedList2Arr

In this function we had to create a linked list to array converter. The main problem in this function is extracting all data from linked list one by one and assign them to data array. To achieve this functionality we have to create a loop that can do both operations. At the same time we have to control if our linked list is empty or not. In the early stage of implementation we just implemented linked list to array converter loop. After some debugging and brainstorm, we created a mechanism that can control if our linked list is empty or not. We implemented this mechanism above from our conversion loop. After creating all of the functionality we had to add error handling mechanisms to our function, therefore we add a control system for empty linked list and when we control it if it is empty we create an error code for empty linked list. If our linked list is not empty and our operations run successfully we created an error code 0 (which means no error).

4.10 WriteErrorLog

WriteErrorLog function is expected to write error logs in to the error log array. To implement it we just gather all the data and store them at the end of the array. The main problem was if we have so much more errors in the program than we can store we have to manage those errors. Because we do not have space for those errors we just block the function before storing more data. If we continued to store more data we would lose other data from the memory because our arrays and linked list are consecutively placed into the memory.

4.11 GetNow

In this function, the main problem was creating the equation of the current time. In the beginning, we did not know how to create a function that gets the current time. We studied the how the system tick timer works and how the tick count increased and each one of us come up with different ideas to create this equation, but at the end, we merged all of our ideas in one base and created a proper equation for getting the current time. When we created our equation properly, there is no obstacle for implementing this equation in the GetNow function. To observe close results when compared to simulation time made us proud.

5 CONCLUSION

In conclusion, we have constructed a sorted set linked list data structure with various functionalities which are allocation-deallocation, insertion, remove and error handling by using Assembly language. We had also faced with challenging problems during the implementation of the expected data structure especially in memory management and edge cases. Memory management and grasping the structure of the allocation table was complicated for us, but once we brainstormed with all of group members we have figured out the structure of allocation table and memory operations. We also had to check edge cases particularly in insert-remove operations to construct a robust program. We have benefited from our data structure knowledge frequently at this point. Besides all of these, we are familiar with core elements of Assembly language and memory management. We have deeper knowledge about the background of a computer program, how to allocate-deallocate memory, which operations are performed by a computer during the run-time of a program. To sum up, this project is a really important facility to experience what we have learned during the lecture and to apply our knowledge about data structures and high level programming languages to Assembly language which make us more well-informed in background of a computer program as computer engineers.

6 APPENDIX

```
         AREA    IN_DATA_AREA, DATA, READONLY
IN_DATA  DCD  0x0001, 0x0001, 0x0013, 0x0003, 0x0002, 0x0002, 0x0002, 0x0003, 0x0025, 0x0003, 0x0001, 0x0000
         DCD  0x0036, 0xFFFF, 0xFFFE, 0x0FFF, 0x0FFE, 0x00FF, 0x00FE, 0x000F, 0x000E
         DCD  0x0000, 0x0001, 0x0002, 0x0003, 0x0004, 0x0005, 0x0006, 0x0007, 0x0008, 0x0009, 0x000A, 0x000B, 0x000C, 0x000D, 0x000E, 0x000F
         DCD  0x0010, 0x0011, 0x0012, 0x0013, 0x0014, 0x0015, 0x0016, 0x0017, 0x0018, 0x0019, 0x001A, 0x001B, 0x001C, 0x001D, 0x001E, 0x001F
         DCD  0x0020, 0x0021, 0x0022, 0x0023, 0x0024, 0x0025, 0x0026, 0x0027, 0x0028, 0x0029, 0x002A, 0x002B, 0x002C, 0x002D, 0x002E, 0x002F
         DCD  0x0030, 0x0031, 0x0032, 0x0033, 0x0034, 0x0035, 0x0036, 0x0037, 0x0038, 0x0039, 0x003A, 0x003B, 0x003C, 0x003D, 0x003E, 0x003F
         DCD  0x0040, 0x0041, 0x0042, 0x0043, 0x0044, 0x0045, 0x0046, 0x0047, 0x0048, 0x0049, 0x004A, 0x004B, 0x004C, 0x004D, 0x004E, 0x004F
         DCD  0x0050, 0x0051, 0x0052, 0x0053, 0x0054, 0x0055, 0x0056, 0x0057, 0x0058, 0x0059, 0x005A, 0x005B, 0x005C, 0x005D, 0x005E, 0x005F
         DCD  0x0060, 0x0061, 0x0062, 0x0063, 0x0064, 0x0065, 0x0066, 0x0067, 0x0068, 0x0069, 0x006A, 0x006B, 0x006C, 0x006D, 0x006E, 0x006F
         DCD  0x0070, 0x0071, 0x0072, 0x0073, 0x0074, 0x0075, 0x0076, 0x0077, 0x0078, 0x0079, 0x007A, 0x007B, 0x007C, 0x007D, 0x007E, 0x007F
         DCD  0x0080, 0x0081, 0x0082, 0x0083, 0x0084, 0x0085, 0x0086, 0x0087, 0x0088, 0x0089, 0x008A, 0x008B, 0x008C, 0x008D, 0x008E, 0x008F
         DCD  0x0090, 0x0091, 0x0092, 0x0093, 0x0094, 0x0095, 0x0096, 0x0097, 0x0098, 0x0099, 0x009A, 0x009B, 0x009C, 0x009D, 0x009E, 0x009F
         DCD  0x00A0, 0x00A1, 0x00A2, 0x00A3, 0x00A4, 0x00A5, 0x00A6, 0x00A7, 0x00A8, 0x00A9, 0x00AA, 0x00AB, 0x00AC, 0x00AD, 0x00AE, 0x00AF
         DCD  0x00B0, 0x00B1, 0x00B2, 0x00B3, 0x00B4, 0x00B5, 0x00B6, 0x00B7, 0x00B8, 0x00B9, 0x00BA, 0x00BB, 0x00BC, 0x00BD, 0x00BE, 0x00BF
         DCD  0x00C0, 0x00C1, 0x00C2, 0x00C3, 0x00C4, 0x00C5, 0x00C6, 0x00C7, 0x00C8, 0x00C9, 0x00CA, 0x00CB, 0x00CC, 0x00CD, 0x00CE, 0x00CF
         DCD  0x00D0, 0x00D1, 0x00D2, 0x00D3, 0x00D4, 0x00D5, 0x00D6, 0x00D7, 0x00D8, 0x00D9, 0x00DA, 0x00DB, 0x00DC, 0x00DD, 0x00DE, 0x00DF
         DCD  0x00E0, 0x00E1, 0x00E2, 0x00E3, 0x00E4, 0x00E5, 0x00E6, 0x00E7, 0x00E8, 0x00E9, 0x00EA, 0x00EB, 0x00EC, 0x00ED, 0x00EE, 0x00EF
         DCD  0x00F0, 0x00F1, 0x00F2, 0x00F3, 0x00F4, 0x00F5, 0x00F6, 0x00F7, 0x00F8, 0x00F9, 0x00FA, 0x00FB, 0x00FC, 0x00FD, 0x00FE, 0x00FF
         DCD  0x0100, 0x0101, 0x0102, 0x0103, 0x0104, 0x0105, 0x0106, 0x0107, 0x0108, 0x0109, 0x010A, 0x010B, 0x010C, 0x010D, 0x010E, 0x010F
         DCD  0x0110, 0x0111, 0x0112, 0x0113, 0x0114, 0x0115, 0x0116, 0x0117, 0x0118, 0x0119, 0x011A, 0x011B, 0x011C, 0x011D, 0x011E, 0x011F
         DCD  0x0120, 0x0121, 0x0122, 0x0123, 0x0124, 0x0125, 0x0126, 0x0127, 0x0128, 0x0129, 0x012A, 0x012B, 0x012C, 0x012D, 0x012E, 0x012F
         DCD  0x0130, 0x0131, 0x0132, 0x0133, 0x0134, 0x0135, 0x0136, 0x0137, 0x0138, 0x0139, 0x013A, 0x013B, 0x013C, 0x013D, 0x013E, 0x013F
         DCD  0x0140, 0x0141, 0x0142, 0x0143, 0x0144, 0x0145, 0x0146, 0x0147, 0x0148, 0x0149, 0x014A, 0x014B, 0x014C, 0x014D, 0x014E, 0x014F
         DCD  0x0150, 0x0151, 0x0152, 0x0153, 0x0154, 0x0155, 0x0156, 0x0157, 0x0158, 0x0159, 0x015A, 0x015B, 0x015C, 0x015D, 0x015E, 0x015F
         DCD  0x0160, 0x0161, 0x0162, 0x0163, 0x0164, 0x0165, 0x0166, 0x0167, 0x0168, 0x0169, 0x016A, 0x016B, 0x016C, 0x016D, 0x016E, 0x016F
         DCD  0x0170, 0x0171, 0x0172, 0x0173, 0x0174, 0x0175, 0x0176, 0x0177, 0x0178, 0x0179, 0x017A, 0x017B, 0x017C, 0x017D, 0x017E, 0x017F
         DCD  0x0180, 0x0181, 0x0182, 0x0183, 0x0184, 0x0185, 0x0186, 0x0187, 0x0188, 0x0189, 0x018A, 0x018B, 0x018C, 0x018D, 0x018E, 0x018F
         DCD  0x0190, 0x0191, 0x0192, 0x0193, 0x0194, 0x0195, 0x0196, 0x0197, 0x0198, 0x0199, 0x019A, 0x019B, 0x019C, 0x019D, 0x019E, 0x019F
         DCD  0x01A0, 0x01A1, 0x01A2, 0x01A3, 0x01A4, 0x01A5, 0x01A6, 0x01A7, 0x01A8, 0x01A9, 0x01AA, 0x01AB, 0x01AC, 0x01AD, 0x01AE, 0x01AF
         DCD  0x01B0, 0x01B1, 0x01B2, 0x01B3, 0x01B4, 0x01B5, 0x01B6, 0x01B7, 0x01B8, 0x01B9, 0x01BA, 0x01BB, 0x01BC, 0x01BD, 0x01BE, 0x01BF
         DCD  0x01C0, 0x01C1, 0x01C2, 0x01C3, 0x01C4, 0x01C5, 0x01C6, 0x01C7, 0x01C8, 0x01C9, 0x01CA, 0x01CB, 0x01CC, 0x01CD, 0x01CE, 0x01CF
         DCD  0x01D0, 0x01D1, 0x01D2, 0x01D3, 0x01D4, 0x01D5, 0x01D6, 0x01D7, 0x01D8, 0x01D9, 0x01DA, 0x01DB, 0x01DC, 0x01DD, 0x01DE, 0x01DF
         DCD  0x01E0, 0x01E1, 0x01E2, 0x01E3, 0x01E4, 0x01E5, 0x01E6, 0x01E7, 0x01E8, 0x01E9, 0x01EA, 0x01EB, 0x01EC, 0x01ED, 0x01EE, 0x01EF
         DCD  0x01F0, 0x01F1, 0x01F2, 0x01F3, 0x01F4, 0x01F5, 0x01F6, 0x01F7, 0x01F8, 0x01F9, 0x01FA, 0x01FB, 0x01FC, 0x01FD, 0x01FE, 0x01FF
         DCD  0x0200, 0x0201, 0x0202, 0x0203, 0x0204, 0x0205, 0x0206, 0x0207, 0x0208, 0x0209, 0x020A, 0x020B, 0x020C, 0x020D, 0x020E, 0x020F
         DCD  0x0210, 0x0211, 0x0212, 0x0213, 0x0214, 0x0215, 0x0216, 0x0217, 0x0218, 0x0219, 0x021A, 0x021B, 0x021C, 0x021D, 0x021E, 0x021F
         DCD  0x0220, 0x0221, 0x0222, 0x0223, 0x0224, 0x0225, 0x0226, 0x0227, 0x0228, 0x0229, 0x022A, 0x022B, 0x022C, 0x022D, 0x022E, 0x022F
         DCD  0x0230, 0x0231, 0x0232, 0x0233, 0x0234, 0x0235, 0x0236, 0x0237, 0x0238, 0x0239, 0x023A, 0x023B, 0x023C, 0x023D, 0x023E, 0x023F
         DCD  0x0240, 0x0241, 0x0242, 0x0243, 0x0244, 0x0245, 0x0246, 0x0247, 0x0248, 0x0249, 0x024A, 0x024B, 0x024C, 0x024D, 0x024E, 0x024F
         DCD  0x0250, 0x0251, 0x0252, 0x0253, 0x0254, 0x0255, 0x0256, 0x0257, 0x0258, 0x0259, 0x025A, 0x025B, 0x025C, 0x025D, 0x025E, 0x025F
         DCD  0x0260, 0x0261, 0x0262, 0x0263, 0x0264, 0x0265, 0x0266, 0x0267, 0x0268, 0x0269, 0x026A, 0x026B, 0x026C, 0x026D, 0x026E, 0x026F
         DCD  0x0270, 0x0271, 0x0272, 0x0273, 0x0274, 0x0275, 0x0276, 0x0277, 0x0278, 0x0279, 0x027A, 0x027B, 0x027C, 0x027D, 0x027E, 0x027F
         DCD  0x0280, 0x0281, 0x0282, 0x0283, 0x0284, 0x0285, 0x0286, 0x0287, 0x0288, 0x0289, 0x028A, 0x028B, 0x028C, 0x028D, 0x028E, 0x028F
         DCD  0x0290, 0x0291, 0x0292, 0x0293, 0x0294, 0x0295, 0x0296, 0x0297, 0x0298, 0x0299, 0x029A, 0x029B, 0x029C, 0x029D, 0x029E, 0x029F
         DCD  0x02A0, 0x02A1, 0x02A2, 0x02A3, 0x02A4, 0x02A5, 0x02A6, 0x02A7, 0x02A8, 0x02A9, 0x02AA, 0x02AB, 0x02AC, 0x02AD, 0x02AE, 0x02AF
         DCD  0x02B0, 0x02B1, 0x02B2, 0x02B3, 0x02B4, 0x02B5, 0x02B6, 0x02B7, 0x02B8, 0x02B9, 0x02BA, 0x02BB
         DCD  0x0000, 0xFFFF, 0x0000
         DCD  0x0258, 0x01F4, 0x0190, 0x012C, 0x022B, 0x00C8, 0x0064, 0x0032, 0x1313, 0x027B
         DCD  0x0000
         DCD  0xEFEF, 0xABCD, 0xBCDE, 0xCDEF, 0x4747, 0x1111, 0x1233, 0x1233, 0x4747
         DCD  0x0000
END_IN_DATA
```

Figure 48: Prepared Data for Test

```
;@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  0x0000, 0x0002, 0x0025, 0x0001, 0x0000, 0x0001, 0x0000, 0x0001, 0x0003, 0x0000, 0x0000, 0x0002
              DCD  0x0049, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0002, 0x0001, 0x0002
              DCD  0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000
              DCD  0x0002
              DCD  0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001, 0x0001
              DCD  0x0002
END_IN_DATA_FLAG
```

Figure 49: Prepared Data Flags for Test