ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 212E

MICROPROCESSOR SYSTEMS
TERM PROJECT

DATE : 31.01.2021

GROUP NO : G7

GROUP MEMBERS:

150150104 : Ece Nur ŞEN

150150109 : Dilara İnan

150150015 : İdil Sezgin

070170450 : Muhammed Enes Deniz

FALL 2020

Contents

1 INTRODUCTION

In this project, we created an assembly program that takes data set and performs operation on data with given instruction set. Our instructions consist of adding an element to linked list, removing element from linked list and transforming linked list to array. In order to create this program, we have worked with Arm Cortex M0+ processor and we used Keil uVision IDE as integrated development environment. Since we didn't own an Arm Cortex M0+ processor, we tested our program with the simulator of Keil uVision. We arranged our processors clock frequency as 8 MHz. Our program creates a SysTick interrupt every 827 microseconds. Whenever an systick interrupt occurs, SysTick_Handler is called. In SysTick_Handler, a new data and its relative flag are read from given in_data array and in_data_flag array. According to its flag, data is sent to a function which might be Insert, Remove and LinkedList2Arr. If an error occurs, the error code and required inputs are sent to WriteErrorLog function. Errors are saved to memory. End of the the SysTick_Handler, we check whether the whether we read every data in in_data array or not. If every data is read, we stopped the systick timer and the program terminates.

2 MATERIALS AND METHODS

2.1 SysTick_Handler

SysTick_Handler function is called when systick counter reached to 0. Firstly, the in_data and its relative in_data_flag is read from the memory. Data represent the value, whereas the flag represent with function this value will be send. If flag is equal to 0, value is sent to Remove function. If flag is equal to 1, value is sent to insert function. If flag is equal to 2, LinkedList2Arr function is called without input. After functions returns with error code; tick count, error code, operation -which is flag- and value -which is data- is sent to WriteErrorLog. Error code that is sent could be 0, which represents no error occured during execution of function. Afterwards, TICK_COUNT will be incremented. We check, whether we reach the end of in_data array or not. If we reached the end of input data array, SysTick_Stop function is called. Finally, SysTick_Handler returns to main loop.

2.2 SysTick_Init

Systick_Init function sets the System Tick Timer registers, start the timer and update the program status register. To conduct these operations firstly we have to calculate the reload value with the given clock frequency: 8 Mhz and Period Of the System Tick Timer

Interrupt: 827 microseconds.

$$Period = (1 + ReloadValue)/F_{cpu}$$
$$827 * 10^{-}6 = (1 + ReloadValue)/8 * 10^{6}$$
$$827 * 8 = (1 + ReloadValue)$$
$$6616 = (1 + ReloadValue)$$
$$ReloadValue = 6615$$

After loading the reload value, we set the enable, clock and interrupt flags as 1. We update the program status register content as 1 to start the timer.

2.3 SysTick_Stop

SysTick_Stop function stops the System Tick Timer. First we take the address of systick control and status register then we load zero value to this address. It makes countflag = 0, clksource = 0, tickint =0 and enable = 0. Then we clear the interrupt flag. After that we take the address of program status to update its value and we give two to program status to say that all data operations are finished.

2.4 Clear Functions

Clear_ErrorLogs and Clear_Alloc functions work in the same way. Clear_Alloc function uses the starting address and size of the allocation table (AT_MEM), while the Clear_ErrorLog function uses the starting address and size of the error log array (LOG_MEM). After the required values are loaded, a register is initialized to keep the index value. The selected indexes of the arrays are reset, then the index is increased and the loop starts again. The index value is compared with the size of the array, when the index is equal to the size, the function ends.

2.5 Init_GlobVars

Init_GlobVars function initializes the globals variables such as TICK_COUNT, FIRST_ELEMENT, INDEX_INPUT_DS, INDEX_ERROR_LOG and PROGRAM_STATUS. Firstly we give zero value to a register, then store the register's value in the global variables.

2.6 Malloc

Malloc function finds and allocates the unused memory node from allocation table. In allocation table there exists 20 words and each word consists from 32 bits. Moreover, each bit points one memory node so in other words in every word there exists 32 memory nodes and in total there exists 20x32 = 640 space for allocation. To visualize this we have created the allocation table's possible look in figure 1. There exists 20 rows which represents words and 32 columns which represents 32-bit.

1st Word	0. bit	1. bit	2. bit	3. bit	4. bit	5. bit	6. bit	7. bit	...	...	...	25. bit	26. bit	27. bit	28. bit	29. bit	30. bit	31. bit
2nd Word	0. bit	1. bit	2. bit	3. bit	4. bit	5. bit	6. bit	7. bit	...	...	...	25. bit	26. bit	27. bit	28. bit	29. bit	30. bit	31. bit
3rd Word	0. bit	1. bit	2. bit	3. bit	4. bit	5. bit	6. bit	7. bit	...	...	...	25. bit	26. bit	27. bit	28. bit	29. bit	30. bit	31. bit
4th Word	0. bit	1. bit	2. bit	3. bit	4. bit	5. bit	6. bit	7. bit	...	...	...	25. bit	26. bit	27. bit	28. bit	29. bit	30. bit	31. bit
5th Word	0. bit	1. bit	2. bit	3. bit	4. bit	5. bit	6. bit	7. bit	...	...	...	25. bit	26. bit	27. bit	28. bit	29. bit	30. bit	31. bit
6th Word	0. bit	1. bit	2. bit	3. bit	4. bit	5. bit	6. bit	7. bit	...	...	...	25. bit	26. bit	27. bit	28. bit	29. bit	30. bit	31. bit
7th Word	0. bit	1. bit	2. bit	3. bit	4. bit	5. bit	6. bit	7. bit	...	...	...	25. bit	26. bit	27. bit	28. bit	29. bit	30. bit	31. bit
8th Word	0. bit	1. bit	2. bit	3. bit	4. bit	5. bit	6. bit	7. bit	...	...	...	25. bit	26. bit	27. bit	28. bit	29. bit	30. bit	31. bit
9th Word	0. bit	1. bit	2. bit	3. bit	4. bit	5. bit	6. bit	7. bit	...	...	...	25. bit	26. bit	27. bit	28. bit	29. bit	30. bit	31. bit
10th Word	0. bit	1. bit	2. bit	3. bit	4. bit	5. bit	6. bit	7. bit	...	...	...	25. bit	26. bit	27. bit	28. bit	29. bit	30. bit	31. bit
11th Word	0. bit	1. bit	2. bit	3. bit	4. bit	5. bit	6. bit	7. bit	...	...	...	25. bit	26. bit	27. bit	28. bit	29. bit	30. bit	31. bit
12th Word	0. bit	1. bit	2. bit	3. bit	4. bit	5. bit	6. bit	7. bit	...	...	...	25. bit	26. bit	27. bit	28. bit	29. bit	30. bit	31. bit
13th Word	0. bit	1. bit	2. bit	3. bit	4. bit	5. bit	6. bit	7. bit	...	...	...	25. bit	26. bit	27. bit	28. bit	29. bit	30. bit	31. bit
14th Word	0. bit	1. bit	2. bit	3. bit	4. bit	5. bit	6. bit	7. bit	...	...	...	25. bit	26. bit	27. bit	28. bit	29. bit	30. bit	31. bit
15th Word	0. bit	1. bit	2. bit	3. bit	4. bit	5. bit	6. bit	7. bit	...	...	...	25. bit	26. bit	27. bit	28. bit	29. bit	30. bit	31. bit
16th Word	0. bit	1. bit	2. bit	3. bit	4. bit	5. bit	6. bit	7. bit	...	...	...	25. bit	26. bit	27. bit	28. bit	29. bit	30. bit	31. bit
17th Word	0. bit	1. bit	2. bit	3. bit	4. bit	5. bit	6. bit	7. bit	...	...	...	25. bit	26. bit	27. bit	28. bit	29. bit	30. bit	31. bit
18th Word	0. bit	1. bit	2. bit	3. bit	4. bit	5. bit	6. bit	7. bit	...	...	...	25. bit	26. bit	27. bit	28. bit	29. bit	30. bit	31. bit
19th Word	0. bit	1. bit	2. bit	3. bit	4. bit	5. bit	6. bit	7. bit	...	...	...	25. bit	26. bit	27. bit	28. bit	29. bit	30. bit	31. bit
20th Word	0. bit	1. bit	2. bit	3. bit	4. bit	5. bit	6. bit	7. bit	...	...	...	25. bit	26. bit	27. bit	28. bit	29. bit	30. bit	31. bit

Figure 1: Allocation Table

From starting the least significant bit of each word we traverse to whole table and when we come up to empty memory node which is denoted by 0 bit we allocate that space for the next element by doing two operation: Firstly we have set the found 0 bit as 1 and then returning its address. To return the node address we have to make a conversion for finding the node's place in data area which is used for the linked list. Similarly to visualize the data area we have created the table in figure 2. There exists again 20 rows to make visual alignment with allocation table and 32 cell(column) with each cell of it consist by 8 byte area this 8 byte is divided as following: 4 byte for value, 4 byte for next value's address.

Row	1st Memory Node	2nd Memory Node	3rd Memory Node	4th Memory Node						30th Memory Node	31st Memory Node	32nd Memory Node
1st Word	8 byte	8 byte	8 byte	8 byte	...	...	...	...	...	8 byte	8 byte	8 byte
2nd Word	8 byte	8 byte	8 byte	8 byte	...	...	...	...	...	8 byte	8 byte	8 byte
3rd Word	8 byte	8 byte	8 byte	8 byte	...	...	...	...	...	8 byte	8 byte	8 byte
4th Word	8 byte	8 byte	8 byte	8 byte	...	...	...	...	...	8 byte	8 byte	8 byte
5th Word	8 byte	8 byte	8 byte	8 byte	...	...	...	...	...	8 byte	8 byte	8 byte
6th Word	8 byte	8 byte	8 byte	8 byte	...	...	...	...	...	8 byte	8 byte	8 byte
7th Word	8 byte	8 byte	8 byte	8 byte	...	...	...	...	...	8 byte	8 byte	8 byte
8th Word	8 byte	8 byte	8 byte	8 byte	...	...	...	...	...	8 byte	8 byte	8 byte
9th Word	8 byte	8 byte	8 byte	8 byte	...	...	...	...	...	8 byte	8 byte	8 byte
10th Word	8 byte	8 byte	8 byte	8 byte	...	...	...	...	...	8 byte	8 byte	8 byte
11th Word	8 byte	8 byte	8 byte	8 byte	...	...	...	...	...	8 byte	8 byte	8 byte
12th Word	8 byte	8 byte	8 byte	8 byte	...	...	...	...	...	8 byte	8 byte	8 byte
13th Word	8 byte	8 byte	8 byte	8 byte	...	...	...	...	...	8 byte	8 byte	8 byte
14th Word	8 byte	8 byte	8 byte	8 byte	...	...	...	...	...	8 byte	8 byte	8 byte
15th Word	8 byte	8 byte	8 byte	8 byte	...	...	...	...	...	8 byte	8 byte	8 byte
16th Word	8 byte	8 byte	8 byte	8 byte	...	...	...	...	...	8 byte	8 byte	8 byte
17th Word	8 byte	8 byte	8 byte	8 byte	...	...	...	...	...	8 byte	8 byte	8 byte
18th Word	8 byte	8 byte	8 byte	8 byte	...	...	...	...	...	8 byte	8 byte	8 byte
19th Word	8 byte	8 byte	8 byte	8 byte	...	...	...	...	...	8 byte	8 byte	8 byte
20th Word	8 byte	8 byte	8 byte	8 byte	...	...	...	...	...	8 byte	8 byte	8 byte

Figure 2: Data Area

The general flow of the malloc function could be summarized as follow: Firstly the word from the allocation table traversed bit by bit and trying to find 0 bit, if there exists no 0 bit in the current word we look the next word this process repeated until we reached 20th word, after that point if we still could not find any 0 bit that means the linked list is full and we return 0 as error code. When we find the 0 bit in other words empty memory node, firstly we set it as 1 and return its corresponding address in the data area by applying the following formula:

$$\&allocated\_area = \&data\_area\_start + ((32X(0\_bit\_found\_row - 1)) + 0\_bit\_found\_cell)X2^3) \quad (1)$$

- 32 = maximum cell number (column)
- &data_area_start = starting address of the linked list
- &allocated_area = allocated memory node address for the next operation

2.7 Free

Free function takes the memory node address as argument and apply undo operation to find memory node's place in the allocation table. After finding the corresponding word an the bit of the word, bit clearing operation conducted and memory free process is accomplished. The undo operation that conducted has two stages: In the first stage firstly we identify the corresponding word in the allocation table by looking whether the memory node that will be freed minus the start of the data_area staring address is greater than 256, if it is, then memory_node address - 256 is take place and it repeats until it less or equal to 256. After finding the word in the first step only the cell value left and it will give us the bit that is need to be cleared, by following this steps bit clear is conducted.

2.8 Insert

Insert function adds the given value to linked list. If there exists another element with same value in the linked list, the given value will not be added to linked list and function returns error code 2. If the linked list is full, value will not be added to linked list and function will return with error code 1. The flow of function is as follow: Function firstly checks whether the linked list is empty or not. If the linked list is empty, function will branch to "InsertFirst" subroutine. If linked list is not empty, function will check whether the given value is smaller than first element of the linked list or not. Linked list is sorted in ascending order. If the given value is smaller than first element of the linked list, function will branch to "InsertFirst" subroutine. Otherwise, the loop phase of Insert function will begin. In the loop, function will firstly check whether the given value is equal to pointed element of linked list. If the values are equal, function will return with error code 2. Otherwise, function will check whether the pointed element of linked list is the last element of the list or not. If it is the last element, function will branch to "InsertMid" subroutine to insert given value to end of linked list. If the pointed element isn't the last element, pointer of linked list will be incremented and will show the next element. If the newly pointed element is larger than given value, function will branch to "InsertMid" subroutine. If not, the loop will continue. In "InsertFirst" subroutine, function will firstly call Malloc function to allocate memory for new value. If Malloc function returns with error, Insert function will return with error code 1, meaning the memory is full. If Malloc returns with address, given value will be saved to address. First element will be configured to show new address and pointer of the new element will point to old first element. Insert function will return with error code 0, insertion successful. Only difference between "InsertMid" and "InsertFirst" subroutines is that the given value will not be added as a first element in the linked list. Therefore, the given value will connected to previous element and current element of the linked list.

2.9 Remove

Remove function searches for the given value in the linked list. If the linked list is empty the function returns with error code 3. If the given value is not found the function returns with error code 4. Then if there is no error, it means given value is found. When the value is found, the function controls is the element at the beginning of the linked list or not. Because if the given value is the first element, the function have to change to the address of FIRST_ELEMENT pointer. If not the function gives the previous elements next pointer to the next element. Then the function calls free function to clear corresponding bits of the element from the allocation table. Lastly the functions returns 0 as success

code.

2.10 LinkedList2Arr

LinkedList2Arr function first clears the content of the AREA_MEM because we will write the elements of the linked list to this array. The function checks if the linked list is empty or not. If the linked list empty then the function returns with error code 5. If not the function iterates every element one by one and stores their values in the array. At the end, the function returns 0 as success code.

2.11 WriteErrorLog

WriteErrorLog function takes four arguments. First, the argument in the R1 register (error code) is checked. If this value is zero, the function is terminated directly because the function operates with non-zero error codes. After this step, the starting address and index value of the error log array are loaded. The current address of the array is determined by adding the index value to the starting address. If this address is not equal to __LOG_MEM then array is not full. Arguments are written to the corresponding addresses sequentially. Then the GetNow function is called to get the 32-bit timestamp value. The return value is taken from the R0 register and written to the last place. The index value is incremented and the function is terminated.

2.12 GetNow

GetNow function calculates the working time of System Tick Timer using the TICK_COUNT variable and System Tick Timer Current Value Register.

TICK_COUNT holds the number of interrupts so far. We can simply multiply TICK_COUNT with period. In our case an interrupt occurs every 827 microseconds. R0 register temporarily holds the result of this multiplication. After, to find the value of the System Tick Timer current register, its address is loaded into register R3 (0xE000E018). The value obtained is not yet correct because the clock counts from reload value to zero. So we need to subtract received value from reload value to find actual current value. To convert this value to microseconds, the formula we use to find the reload value is used.

$$Period = (1 + ReloadValue)/FCPU$$

$$CurrentTime(seconds) = (1 + CurrentValue)/8 * 10^6 \quad (2)$$

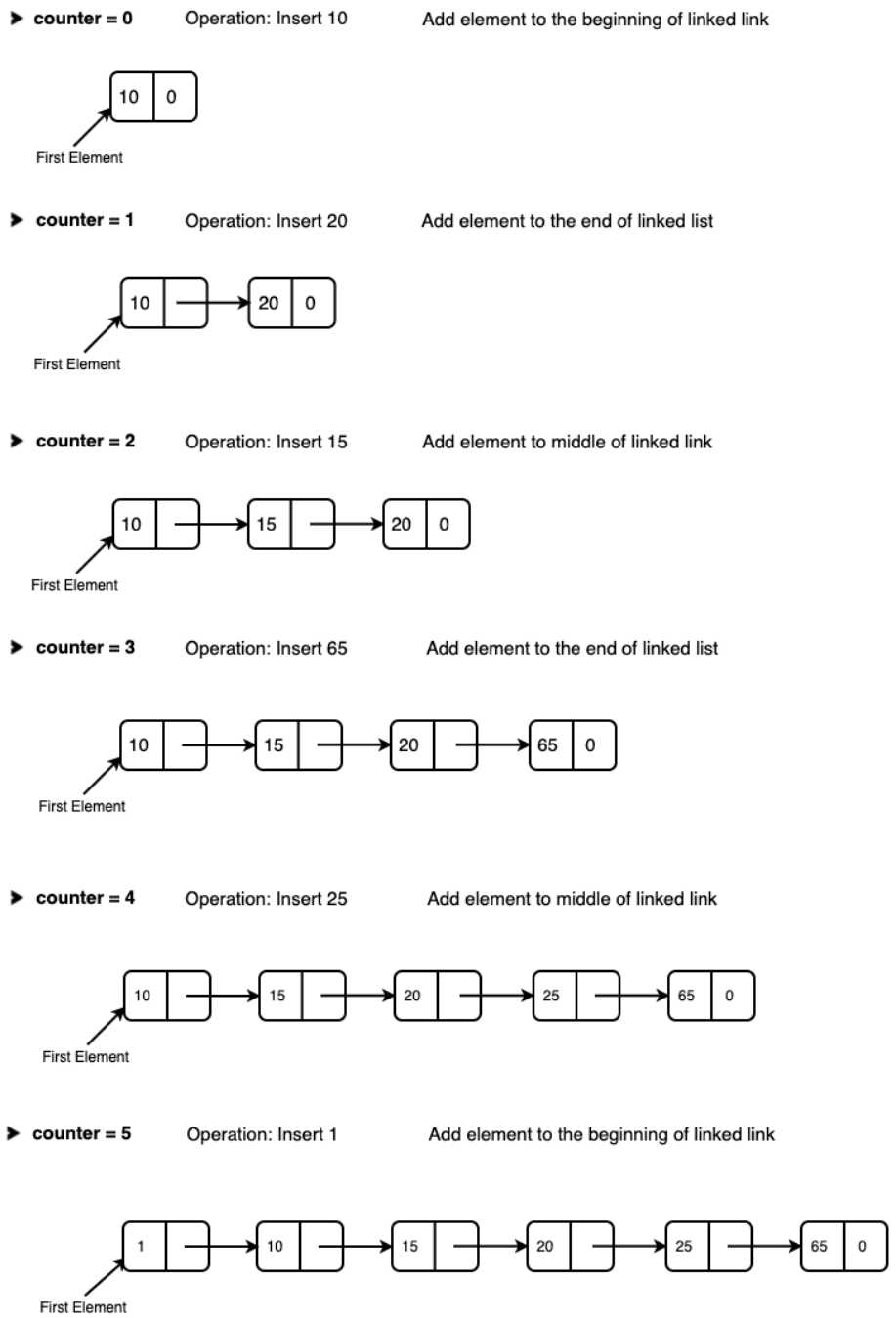
$$CurrentTime(microseconds) = (1 + CurrentValue)/8 \quad (3)$$

Finally current time obtained in microseconds added to R0 and function ends.

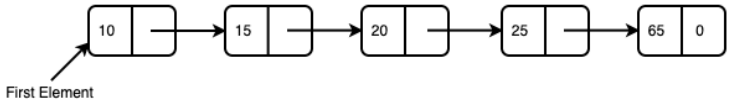
3 RESULTS

3.1 Given Test Case

In Figure 3 you can see a schema for the given test case.



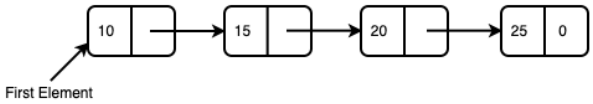
➤ **counter = 6** Operation: Remove 1 Remove element from the beginning of linked link



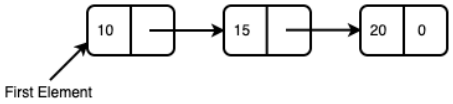
➤ **counter = 7** Operation: Remove 12 Remove non existing element

Return: Error code 4

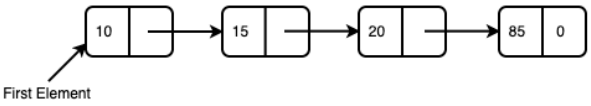
➤ **counter = 8** Operation: Remove 65 Remove element from the end of linked list



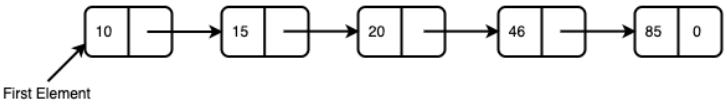
➤ **counter = 9** Operation: Remove 25 Remove element from the end of linked list



➤ **counter = 10** Operation: Insert 85 Add element to the end of linked list



➤ **counter = 11** Operation: Insert 46 Add element to middle of linked link



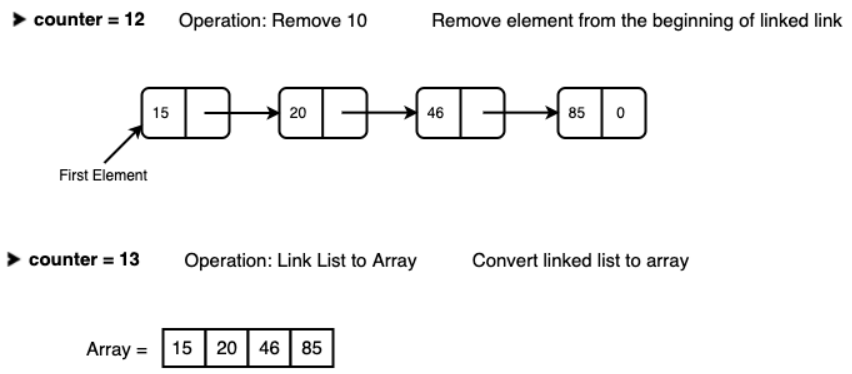


Figure 3: Result of given test case

When we run the given test case we had one error which is deletion of 12. Figure 4 shows the error log of given test case. In the first four digit of error log if 1C which is 28 in decimal. It means the index of 12 is 7. Then the next two digits shows 04 which is the error code. Next two digit is 00 which is operation. Next eight digit shows 12 which is input. And last 8 digits shows that the current System Tick Timer working time in microseconds.

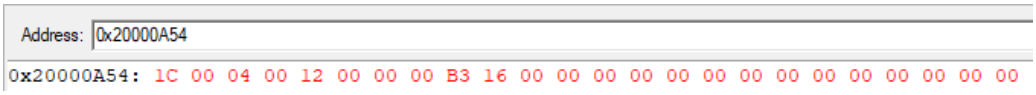
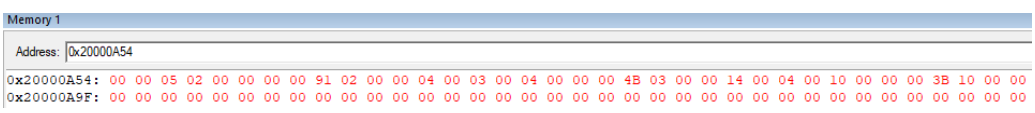


Figure 4: Error log of given test case

3.2 Our Test Case

We created a new test case to try all operations and other errors. Our IN_DATA elements are: 0x00, 0x04, 0x18, 0x09, 0x07, 0x10, 0x13, 0x07, 0x09, 0x18, 0x00 and our IN_DATA_FLAG elements are 0x02, 0x00, 0x01, 0x01, 0x01, 0x00, 0x01, 0x00, 0x01, 0x00, 0x02. When we run the program with our test case, in first element we had error code 5 which is converting empty linked list to array. In second element we had error code 3 which is remove element when linked list is empty. In sixth element we had error code 4 which is remove element when the element is not in the linked list. Lastly, in ninth element we had error code 2 which is insert the duplicated value. You can see the error_log array in the Figure 5.



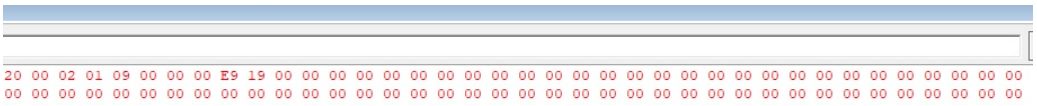


Figure 5: Error log of our test case

4 DISCUSSION

In this project we have to create sorted set linked list from scratch to end. Memory organization was solely handled by ourselves and we have created our own malloc and free functions. Our prior knowledge about sorted set linked list had definitely ease the process of understanding the project general framework and context. Yet, the unfamiliarity of the development environment and assembly language created many obstacles in our advancement.

In order to progress securely, we have created checkpoints and after we reach each checkpoint we have tested our implementations. Firstly we have implemented the functions from b to g, in there we make sure that our SysTick registers and timer functions work properly. In the next check point we have prepared then tested malloc and insert functions in order to make sure our memory allocation and linked list creation steps works accurately. Lastly, we have tested remove, linkedlist2array, free, writeErrorlog and Getnow functions to make sure our memory deallocation, linkedlist convertion to array and error codes functions work successfully.

Through our progress we have faced many minor and major obstacles. The problems that are related with the unfamiliarity with ARM development studio are the following:

- Error L6286E: Relocation #REL:6 : The following problem is occurred since we have tried to branch the functions using conditional branching commands since we can jump only + - 254 addresses in conditional branch we should rather use BL for to branch to function.

- Returning from functions: Since this was a major project, we had to make lots of function calls inside of function calls. When we used BL for calling functions, BL alters LR in order to return from function call using BX. However, LR is just overwritten with every BL, instead of pushing to a stack. After we observed that our program doesn't not return from function calls when we call more than one function, we realized that we have to save LR values. Hence, we started to use stack to push LR values and this solved our function return problem.

The problems we have face because of the complexity of the project concepts are the following:

- Memory organization: In our aspect the memory organization is one of the hardest part of the project and we have work on this problem together. To ease our understanding we have create visualization for the memory tables. Employing lots of our work power on the debugging process has created a workload for us.

- In insert and remove functions, adding or removing element from linked list differs when the element is first element of the linked list. Thus, while writing this functions, we had to keep that difference in mind.

The project was covering nearly all subjects discussed in the fall term. During our progress, we have embodied most of the concepts we have learned and able to gain some basic fundamentals on assembly programming. Since we could not physically gather and come together we have experienced some communication problems yet we have accomplished to overcome them.

5 CONCLUSION

Implementing a sorted set linked list is a complicated process even with a higher level programming language. We faced many difficulties such as lack of registers, incorrect LR values, pointer usage and variety of errors. The most challenging part was that mistakes in a function could only be noticed when other functions gave unreasonable errors. At first, we had difficulties returning from functions because we lost correct LR values constantly. To overcome this, we tried to return from nested functions with B, but it turned out that this is not the correct way. At the end, we learned the correct way to return functions. After completing the code, with the help of the test cases we prepared, we fixed the errors that we could not predict at first and successfully completed the project.