ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 212E

MICROPROCESSOR SYSTEMS
TERM PROJECT

DATE : 31.01.2021

GROUP NO : G18

GROUP MEMBERS:

150170087 : Sırrı Batuhan ÇOKSAK

150170039 : Fatih MURAT

150170069 : Furkan Yusuf GÜRAY

150160123 : Matthew Oğuzhan DİRLİK

150160142 : Ekin Zuhat YAŞAR

FALL 2020

Contents

1 INTRODUCTION

In this project we created a sorted set linked list structure using assembly language on Keil uVision5. We have implemented our own insert, delete, malloc and free functions that go with our system tick handler function. Our group had the following values to use in our design;

- CPU Clock Frequency: 32 MHz
- Period Of the System Tick Timer Interrupt: 602 µs

2 MATERIALS AND EXPERIMENT

A list of elements used in this project is given below.

- Keil uVision5
- Notes and videos from lectures

2.1 SysTick_Handler()

The SysTick_Handler function is where our code reads the input data and flag, and executes the corresponding action according to the table given in the homework instructions. This function also ensures that the program writes an error log in the case of an error occurring during one of the operations that it attempts to execute. Assembly code part for the function is given as in Listing 1.

```
SysTick_Handler FUNCTION
        EXPORT SysTick_Handler
        PUSH {LR, R0, R1}        ; Since in the main r0 and r1 is constantly
                                   used push them here

        ; Block that increments tick count
        LDR R1, =TICK_COUNT      ; Get the tick count address
        LDR R0, [R1]             ; Load the tick count value
        ADDS R0, R0, #1          ; Increment tick count value
        STR R0, [R1,#0]          ; Store the incremented value to its
    address

        LDR R7, =INDEX_INPUT_DS  ; Get the INDEX_INPUT_DSs address
        LDR R0, [R7,#0]          ; Get the value in the INDEX_INPUT_DS
        LDR R6, =IN_DATA         ; Get the data IN_DATA address
        LDR R3, [R6,R0]          ; Get the value in IN_DATA using index
```

```
            LDR R6, =IN_DATA_FLAG      ; Get the IN_DATA_FLAG
                                         address
            LDR R2, [R6,R0]            ; Get the value in IN_DATA_FLAG using index

            CMP R2, #2                 ; Check if in_data_flag is 2
            BEQ OP2                    ; If so go to OP2 label
            CMP R2, #1                 ; Check if in_data_flag is 1
            BEQ OP1                    ; If so go to OP1 label
            CMP R2, #0                 ; Check if in_data_flag is 0
            BEQ OP0                    ; If so go to OP0 label

            MOVS R1, #6                ; Operation is not found code saved to R1
            B Endf                     ; Branch to Endf label

OP2         PUSH {R0}                  ; Push R0 register first just in case
            PUSH {R1-R7}               ; Push registers
            BL LinkedList2Arr          ; Call corresponding function
            POP {R1-R7}                ; Pop registers (r0 is the return register)
            MOVS R1, R0                ; Move the error code to R1
            POP {R0}                   ; R0s value back
            B Endf                     ; Branch to Endf label

OP1         PUSH {R0}                  ; Push R0 register first just in case
            PUSH {R1-R7}               ; Push registers
            MOVS R0, R3                ; Move the data to R0 since it is the
                                         parameter to insert function
            BL Insert                  ; Call Insert Function
            POP {R1-R7}                ; Pop registers (r0 is the return register)
            MOVS R1, R0                ; Move the error code to R1
            POP {R0}                   ; R0s value back
            B Endf                     ; Branch to Endf label

OP0         PUSH {R0}                  ; Push R0 register first just in case
            PUSH {R1-R7}               ; Push registers
            MOVS R0, R3                ; Move the data to R0 since it is the
                                         parameter to insert function
            BL Remove                  ; Call Insert Function
            POP {R1-R7}                ; Pop registers (r0 is the return register)
            MOVS R1, R0                ; Move the error code to R1
            POP {R0}                   ; R0s value back
            B Endf                     ; Branch to Endf label

Endf        PUSH {R0-R7}               ; Push registers
            BL WriteErrorLog           ; Call WriteErrorLog
            POP {R0-R7}                ; Pop registers
```

```
        ADDS R0, #4              ; Increment Index of the input dataset
        STR R0, [R7]             ; Update the INDEX_INPUT_DS value
        CMP R2, #2               ; If operation was LinkedList2Arr then go
                                   to Stop_ label
        BEQ Stop_                ; Branch if equal to Stop_ label

scont1  POP {R0,R1}              ; Pop the mains r0 and r1 values
        POP {PC}                 ; Pop LR to PC

Stop_   BL SysTick_Stop          ; Call SysTick_Stop
        B scont1                 ; Back to scont1 label

        ENDFUNC
```

Listing 1: SysTick_Handler() Function Codes

Once at the end of the ISR, we check to see if all the data has been read and if so, the timer is stopped.

2.2 SysTick_Init()

This function initializes System Tick Timer registers, starts the timer, and updates the program status register as desired. In our project, the CPU clock frequency is 32 MHz and period of the System Tick Timer is 602 μs as given. We calculated the reload value according to the formula below;

- Reload Value = (Timer / Clock Period) - 1

From that formula, we obtained our Reload Value as 19263. After that, we load 19263 value to SysTick Reload Value Register which is located in the address of 0xE000E014.

After loading the Reload Value to SysTick Reload Value Register, we clear the value in SysTick Current Value Register which its address is 0xE000E014 by loading 0 to it.

And finally, we set 1 to Enable, Clock and Tickint bits in SysTick Control and Status Register which its address is 0xE000E010 by loading 7 (b111) to it.

When we are done with this registers, we load 1 to PROGRAM_STATUS which means that required system tick initializations are done and program is ready to go (Timer started). Assembly code part for the function is given as in Listing 2.

```
SysTick_Init   FUNCTION
        LDR R0, =0xE000E010;     ; Load the address of SYST_CSR
        LDR R1, = 19263          ; RV+1 = 602*10^-6 *  32* 10^6
                                   hence reload value is 19263
        STR R1, [R0, #4]         ; Store reload value to SYST_RVR
        MOVS R1, #0              ; Move 0 to r1
```

```
        STR R1, [R0, #8]            ; Clear the value in SYST_CVR
        MOVS R1, #7                 ; Move 7 to r1
        STR R1 , [R0]               ; Set ENABLE, CLOCK And TICKINT to
                                      1 in SYST_CSR
        MOVS R0, #1                 ; Move 1 to r1
        LDR R1, =PROGRAM_STATUS     ; Get the address of program status
        STR R0, [R1,#0]             ; Store 1 to that address
        BX LR                       ; Branch with link register
        ENDFUNC
```

Listing 2: SysTick_Init() Function Codes

2.3 SysTick_Stop()

In this function, we did the reverse operations of operations that we did in SysTick_Init(). First we go to SysTick Control and Status Register which its address is 0xE000E010 which is told before. After that we set the value in the register to 0 (b000). This operation sets 0 to Enable, Clock and Tickint bits in SysTick Control and Status Register.

When we are done with resetting the value inside the register, we load 2 to PROGRAM_STATUS which means that all data operations are finished. Overall, Figure x shows the flag codes of the program status variable. Assembly code part for the function is given as in Listing 3.

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

Figure 1: Flag Codes of the Program Status

```
SysTick_Stop FUNCTION
        LDR R3, =0xE000E010         ; Load the address of SYST_CSR
        MOVS R4, #0                 ; Move 0 to r4
        STR R4 , [R3,#0]            ; Set ENABLE, CLOCK And TICKINT to 0 in
                                      SYST_CSR
        MOVS R4, #2                 ; Move 2 to r4
        LDR R3, =PROGRAM_STATUS     ; Get the address of program status
        STR R4, [R3,#0]             ; Store 2 to that address
        BX LR                       ; Branch with link register
```

```
        ENDFUNC
```

Listing 3: SysTick_Stop() Function Codes

2.4 Clear_Alloc()

In this function, we are asked to clear all bits in the allocation table. Firstly we load starting address of the allocation table to R3 register and load size of the allocation table to R4 register. We load 0 to R5 for using as index value and load 0 to R6 to constant value. Then, we control our index value will reach size of allocation table. If index value reach the size of allocation table, loop is break, branch CONT1 label and function return with BX LR. Else, store the 0 value to the element indicated by the index value in the allocation table. After that, we increment the index value by 4 because of allocation table elements type is word and each word take 4 bytes in memory. Then, branch to LOOP1 label and the loop will continue until the index value reaches the allocation table size. Therefore, we can clear all bit in allocation table by loading 0 value to each element. Assembly code part for the function is given as in Listing 4.

```
Clear_Alloc FUNCTION
        LDR R3, =AT_MEM             ; Adress of the Allocation table start
        LDR R4, =AT_SIZE            ; Size of the Allocation table
        MOVS R5, #0                 ; r5 will be used as an index
        MOVS R6, #0                 ; r6 is the constant 0 value

LOOP1
        CMP R5, R4                  ; Compare index with size if
        BEQ CONT1                   ; If they are equal go to CONT1 label
        STR R6, [R3, R5]            ; Store the constant value to AT_MEM with
                                      r5 offset
        ADDS R5 , R5, #4            ; Increase the index
        B LOOP1                     ; Go back to LOOP1 label

CONT1
        BX LR                       ; Branch with link register
        ENDFUNC
```

Listing 4: Clear_Alloc() Function Codes

2.5 Clear_ErrorLogs()

This function clears all cells in the Error Log Array. The function work same as Clear_Alloc(). Firstly we load starting address of the error log array to R3 register and load array size to R4 register. Then we load 0 as index value to R5 register, load constant 0 value to R6 register. If index value will reach array size, loop is break, branch CONT1 label and function return with BX LR. Else, store the 0 value to the element indicated by the index value in the allocation table. After, increment index value by 4 and branch LOOP2 label. This loop will continue until the index value reaches the array size. Assembly code part for the function is given as in Listing 5.

```
Clear_ErrorLogs FUNCTION
          LDR R3, =LOG_MEM          ; Adress of the error log memory start
          LDR R4, =LOG_ARRAY_SIZE   ; Size of the error log memory table
          MOVS R5, #0               ; r5 will be used as an index
          MOVS R6, #0               ; r6 is the constant 0 value

LOOP2
          CMP R5, R4                ; Compare index with size if
          BEQ CONT2                 ; If they are equal go to CONT2 label
          STR R6, [R3, R5]          ; Store the constant value to AT_MEM with
                                      r5 offset
          ADDS R5 , R5, #4          ; Increase the index
          B LOOP2                   ; Go back to LOOP2 label

CONT2
          BX LR                     ; Branch with link register
          ENDFUNC
```

Listing 5: Clear_Alloc() Function Codes

2.6 Init_GlobVars()

In this function, we are asked to initialize all global values. We load 0 value to R0 register. Then we use the R1 register to get global values and store a value of 0, because every element must be initialized as 0. We will execute a load and store process for all global variables. At the and of, return the function with BX LR. Assembly code part for the function is given as in Listing 6.

```
Init_GlobVars FUNCTION
          MOVS R0, #0               ; Move constant 0 to r0
          LDR R1, =TICK_COUNT       ; Get the address of tick_count
          STR R0, [R1,#0]           ; Store 0 to tick_count
          LDR R1, =FIRST_ELEMENT    ; Get the address of FIRST_ELEMENT
          STR R0, [R1,#0]           ; Store 0 to FIRST_ELEMENT
          LDR R1, =INDEX_INPUT_DS   ; Get the address of INDEX_INPUT_DS
          STR R0, [R1,#0]           ; Store 0 to INDEX_INPUT_DS
          LDR R1, =INDEX_ERROR_LOG; Get the address of INDEX_ERROR_LOG
          STR R0, [R1,#0]           ; Store 0 to INDEX_ERROR_LOG
```

```
        LDR R1, =PROGRAM_STATUS  ; Get the address of PROGRAM_STATUS
        STR R0, [R1,#0]          ; Store 0 to PROGRAM_STATUS
        BX LR                    ; Branch with link register
        ENDFUNC
```

Listing 6: Init_GlobVars() Function Codes

2.7 Malloc()

In this function, we are expected to find the unused memory node and allocates it using the allocation table. First of all, we load starting address of our linked list to the R0 register, the number of allocation table to R1 register, 0 which is the index value to R2 register, 0 constant value to R3 register, and the starting address of the allocation table to R4 register. Then, we control index value will reach to the number of allocation table value. If it reaches, branch MALLOCE label and returns error value 0 with R0 register because the linked list is full this situation. Else, we load the first line, which is first word of allocation table, to R5 register and 1, which is shifting number, to R6 register. Shifting number will shift left by one each MALLOCL2 loop and after 32 shift, our shifting number is equal 0 value because of overflow so, MALLOCL2 loop will run 32 times. After that, we control our shifting number equal to 0. if it is, branch MALLOCC1 label increment R2 register by one and R4 register by four because of going to next line in the allocation table. Else, we load the word in the R5 register to R7 register. We do 'and' operation R7 register with R6 register which is shifting number. If the bit for which we apply the 'and' operation is 0, the operation will return 0. Therefore, there is an space in that bit branch ENDM label, set that bit 1 with ORRS and STR operation and return function. Else, the bit is 1 so there is no space that bit. We increment R0 value by 8 because of going to next data on linked list. We do logical shift left operation by one to shifting value for controlling next bit on allocation table and branch MALLOCL2 label. Assembly code part for the function is given as in Listing 7.

```
Malloc  FUNCTION
        LDR R0, =DATA_MEM        ; Get the address of DATA_MEM
        LDR R1, =NUMBER_OF_AT    ; Get the global value NUMBER_OF_AT
        MOVS R2, #0              ; Index starting from 0
        MOVS R3, #0              ; Constant 0
        LDR R4, =AT_MEM          ; Address of the Allocation table start

MALLOCL1
        CMP R2, R1               ; Compare index with the NUMBER_OF_AT which
                                   is the size for this loop
        BHI MALLOCE              ; If they are equal go to MALLOCE label

```

```
        LDR R5, [R4]            ; Load the number in r4 to r5
        MOVS R6, #1             ; Shifting the number

MALLOCL2
        CMP R6, R3              ; Loop 32 times
        BEQ MALLOCC1            ; If r6 becomes 0 go to MALLOCC1 label
        MOVS R7, R5             ; Move r5 to r7
        ANDS R7, R7, R6         ; And r7 with r6 and save it to r7
        CMP R7 , R3             ; Compare the added value to 0
        BEQ ENDM                ; If the result is zero then there is space
        ADDS R0, #8             ; Else increment data mem pointer by 8
                                  since it has 2 int values for each node
        LSLS R6, R6,#1          ; Shift the shifting number to left by 1
        B MALLOCL2              ; Branch to MALLOCL2 label

MALLOCC1
        ADDS R2, #1             ; Increment the index
        ADDS R4, #4             ; Increment to address pointer
        B MALLOCL1              ; Go to MALLOCL1 label

MALLOCE
        MOVS R0, #0             ; Move 0 to r0 if no space is found

ENDM
        ORRS R5,R5,R6           ; Or the values in shifting number and
                                  current word
        STR R5, [R4]            ; Store it to current word location on
                                  allocation table
        BX LR                   ; Branch with link register

        ENDFUNC
```

Listing 7: Malloc() Function Codes

2.8 Free(address)

In this function we were expected to clear the corresponding bit of the freshly deleted value from the allocation table, using the address parameter passed from Remove(value) to here. We use a similar algorithm to that in Malloc() in a reverse engineered pattern to calculate the exact bit we need to reset in our allocation table. We look through the table treating it line by line. When we get our corresponding bit we use AND operation to only change the bit we need to change and not do collateral damage to the rest of our line. Assembly code part for the function is given as in Listing 8.

```
Free        FUNCTION
            LDR R1, =DATA_MEM               ; Address of the DATA start
            MOVS R2 ,#1                     ; Iterator that travels inside the line
            MOVS R3 ,#0                     ; Constant 0
            MOVS R4 ,#0                     ; Iterator that hold the line count

FREEL1
            CMP R1, R0                      ; Check if we found the bit
            BEQ FREEC2                      ; if we found our bit, break the loop
            ADDS R1 ,#8                     ; Iterate DATA_MEM
            LSLS R2 , R2 ,#1                ; LSLS to iterate the index inside our line
            CMP R2,R3                       ; If index overflows, call next iteration
            BEQ FREEF1                      ; Increase the line counter

FREEF1
            ADDS R4,#4                      ; Increase the line counter
            MOVS R2 ,#1                     ; Reset the index traveling inline
            B FREEL1                        ; Go back to the loop

FREEC2
            LDR R5, =AT_MEM                 ; Address of the Allocation table start
            LDR R6, [R5,R4]                 ; Write the number to r1
            MVNS R2, R2                     ; Invert R2
            ANDS R6,R6,R2                   ; Only change the necessary bit
            STR R6, [R5,R4]                 ; Store the new A.T. line
            BX LR                           ; Go back with Link Register

            ENDFUNC
```

Listing 8: Free(address) Function Codes

2.9 Insert(value)

In this function, we were to use the parameter which was the value that we were to insert. We store the first element and it's address, then check if our list is empty. Empty list means we branch to INSERT_TO_START. This label handles the first element placement. As all other labels, this has a Malloc() call and then an error check to control if Malloc() returned an error or not. Otherwise, we store our address to the head and write our value inside the head and exit the insert() call.

However, If the list is not empty we check if the value we checked is equal to the first value stored. If it is, we give duplicate error in the label INSERT_EQ_ERROR, if not we go on to check if it is a lower value, which means new value must be the new head. We insert to head in INSERT_TO_HEAD. INSERT_EQ_ERROR just returns the operation error value, but INSERT_TO_HEAD calls Malloc() and update value and addresses on Malloc success. If we have not branched in any of the conditions listed above we find ourselves in the INSERTL1 which stands for loop 1.

INSERTL1 is the main loop in which we do most of our work in. This loop has conditions to check if we are inserting to the end(INSERT_TO_END), duplicate error(INSERT_EQ_ERROR) and the main ending sequence condition which is INSERT_F. We check necessary conditions to decide if we need to branch out in these labels or iterate even further. If no iterations left, we take the decided value and branch into INSERT_F, which brings us to:

INSERT_F is the ending sequence. Most of the operations tend to end here. We protect the integrity of linked list general structure by connecting necessary addresses to the incoming and outgoing address of our newly inserted value. Assembly code part for the function is given as in Listing 9.

```
Insert   FUNCTION
         LDR R2, =FIRST_ELEMENT      ; Get address of first element global var
         LDR R4, [R2,#0]             ; Get address of the first element in LS
         CMP R4 , #0                 ; Check if the list is empty
         BEQ INSERT_TO_START         ; If so add to start
         MOV R2, R4                  ; Move the address of first element in LS
         LDR R4, [R2,#0]             ; Load the value in first element in the
                                       linked list to R4
         CMP R0 , R4                 ; Check if the value is equal to the first
                                       value in the ls
         BEQ INSERT_EQ_ERROR         ; If so go to INSERT_EQ_ERROR label
         CMP R0 , R4                 ; Check if the value is lower than the
                                       first value in the ls
         BLO INSERT_TO_HEAD          ; If so new value will be the head so go to
                                       INSERT_TO_HEAD LABEL

INSERTL1
         MOV R3, R2                  ; Iter tail
         LDR R2, [R2,#4]             ; Move the iter
         CMP R2 , #0                 ; Check if the link list end has been
                                       reached
         BEQ INSERT_TO_END           ; If so go to INSERT_TO_END label
         LDR R4, [R2,#0]             ; Get the iter value and save it to R4
         CMP R0, R4                  ; Check if the iter value is equal to the
                                       value to be inserted
         BEQ INSERT_EQ_ERROR         ; If so go to INSERT_EQ_ERROR label
         CMP R0, R4                  ; Check if the iter value is greater than
    the
                                       value to be inserted
         BLO INSERT_F                ; If so go to INSERT_F label
```

```
        B INSERTL1                  ; Else go to INSERTL1

INSERT_TO_START
        MOVS R1, R0                 ; Move to be inserted val to R1
        PUSH {LR,R1}                ; PUSH R1 AND LR
        BL Malloc                   ; Call malloc
        CMP R0, #0                  ; Check if malloc gave an error
        BEQ INSERT_SPACE_ERROR      ; If so branch to INSERT_SPACE_ERROR
        POP {R1}                    ; POP R1
        LDR R2, =FIRST_ELEMENT      ; Get the head
        STR R0, [R2,#0]             ; Store the address to head
        STR R1, [R0]                ; Write the value to the address
        MOVS R0, #0                 ; Move 0 to R0 as a successful task code
        POP {PC}                    ; POP LR to PC

INSERT_TO_HEAD
        MOVS R1, R0                 ; Move to be inserted val to R1
        PUSH {LR,R1,R2}             ; PUSH R1, R2 AND LR
        BL Malloc                   ; Call malloc
        POP {R1,R2}                 ; POP R1, R2
        CMP R0, #0                  ; Check if Malloc gave and error
        BEQ INSERT_SPACE_ERROR      ; If so branch to INSERT_SPACE_ERROR
        LDR R7, =FIRST_ELEMENT      ; Get the head
        STR R0, [R7,#0]             ; Store the address to head
        STR R1, [R0]                ; Write the value to the address
        STR R2, [R0,#4]             ; Update the address to previous head value

        MOVS R0, #0                 ; Move 0 to R0 as a succesfull task code
        POP {PC}                    ; POP LR to pc

INSERT_TO_END
        MOVS R1, R0                 ; Move to be inserted val to R1
        PUSH {LR, R1,R3}            ; PUSH R1, R3 AND LR
        BL Malloc                   ; Call Malloc
        POP {R1,R3}                 ; POP R1, R3
        CMP R0, #0                  ; Check if malloc gave and error
        BEQ INSERT_SPACE_ERROR      ; If so branch to INSERT_SPACE_ERROR
        STR R0, [R3,#4]             ; Store the address to previous values
                                      address part
        STR R1, [R0]                ; Write the value to the address
        MOVS R0, #0                 ; Move 0 to R0 as a succesfull task code
        POP {PC}                    ; POP LR to pc

INSERT_F
        MOVS R1, R0                 ; Move to be inserted val to R1
```

```
        PUSH {LR,R1,R2,R3}        ; PUSH R1, R2, R3 AND LR
        BL Malloc                 ; Call Malloc
        POP {R1,R2,R3}            ; POP R1, R2, R3
        CMP R0, #0                ; Check if malloc gave an error
        BEQ INSERT_SPACE_ERROR    ; If so branch to INSERT_SPACE_ERROR
        STR R0, [R3,#4]           ; Store the address to previous values
                                    address part
        STR R1, [R0]              ; Write the value to the address
        STR R2, [R0,#4]           ; Update the address to previous head value

        MOVS R0, #0               ; Move 0 to r0 as a succesfull task code
        POP {PC}                  ; POP LR to pc

INSERT_SPACE_ERROR
        MOVS R0, #1               ; Move 1 ti r0 as a There is no allocable
                                    area error code
        BX LR                     ; Branch with LR (Theres no additional
                                    function call hence LR is still at same
                                    position)
INSERT_EQ_ERROR
        MOVS R0, #2               ; Move 2 to r0 as a Same data is in the
                                    array error code
        BX LR                     ; Branch with LR (Theres no additional
                                    function call hence LR is still at same
                                    position)
```

Listing 9: Insert(value) Function Codes

2.10 Remove(value)

The remove function works similar to insert in that it also takes the value ot be removed as a parameter, and traverses the list to find it's location. We used the fact that the elements in the list are in ascending order to find the desired values position in the list. If the value is equal to the first element in the list then it is the head of the list and we send it to a subroutine, as removing the head of the list requires different operations to removing an element from the middle of the list. If the value to be removed is smaller than the first element of the list, then we can be sure it doesn't exist in our list and we can pass it to the REMOVE_NOT_FOUND label for error handling.

In case an error is not found and the value to be removed is not the head of the list, the code moves on to the REMOVEL1 label. In this label we use an iterator stored in the R2 register to traverse the linked list to find the desired value to be removed. If the value at the position of the iterator is not equal to the target value, the iterator is incremented and the code loops again. Once the value is found the code branches to the REMOVE_F label where the node can be removed from the list. If the value isn't found and the iterator reaches the last element in the list, we move to the REMOVE_NOT_FOUND label for error handling.

In the case of a successful removal, the code moves to the REMOVE_F label once it finds the target value in the list. In this label we push our values and free the memory of the node to be deleted, then pop to restore our values. We find the address of the deleted node and use temporary registers to update the address part in the tail to the next node to keep the linked lists structure. Assembly code part for the function is given as in Listing 10.

```
Remove   FUNCTION
         LDR R2, =FIRST_ELEMENT   ; Get address of first element global var
         LDR R4, [R2,#0]          ; Get address of the first element in LS
         CMP R4 , #0              ; check if the list is empty
         BEQ REMOVE_EMPTY_ERR     ; If so branch to error part

         MOV R2, R4               ; Move address of the first element in LS
         LDR R4, [R2,#0]          ; Load the value in first element in the
                                    linked list to R4

         CMP R0 , R4              ; Check if the value is equal to the first
                                    value in the LS
         BEQ REMOVE_HEAD          ; If so go to REMOVE_HEAD label
         CMP R0 , R4              ; Check if the value is lower than first
                                    value in the LS
         BLO REMOVE_NOT_FOUND     ; If so go to REMOVE_NOT_FOUND label

REMOVEL1
         MOV R3, R2               ; ITER TAIL
         LDR R2, [R2,#4]          ; Move the iter
         CMP R2 , #0              ; Check if the linked list end has been
                                    reached
         BEQ REMOVE_NOT_FOUND     ; If so go to REMOVE_NOT_FOUND label
         LDR R4, [R2,#0]          ; Get the value in the iter and save to R4
         CMP R0, R4               ; Check if the iters value is equal to the
                                    to be removed value
         BEQ REMOVE_F             ; If so go to REMOVE_F LABEL
         CMP R0, R4               ; Check if the iters value is greater than
                                    the to be removed value
         BLO REMOVE_NOT_FOUND     ; If so go to REMOVE_NOT_FOUND label
         B REMOVEL1               ; Else go to REMOVEL1

REMOVE_HEAD
```

```
        MOVS R1,R0                  ; Move R0s value to R1 as a failsafe
        MOVS R0,R2                  ; Move address to R0
        PUSH {LR,R1,R2}             ; Push to preserve values
        BL Free                     ; Free memory
        POP {R1,R2}                 ; Restore values
        LDR R3, =FIRST_ELEMENT      ; Load the first element global VARIABLES
                                      address to R3
        LDR R4, [R3]                ; Load the R3 value to R4 so that it points
                                      to first element
        LDR R4, [R4,#4]             ; Load the address in the first element to
                                      R4 since it will be the next head
        STR R4, [R3,#0]             ; Store the address to R3
        MOVS R3, #0                 ; Move 0 to R3, R3s previous job is
                                      finished
        STR R3, [R2,#0]             ; Set the value in the address to 0
        STR R3, [R2,#4]             ; Set the address in the address to 0
        MOVS R0, #0                 ; Move 0 to R0 as a successful task code
        POP {PC}                    ; Pop LR to pc

REMOVE_F
        MOVS R1,R0                  ; Move R0s value to R1 as a failsafe
        MOVS R0,R2                  ; Move address to R0
        PUSH {LR,R1,R2,R3}          ; Push to preserve values
        BL Free                     ; Free memory
        POP {R1,R2,R3}              ; Restore values
        LDR R4, [R2,#4]             ; Get the address of the to be deleted node
                                      and save it to R4
        STR R4, [R3,#4]             ; Update address part in tail to next node
        MOVS R3, #0                 ; Move 0 to R3, R3s previous job is
                                      finished
        STR R3, [R2,#0]             ; Set the value in the address to 0
        STR R3, [R2,#4]             ; Set the address in the address to 0
        MOVS R0, #0                 ; Move 0 to R0 as a successful task code
        POP {PC}                    ; Pop LR to pc

REMOVE_EMPTY_ERR
        MOVS R0, #3                 ; Move 3 to r0 as a "the linked list is
                                      empty" error code
        BX LR                       ; Branch with LR (Theres no additional
                                      function call hence LR is still at same
                                      position)

REMOVE_NOT_FOUND
        MOVS R0, #4                 ; Move 4 ti r0 as a "the element is not
                                      found" error code
```

```
        BX LR                        ; Branch with LR (Theres no additional
                                       function call hence LR is still at same
                                       position)
        ENDFUNC
```

Listing 10: Remove(value) Function Codes

2.11 LinkedList2Arr()

In this function, we simply implement the linked list structure to turn it into an array. We use FIRST_ELEMENT and use the address attached to it, until we arrive at the last element pointing to NULL. This algorithm has been implemented in other classes before, we just developed an Assembly version.

```
LinkedList2Arr  FUNCTION
        LDR R3, =ARRAY_MEM           ; Adress of the Array mem start
        LDR R4, =ARRAY_SIZE          ; Size of the Array mem
        MOVS R5, #0                  ; r5 will be used as an index
        MOVS R6, #0                  ; r6 is the constant 0 value

LOOP3
        CMP R5, R4                   ; Compare index with size
        BEQ CONT3                    ; If they are equal go to CONT3
        STR R6, [R3, R5]             ; Store the constant zero value to
                                       ARRAY_MEM with index offset
        ADDS R5 , R5, #4             ; Increment index by int size
        B LOOP3                      ; Branch to LOOP3 label

CONT3
        MOVS R5, #0                  ; r5 will be used as an index
        LDR R0, =FIRST_ELEMENT       ; Get the address of first element
                                       global var
        LDR R1, [R0,#0]              ; Get the address of the first
                                       element in ls
        CMP R1 , #0                  ; Check if the list is empty
        BEQ LL2A_EMPTY_ERR           ; If so branch to error part
        MOV R0, R1                   ; Move the address of the first
                                       element in ls
        LDR R1, [R0,#0]              ; Load the value in first element
                                       in the linked list to r1

LL2A_LOOP
        CMP R0, #0                   ; Check if the end of the has
                                       been reached
        BEQ LL2A_END                 ; Branch to LL2A_END label
```

```
        LDR R1, [R0,#0]          ; Load the value in of the current
                                   element to r1
        STR R1, [R3, R5]         ; Save the value to array mem
        ADDS R5 , R5, #4         ; Increment array mem index
        LDR R0, [R0,#4]          ; Move the iter
        B LL2A_LOOP              ; Branch to LL2A_LOOP label

LL2A_END
        MOVS R0, #0              ; Gave the error code 0 for
                                   successful operation
        BX LR                    ; Branch with link register

LL2A_EMPTY_ERR
        MOVS R0, #5              ; Gave the corresponding error code
        BX LR                    ; Branch with link register

        ENDFUNC
```

Listing 11: LinkedList2Arr() Function Codes

2.12 WriteErrorLog(Index, ErrorCode, Operation, Data)

In this function, we are expected to stores the error log of the input dataset operations. Firstly, we combine Index, ErrorCode and Operation parameters to 32 bit variable. To do this, we do logic shift left by 16 to R0 register which kept Index variable. So the first 16 bits of our 32 bit variable is the index variable. Then we shift Error code, which is R1 register, to left 24 bit so that clears first 24 bit and we shift right 16 bit to gets to its place. After, we shift Opcode, which is R2 register, to left 24 bit so that clears first 24 bit and we shift right 24 bit to gets to its place. Finally we use ORRS operations to combine R1 and R2 values to R0 register. Then, we load index error log address to R1 register and the index error log value to R2 register. PUSH the used register before call the GetNow function. GetNow function return current time with R0 register. We load R0 to R4 and POP used registers back. After that, we load LOG_MEM's starting address to R5 and store the combined Index, ErrorCode and Opcode to LOG_MEM. We increment index error log value by 4 so that it will point to next position and we store the data to LOG_MEM. Again we increment index error log value and store the time stamp like before. Then, we increment index error log value by 4 and store the new index error log value to its position in memory. POP the value LR which was saved in stack to PC so that program returns to function call. Assembly code part for the function is given as in Listing 11.

```
WriteErrorLog FUNCTION
```

```
; R0 INDEX DS        16 BIT ,
; R1 ERROR CODE      8
; R2 OPCODE          8
; R3 DATA            32
; ALSO ADD TIME      32

LSLS R0,R0, #16            ; Shift R0 to left for 16 bits
LSLS R1,R1, #24            ; Shift error code to left for 24 bits so
                             that it clears first 24 bits
LSRS R1,R1, #16            ; Shift error code to right for 16 bits so
                             that it gets to its place
LSLS R2,R2, #24            ; Shift OPCODE to left for 24 bits so that
                             it clears first 24 bits
LSRS R2,R2, #24            ; Shift OPCODE to right for 24 bits so that
                             it gets to its place

ORRS R0,R0, R1             ; Combine index DS and error code
ORRS R0,R0, R2             ; Join OPCODE to them

LDR R1, =INDEX_ERROR_LOG; Get the INDEX_ERROR_LOG address
LDR R2, [R1]               ; Get the value in index error log

PUSH {LR,R0,R1,R2,R3}      ; Push the used registers to save them
BL GetNow                  ; Call the GETNOW function, it will return
                             current time in R0
MOVS R4, R0                ; Get the current time value, save it to R4
POP {R0,R1,R2,R3}          ; POP the used registers back
LDR R5, =LOG_MEM           ; Get the LOG_MEMs starting address
STR R0, [R5,R2]            ; Store the combined index DS, ERRORCODE
                             and OPCODE to LOG_MEM
ADDS R2,R2,#4              ; Increment INDEX_ERROR_LOGS value by 4
                             bytes to point to next position
STR R3, [R5,R2]            ; Store the data to LOG_MEM
ADDS R2,R2,#4              ; Increment INDEX_ERROR_LOGS value by 4
                             bytes to points to next position
STR R4, [R5,R2]            ; Store the time stamp TO LOG_MEM
ADDS R2,R2,#4              ; Increment INDEX_ERROR_LOGS value by 4
                             bytes to points to next position
STR R2, [R1]               ; Store the INDEX ERROR LOG value to its
                             position in memory
POP {PC}                   ; Pop the value LR saved in the stack to PC
                             to return to function call
ENDFUNC
```

Listing 12: WriteErrorLog() Function Codes

2.13 GetNow()

In GetNow() function, we first get our unique group value which is 602 microseconds. We proceed by getting our total tick count so far. If we add 1 to total tick count and multiply it by 602 microseconds we get the total time at the end of our current tick. If we then subtract the value inside our reload register from this microsecond value we can achieve the current time passed in microseconds. Then we pass the value in R0 then return R0 as we were asked. Assembly code part for the function is given as in Listing 12.

```
GetNow  FUNCTION
        LDR R3, =POTSTTI            ; Get the 602 ms value
        LDR R4, =TICK_COUNT         ; Get the tick count
        LDR R4, [R4]
        ADDS R4,R4, #1              ; Increase tick count value by 1
        MULS R3, R4, R3             ; Multiply tick count+1 with 602 ms
        LDR R4, =0xE000E018         ; Get the value in reload register
        LDR R4, [R4]
        LSRS R4, R4, #5             ; Divide it to 32 tihs value is the the
                                      amount of ms to next tick
        SUBS R3, R3, R4             ; Subtract amount of next ms from the
                                      tick count+1 * 602
        MOVS R0,R3                  ; Move R3 to R0 since R0 is the return
                                      value
        BX LR
        ENDFUNC
```

Listing 13: GetNow() Function Codes

3 RESULTS

After we are done with the implementation of the assembly code, we tested our program with the inputs which are given at the beginning of the template code file. Results are as in the figures below.

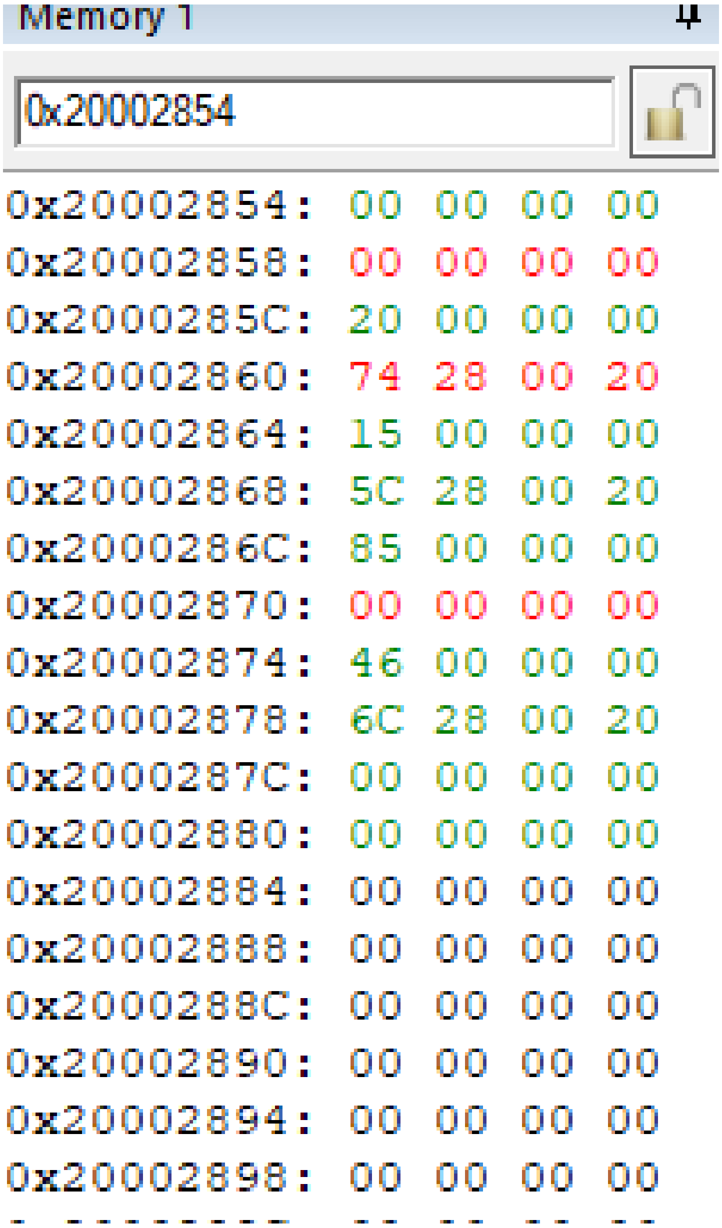

Figure 2: DATA_MEM area from memory

Figure 2 shows us our linked list. First lines shows data value, second lines shows next address value. Our linked list have to start 15 value according to input given. 15 value in 0x20002864 address according to figure 2. When we look next address, it shows 20 value's address and it goes on. Our last data is 85 and when we look next address, it is null value. Therefore, our data and address values are correct.

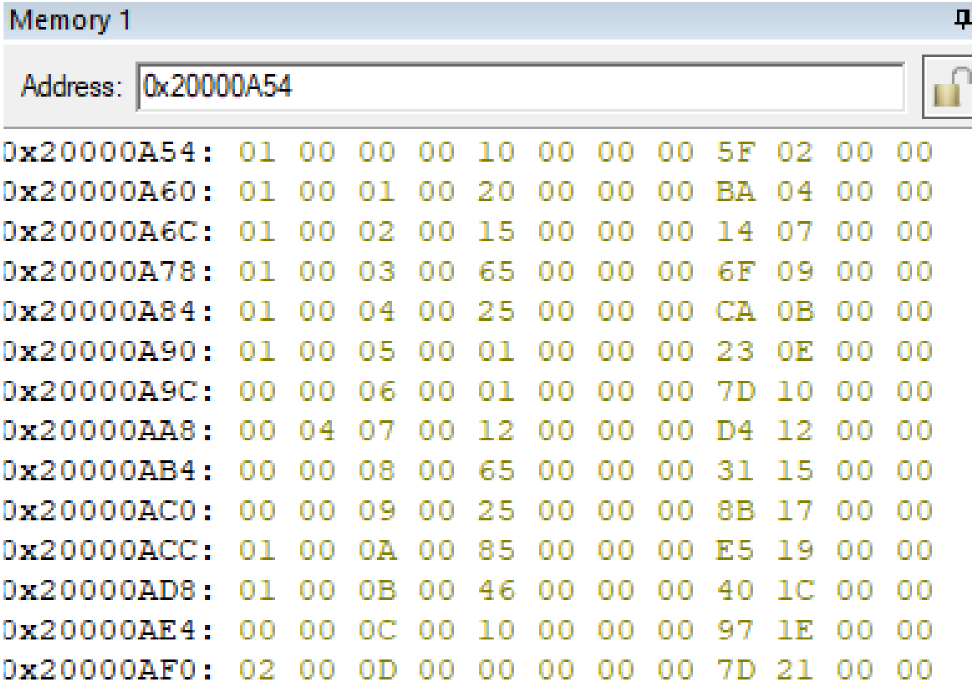

Figure 3: LOG_MEM area from memory

Figure 3 shows us our error log area in memory according to inputs. First operation is insert the 0x10 element to linked list. When we look at first line error log area, first 4 values show index of input dataset which is 1, after 2 values show error code of the operation which is 0, after 2 values show operation of the current input data which is 0, after 8 values show current data to which is 10 and finally last 8 values show current systick time working time in milliseconds. All of the error logs are correct according to inputs.

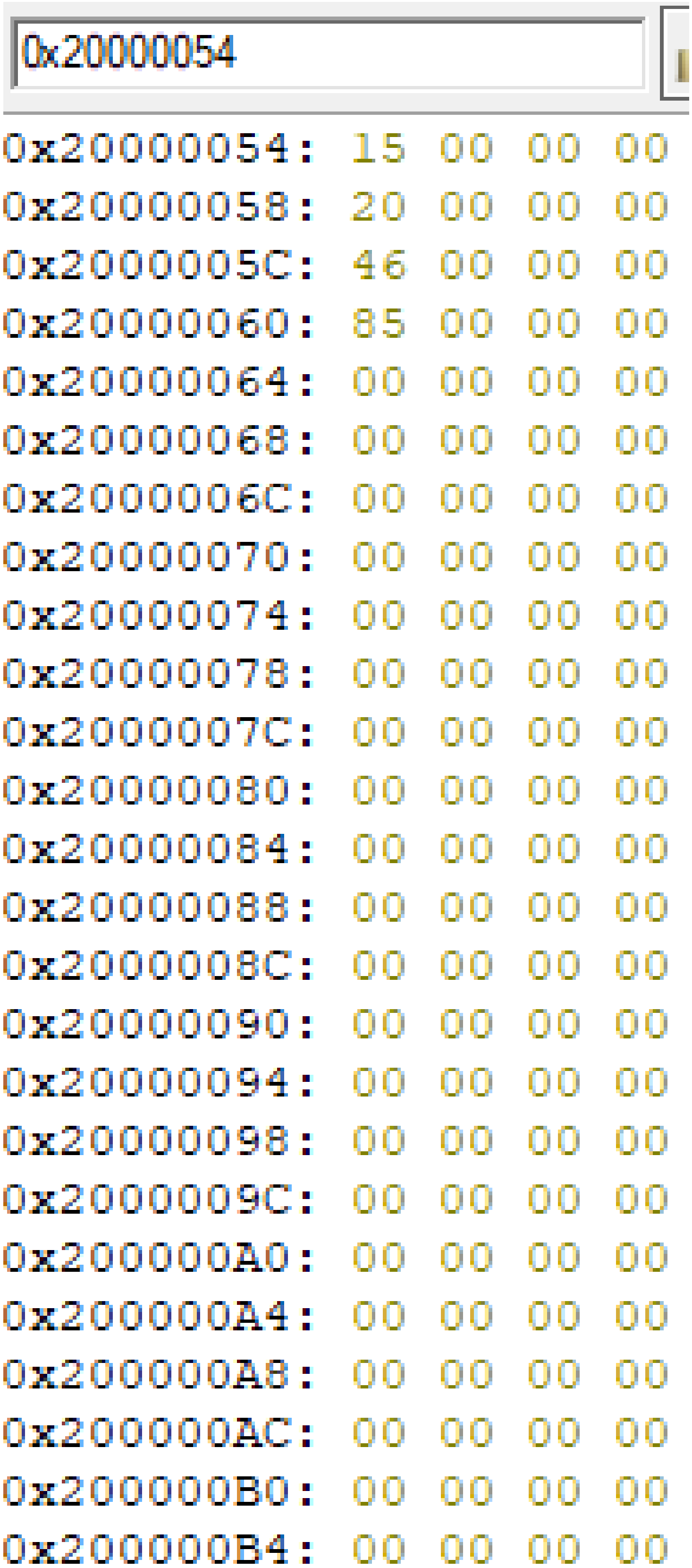

Figure 4: ARR_MEM area from memory

Figure 4 shows the memory area which is allocated for the elements of the linked list after the list is converted into array. Using LinkedList2Arr() function, we convert our linked list to array and store the numbers in consecutive memory block which is array. Start address of this memory block is named as ARRAY_MEM and start in the address of 0x20000054. As a result of the figure, we see that our list had values of 0x15, 0x20, 0x46 and 0x85. Now, these values are stored in the ARRAY_MEM block of the memory.

4 DISCUSSION

In SysTick_Init() function, we simply initialized the registers related to system timer which are SYST_CSR, SYST_RVR and SYST_CVR. Apart from that, we set the program status accordingly. Since those operations were already covered during the recitation of the lecture, we had background information about them. So, there was nothing extra in this function and we simply implemented it.

In SysTick_Handler() we read the input data and operation that goes with it. OP0 is Remove operation, OP1 is Insert and OP2 is the end of our program essentially. OP2 ends our program by calling SysTick_Stop to set zeroes across SysTick registers and loading value 2 into PROGRAM_STATUS to mark the finish.

In SysTick_Stop() function, we simply reset some registers related to system timer that we initialized in SysTick_Init(). Implementing this function was even easier compared to SysTick_Init(). Again, there was not any challenging parts in this function and we simply implemented it.

In Clear_Alloc() function, we can traverse allocation table line by line and set values to 0. We can easily implement that with one loop.

Clear_ErrorLogs() function very similar to Clear_Alloc(), we can easily traverse all error logs elements and set values to 0 with one loop.

In Init_GlobVars() function, we can easily get global values in order and initialize 0.

In Malloc() function firstly, we had difficulty understanding how many rows and columns the allocation table consist of. Later on, we realize that table has 32 column because of the word is 32 bits and it has 20 row because of the number of word depend to NUMBER_OF_AT constant number. After that, we couldn't figure out how can one bit will point to node. We realized that the allocation table and the linked list are working synchronously, and when we move 1 bit in the allocation table, we need to move on to the next node in the linked list. After the getting over problems we can easily implement Malloc() function.

In Free(address) function, since we learnrd the allocation table structure in the Malloc() function, it was quite easy to reset the bit holding the required address value.

In Insert(value) function, we already knew how we should add a value to a sorted linked list in C/C++ form. We had a little difficulty applying this situation to assembly code, but since we knew the general algorithm, we could do it without any errors.

In Remove(value) function we can easily implement with help to Insert function. That are very similar functions.

In LinkedList2Arr() function, we were able to easily write the elements to the array, starting at the head of our linked list and traverse all nodes. That is to easy for us because we already knew the algorithm which traverses all element in the linked list.

In WriteErrorLog() function, we can easily place the parameters given in the function with shifting operations and combined them with 'or' operation. With the GetNow function we get current time. Finally we can store Index, ErrorCode and other variables with STR operations easily.

In GetNow() function, we use our group value 602 microseconds. We decided easiest way would be multiplying this value and adding the last unfinished tick's countdown value to find an accurate answer. So we are doing just that.

5 CONCLUSION

As a result of the final term project, we were expected to implement sorted set linked list structure using assembly language, and we believe that we successfully implemented all the functions as desired with the well communication and task distribution between team members.

While doing the project, we faced some difficulties about the logic behind the some implementations such as SysTick, Malloc() and Insert(). We handled the problems of SysTick and Insert() with the well use of sources. Especially understanding the allocation table in the Malloc() was really hard for us since it did not mean anything to us at the first step. Again, with the spent of hours and brainstorming together, we figured out and implemented the assembly code successfully.

What we have learnt during the project is listed as below;

- How to use Keil uVision5 with Arm Cortex M0+ assembly code.
- What is System Tick Timer and how to use it.
- How to debug assembly code by using Keil uVision5 debugger tools.
- How to pass parameters to function and how to return a value from a function.
- How to implement non-nested and nested loops.

- Basic and important Arm Cortex M0+ assembly instructions such as DCD, EQU, ALIGN, BL, LDR, CMP, BNE, B, ADDS, STR, BEQ, MOVS, PUSH, POP, BX, LSLS, ANDS, ORRS, MVNS, LSRS and MULS.
- How memory structure is organized and how to follow it.
- Seeing how one of the most important functions which is malloc works.

To sum up, we had already have rather strong background information about assembly language because of the quizzes, homeworks and recitations from the lecture. With this final term project, we really strengthened our information about memory structures and assembly language. Thanks to the final project, we carried our general information about how computers works one step further.

REFERENCES

[1] Overleaf documentation https://tr.overleaf.com/learn.

[2] Keil uvision5 https://www2.keil.com/mdk5/uvision/.