Department of Computer Science and Engineering

University of Dhaka

2nd Year 1st Semester 2025

CSE 2106: Microprocessor and Assembly Language Lab

Assignment 2

PayrollSys-32

ARM-Based Employee Payroll & Salary Processing System

Submitted by

Group 17

Shashwata Nandi (JN-20)

Swarlok Samadder (JN-36)

Mohammad Mahmudul Kabir Fahmid (EK-40)

Submitted to

Dr. Mosarrat Jahan

Associate Professor, Dept. of CSE, DU

&

Palash Roy

Lecturer, Dept. of CSE, DU

Solution Technique and Module-wise Output Explanation

Introduction

PayrollSys-32 is an ARM Assembly based payroll processing system designed for a Cortex-M(specifically, Cortex-M4) microprocessor. The system processes employee records, attendance, overtime, allowances, tax, deductions, bonuses, and generates final pay-slips. All modules are implemented as separate subroutines and work together through controlled memory access.

The system stores employee data in RAM, performs calculations step by step, and finally generates a readable pay-slip using a UART debug buffer.

The code has been implemented in three sections in `__main` function:

1. Data Initialization
    a. Module 0
    b. Module 1

   These modules initialize the employee data.
2. Process Loop
    a. Module 2
    b. Module 3
    c. Module 4
    d. Module 5
    e. Module 6
    f. Module 7
    g. Module 10

   These modules implement various aspects (leave management, overtime calculation, tax computation, etc.) for each of the employees that requires processing.
3. After Loop
    a. Module 8
    b. Module 9
    c. Module 11

   These modules work with the already created employee structure (sort by salary, summary creation and pay-slip printing).

Module 0 – Test Data Initialization

Technique Used

- This module prepares all required input data before payroll processing begins.
- Attendance data is stored in ROM and copied into RAM.
- Each employee gets a 31-day attendance log.
- Overtime hours and performance scores are also copied from ROM to RAM.
- Loops are used to copy data byte-by-byte using LDRB and STRB.

Memory Usage

- ■ Attendance logs → 0x20001000 + index × 0x100
- ■ OT hours → 0x20002000
- ■ Performance scores → 0x20006000

Output

RAM now contains:

- ➢ Valid attendance logs
- ➢ OT hours for each employee
- ➢ Performance scores for each employee

Code Execution Flow

```
PUSH {R4-R7,LR} saves registers.

R0 = ATT_TABLE_ROM, R1 = ATT_LOG_ADDR, R2 = 0 (employee index).

Attendance copy (5 employees)

Loop runs while R2 < NUM_EMPS.

R3 = R2 << 8 → multiply employee index by 256 (0x100).

R4 = R1 + R3 → destination block: 0x20001000 + index*0x100.

R5 = 32 → bytes per employee.

R6 = R2 << 5 → multiply index by 32.

R7 = R0 + R6 → source block in ROM.

Copy 32 bytes

Loop while R5 != 0:

LDRB R3, [R7], #1 reads a byte and moves source pointer.
```

```
STRB R3, [R4], #1 stores it and moves destination pointer.

R5-- until done.

R2++ move to next employee.

OT copy (5 bytes)

R0 = OT_TABLE_ROM, R1 = OT_LOG_ADDR, R2 = NUM_EMPS

Loop copies one byte each into 0x20002000 + i.

Score copy (5 bytes)

R0 = SCORE_TABLE_ROM, R1 = SCORE_ADDR, R2 = NUM_EMPS

Loop copies one byte each into 0x20006000 + i.

POP {R4-R7,PC} returns.
```

*Important Clarification: We initially put the data in ROM because it represents fixed test inputs (attendance patterns, OT hours, scores) that should not change at program start. Storing them in ROM saves RAM space, and ensures the system starts with known, reliable data every time. Then, we use ROM-to-RAM copying in Module 0 because test data (attendance, OT hours, and scores) is stored in ROM as constants, but all processing in the payroll system is designed to work on RAM-based data. Module 0 copies this data into RAM so that other modules can read, modify, and process it dynamically.

Module 1 – Employee Record Initialization

Technique Used

- This module creates the employee database.
- A fixed structure is designed for each employee.
- Each field (ID, name pointer, salary, grade, department, etc.) is written at a specific offset. The offset is as given below:

Field	Offset	Size
Employee ID	0x00	4 B
Name pointer	0x04	4 B

Base salary	0x08	4 B
Grade	0x0C	1 B
Dept	0x0D	1 B
Bank account	0x10	4 B
Attendance pointer	0x14	4 B
Allow table ptr	0x18	4 B
Present days	0x1C	1 B
Flags	0x1D	1 B
Deduction	0x20	4 B
OT pay	0x24	4 B
Allowance	0x28	4 B
Bonus	0x2C	4 B
Tax	0x30	4 B
Net salary	0x34	4 B

- Five employee records are stored sequentially in RAM.
- All numeric fields are aligned correctly (bytes and words).

<u>Memory Usage</u>

- Employee records start at 0x20000000
- Each employee occupies 64 (Hex: 0x00000040) bytes

<u>Output</u>

- A complete employee table exists in RAM.
- All fields are initialized to known values.
- Name pointers correctly point to strings in ROM.

<u>Code Execution Flow</u>

```
R4 = EMP_BASE_ADDR → points to first employee struct at 0x20000000.
```

```
For each employee:

Store ID: STR R0, [R4, #OFF_ID]

Store name pointer: STR R0, [R4, #OFF_NAMEPTR]

Store base salary: STR R0, [R4, #OFF_BASE]

Store grade byte: STRB R0, [R4, #OFF_GRADE]

Store dept byte: STRB R0, [R4, #OFF_DEPT]

Store bank: STR R0, [R4, #OFF_BANK]

Store attendance pointer: STR R0, [R4, #OFF_ATTPTR]

Store allowance pointer: STR R0, [R4, #OFF_ALLOWPTR]

Clears computed fields to 0:

Present, flags, deduction, OT, allowance, bonus, tax, net.
```

*Important Clarification: LTORG instruction was used in different places of our code when we faced error while merging the modules. This was recommended in the Build Output of Keil. LTORG is an instruction used to force the assembler to place literal constants (used with LDR = value) close to the code, so they remain within valid PC-relative range and load correctly at runtime.

Module 2 – Attendance Data Loader

Technique Used

- This module counts the number of present days in a month.
- Attendance data is read from RAM (31 bytes).
- Each byte represents one day (1 = present, 0 = absent).
- A loop adds all present days.
- The result is stored in the employee structure.

Memory Usage

- Attendance loads from memory locations 0x20001000, 0x20001100, 0x20001200, 0x20001300 & 0x20001400.

Output

- ➢ `present_days` field is updated for each employee.

Code Execution Flow

```
R4 = ATT_LOG_ADDR

R2 = R0 and R2 << 8 → index × 256 (0x100)

R4 = R4 + R2
Now R4 points to this employee's attendance block.

R5 = 0 present counter

R6 = 0 day index


Loop 31 days

LDRB R7, [R4, R6] reads one attendance byte (0/1)

R5 = R5 + R7 adds it

R6++ day index

Stop when R6 == 31


Store result

STRB R5, [R1, #OFF_PRESENT]
```

Module 3 – Leave Management & Deduction Logic

Technique Used

- This module checks if an employee has too many absences.
- Minimum required present days = 22.
- If present days are less than 22:
    - Leave deficit is calculated using the formula: *leave_deficit = 22 − present_days.*
- Deduction is computed using the formula: *leave_deduction = leave_deficit × 300.*
- If deficit exceeds 5 days, a warning flag is set.

Output

- ➢ Deduction amount stored in employee record.
- ➢ Leave-deficit flag set when required.

Code Execution Flow

```
Load flags: LDRB R4, [R5, #OFF_FLAGS]

Clear leave deficit flag:

BIC R4, R4, #FLAG_LDEFICIT

STRB R4, [R5, #OFF_FLAGS]


Load present days: LDRB R0, [R5, #OFF_PRESENT]

Compare with minimum 22:

CMP R0, #22

If present >= 22, branch to NoDed3 (no deduction case)


If present < 22

R2 = 22 - present (leave deficit)

R3 = 300

R3 = R2 * R3 → deduction

Store deduction: STR R3, [R5, #OFF_DED]


Set LDEFICIT flag if deficit > 5

CMP R2, #5

If deficit > 5:

read flags again, OR with FLAG_LDEFICIT

store updated flags


If present ≥ 22

deduction = 0 stored in OFF_DED
```

Module 4 – Overtime Calculation Module

Technique Used

- This module calculates overtime payment using a lookup table.
- OT hours are read from RAM.
- Job grade is used as an index into an OT rate table.
- Rate values:
    - Grade A → 250
    - Grade B → 200
    - Grade C → 150
- OT pay is calculated by multiplication.
- Formula used to calculate: $ot\_pay = ot\_hours \times rate$.

Memory Usage

- OT hours stored from address 0x20002000.

Output

- OT pay stored in employee record.

Code Execution Flow

```
R6 = OT_LOG_ADDR

R6 = R6 + R4 → points to OT byte for this employee

LDRB R0, [R6] → R0 = OT hours

LDRB R1, [R5, #OFF_GRADE] → grade (0/1/2)


Grade safety

If grade > 2, force grade = 2 (C)


Lookup OT rate

R2 = OT_RateTable
```

```
LDRB R2, [R2, R1] → rate selected based on grade


Compute OT pay

R3 = R0 * R2

Store OT pay:

STR R3, [R5, #OFF_OT]
```

Module 5 – Allowance Processor

Technique Used

- This module computes allowances.
- HRA is calculated as *base_salary / 5* (20%).
- Medical allowance is fixed at 3000.
- Transport allowance depends on department:
    - IT: 5000
    - HR: 4000
    - Admin: 3500
- All allowances are added together.
- Formula used to calculate: *total_allowance = HRA + 3000 + transport.*

Output

➢ Total allowance stored in employee record.

Code Execution Flow

```
Load base salary: LDR R0, [R5, #OFF_BASE]

R1 = 5

UDIV R6, R0, R1 → HRA = base/5 (20%)

R7 = 3000 medical allowance


Transport selection by department
```

```
Load dept code: LDRB R3, [R5, #OFF_DEPT]

If dept == 0 → R4 = 5000

If dept == 1 → R4 = 4000

Else → R4 = 3500


Total allowance

R0 = R6 + R7

R0 = R0 + R4

Store:

STR R0, [R5, #OFF_ALLOW]
```

Module 6 – Tax Computation Using Slab System

Technique Used

- This module applies slab-based tax rules.
- Gross salary is calculated using the formula: gross_salary = base_salary + total_allowance + ot_pay – leave_deduction.
- Tax slabs are checked using `CMP`, `BLE`, `BHI`, and `BGE`.
- Tax is calculated as per the formula: *tax = (gross_salary × rate_value) / 100*.
- It is important to note that bonus is excluded from tax.

Output

➢ Tax amount stored in employee record.

Code Execution Flow

```
Step 1: compute gross

R0 = base

add allowance and OT:

R0 = R0 + [OFF_ALLOW]
```

```
R0 = R0 + [OFF_OT]

subtract deduction:

R0 = R0 - [OFF_DED]

Now R0 = gross salary

Step 2: choose tax rate in R4 using BLE, BHI, BGE

Compare gross with 30000:

if gross <= 30000, go to Tax0_rate (rate = 0)

Compare gross with 120000:

if gross > 120000, go to Tax15_rate (rate = 15)

Compare gross with 60001:

if gross >= 60001, go to Tax10_rate (rate = 10)

Otherwise:

go to Tax5_rate (rate = 5)


Step 3: compute tax = (gross * rate) / 100

R2 = gross * rate

R6 = 100

UDIV R7, R2, R6

R2 = R7

Store:

STR R2, [R5, #OFF_TAX]
```

Module 7 – Net Salary Calculator

Technique Used

- This module combines all payroll components.
- Net salary is calculated as per the formula: *net_salary = base_salary + allowance + ot_pay − tax − leave_deduction.*
- Overflow detection is done after each arithmetic operation.
- If overflow occurs, a warning flag is set.

Output

- ➢ Net salary stored in employee record.
- ➢ Overflow flag updated when needed.

Code Execution Flow

```
Load flags byte, clear overflow flag:

LDRB R4

BIC R4, R4, #FLAG_OVERFLOW


Compute net step-by-step in R0

R0 = base

Add allowance:

ADDS R0, R0, allow

If overflow, set overflow flag in R4


Add OT:

ADDS R0, R0, ot

If overflow, set flag


Subtract tax:

SUBS R0, R0, tax

If overflow, set flag
```

```
Subtract deduction:

SUBS R0, R0, ded

If overflow, set flag


Store results

STR R0, [R5, #OFF_NET]

STRB R4, [R5, #OFF_FLAGS]
```

Module 8 – Sorting Employees by Net Salary

Technique Used

- This module sorts employees by net salary.
- Employee table is copied to a new memory area.
- *Bubble sort* is applied.
- Employees are sorted in descending order.
- Entire structures are swapped, not just salaries.

Memory Usage

- Sorted list stored at 0x20005000

Output

- Sorted employee list by net salary.

Code Execution Flow

```
Step 1: Copy original table into sorted region

R0 = EMP_BASE_ADDR, R1 = SORT_DEST_ADDR

For each employee:

copy 16 words (64 bytes) using loop:

LDR from original
```

```
STR into sorted area


Step 2: Bubble sort in sorted region

Outer loop repeats passes

Inner loop compares adjacent employees:

load net salaries of two records:

R2 = net of current

R3 = net of next


If current < next → swap full 64 bytes (16 words)


Sorting ends with descending order by net
```

Module 9 – Department Salary Summary

Technique Used

- This module calculates department-wise salary totals.
- Employee records are scanned one by one.
- Net salary is added to the corresponding department total.
- Three totals are maintained:
    - `Total_IT`
    - `Total_HR`
    - `Total_Admin`

Output

➢ Total salary cost per department stored in RAM.

➢ Output can be viewed directly from the '**Watch**' window in Keil using the three memory variables: `Total_IT`, `Total_HR` and `Total_Admin`.

Code Execution Flow

```
R4 = EMP_BASE_ADDR, employee pointer

R6 = 0 total IT

R7 = 0 total HR

R3 = 0 total Admin

Loop over employees:

read dept byte (OFF_DEPT)

read net salary (OFF_NET)

add net salary to correct total based on dept value


Store totals into memory variables:

Total_IT, Total_HR, Total_Admin
```

Module 10 – Bonus & Performance Rating Engine

Technique Used

- This module calculates bonuses based on performance scores.
- Scores are read from RAM.
- Bonus percentage depends on score range.
- Bonus is calculated as a percentage of base salary.
- An important point to note is that bonus is stored separately and not added to net salary.

Memory Usage

- Performance scores are stored at 0x20006000.

Output

- Bonus amount stored in employee record.

Code Execution Flow

```
R6 = SCORE_ADDR + index
```

```
LDRB R0, [R6] → score

R1 = base salary

Decide percentage in R3:

if score >= 90 → 25

else if score >= 75 → 15

else if score >= 60 → 8

else → bonus stays 0


Compute bonus

R2 = base * percentage

UDIV R7, R2, #100

R2 = R7

Store:

STR R2, [R5, #OFF_BONUS]
```

Module 11 – UART Pay-Slip Generator (UART Debug Buffer, not Hardware implementation)

Technique Used

- This module prints salary information for all employees.
- Employee records are accessed one by one.
- Numeric values are converted to ASCII using recursion.
- Output is written to a debug buffer instead of hardware UART *(The required functions for hardware implementation have been written but left as comment)*.
- Warning messages are printed if flags are set.

Printed Fields

- Employee ID
- Net Salary

- Tax
- Allowance
- Bonus
- Alert messages *(if any)*
- Final Pay *(Net + Bonus)*

Output

- Complete pay-slip text stored in `TX_Buffer`.
- Output can be viewed directly from memory in Keil.

Code Execution Flow

```
Loops employee index from 0 to 4

Computes employee pointer:

R6 = EMP_BASE_ADDR + (index << 6) because 64 bytes per record


Prints strings + numbers:

Uses UART_SendString for labels

Uses PrintNumber for integers


Checks flags:

If FLAG_LDEFICIT set → prints leave alert

If FLAG_OVERFLOW set → prints overflow alert


Final pay is computed:

Final = Net + Bonus


Output is stored in TX_Buffer using UART_SendChar
```

*Important Clarification: The implementation of this module requires knowledge beyond our course syllabus, since we have to work with UART hardware implementation. But we have simplified the implementation by using RAM Buffer for debugging which allows us to view the pay-slip directly in the '**Memory**' window in Keil.

Since this project is implemented without physical UART support, a debug buffer (`TX_Buffer`) is used to simulate UART output. All characters that would normally be transmitted are written sequentially into this memory buffer, allowing the output to be verified using the Keil debugger.

The module loops through all employee records stored at `EMP_BASE_ADDR`. For each employee, the correct structure address is calculated using the employee index multiplied by the structure size (64 bytes). A separator line is printed between employees to improve readability.

For every employee, the module prints the employee ID, net salary, tax, allowance, bonus, and final pay. Text labels are stored in ROM and printed using the `UART_SendString` routine, while numeric values are converted to decimal characters using the `PrintNumber` function. Line breaks are added after each field to format the output clearly.

The module also checks payroll alert flags. If an employee has excessive leave or a salary overflow condition, corresponding warning messages are printed. The final pay is calculated by adding the bonus to the net salary and then displayed.

What happens in our code is that instead of interacting with UART hardware registers, the `UART_SendChar` function writes characters into `TX_Buffer` and advances `TX_Index`. This design preserves the complete payslip output in memory for debugging and verification.

Final System Outcome

The PayrollSys-32 system successfully:

- Initializes and manages employee records
- Processes attendance, overtime, allowances, deductions, tax, and bonus
- Detects payroll anomalies using flags
- Sorts employees by salary
- Produces department-wise salary summaries
- Generates readable pay-slips without hardware dependency

The system demonstrates modular ARM Assembly programming, structured memory usage, and full payroll computation in an embedded environment.

Expected Output for our Test Input Data

Data Section of the Submitted Code

```
;READONLY DATA(ROM)

        AREA    |.rodata|, DATA, READONLY
        ALIGN

;Employee names(for NamePtr fields)
Emp0Name              DCB "Neymar",0
Emp1Name              DCB "Suarez",0
Emp2Name              DCB "Messi",0
Emp3Name              DCB "Iniesta",0
Emp4Name              DCB "Xavi",0

;Attendance patterns

ATT_TABLE_ROM
        ;Emp 0(31 days: mostly present, 1 absence)
        DCB 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1
        DCB 0
        ;Emp 1
        DCB 1,1,1,1,1,1,1,1,0,1,1,1,1,0,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1
        DCB 0
        ;Emp 2
        DCB 1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
        DCB 0
        ;Emp 3 (more absences early)
        DCB 0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
        DCB 0
        ;Emp 4 (all present)
        DCB 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
        DCB 0

;OT rate lookup table by grade: index 0=A, 1=B, 2=C
OT_RateTable
        DCB    250,200,150

;OT hours(ROM)
OT_TABLE_ROM
        DCB     2,0,5,1,3

;Scores(ROM)
SCORE_TABLE_ROM
        DCB     92,80,70,55,88

;Strings for UART labels
Str_ID          DCB "ID: ",0
Str_Net         DCB "Net Salary: ",0
Str_Tax         DCB "Tax: ",0
```

```
Str_Allow       DCB "Allowance: ",0
Str_Bonus       DCB "Bonus: ",0
Str_Final       DCB "Final Pay: ",0
Str_Sep         DCB "------------------------------",13,10,0
Str_Alert_LD   DCB "** ALERT: Leave deficit > 5 days **",13,10,0
Str_Alert_OF   DCB "** ALERT: Salary overflow detected **",13,10,0


; READWRITE DATA

        AREA    |.data|, DATA, READWRITE
        ALIGN

Total_IT        DCD     0
Total_HR        DCD     0
Total_Admin     DCD     0

;UART debug buffer

TX_BUF_SIZE     EQU     1024

TX_Buffer       SPACE   TX_BUF_SIZE   ; raw bytes of UART output
TX_Index        DCD     0             ; current write index
```

Module 1 (Only Employee ID verification, for example)

Employee #	Employee ID	ID Address	ID Hex Value	Bytes (Little Endian)
0	1001	0x20000000	0x000003E9	E9 03 00 00
1	1002	0x20000040	0x000003EA	EA 03 00 00
2	1003	0x20000080	0x000003EB	EB 03 00 00
3	1004	0x200000C0	0x000003EC	EC 03 00 00
4	1005	0x20000100	0x000003ED	ED 03 00 00

Module 2

Employee #	Present Days	Address	Hex Value	Bytes (Little Endian)
0	30	0x2000001C	1E	1E
1	28	0x2000005C	1C	1C

2	30	0x2000009C	1E	1E
3	18	0x200000DC	12	12
4	31	0x2000011C	1F	1F

Module 3

Employee #	Present Days	Deficit	Deduction	Deduction Address	Deduction Hex Value	Flags Address	Flags Hex Value
0	30	0	0	0x20000020	0x00000000	0x2000001D	00
1	28	0	0	0x20000060	0x00000000	0x2000005D	00
2	30	0	0	0x200000A0	0x00000000	0x2000009D	00
3	18	4	1200	0x200000E0	0x000004B0	0x200000DD	00
4	31	0	0	0x20000120	0x00000000	0x2000011D	00

Module 4

Employee #	Grade	Hours	Rate	OT Pay	OT Pay Hex Value	OT Address	Bytes (Little Endian)
0	A (0)	2	250	500	0x000001F4	0x20000024	F4 01 00 00
1	B (1)	0	200	0	0x00000000	0x20000064	00 00 00 00
2	A (0)	5	250	1250	0x000004E2	0x200000A4	E2 04 00 00
3	C (2)	1	150	150	0x00000096	0x200000E4	96 00 00 00
4	B (1)	3	200	600	0x00000258	0x20000124	58 02 00 00

Module 5

Employee #	Allowance	Allowance Hex Value	Allowance Address	Bytes (Little Endian)
0	18,000	0x00004650	0x20000028	50 46 00 00
1	15,000	0x00003A98	0x20000068	98 3A 00 00
2	20,000	0x00004E20	0x200000A8	20 4E 00 00
3	13,500	0x000034BC	0x200000E8	BC 34 00 00
4	17,000	0x00004268	0x20000128	68 42 00 00

<u>Module 6</u>

Employee #	TAX	TAX Hex Value	TAX Address	Bytes (Little Endian)
0	6850	0x00001AC2	0x20000030	C2 1A 00 00
1	2750	0x00000ABE	0x20000070	BE 0A 00 00
2	8125	0x00001FBD	0x200000B0	BD 1F 00 00
3	2372	0x00000944	0x200000F0	44 09 00 00
4	6260	0x00001874	0x20000130	74 18 00 00

<u>Module 7</u>

Employee #	Net Salary	Net Salary Hex Value	Net Salary Address	Bytes (Little Endian)
0	61, 650	0x0000F0D2	0x20000034	D2 F0 00 00
1	52, 250	0x0000CC1A	0x20000074	1A CC 00 00
2	73, 125	0x00011DA5	0x200000B4	A5 1D 01 00
3	45, 078	0x0000B016	0x200000F4	16 B0 00 00
4	56, 340	0x0000DC14	0x20000134	14 DC 00 00

<u>Module 8</u>

So the final order in 0x20005000 (sorted table) is:

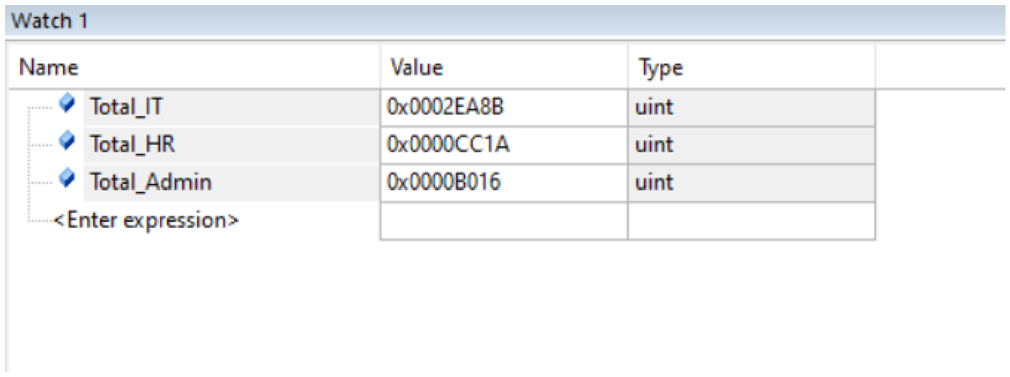
1. Highest Net Salary → Emp 2 (ID 1003, net 73,125)
2. Next → Emp 0 (ID 1001, net 61,650)
3. Next → Emp 4 (ID 1005, net 56,340)
4. Next → Emp 1 (ID 1002, net 52,250)
5. Lowest → Emp 3 (ID 1004, net 45,078)

Sorted Employee #	Original Employee #	Employee ID	Sorted Address to Check	Hex Value	Bytes (Little Endian)
0	2	1003	0x20005000	0x000003EB	EB 03 00 00
1	0	1001	0x20005040	0x000003E9	E9 03 00 00
2	4	1005	0x20005080	0x000003ED	ED 03 00 00
3	1	1002	0x200050C0	0x000003EA	EA 03 00 00
4	3	1004	0x20005100	0x000003EC	EC 03 00 00

Module 9

Department	Variable name	Value (Decimal)	Value (Hex)	Bytes (Little Endian)
IT	Total_IT	191,115	0x0002EA8B	8B EA 02 00
HR	Total_HR	52,250	0x0000CC1A	1A CC 00 00
Admin	Total_Admin	45,078	0x0000B016	16 B0 00 00

*Screenshot:



Watch 1

Name	Value	Type
Total_IT	0x0002EA8B	uint
Total_HR	0x0000CC1A	uint
Total_Admin	0x0000B016	uint
<Enter expression>		

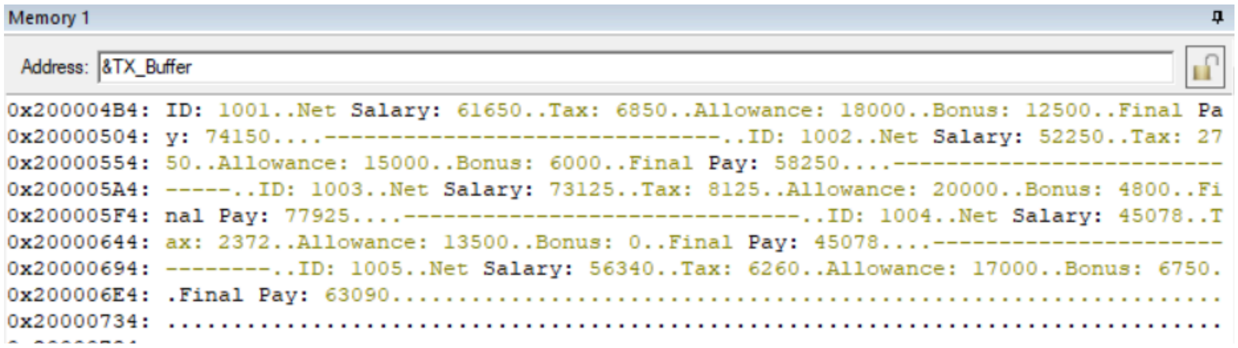
Module 10

Employee #	Bonus	Bonus Hex Value	Bonus Address	Bytes (Little Endian)

E0	12,500	0x000030D4	0x2000002C	D4 30 00 00
E1	6,000	0x00001770	0x2000006C	70 17 00 00
E2	4,800	0x000012C0	0x200000AC	C0 12 00 00
E3	0	0x00000000	0x200000EC	00 00 00 00
E4	6,750	0x00001A5E	0x2000012C	5E 1A 00 00

Module 11

*Screenshot:



```
0x200004B4: ID: 1001..Net Salary: 61650..Tax: 6850..Allowance: 18000..Bonus: 12500..Final Pa
0x20000504: y: 74150....------------------------------..ID: 1002..Net Salary: 52250..Tax: 27
0x20000554: 50..Allowance: 15000..Bonus: 6000..Final Pay: 58250....-------------------------
0x200005A4: -----..ID: 1003..Net Salary: 73125..Tax: 8125..Allowance: 20000..Bonus: 4800..Fi
0x200005F4: nal Pay: 77925....------------------------------..ID: 1004..Net Salary: 45078..T
0x20000644: ax: 2372..Allowance: 13500..Bonus: 0..Final Pay: 45078....----------------------
0x20000694: --------..ID: 1005..Net Salary: 56340..Tax: 6260..Allowance: 17000..Bonus: 6750.
0x200006E4: .Final Pay: 63090...............................................................
0x20000734: ................................................................................
```