Contents

STRUCTURES 2
INDIRECT AND INDEXED OPERANDS STRUCTS 20
PERFORMANCE OF ALIGNED AND MISALIGNED STRUCTS 27
STRUCTURES CONTAINING STRUCTURES 31
UNIONS IN ASSEMBLY 37
MACROS IN ASSEMBLY 40
ECHO AND LOCAL DIRECTIVES 44
CONDITIONAL ASSEMBLY DIRECTIVES 63
SPECIAL OPERATORS FOR MACROS 75
REPEAT BLOCKS 81

STRUCTURES

Structures in Assembly Language

- A **structure** in assembly language is defined using the **STRUCT** and **ENDS** directives.
- A structure is a **user-defined data type** that groups related data together.
- Inside a structure, **fields (members)** are defined using the **same syntax as ordinary variables**.
- Each field has:
  - A **name**
  - A **data type** (BYTE, WORD, DWORD, QWORD, etc.)
- Structures can contain **many fields**, limited only by memory and practicality.
- The assembler automatically:
  - Calculates the **offset** of each field
  - Determines the **total size** of the structure
- Structures can include:
  - Simple data types (BYTE, DWORD, etc.)
  - Arrays
  - Other structures (nested structures)
- A structure definition **does not allocate memory** by itself.
- Memory is allocated only when a **structure variable** is declared using the structure type.
- Structures make assembly programs:
  - **More readable**
  - **Easier to maintain**
  - **More similar to high-level language data organization**

```
0890 name STRUCT
0891 field-declarations
0892 name ENDS
```

The following structure defines an employee structure:

```
0900 Employee STRUCT
0901     IdNum BYTE "000000000"
0902     LastName BYTE 30 DUP(0)
0903     Years WORD 0
0904     SalaryHistory DWORD 0,0,0,0
0905 Employee ENDS
```

Structure Field Description

- **IdNum**
  - A **byte-sized field**
  - Initialized to the string **"000000000"**
  - Represents an ID stored as **ASCII characters**, not a numeric value
- **LastName**
  - A **30-byte array of bytes**
  - Initialized to **all zeros**
  - Typically used to store a null-terminated string
- **Years**
  - A **2-byte WORD**
  - Initialized to **0**
  - Often used to store years of service or age
- **SalaryHistory**
  - A **4-byte DWORD** *(or an array of DWORDs, depending on context)*
  - Initialized to **all zeros**

Note on DUP Operator

The **DUP operator** is used to create and initialize arrays.

Syntax example:

```
30 DUP(0)
```

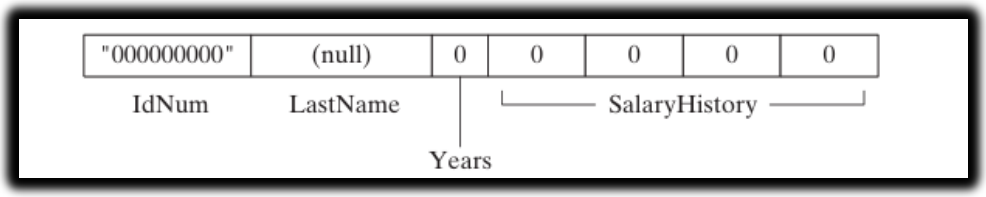
This creates an array of **30 bytes**, each initialized to **0**.

DUP is commonly used to:

- Zero-initialize strings
- Initialize arrays inside structures

Structure Memory Layout

"In memory, a structure is stored as a contiguous block, with each field placed sequentially at the next available address. The diagram below illustrates the memory layout of the Employee structure."



Memory Layout:

Memory Record Structure

FIELD NAME	MEMORY SIZE	ASSEMBLY TYPE
IdNum	1 Byte	BYTE
LastName	30 Bytes	DB 30 DUP(?)
Years	2 Bytes	WORD
SalaryHistory	4 Bytes	DWORD

Total Structure Size: 37 Bytes

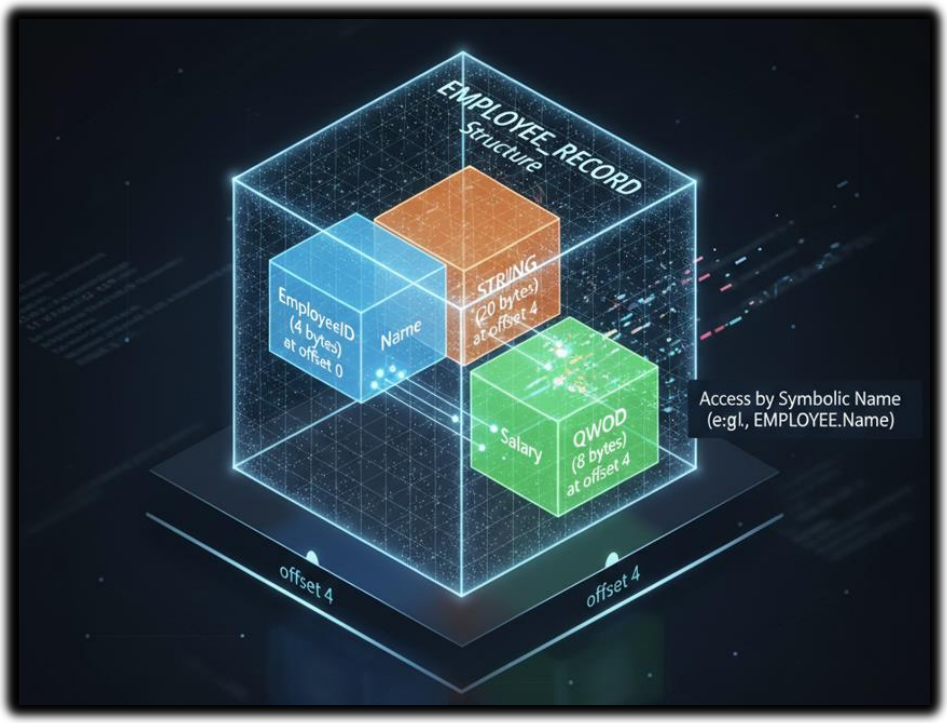
Using Structures in Assembly Language (MASM)

I. What Is a Structure?

A **structure** is a user-defined data type that groups related fields (members) under a single name.
Each field has its own type, size, and offset within the structure.

Structures are mainly used to:

- Represent records (e.g., employees, students)
- Organize related data in memory
- Access fields using symbolic names instead of raw offsets



II. Declaring Structure Variables

General Syntax

```
identifier STRUCTURE_TYPE <initializer-list>
```

Components Explained

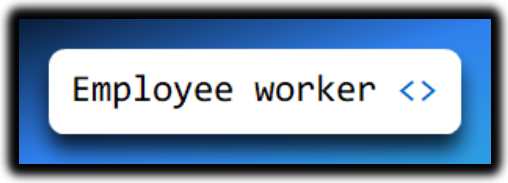
- **identifier**
  Name of the structure variable (instance)
- **STRUCTURE_TYPE**
  Name of the previously defined structure
- **initializer-list (optional)**
  - Values used to initialize structure fields
  - Must follow the **same order** as fields in the structure definition
  - Enclosed in angle brackets < >

Default Initialization

If the initializer list is **empty** or omitted:

- Fields are initialized to **default values**
- Numeric fields → 0
- Character fields → 0 or empty

Example



Explanation:

- Declares a structure variable named worker
- Type: Employee
- All fields are initialized to default values

III. Referencing a Structure Variable

Once a structure variable is declared, it behaves like a block of memory.

Example

```
mov dx, worker.Years
```

Accesses the Years field inside the worker structure

The assembler translates this into a memory access using:

```
base address of worker + offset of Years
```

IV. Referencing Structure Members (Dot Operator)

Syntax

```
structureVariable.memberName
```

Explanation

- structureVariable → name of the structure instance
- memberName → name of a field inside the structure

Example

```
mov dx, worker.SalaryHistory
```

- Reads the SalaryHistory field from the worker structure

- This is the **preferred and safest** way to access structure members

V. Using the OFFSET Operator with Structures

Purpose: The OFFSET operator retrieves the **address** of a structure or one of its members — not the value stored there.

Syntax

```
OFFSET structureVariable.memberName
```

Example

```
mov edx, OFFSET worker.LastName
```

What this does:

- Loads the address of LastName into EDX
- Commonly used when:
  - Passing addresses to procedures
  - Working with string or array fields

VI. Declaring Multiple Structure Variables

You can declare multiple variables of the same structure type.

Example

```
Employee1 Employee
Employee2 Employee
```

Explanation:

- Declares two independent structure instances
- Each occupies its own block of memory
- Fields are accessed independently

VII. Accessing Structure Fields (Dot Operator – Recommended)

Example

```
mov Employee1.IdNum, 123456789
```

- Stores 123456789 in the IdNum field of Employee1
- Clear, readable, and offset-safe
- Automatically adapts if structure layout changes

VIII. Accessing Structure Fields Using Displacement (⚠ Advanced / Risky)

Syntax

```
structureVariable[displacement]

; Example

mov Employee2[2], 5
```

Explanation

- The displacement is measured **in bytes**, not fields
- It is calculated from the **start of the structure**
- In this example: Offset 2 happens to correspond to the Years field

⚠ **Important Correction (2022 note fix):**
The displacement is **NOT** "the second field" — it is the **byte offset** of the field.

Why This Is Dangerous

- Breaks if:
  - Field sizes change
  - Fields are reordered
- Hard to read and maintain
- Easy to introduce bugs

Preferred approach:

```
mov Employee2.Years, 5
```

IX. Memory Layout Insight (Conceptual)

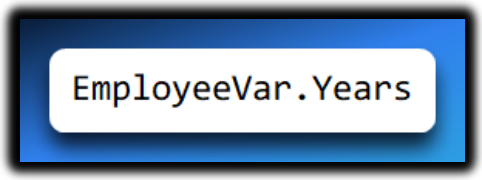
Example structure:

```
Employee STRUCT
  IdNum       DWORD   ; offset 0
  Years       WORD    ; offset 4
  Salary      DWORD   ; offset 6
Employee ENDS
```

Visual Layout

```
Offset 0 → IdNum
Offset 4 → Years
Offset 6 → Salary
```

The assembler computes these offsets automatically when you use:



X. Common Pitfalls

- Confusing **field order** with **byte offset**
- Using hard-coded displacements instead of symbolic names
- Forgetting that OFFSET gives an address, not data
- Assuming structures behave like high-level language objects

XI. Key Takeaways

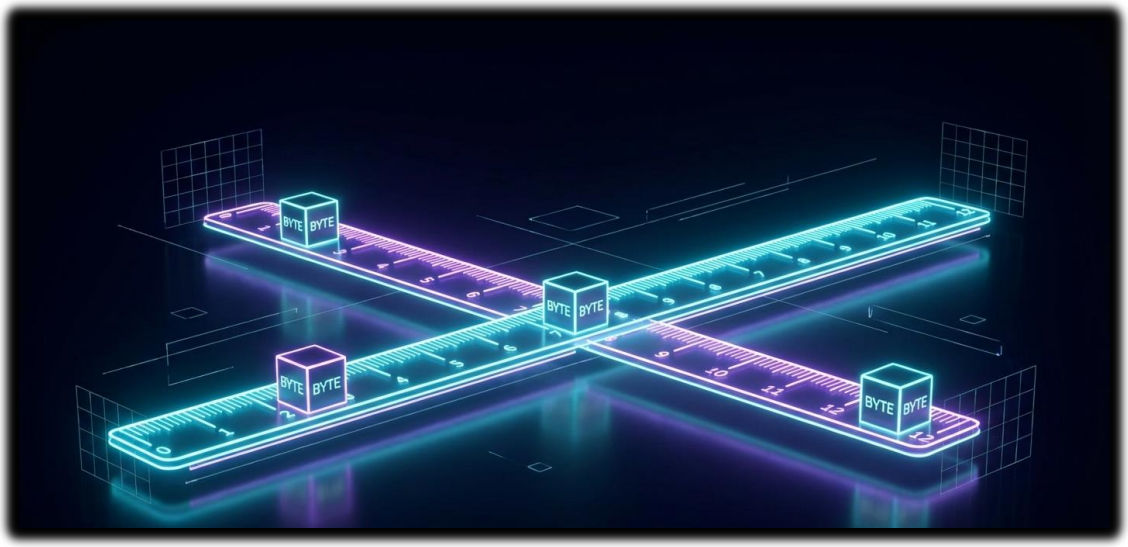
- Structures group related data into a single memory block
- Use **dot notation** for clarity and safety
- OFFSET is used when you need addresses
- Displacement access is possible but discouraged
- Assemblers compute field offsets — let them do it

Aligning Structure Fields

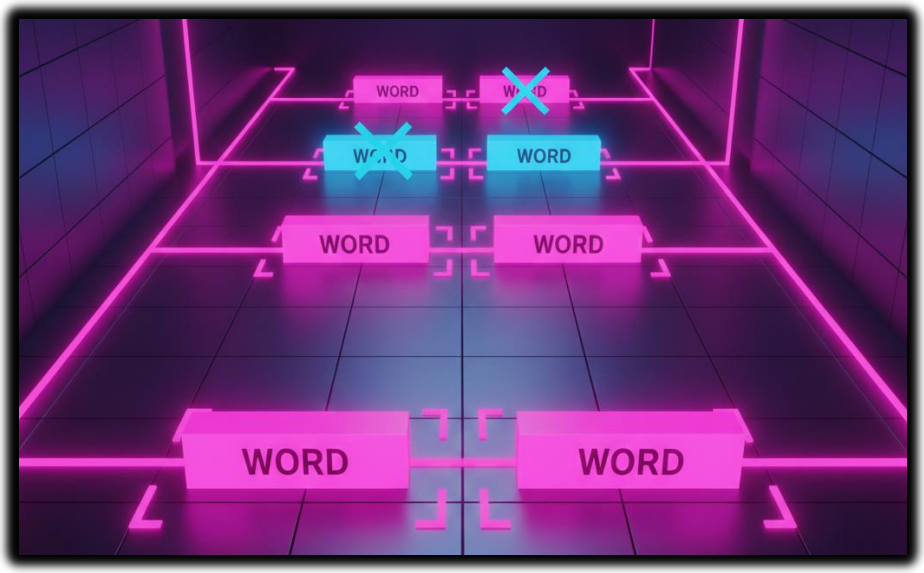
To achieve **optimal memory I/O performance**, structure members should be **aligned according to their data types**.

Alignment means placing data at memory addresses that are multiples of the data size.

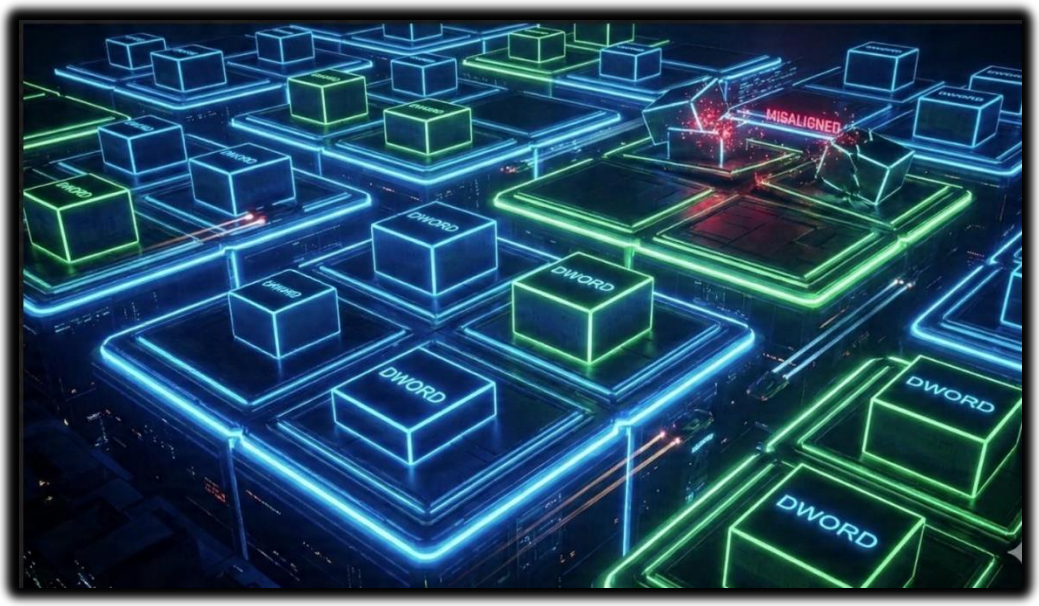
A **BYTE** member should be aligned on a **byte boundary**.



A **WORD** member should be aligned on a **word (2-byte) boundary**.



A **DWORD** member should be aligned on a **doubleword (4-byte) boundary**.



Proper alignment allows the CPU to **access data in fewer memory cycles**.

If structure members are **not properly aligned**, the CPU may require **additional memory accesses**, reducing performance.

Assemblers may insert **padding bytes** between fields to maintain proper alignment.

The **Microsoft C/C++ compilers** and **Win32 API functions** follow specific alignment rules, as summarized in the alignment table (referenced below).

Member Type	Alignment
BYTE, SBYTE	8-bit (byte) boundary
WORD, SWORD	16-bit (word) boundary
DWORD, SDWORD	32-bit (doubleword) boundary
QWORD	64-bit (quadword) boundary
REAL4	32-bit (doubleword) boundary
REAL8	64-bit (quadword) boundary
Structure	Largest alignment requirement of any member
Union	Alignment requirement of the first member

Data Alignment in Assembly Language (Structures & Variables)

I. What Is Data Alignment?

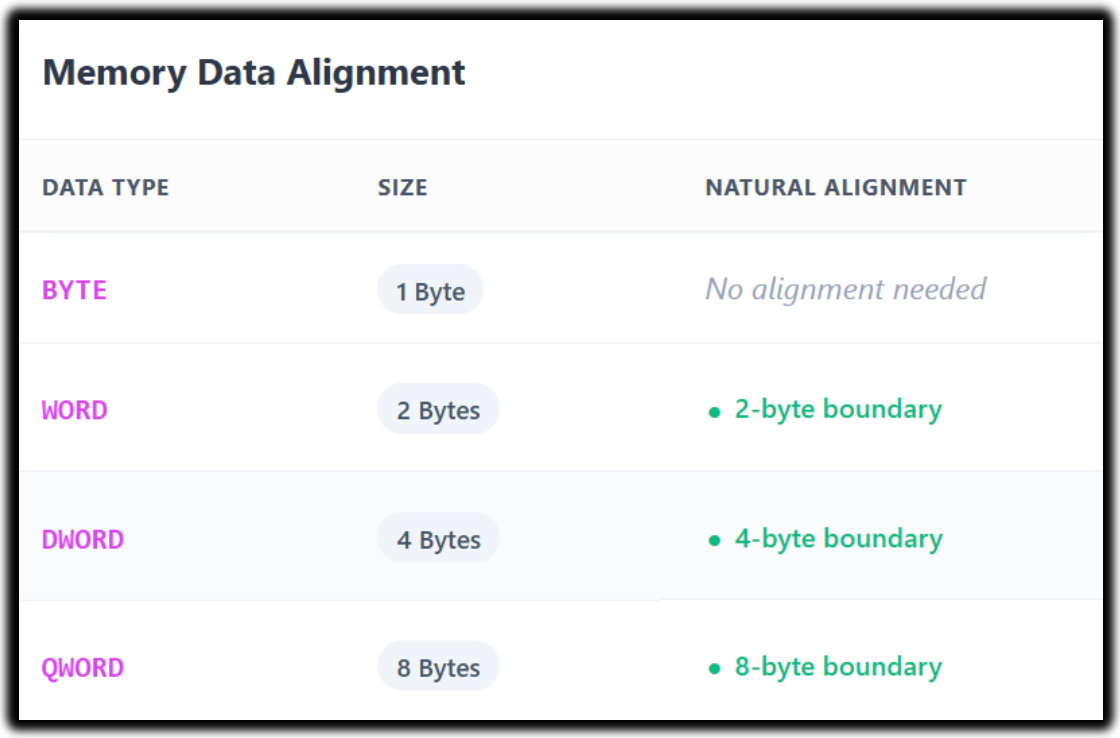
Data alignment means placing data in memory at addresses that are multiples of its size. Modern CPUs access aligned data more efficiently than unaligned data.

Example:

- A WORD (2 bytes) is best placed at an **even address**
- A DWORD (4 bytes) is best placed at an address divisible by **4**

II. Key Alignment Rules (x86)

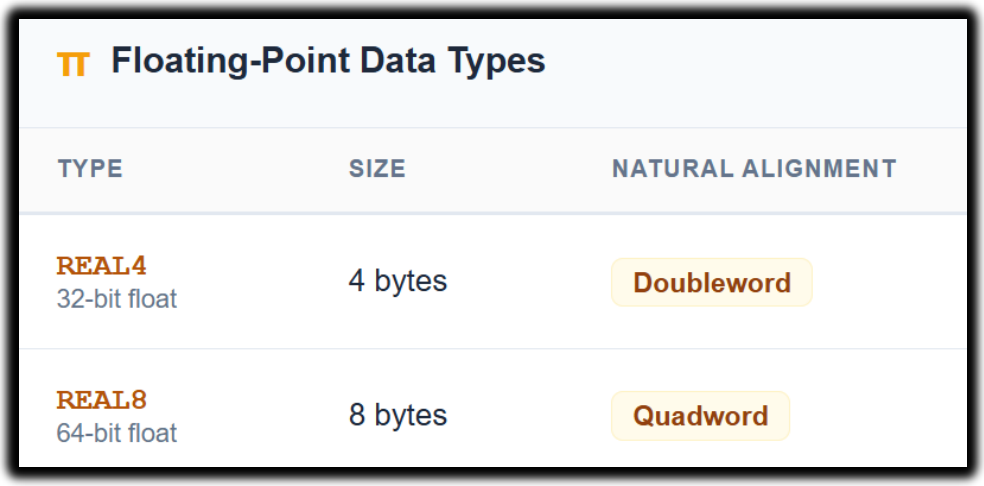
Natural Alignment of Basic Data Types



Memory Data Alignment

DATA TYPE	SIZE	NATURAL ALIGNMENT
BYTE	1 Byte	*No alignment needed*
WORD	2 Bytes	• 2-byte boundary
DWORD	4 Bytes	• 4-byte boundary
QWORD	8 Bytes	• 8-byte boundary

Floating-Point Alignment



Structure and Union Alignment

- **Structures**
  - Aligned to the **largest alignment requirement** of any member
- **Unions**
  - Aligned to the **alignment of the first member**

The assembler automatically calculates padding unless explicitly overridden.

3. Why Alignment Matters

Performance Impact

- Aligned data:
  - Can be fetched in a **single memory access**
- Unaligned data:
  - May require **multiple memory reads**
  - Can cause CPU penalties or microcode fixes

Practical Impact - Small performance loss for single variables

Large performance degradation when:

- Iterating over arrays of structures
- Processing large datasets

4. Automatic vs Manual Alignment

Automatic Alignment (Default)

- MASM aligns:
  - Variables
  - Structure fields
- Based on data type requirements
- Usually sufficient for most programs

Manual Alignment (When Needed)

- Required when:
  - Mixing different-sized fields
  - Matching external data formats
  - Optimizing critical data paths

5. ALIGN Directive

Purpose: The ALIGN directive forces the **next variable or field** to start at a specific boundary.

Syntax:

Where boundary can be:

- BYTE
- WORD
- DWORD
- QWORD

6. Aligning Standalone Variables

Example

```
.data
ALIGN DWORD
myVar DWORD ?
```

Explanation

- Ensures myVar starts at a **4-byte boundary**
- Prevents accidental misalignment due to previous data

7. Aligning Structure Fields

Problem Without Alignment - Fields may start at suboptimal addresses depending on preceding fields' sizes.

Correctly Aligned Structure Example:

```
Employee STRUCT
  IdNum          BYTE 10 DUP('o')
  LastName       BYTE 30 DUP(0)

  ALIGN WORD
  Years          WORD 0

  ALIGN DWORD
  SalaryHistory  DWORD ?
Employee ENDS
```

8. How Padding Works (Memory Layout)

Field Sizes & Offsets

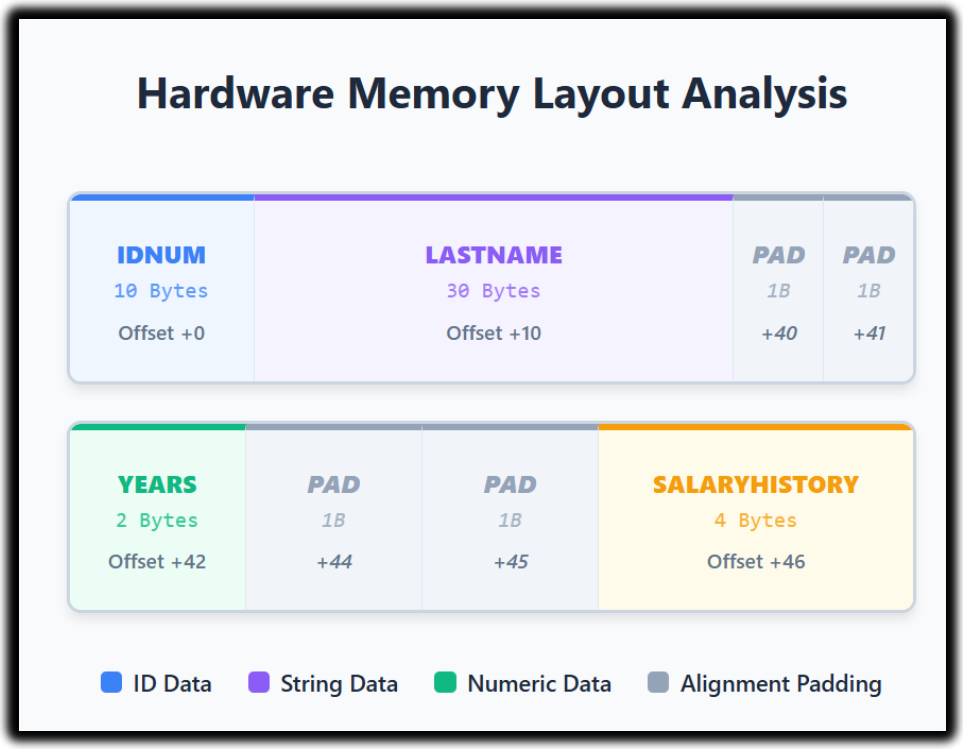
Aligned Memory Structure (Padded)

FIELD NAME	SIZE	OFFSET	TYPE
IdNum	10 Bytes	+0	DATA
LastName	30 Bytes	+10	DATA
Padding (for WORD)	*2 Bytes*	*+40*	PADDING
Years	2 Bytes	+42	DATA
Padding (for DWORD)	*4 Bytes*	*+44*	PADDING
SalaryHistory	4 Bytes	+48	DATA

Total Structure Size:

- **52 bytes**
- Rounded up to alignment requirement if used in arrays

9. Visual Memory Layout (Suggested Diagram)



This shows how alignment introduces **padding bytes** to meet boundaries.

10. Why Structure Alignment Is Critical

- Affects:
  - Access speed
  - Cache efficiency
  - SIMD operations
- Especially important for:
  - Arrays of structures
  - Repeated memory access loops

11. Common Mistakes

- Assuming alignment is based on **field order**, not size
- Forgetting that padding bytes consume memory
- Manually hardcoding offsets instead of using field names
- Overusing ALIGN when the assembler already handles it

12. Key Takeaways

- Each data type has a **natural alignment**
- Structures align to their **largest field**
- Unaligned memory access hurts performance
- ALIGN lets you explicitly control field placement
- Always let the assembler compute offsets when possible

INDIRECT AND INDEXED OPERANDS STRUCTS

Indirect Operands

Indirect and Indexed Operands with Structures (x86 / MASM)

When working with **structures and arrays of structures**, direct addressing (StructVar.Field) is often not flexible enough.

Instead, assembly provides **indirect** and **indexed** operands that allow you to:

- Access structure members using registers
- Traverse arrays of structures
- Write loops and dynamic memory access code

What Is an Indirect Operand?

An **indirect operand** accesses memory **through a register** that holds an address.

- The register contains the **base address**
- The instruction dereferences that register
- Commonly used when:
  - Passing structure addresses to procedures
  - Accessing dynamically selected structures

PTR Operator (Why It Is Needed)

MASM requires **type information** when dereferencing memory through registers.

The PTR operator:

- Tells the assembler **what data type** the memory represents
- Allows access to structure members using dot notation

Syntax

```
(StructureType PTR [register]).member
```

Example: Accessing a Structure via Indirection

```
0992 mov esi, OFFSET worker
0993 mov ax, (Employee PTR [esi]).Years
```

Step-by-Step Explanation

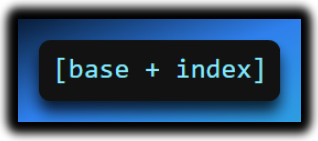
1. OFFSET worker loads the address of the structure into ESI
2. [ESI] dereferences that address
3. Employee PTR tells MASM the memory layout
4. .Years accesses the correct offset inside the structure

Without PTR, MASM does not know how to interpret [esi]

2. Indexed Operands (Register + Offset Addressing)

What Is an Indexed Operand?

An **indexed operand** accesses memory using:



This is especially useful for **arrays of structures**.

Typical Use Case

- Base address → start of the array
- Index register → offset to a specific element
- Structure type → defines field offsets

3. Arrays of Structures

Declaring an Array of Structures

```
1000 .data
1001     department Employee 5 DUP(<>)
1002 .code
1003     mov esi, TYPE Employee
1004     ; index = 1
1005     mov department[esi].Years, 4
```

- Creates an array of 5 Employee structures
- Each element is placed contiguously in memory

4. Accessing an Array Element Using Indexing

Correct Concept

To access element i:

```
Address = base + (i × TYPE Structure)
```

Corrected Example (Fixing 2022 Notes)

```
mov esi, TYPE Employee      ; size of one structure
mov Department[esi].Years, 4
```

⚠ **Important Correction:**

- This accesses the **second element**, because:
  - esi = TYPE Employee
  - Offset = 1 × structure size

The original notes incorrectly implied esi held an index — it actually holds a **byte offset**.

Preferred & Clearer Form

```
mov esi, TYPE Employee
mov (Employee PTR Department[esi]).Years, 4
```

This ensures:

- Correct type interpretation
- Safe member access

5. Indirect + Indexed Addressing Combined

This is the **most common pattern** when looping through arrays of structures.

6. Looping Through an Array of Structures (Full Example)

Code:

```
; Loop Through Array of Structures
INCLUDE Irvine32.inc

NumPoints = 3

.data
ALIGN WORD
AllPoints COORD NumPoints DUP(<0,0>)

.code
main PROC

  mov edi, 0              ; byte offset (index)
  mov ecx, NumPoints      ; loop counter
  mov ax, 1               ; starting X and Y values

L1:
  mov (COORD PTR AllPoints[edi]).X, ax
  mov (COORD PTR AllPoints[edi]).Y, ax

  add edi, TYPE COORD     ; move to next structure
  inc ax                  ; update values
  loop L1

  exit
main ENDP
END main
```

7. How This Loop Works (Step-by-Step)

Initialization

- EDI = 0 → start of the array
- ECX = NumPoints → loop count
- AX = 1 → initial coordinate value

Inside the Loop

Access current structure:

```
AllPoints + EDI
```

Write to X field

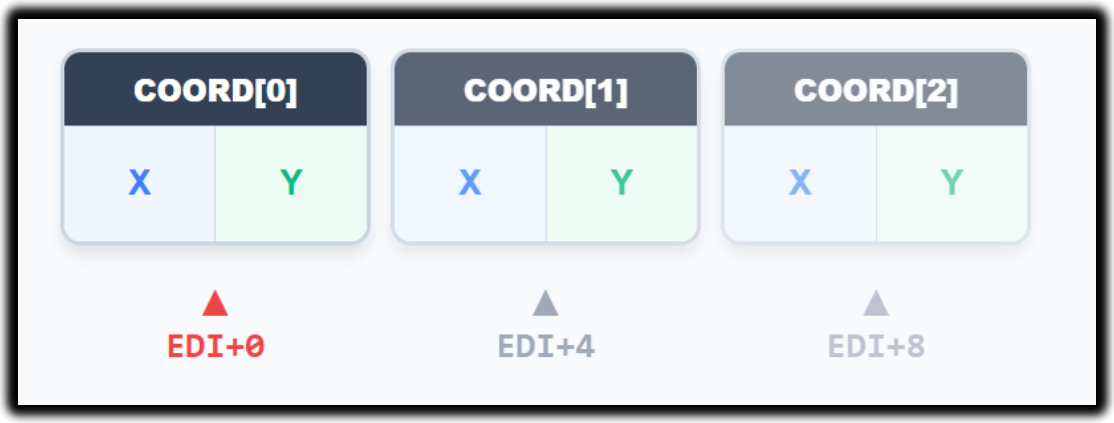
Write to Y field

Advance to next element:

```
EDI += TYPE COORD
```

1. Increment coordinate value
2. Loop until ECX = 0

8. Memory Layout Visualization (Recommended)



After each iteration:

```
EDI += sizeof(COORD)
```

9. Common Pitfalls (Very Exam-Relevant)

- Treating index registers as **element numbers**, not byte offsets
- Forgetting to multiply index by TYPE Structure
- Omitting PTR when dereferencing registers
- Using displacement values instead of symbolic field names

10. Key Differences: Indirect vs Indexed

Addressing Modes Comparison

Direct vs. Indexed Access in x86 Assembly

FEATURE	INDIRECT ADDRESSING	INDEXED ADDRESSING
Uses Register	YES	YES
Uses Offset/Label	NO	YES
Best Use Case	Single objects or structures passed by pointer.	Arrays, tables, and lists where items are stepped through.
Requires PTR	YES	YES
Code Example	`mov eax, [ebx]`	`mov eax, myArr[esi]`

11. Key Takeaways

- **Indirect operands** use registers to hold structure addresses
- **Indexed operands** use offsets to access array elements
- PTR is mandatory for structure member access via registers
- Index registers always represent **byte offsets**
- TYPE Structure is critical for safe iteration

PERFORMANCE OF ALIGNED AND MISALIGNED STRUCTS

Aligned structure members are accessed by the CPU **more efficiently** than misaligned members.

When data is **properly aligned**, the CPU can access it using **fewer instructions and memory cycles**.

Misaligned structure members may require the CPU to perform **multiple memory accesses** to read or write a single data item.

The **performance impact** of misalignment depends on:

- The **CPU architecture**
- The **data size** of the member
- The **layout of the structure**

On some architectures, misaligned access causes a **performance penalty**, while on others it may even cause a **hardware exception**.

In general, **misaligned structures reduce execution speed** compared to aligned structures.

Performance testing code can be used to **compare access times** between aligned and misaligned structure members.

```
.data
    ALIGN DWORD
    startTime DWORD ?
    ; align startTime
    emp Employee <>
    ; or: emp EmployeeBad <>
.code
    call
    GetMSeconds
    ; get starting time
    mov
    startTime,eax
    mov
    ecx,0FFFFFFFFh
    ; loop counter
    L1:
    mov
    emp.Years,5
    mov
    emp.SalaryHistory,35000
    loop
    L1
    call
    GetMSeconds
    ; get starting time
    sub
    eax,startTime
    call
    WriteDec
    ; display elapsed time
```

Example: Performance Impact of Aligned vs. Misaligned Structures

- The code example:
  - Retrieves the **system time**.
  - Executes a **loop that accesses structure fields**.
  - Calculates the **elapsed time**.
- The variable emp can be declared as either:
  - **Employee** → aligned structure
  - **EmployeeBad** → misaligned structure
- Observed results:
  - Using **Employee**: elapsed time = **6141 ms**
  - Using **EmployeeBad**: elapsed time = **6203 ms**
  - **Difference** = 62 ms (small but measurable)
- Interpretation:
  - Even **minor misalignment** can **negatively affect performance**.
  - Impact depends on **CPU architecture** and **structure layout**.
  - On some systems, misalignment can cause a **larger performance penalty**.
- **Conclusion / Best Practice:**
  - Always **align structure members** to their natural boundaries.
  - Proper alignment ensures **efficient CPU access** and **better performance**.

Example: Displaying the System Time

The following is a step-by-step explanation of the program ShowTime.asm in depth:

```
1067 ; Structures (ShowTime.ASM)
1068 INCLUDE Irvine32.inc
1069 .data
1070     sysTime SYSTEMTIME <>
1071     XYPos COORD <10,5>
1072     consoleHandle DWORD ?
1073     colonStr BYTE ":",0
1074 .code
1075     main PROC
1076     ; Get the standard output handle for the Win32 Console.
1077     INVOKE GetStdHandle, STD_OUTPUT_HANDLE
1078     mov consoleHandle,eax
```

- The **first part of the program** defines the **data structures** that will be used.
- sysTime → **SYSTEMTIME** structure, used to **store the current system time**.
- XYPos → **COORD** structure, used to **store the cursor position** in the console.
- consoleHandle → **DWORD** variable, used to **store the handle to the standard output**.
- **GetStdHandle** function: retrieves the **handle to the standard output**.
- The **console handle** is then used by:
  - **SetConsoleCursorPosition** → positions the cursor in the console.
  - **WriteString** → writes data to the console at the current cursor position.

```
1083 ; Set the cursor position and get the system time.
1084 INVOKE SetConsoleCursorPosition, consoleHandle, XYPos
1085 INVOKE GetLocalTime, ADDR sysTime
```

Setting Cursor Position and Retrieving System Time

The program **sets the cursor position** and **retrieves the system time**.

SetConsoleCursorPosition function:

- Sets the **cursor position** in the console to the specified coordinates (XYPos).

GetLocalTime function:

- Retrieves the **current system time** and stores it in the sysTime structure.

```
1090 ; Display the system time (hh:mm:ss).
1091 movzx eax,sysTime.wHour
1092 ; hours
1093 call WriteDec
1094 mov edx,OFFSET colonStr
1095 ; ":"
1096 call WriteString
1097 movzx eax,sysTime.wMinute
1098 ; minutes
1099 call WriteDec
1100 call WriteString
1101 movzx eax,sysTime.wSecond
1102 ; seconds
1103 call WriteDec
1104 call Crlf
```

Displaying System Time on the Console

- The program **displays the system time** to the console.
- **WriteDec** function:
  - Writes a **decimal number** to the console (e.g., hours, minutes, seconds).
- **WriteString** function:
  - Writes a **string** to the console (e.g., labels like “Hour: ”).
- **Crlf** function:
  - Writes a **carriage return and line feed** to move the cursor to the next line.

```
1108 call WaitMsg
1109 ; "Press any key..."
1110 exit
1111 main ENDP
1112 END main
```

Displaying a Message and Waiting for User Input

WaitMsg function:

- Displays a **message** to the console.
- **Waits for the user** to press a key before continuing.

Conclusion

The ShowTime.asm program demonstrates how to use structures in assembly language.

Structures allow grouping of related data, improving:

- Code readability
- Maintainability

Proper use of structures, along with **alignment considerations**, ensures **efficient CPU access** and **better performance**.

STRUCTURES CONTAINING STRUCTURES

A **nested structure** occurs when a **structure contains instances of other structures**.

This allows grouping of **complex or related data** within a larger structure.

Example: Rectangle structure can have two fields:

- upperLeft → a **COORD** structure
- lowerRight → a **COORD** structure

Nested structures help in organizing data hierarchically and make code more readable and maintainable.

Accessing fields in a nested structure uses the syntax:

```
Rectangle STRUCT
    UpperLeft COORD <>
    LowerRight COORD <>
Rectangle ENDS
```

Or

```
Rectangle.upperLeft.X
Rectangle.upperLeft.Y
Rectangle.lowerRight.X
Rectangle.lowerRight.Y
```

Rectangle Structure with Nested COORDs

A Rectangle structure **contains two COORD structures**:

- **UpperLeft** → represents the upper-left corner of the rectangle
- **LowerRight** → represents the lower-right corner of the rectangle

Rectangle variables can be **declared in different ways**:

- **Without overrides** → all fields use default/uninitialized values
- **With overrides** → individual COORD fields can be initialized with specific values

Examples of declarations:

- rect1 RECTANGLE <> → default/uninitialized
- rect2 RECTANGLE <UpperLeft <10, 20>, LowerRight <50, 80>> → all fields initialized
- rect3 RECTANGLE <UpperLeft <5, 15>, LowerRight <>> → partially initialized

Nested field access uses the syntax:

```
RectangleVar.CoordField.SubField
```

Or

```
rect1 Rectangle < >
rect2 Rectangle { }
rect3 Rectangle { {10,10}, {50,20} }
rect4 Rectangle < <10,10>, <50,20> >
```

This structure allows **grouping related coordinate data** in a readable and maintainable way.

Declaring and Accessing Nested Structure Variables

Rectangle variables can be declared in multiple ways:

- rect1, rect2 → created with default values for UpperLeft and LowerRight.
- rect3 → created with specified values for UpperLeft and LowerRight.
- rect4 → alternative syntax for specifying field values.

Accessing fields of a structure uses **dot notation**:

- Example: move 10 to the **X coordinate** of rect1's UpperLeft:

```
mov rect1.UpperLeft.X, 10
```

Accessing fields via indirect addressing

Example: move 10 to the **Y coordinate** of UpperLeft in a structure pointed to by esi:

```
mov esi,OFFSET rect1
mov (Rectangle PTR [esi]).UpperLeft.Y, 10
```

Using OFFSET operator to get pointers to structure fields (including nested fields):

Example: move 50 to the **X coordinate** of LowerRight in rect2:

```
1141 mov edi,OFFSET rect2.LowerRight
1142 mov (COORD PTR [edi]).X, 50
```

- Dot notation works for **direct variable access**.
- Indirect addressing + ptr allows **pointer-based access**.
- OFFSET operator returns **memory address of a field**, useful for **nested structures**.

Example program

```
; Drunkard's Walk
; Drunkard's walk program simulates a professor's random path in an imaginary grid.
; The professor starts at coordinates 25, 25 and wanders around the immediate area.

INCLUDE Irvine32.inc

; Constants
WalkMax = 50    ; Maximum number of steps
StartX = 25     ; Starting X-coordinate
StartY = 25     ; Starting Y-coordinate

; Define a structure to store the path and number of steps
DrunkardWalk STRUCT
    path COORD WalkMax DUP(<0,0>) ; Array of COORD objects for the path
    pathsUsed WORD 0              ; Number of steps taken
DrunkardWalk ENDS

; Prototypes
DisplayPosition PROTO currX:WORD, currY:WORD

.data
aWalk DrunkardWalk <> ; Create an instance of the DrunkardWalk structure

.code
main PROC
    mov esi,OFFSET aWalk ; Get the address of the aWalk structure
    call TakeDrunkenWalk ; Simulate the professor's walk
    exit
main ENDP
;-----------------------------------------------------
TakeDrunkenWalk PROC
LOCAL currX:WORD, currY:WORD

; Takes a walk in random directions (north, south, east, west).
; Receives: ESI points to a DrunkardWalk structure
; Returns: the structure is initialized with random values
;-----------------------------------------------------

pushad ; Preserve registers

; Initialize EDI with the address of the path array
mov edi,esi
add edi,OFFSET DrunkardWalk.path

mov ecx,WalkMax  ; Set loop counter
mov currX,StartX ; Initialize current X-location
mov currY,StartY ; Initialize current Y-location

Again:
; Insert the current location in the array.
mov ax,currX
mov (COORD PTR [edi]).X,ax ; Store X-coordinate
mov ax,currY
mov (COORD PTR [edi]).Y,ax ; Store Y-coordinate

INVOKE DisplayPosition, currX, currY ; Display the current position

mov eax,4 ; Choose a random direction (0-3)
call RandomRange
```

```
.IF eax == 0
    ; North
    dec currY
.ELSEIF eax == 1
    ; South
    inc currY
.ELSEIF eax == 2
    ; West
    dec currX
.ELSE
    ; East (EAX = 3)
    inc currX
.ENDIF

add edi,TYPE COORD ; Move to the next COORD
loop Again

Finish:
mov (DrunkardWalk PTR [esi]).pathsUsed, WalkMax ; Set pathsUsed to WalkMax
popad ; Restore registers
ret
TakeDrunkenWalk ENDP

```

```
;-----------------------------------------------------
DisplayPosition PROC currX:WORD, currY:WORD
; Display the current X and Y positions.
;-----------------------------------------------------

.data
commaStr BYTE ",",0 ; Comma string

.code
pushad ; Preserve registers

; Display the current X position
movzx eax,currX
call WriteDec

mov edx,OFFSET commaStr ; Load the comma string
call WriteString

; Display the current Y position
movzx eax,currY
call WriteDec

call Crlf ; Move to the next line

popad ; Restore registers
ret
DisplayPosition ENDP

END main
```

Drunkard's Walk Program (Simulates a Professor's Random Walk)

1. Include Directives

INCLUDE Irvine32.inc

Provides **console I/O** functions and other utilities.

2. Constants

WalkMax → Maximum number of steps in the walk.

StartX, StartY → Starting coordinates of the professor.

3. Structure Definition

DrunkardWalk STRUCT ... ENDS

- path → Array of **COORD** objects to store the professor's path.
- pathsUsed → WORD to track the **number of steps taken**.

4. Function Prototypes

- DisplayPosition → Displays the professor's **current X and Y coordinates**.

5. Data Section (.data)

Defines program data, including:

aWalk → An **instance of the DrunkardWalk structure**.

6. Code Section (.code)

Main Procedure

- Entry point of the program.
- Initializes ESI with the address of aWalk.
- Calls TakeDrunkenWalk to simulate the professor's walk.

TakeDrunkenWalk Procedure

Simulates the **random walk**:

- Initializes local variables currX and currY to **starting coordinates**.
- Loop runs until **WalkMax** steps are completed:
    - Inserts current coordinates into path array.
    - Calls DisplayPosition to show current position.
    - Randomly selects **next direction** (north, south, east, west).
- Updates pathsUsed in aWalk to indicate the **total steps taken**.

DisplayPosition Procedure

- Displays **X and Y coordinates** with proper formatting:
    - WriteDec → prints numbers.
    - WriteString → prints commas or text.
    - Crlf → moves cursor to **next line**.

7. Program Termination

- Walk is complete.
- END main → Marks **end of the program**.

8. Short note:

- Uses a **structure** to manage data related to the professor's walk.
- Tracks the **path** and **number of steps**.
- Randomly selects steps until the **maximum is reached**.
- Demonstrates **structure usage, arrays, loops, and console I/O** in assembly.

UNIONS IN ASSEMBLY

A **union** is a user-defined data type that **groups multiple fields at the same memory offset**.

- **Key property:** Only **one field exists at a time** in memory; all fields share the same location.
- **Size of a union:** Determined by the **largest field** it contains.
- **Declaration:** Use the **UNION** and **ENDS** directives.

Example of a union declaration:

```
1265 myUnion UNION
1266     field1 DWORD 0
1267     field2 WORD 0
1268     field3 BYTE 0
1269 myUnion ENDS
```

Field initialization rules:

- Each field in a union can have **only one initializer**.
- Fields follow the same rules as structures, but since all share the same memory, initializing one affects the union as a whole.

Nesting Unions in Structures

You can include a union **inside a structure** to combine flexibility and organization.

Two main ways to nest a union:

Declare the union directly inside the structure

```
1277 MyStructure STRUCT
1278     UNION MyUnion
1279         field1 DWORD ?
1280         field2 WORD ?
1281         field3 BYTE ?
1282     ENDS
1283 MyField DWORD ?
1284 MyStructure ENDS
```

Use the union's name in a declaration inside the structure/ Declare a union separately and include it as a field in the structure.

```
1290 MyStructure STRUCT
1291     MyUnion MyField
1292         field1 DWORD ?
1293         field2 WORD ?
1294         field3 BYTE ?
1295 MyStructure ENDS
```

Declaring and Using Union Variables

Declaring a union variable is similar to declaring a structure variable.

However, you can only have one initializer:

```
myVar MyUnion <1234h>
```

To use a specific field of a union variable, you supply the field's name. For example:

```
mov myVar.field1, eax
mov myVar.field2, bx
```

Unions Can Contain Structures

Unions can contain structures as their fields.

This allows a single memory location to represent **different types of structured data** depending on the situation.

Example: KEY_EVENT_RECORD (Windows API)

- The structure contains a union named **uChar**.
- The union can store:
  - A **Unicode character**
  - An **ASCII character**
- Depending on the **keyboard event type**, the program uses the appropriate field.

Advantages of combining unions and structures in assembly:

- Efficient **memory usage**: multiple types share the same memory space.
- Flexible handling of **alternative data formats**.
- Makes code **more organized and readable** when dealing with complex data like input events or records.

Key point:

- Unions allow **storing data of different sizes in the same memory space**, while structures help **group related fields together**.
- Together, they provide a **powerful way to organize complex data** in assembly language programs.

MACROS IN ASSEMBLY

What is a Macro?

- A **macro** is a named block of assembly statements.
- It can be **invoked multiple times** in a program.
- When called, the macro's **code is inserted directly** where it's invoked (**inline expansion**).
- No actual CALL instruction is used — the code is just copied into place.

Placement of Macros:

- Usually defined at the **beginning of the source code**.
- Can also be placed in a **separate file** and included via the INCLUDE directive.

How Macros Work:

- Expanded during the **assembler's preprocessing step**.
- Every invocation is replaced with a **copy of the macro's code**.
- If a macro is defined but never called, its code **does not appear** in the compiled program.

Defining Macros:

- Use the MACRO and ENDM directives.
- Example syntax:

```
macroname MACRO parameter-1, parameter-2...
statement-list
ENDM
```

Or

```
MyMacro MACRO
    ; Assembly instructions go here
ENDM
```

Macros with Parameters in Assembly

What are Parameters?

- Named placeholders for values passed to a macro.
- Can be text, integers, variable names, or other values.
- The preprocessor treats them as text; no type checking occurs at macro expansion.
- Type checking happens during assembly.

Example: mPutchar Macro

```
mPutchar MACRO char
    push eax
    mov al, char
    call WriteChar
    pop eax
ENDM
```

Takes a **single parameter** char.

- Temporarily saves eax to preserve its value.
- Moves char into AL and calls WriteChar to display it.
- Restores eax after the call.

How it Works:

When invoked, the **macro code is inserted inline**, replacing the macro call.

Example:

Expands into the four assembly statements above, displaying 'A' on the console.

Key Advantages of Parameterized Macros:

- Avoids repetitive coding.
- Makes programs **more readable and maintainable**.
- **Flexible and reusable** — can handle different arguments for different situations.

Macros with parameters are a **powerful tool** for modular assembly code.

They let you **reuse code efficiently** without function call overhead.

Using **mPutchar** used for printing a single character to the console:

```
mPutchar MACRO char
    push eax       ; Push the value in the eax register
    mov al, char   ; Move the character 'char' into the al register
    call WriteChar ; Call the WriteChar procedure to display the character
    pop eax        ; Pop the original value back into eax
ENDM


;You'd use it like this:


.code
; ... Your code here ...
mPutchar 'A' ; Display the character 'A' on the console
; ... Your code here ...
```

Invoking (Using) a Macro

- To use a macro, just write its **name** in your program.
- You can also give it **arguments** (extra information it needs).
- The arguments you give replace the placeholders inside the macro.
- The order matters: the first argument goes to the first placeholder, the second to the second, and so on.
- If you give **too many arguments**, the assembler warns you.
- If you give **too few arguments**, the missing ones stay empty.
- When the program runs, the macro is expanded into the actual instructions it represents.

Debugging Macros

Macros can be tricky to debug because they expand into hidden code.

To check what's really happening:

- Look at the **listing file (.LST)** after assembling. It shows how macros were expanded.
- In tools like **Visual Studio Community**(not VSCode), you can open the **disassembly view** to see the actual machine code created from the macro.

```
1340 mPutchar MACRO char:REQ
1341     push eax            ; Save the value in the eax register
1342     mov al, char        ; Move the character from 'char' to the al register
1343     call WriteChar      ; Call the WriteChar procedure to display the character
1344     pop eax             ; Restore the original value in eax
1345 ENDM
```

Extra Features of Macros

- You can mark some arguments as **required** using the REQ qualifier.
- If you forget to provide a required argument, the assembler gives an error.
- You can hide comments inside macros by using **double semicolons (;;)**.
- This way, notes you write inside the macro won't appear in the expanded code.

Summary

- Think of macros as **predefined mini-functions** in assembly.
- You call them by name, give them arguments, and they expand into the instructions you need.
- They save time, reduce repetition, and make your code easier to manage.
- But remember: debugging can be harder, so always check how they expand.

ECHO AND LOCAL DIRECTIVES

📣 The ECHO Directive in Assembly

What is ECHO?

- The **ECHO directive** is a special instruction used during assembly (when your source code is being converted into machine code).
- It doesn't affect the program's execution at runtime. Instead, it **prints a message to the assembler's output window or console while assembling**.
- Think of it as a way to leave "breadcrumbs" or **status messages** for yourself or other programmers.

Why Use ECHO?

- **Debugging aid**: Helps you see when certain macros are being expanded.
- **Informative messages**: Lets you print reminders or notes during assembly.
- **Tracking flow**: Useful when your program has many macros—you can confirm which ones are being expanded and in what order.

Example Macro with ECHO

Here's the macro explained line by line:

```
mPutchar MACRO char:REQ
ECHO Expanding the mPutchar macro
push
eax
mov
al,char
call
WriteChar
pop
eax
ENDM
```

mPutchar MACRO char: REQ

- Defines a macro named mPutchar.
- It takes one required parameter: char.
- If you forget to provide char, the assembler will throw an error.

ECHO Expanding the mPutchar macro

- When the assembler reaches this line, it prints the message: *"Expanding the mPutchar macro"*
- This happens **during assembly**, not when the program runs.
- It's like a debug log telling you: "Hey, I'm expanding this macro right now."

push eax

- Saves the current value of the eax register onto the stack.
- This prevents overwriting whatever was in eax before.

mov al, char

- Moves the character you passed into the macro into the al register (the lower 8 bits of eax).
- Example: If you called mPutchar 'A', then 'A' is placed into al.

call WriteChar

- Calls a procedure named WriteChar.
- This procedure is responsible for displaying the character stored in al onto the console.

pop eax

- Restores the original value of eax from the stack.
- This ensures the macro doesn't accidentally change eax permanently.

ENDM - Marks the end of the macro definition.

What Happens When You Use It

Suppose you write: **mPutchar 'A'**

During assembly:

- The assembler prints: *"Expanding the mPutchar macro"* (thanks to ECHO).
- Then it replaces the macro call with the actual instructions (push eax, mov al, 'A', call WriteChar, pop eax).

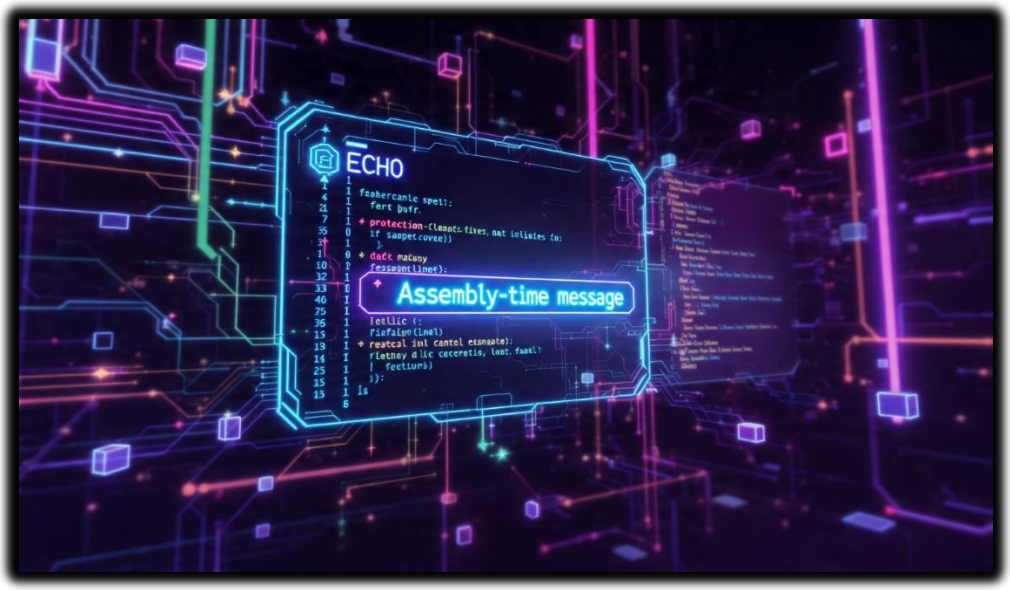
At runtime:

- The program executes those instructions and displays the character 'A' on the console.

So, you get two layers of feedback:

- One during assembly (ECHO message).
- One during execution (the character output).

ECHO = Assembly-time message tool



- It's like a **debugging print statement**, but for the assembler, not the running program.
- Helps track macro expansions, confirm flow, and leave notes.
- Doesn't affect the final machine code—only what you see while assembling.

LOCAL Directive in Macros

Problem with Labels in Macros:

- Macros often contain **labels** for jumps, loops, or internal references.
- If a macro is **invoked multiple times**, the **same label is redefined** each time.
- Redefining labels causes **assembler errors** due to duplicate names.

Solution – LOCAL Directive:

- Use the **LOCAL** directive to declare labels inside a macro.
- The assembler **generates a unique name** for each invocation of the macro.
- This allows **safe reuse of macros** with internal labels without conflicts.

Example:

```
MyMacro MACRO
    LOCAL loopStart
    mov eax, 0
loopStart:
    inc eax
    cmp eax, 10
    jl loopStart
ENDM
```

Or

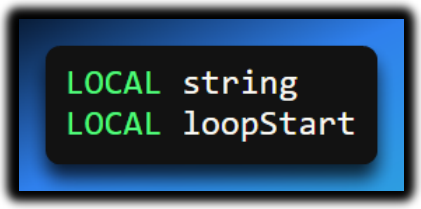
```
makeString MACRO text
LOCAL string
.data
    string BYTE text,0
ENDM
```

- loopStart is **local to the macro invocation**.
- Each time MyMacro is called, a **unique loopStart label** is created.
- Avoids **duplicate label errors** in programs with multiple macro calls.

Using LOCAL in Macros - Purpose of LOCAL:

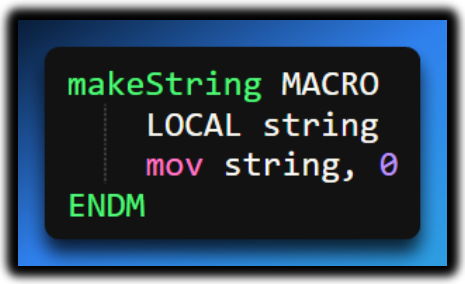
- Prevents **label redefinition errors** when a macro is invoked multiple times.
- Ensures that **each macro call gets unique labels or variables**.
- Makes macros **reusable, modular, and safer**.

How It Works: Place the **LOCAL** keyword before a label or variable inside the macro:



The assembler generates a **unique name** for each invocation of the macro.

Example – makeString Macro:



If makeString is called multiple times, each string variable is **unique**.

Avoids conflicts or overwriting memory in the program.

What Labels Can Be Local:

- **Code labels** (for jumps or loops)
- **Data labels/variables**

Interaction with ECHO:

- **ECHO** can be used to print **assembly-time messages** for debugging.
- **LOCAL** prevents label collisions, making macro debugging and tracing easier.

Key Benefits of Using LOCAL:

- Makes macros **reusable and modular**
- Ensures **safe internal jumps and loop labels**
- Simplifies **macro maintenance in large assembly programs**

Macros Containing Code and Data

- In assembly language, **macros are not limited to just instructions**.
- A macro can include both **executable code** (instructions like mov, add, jmp) and **data declarations** (variables, arrays, or constants).
- This allows you to create a **self-contained reusable block** that can initialize data and perform operations on it whenever the macro is invoked.
- For example, a macro could define a **temporary variable** and immediately perform calculations or manipulate that variable as part of its code.
- This feature makes macros extremely **flexible**, enabling both **modular code design** and **automatic data setup** without having to write separate variable declarations elsewhere.

Example: mWrite Macro

```
1375 mWrite MACRO text
1376     LOCAL string    ; Define a local label
1377     .data           ; Switch to the data section
1378     string BYTE text, 0   ; Define the string
1379
1380     .code           ; Switch back to the code section
1381     push edx        ; Push edx onto the stack
1382     mov edx, OFFSET string    ; Load the address of the string into edx
1383     call WriteString        ; Call the WriteString procedure with the address of the string
1384     pop edx         ; Restore the value of edx
1385 ENDM
```

Defines a reusable **function-like macro** that displays a string on the console.

Makes it easy to display messages **without writing repetitive code**.

Takes a single parameter: text — the string to display.

Local Label:

- Defines a **local label** (e.g., string) using the LOCAL directive.
- This label uniquely identifies the string in the **.data section** for each macro invocation.

Data Declaration:

- The string parameter (text) is associated with the local label in memory.

Code Section:

- Pushes the **address of the string** onto the stack.
- Calls the **WriteString procedure** to display the string on the console.
- Pops the address off the stack to **restore the previous state**.

Benefits:

- Each invocation gets its **own unique string label**.
- Avoids **label conflicts** if the macro is called multiple times.
- Allows you to **display strings easily** without manually declaring variables or writing repeated code.
- Makes macros **modular, reusable, and easier to maintain**.

Example Usage:

```
mWrite "Hello, World!"
mWrite "Assembly Macros are powerful!"
```

- Each call **expands inline** into code that pushes, calls WriteString, and pops.
- Each string gets a **unique memory label**, thanks to LOCAL.

Nested Macros

A macro invoked from another macro is called a nested macro. When the assembler's preprocessor encounters a call to a nested macro, it expands the macro in place.

This means that the parameters passed to the enclosing macro are passed directly to its nested macros. Example:

```
1390 mWriteln MACRO text
1391     mWrite text    ; Invoke the mWrite macro with the given text
1392     call Crlf     ; Call the Crlf procedure to add a new line
1393 ENDM
```

This mWriteln macro simplifies the process of writing a line of text. It calls the mWrite macro to display the provided text and then adds a new line by calling the Crlf procedure.

This macro defines a function called mWriteln() that writes a string to the console and appends an end of line. The macro invokes the mWrite() macro to display the string, and then calls the Crlf() procedure to append an end of line.

The text parameter is passed directly to the mWrite() macro. Example:

```
mWriteln "My Sample Macro Program"
```

When the assembler expands this statement, it will first expand the mWriteln() macro. This will result in the following code:

```
1406 mWrite "My Sample Macro Program"
1407 call Crlf
```

This code will then be expanded to the following instructions:

```
1412 push edx          ; Push the value in the edx register onto the stack
1413 mov edx, OFFSET ??0002   ; Load the address of label ??0002 into edx
1414 call WriteString  ; Call the WriteString procedure with the address in edx
1415 pop edx           ; Pop the value from the stack back into edx
1416 call Crlf         ; Call the Crlf procedure to add a new line
```

Tips for Creating Macros

:

Keep macros short and simple. This will make them easier to understand and maintain.

Use a modular approach when creating macros. This means breaking down complex macros into smaller, simpler macros. This will make the macros more reusable and flexible.

Use the LOCAL directive to avoid label redefinition errors.

Document your macros clearly. This will help other programmers understand how to use them.

Macros can be a powerful tool for simplifying and improving code. However, it is important to use them carefully and to follow the tips above to avoid problems.

Irvine32/64 library and Macros:

```
; Using the Book's Macro Library (32-Bit Mode Only)
; To enable the library, include the following line after your existing INCLUDE:
INCLUDE Macros.inc

; Macros in the Macros.inc Library:
; mDump - Displays a variable using its name and default attributes.
; mDumpMem - Displays a range of memory.
; mGotoxy - Sets the cursor position in the console window buffer.
; mReadString - Reads a string from the keyboard.
; mShow - Displays a variable or register in various formats.
; mShowRegister - Displays a 32-bit register's name and contents in hexadecimal.
; mWrite - Writes a string literal to the console window.
; mWriteSpace - Writes one or more spaces to the console window.
; mWriteString - Writes a string variable's contents to the console window.

; Example usage of the mDumpMem macro:
; mDumpMem OFFSET array, LENGTHOF array, TYPE array
; Displays the 'array' variable using its default attributes.
```

In summary, the "Macros.inc" library contains various macros that simplify common tasks in assembly programming, such as displaying variables, manipulating cursor positions, reading from the keyboard, and more.

You can enable this library by including it in your program, and then you can use the provided macros to streamline your code.

==========

Let's expand on them a bit:

mDumpMem macro

The **mDumpMem macro** is used to display a block of memory in the console window. It requires three arguments: the memory offset, the number of items to display, and the size of each memory component. The macro internally invokes the DumpMem library procedure, passing the three arguments to it (ESI, ECX, and EBX, respectively). Here's a cleaned-up version of the explanation:

```
1440 mDumpMem MACRO address:REQ, itemCount:REQ, componentSize:REQ
1441
1442   push ebx
1443   push ecx
1444   push esi
1445
1446   mov esi, address
1447   mov ecx, itemCount
1448   mov ebx, componentSize
1449
1450   call DumpMem
1451
1452   pop esi
1453   pop ecx
1454   pop ebx
1455
1456 ENDM
```

The mDumpMem macro displays a memory dump using the DumpMem procedure.

It takes 3 parameters:

- address - Offset of memory block to dump
- itemCount - Number of components to display
- componentSize - Size in bytes of each component

It avoids passing EBX, ECX, ESI to prevent corrupting registers used internally.

Example usage:

```
mDumpMem OFFSET array, LENGTHOF array, TYPE array
```

This would display the contents of the array memory block, with number of items and size based on the array's declared size and type.

It shows both the memory offset and hex dump of the values, with the componentSize determining the format.

So mDumpMem provides a convenient way to dump memory blocks for debugging.

==========

mDump macro

The mDump macro displays the address and hexadecimal contents of a variable.

It takes two parameters:

- varName - Name of the variable to dump
- useLabel - Optional label to display

The size and format match the variable's declared type.

If useLabel is passed a non-blank value, it will print the variable name.

```
1463 mDump MACRO varName:REQ, useLabel
1464
1465 call Crlf
1466
1467 IFNB <useLabel>
1468   mWrite "Variable name: &varName"
1469 ENDIF
1470
1471 mDumpMem OFFSET varName, LENGTHOF varName, TYPE varName
1472
1473 ENDM
```

The &varName substitution operator inserts the actual variable name into the string.

IFNB checks if useLabel was passed a non-blank value.

mDumpMem is called to print the hex dump using the variable's attributes.

So mDump provides a convenient way to quickly dump variables for debugging.

==========

mGotoxy macro

The mGotoxy macro positions the cursor in the console window.

It takes two BYTE parameters for the X and Y coordinates.

Avoid passing DH and DL to prevent register conflicts.

```
1477 mGotoxy MACRO X:REQ, Y:REQ
1478
1479   push edx
1480
1481   mov dh,Y
1482   mov dl,X
1483
1484   call Gotoxy
1485
1486   pop edx
1487
1488 ENDM
```

Register conflict example:

If DH and DL are passed as arguments, the expanded code would be:

```
1492 push edx
1493 mov dh,dl ; Y value overwritten
1494 mov dl,dh ; X value now incorrect
1495 call Gotoxy
1496 pop edx
```

DH is overwritten by DL before it can be copied to DL.

So the macro documentation warns not to pass DH/DL to avoid this problem.

In summary, mGotoxy sets the cursor position, but care must be taken with register arguments to prevent conflicts. The macro code/docs make this clear.

==========

mReadString macro

The mReadString macro reads keyboard input into a buffer.

It takes one parameter:

- • • **varName -** Name of the buffer to store the input string

Avoids using ECX and EDX internally to prevent register conflicts.

```
1500 mReadString MACRO varName:REQ
1501
1502   push ecx
1503   push edx
1504
1505   mov edx,OFFSET varName
1506   mov ecx,SIZEOF varName
1507
1508   call ReadString
1509
1510   pop edx
1511   pop ecx
1512
1513 ENDM
```

```
1517 firstName BYTE 30 DUP(?)
1518
1519 mReadString firstName
```

This would call ReadString to input a string from the keyboard into the firstName buffer.

So mReadString encapsulates the details of calling ReadString to simplify inputting strings.

==========

mShow macro

The mShow macro displays a register or variable's name and contents in different formats. It is useful for debugging.

mShow takes a register/variable name followed by format specifiers:

- **• • H - Hexadecimal**
- **D - Decimal (unsigned by default)**
- **I - Signed decimal**
- **B - Binary**
- **N - Newline**

You can combine multiple formats like "HDB" and add multiple newlines. The default is "HIN".

Display AX in multiple formats:

```
1524 mov ax,4096
1525 mShow AX          ; HIN (hex, signed decimal, newline)
1526 mShow AX,DBN      ; Decimal, binary, newline
```

```
1529 ;Output
1530 AX = 1000h = 4096d
1531 AX = 4096d = 00010000 00000000b
```

Display multiple registers on one line:

```
1540 mov ax,1
1541 mov bx,2
1542 mov cx,3
1543 mov dx,4
1544
1545 mShow AX,D
1546 mShow BX,D
1547 mShow CX,D
1548 mShow DX,DN
```

```
1552 ;Output
1553 AX = 1d BX = 2d CX = 3d DX = 4d
```

Display variable in decimal with newline:

```
1555 mydword DWORD ?
1556
1557 mShow mydword,DN
```

==========

mShowRegister macro

mShowRegister displays a 32-bit register's name and hexadecimal contents. It is useful for debugging.

It takes two parameters:

- • • **regName -** String to display as the register name
- • • **regValue -** The 32-bit register value

```
mShowRegister EBX, ebx
```

Displays:

EBX=7FFD9000

Implementation

The macro does the following:

- Declares a local string variable tempStr to hold the label
- Pushes EAX and EDX to preserve registers used internally
- Builds the label string with the name and '='
- Calls WriteString to display the label
- Pops EDX to restore it
- Moves the register value into EAX
- Calls WriteHex to display the hex value
- Pops EAX to restore it

Some key points:

- Local string variable avoids modifying caller's code
- Register pushing/popping prevents corruption
- WriteString and WriteHex handle the output
- Substitution inserts regName and regValue from caller

So mShowRegister encapsulates the details of displaying a register's name and value for debugging. The caller simply specifies the name and register.

==========

The mWriteSpace macro writes one or more spaces to the console window. It takes an optional integer parameter specifying the number of spaces to write. The default value is one.

The mWriteString macro writes the contents of a string variable to the console window. It takes a single parameter, which is the name of the string variable to write.

Here is a more detailed explanation of how these macros work:

mWriteSpace

The mWriteSpace macro defines a local label named spaces. This label is used to identify a string of spaces in the .data section of the program.

The macro then defines the code for the mWriteSpace() function. This code pushes the address of the spaces string onto the stack, calls the WriteString() function to display the string, and then pops the address of the stack.

Example:

```
mWriteSpace 5
```

When the assembler expands this statement, it will first expand the mWriteSpace() macro. This will result in the following code:

```
1571 push edx
1572 mov edx,OFFSET spaces
1573 call WriteString
1574 pop edx
```

This code will then be expanded to the following instructions:

```
1577 push edx
1578 mov edx,5
1579 rep movsb
1580 pop edx
```

The rep movsb instruction will copy 5 bytes from the source (the EDX register) to the destination (the console window). The source will be incremented by one after each byte is copied, and the destination will be incremented by one after each byte is copied.

==========

mWriteString

The mWriteString macro defines a function called mWriteString() that writes the contents of a string variable to the console window. The macro takes a single parameter, which is the name of the string variable to write.

The macro first saves the EDX register on the stack. This is because the WriteString() function uses the EDX register to store the address of the string to write.

The macro then loads the address of the string variable into the EDX register. Finally, the macro calls the WriteString() function and then pops the EDX register from the stack.

Example:

```
1583 .data
1584     str1 BYTE "Please enter your name: ",0
1585 .code
1586     mWriteString str1
```

When the assembler expands this statement, it will first expand the mWriteString() macro. This will result in the following code:

```
1590 push edx
1591 mov edx,OFFSET str1
1592 call WriteString
1593 pop edx
```

This code will then be expanded to the following instructions:

```
1595 push edx
1596 mov edx,OFFSET str1
1597 call WriteString
1598 pop edx
```

The WriteString() function will then write the contents of the str1 variable to the console window.

Conclusion

The mWriteSpace and mWriteString macros can be useful for simplifying code and making it more readable. For example, the following code:

```
WriteString("Please enter your name: ");
```

can be rewritten using the macros as follows:

```
mWriteString "Please enter your name: "
```

Here is the Wraps.asm program explained with the code rewritten for clarity:

Purpose:

Demonstrates use of wrapper macros for common procedures.

Macros used:

- mGotoxy - Set cursor position
- mWrite - Display formatted output
- mWriteString - Display a string
- mReadString - Input a string
- mDumpMem - Hex dump memory

```
1610 INCLUDE Irvine32.inc
1611 INCLUDE Macros.inc
1612 .data
1613     array DWORD 1,2,3,4,5,6,7,8
1614     firstName BYTE 31 DUP(?)
1615     lastName BYTE 31 DUP(?)
1616 .code
1617     main PROC
1618       mGotoxy 0,0       ; Position cursor
1619       ; Display heading
1620       mWrite <"Sample Macro Program",0dh,0ah>
1621       ; Input first name
1622       mGotoxy 0,5
1623       mWrite "Please enter your first name: "
1624       mReadString firstName
1625       ; Input last name
1626       call Crlf
1627       mWrite "Please enter your last name: "
1628       mReadString lastName
1629       ; Display name
1630       call Crlf
1631       mWrite "Your name is "
1632       mWriteString firstName
1633       mWriteSpace
1634       mWriteString lastName
1635       ; Display array
1636       call Crlf
1637       mDumpMem OFFSET array, LENGTHOF array, TYPE array
1638       exit
1639     main ENDP
1640     END main
```

Here are the review questions rewritten:

1. When a macro is invoked, does the assembler automatically insert CALL and RET instructions in the generated code? Explain.
2. Where is macro expansion handled - at assembly time or runtime?
3. What is the main advantage of using macros with parameters compared to macros without parameters?

4. Can a macro definition appear before or after the code that invokes it, as long as it is in the code segment? Explain.
5. How does replacing a procedure call with a macro invocation typically affect code size if the macro is called multiple times? Explain.
6. Can a macro contain data definitions like DW and DB? Explain.

Here are answers to the review questions:

1. False - CALL and RET instructions are not automatically inserted when a macro is invoked. The macro expansion is inserted directly into the code.
2. True - Macro expansion is handled by the assembler's preprocessor before the code is assembled.
3. Macros with parameters are more flexible since they can accept arguments from the caller. This avoids having to modify the macro definition for different uses.
4. True - As long as it is in the code segment, a macro can appear anywhere, before or after its usage. The assembler handles macros separately during preprocessing.
5. True - If a macro is invoked multiple times, the code will be expanded/duplicated each time, increasing the overall code size compared to calling a single procedure.
6. False - Macros can contain data definitions, though the data is inserted wherever the macro is expanded.

CONDITIONAL ASSEMBLY DIRECTIVES

Here is an organized summary of the key conditional assembly directives:

- • • **IF, ELSE, ENDIF -** Conditionally assemble code blocks based on a condition
- • • **IRP, IRPC -** Repeat a code block for each parameter value
- • • **REPT -** Repeat a block a specified number of times
- • • **WHILE, ENDW -** Repeat a block while a condition is true
- • • **EXITM -** Exit macro expansion early
- • • **LOCAL -** Declare local macro symbols

These allow macros to contain conditional logic, repetition, local symbols, early exit, and other advanced logic.

The important thing is that they work at assembly time, not runtime. The assembler evaluates them to determine what code to include or exclude in the final program. This makes macros much more powerful than simple text substitution.

Conditional-assembly directives

Conditional-assembly directives can be used to control the assembly of code based on certain conditions.

This can be useful for creating macros that are more flexible and adaptable. The general syntax for conditional-assembly directives is as follows:

```
1645 IF condition
1646     statements
1647
1648 [ELSE
1649     statements]
1650
1651 ENDIF
```

The IF directive checks the condition specified in its argument. If the condition is true, the statements following the IF directive are assembled.

If the condition is false, the statements following the ELSE directive are assembled (if one is present).

Here is a table of the most common conditional-assembly directives:

Directive	Description
IF *expression*	Permits assembly if the value of *expression* is true (nonzero). Possible relational operators are LT, GT, EQ, NE, LE, and GE.
IFB <*argument*>	Permits assembly if *argument* is blank. The argument name must be enclosed in angle brackets (<>).
IFNB <*argument*>	Permits assembly if *argument* is not blank. The argument name must be enclosed in angle brackets (<>).
IFIDN <*arg1*>,<*arg2*>	Permits assembly if the two arguments are equal (identical). Uses a case-sensitive comparison.
IFIDNI <*arg1*>,<*arg2*>	Permits assembly if the two arguments are equal. Uses a case-insensitive comparison.
IFDIF <*arg1*>,<*arg2*>	Permits assembly if the two arguments are unequal. Uses a case-sensitive comparison.
IFDIFI <*arg1*>,<*arg2*>	Permits assembly if the two arguments are unequal. Uses a case-insensitive comparison.
IFDEF *name*	Permits assembly if *name* has been defined.
IFNDEF *name*	Permits assembly if *name* has not been defined.
ENDIF	Ends a block that was begun using one of the conditional-assembly directives.
ELSE	Terminates assembly of the previous statements if the condition is true. If the condition is false, ELSE assembles statements up to the next ENDIF.
ELSEIF *expression*	Assembles all statements up to ENDIF if the condition specified by a previous conditional directive is false and the value of the current expression is true.
EXITM	Exits a macro immediately, preventing any following macro statements from being expanded.

Table 2:

Directive	Description
`.IF`	Assembles the following statements if the condition is true.
`.ELSE`	Assembles the following statements if the condition is false (and no previous `.ELSE` directive was true).
`.ENDIF`	Marks the end of a conditional-assembly block.

The following example shows how to use conditional-assembly directives to create a macro that can be used to print a message to the console:

```
; PrintMessage Macro
; Displays a message using the mWrite macro if the message is provided.
; Parameters:
;   - message: The message to be displayed
;
; Usage example:
; PrintMessage "Hello, world!"

PrintMessage MACRO message
    .IF message
        mWrite message
    .ENDIF
ENDM
```

This macro takes a single parameter, message, which is the message to be printed to the console.

The IF directive checks to see if the message parameter is empty. If it is not empty, the mWrite macro is used to print the message to the console.

Here is an example of how to use the PrintMessage macro:

```
PrintMessage "Hello, world!"
```

This will print the message "Hello, world!" to the console.

Conditional-assembly directives can be used to create macros that are more flexible and adaptable.

For example, the PrintMessage macro could be modified to support different types of messages, such as error messages, warning messages, and informational messages.

It is important to note that conditional-assembly directives are evaluated at assembly time, not at runtime.

This means that the condition specified in a conditional-assembly directive must be constant and cannot be evaluated based on runtime values.

Checking for Missing Arguments

To prevent errors caused by missing arguments in a macro, you can use conditional directives like IFB (if blank) or IFNB (if not blank) to check if an argument is provided. Here's a simplified explanation and code example:

```
1675 mWriteString MACRO string
1676     IFB <string>
1677         ECHO ----------------------------------------
1678         ECHO * Error: Parameter missing in mWriteString
1679         ECHO * (No code generated)
1680         ECHO ----------------------------------------
1681         EXITM
1682     ENDIF
1683
1684     push edx
1685     mov edx, OFFSET string
1686     call WriteString
1687     pop edx
1688 ENDM
```

In assembly language macros, it's important to handle missing arguments to avoid errors during macro expansion. A missing argument can lead to invalid instructions when the macro is expanded.

To address this issue, you can use conditional directives:

IFB (if blank) returns true if a macro argument is blank, meaning it's not provided. IFNB (if not blank) returns true if a macro argument is not blank, indicating it's provided.

Let's take an example where we have a macro called mWriteString. This macro displays a string using the WriteString procedure, but it needs a string argument to work correctly.

In this example, if the string argument is missing, the macro will display an error message during assembly and won't generate any code. This helps ensure that your macros are used correctly with the required arguments.

Default Argument Initializers

Macros can have default argument initializers.

This means that if a macro argument is missing when the macro is called, the default argument will be used instead.

The syntax for a default argument initializer is as follows:

```
paramname := <argument>
```

Spaces before and after the operators are optional.

For example, the following macro has a default argument initializer for the text parameter:

```
; mWriteln Macro
; This macro writes a string to the console followed by a carriage return and line feed (CRLF).

mWriteln MACRO text:=<" ">
    ; Check if a text argument is provided. If not, use a space as the default.
    IFB <text>
        mWrite " " ; Display a space if no text is provided
    ELSE
        mWrite text ; Display the provided text
    ENDIF

    ; Call the Crlf procedure to move to the next line.
    call Crlf
ENDM
```

This macro first checks if a text argument is provided. If it's missing, it uses a space as the default value.

Then, it calls the mWrite macro to display the text and adds a line break by invoking the Crlf procedure.

If the mWriteln macro is called with no arguments, the default argument initializer (" ") will be used for the text parameter.

This will cause the macro to print a single space followed by an end of line to the console.

Boolean Expressions

The assembler allows the following relational operators to be used in constant Boolean expressions:

```
<:      ;Less than
>:      ;Greater than
==:     ;Equal to
!=:     ;Not equal to
<=:     ;Less than or equal to
>=:     ;Greater than or equal to
```

These operators can be used in conjunction with the IF and other conditional directives to control the assembly of code.

For example, the following code uses the IF directive to check if the value of the x variable is greater than 10:

```
IF x > 10
    ; Assemble the following code if x is greater than 10
    ; ...
ENDIF
```

In this code snippet, the instructions within the IF block will be assembled only if the condition x > 10 is true.

If x is indeed greater than 10, the code within the IF block will be processed; otherwise, it will be skipped.

IF, ELSE, and ENDIF directives

The **IF, ELSE, and ENDIF directives** are used to control the assembly of code based on certain conditions.

The IF directive checks the condition specified in its argument. If the condition is true, the statements following the IF directive are assembled.

If the condition is false, the statements following the ELSE directive are assembled (if one is present).

The ENDIF directive marks the end of a conditional-assembly block.

The following is an example of how to use the IF, ELSE, and ENDIF directives:

```
1732 IF userAge > 18
1733     ; Assemble this code if the user's age is greater than 18
1734     mov edx, OFFSET adultMessage
1735 ELSE
1736     ; Assemble this code if the user's age is 18 or younger
1737     mov edx, OFFSET minorMessage
1738 ENDIF
```

In this example, the code chooses different messages to display based on whether the userAge is greater than 18 or not. If it's greater than 18, it displays the "adultMessage," otherwise, it displays the "minorMessage."

mGotoxyConst macro

The mGotoxyConst macro is an example of a macro that uses the IF, ELSE, and ENDIF directives to control the assembly of code.

The macro takes two parameters, X and Y, which must be constant expressions.

The macro checks to see if the values of X and Y are within the valid ranges of 0 to 79 and 0 to 24, respectively.

If either of the values is outside of the valid range, the macro displays a warning message and sets a flag.

If the flag is set, the macro exits. Otherwise, the macro assembles the code to move the cursor to the specified coordinates.

The following is an example of how to use the mGotoxyConst macro:

```
mGotoxyConst 10, 20
```

This will move the cursor to column 10, row 20.

The IF, ELSE, and ENDIF directives can be used to create more complex and flexible macros.

The IFIDN and IFIDNI Directives

```
1748 mReadBuf MACRO bufferPtr, maxChars
1749 ;
1750 ; Reads from the keyboard into a buffer.
1751 ; Receives: offset of the buffer, count of the maximum
1752 ; number of characters that can be entered. The
1753 ; second argument cannot be edx or EDX.
1754 ;-----------------------------------------------------
1755 IFIDNI <maxChars>,<EDX> ; Check if the second argument is EDX (case-insensitive)
1756     ECHO Warning: Second argument to mReadBuf cannot be EDX
1757     ECHO ************************************************
1758     EXITM ; Exit the macro
1759 ENDIF
1760
1761 push ecx
1762 push edx
1763 mov edx, bufferPtr ; Move the buffer offset to EDX
1764 mov ecx, maxChars ; Move the maximum character count to ECX
1765 call ReadString
1766 pop edx
1767 pop ecx
1768 ENDM
```

This code defines a macro mReadBuf that reads from the keyboard into a buffer. It checks if the second argument, maxChars, is equal to EDX in a case-insensitive manner using IFIDNI.

If it is, it displays a warning message and exits the macro. If not, it proceeds to read from the keyboard into the specified buffer.

IFIDNI:

IFIDNI is a conditional assembly directive.

It is used to check if two symbols (or macro parameter names) are equal in a case-insensitive manner.

In the code, it's used to compare maxChars with EDX to ensure that the second argument is not equal to EDX.

If the comparison is true, it means the second argument is EDX, and a warning message is displayed.

If the comparison is false, the code proceeds to read input into the buffer.

ECHO:

ECHO is used to write a message to the console during assembly. In the code, it's used to display a warning message when the second argument is EDX.

E

XITM:

EXITM is used to exit a macro.

In the code, if the second argument is EDX, the macro execution is halted using EXITM. push and pop:

These instructions are used to push and pop values onto and from the stack, respectively.

In the code, push ecx and push edx are used to save the values of registers ECX and EDX on the stack. This is done to protect the original values of these registers.

Later, pop edx and pop ecx are used to restore the original values of these registers before exiting the macro.

mov:

The mov instruction is used to move data between registers and memory locations.

In the code, mov edx, bufferPtr is used to load the address of the buffer into the EDX register, and mov ecx, maxChars is used to load the value of maxChars into ECX before calling ReadString.

Overall, the code ensures that the second argument passed to the mReadBuf macro is not EDX and issues a warning if it is.

If the argument is not EDX, the code continues to read input into the specified buffer. The use of conditional assembly directives and stack manipulation ensures proper execution of the macro.

Summing a matrix row

```
1783 mCalc_row_sum MACRO index, arrayOffset, rowSize, eltType
1784     push ebx
1785     push ecx
1786     push esi
1787
1788     mov eax, index
1789     mov ebx, arrayOffset
1790     mov ecx, rowSize
1791     shr ecx, (TYPE eltType / 2) ; Adjust rowSize for the element type
1792
1793     add ebx, eax ; Calculate the row offset
1794     mov eax, 0    ; Initialize accumulator
1795     mov esi, 0    ; Initialize column index
1796
1797 L1:
1798     IFIDNI <eltType>, <DWORD>
1799         mov edx, eltType PTR[ebx + esi*(TYPE eltType)]
1800     ELSE
1801         movzx edx, eltType PTR[ebx + esi*(TYPE eltType)]
1802     ENDIF
1803
1804     add eax, edx
1805     inc esi
1806     loop L1
1807
1808     pop esi
1809     pop ecx
1810     pop ebx
1811 ENDM
```

Here's what the macro does:

It takes four parameters: index, arrayOffset, rowSize, and eltType.

The rowSize is adjusted based on the eltType parameter, ensuring it represents the number of elements in each row.

The macro initializes the registers and sets up the required variables.

It calculates the row offset and initializes the accumulator.

Inside the loop labeled L1, it uses an IFIDNI conditional to check if the eltType is DWORD.

Depending on this condition, it reads the elements from memory correctly.

It accumulates the elements into eax and increments the column index.

The loop continues until all elements are processed.

Finally, the macro cleans up by popping the registers from the stack.

This simplified version of the macro calculates the sum of a row in an array of different element types, taking into account the size of each element type.

```
; Define the mCalc_row_sum macro
mCalc_row_sum MACRO index, arrayOffset, rowSize, eltType
    LOCAL L1

    ; Save registers
    push ebx
    push ecx
    push esi

    ; Set up required registers
    mov eax, index
    mov ebx, arrayOffset
    mov ecx, rowSize

    ; Calculate the row offset
    mul ecx             ; row index * row size
    add ebx, eax        ; row offset

    ; Prepare the loop counter
    shr ecx, (TYPE eltType / 2)  ; Calculate the number of elements in a row

    ; Initialize accumulator and column index
    mov eax, 0           ; Accumulator
    mov esi, 0           ; Column index

```

```
    L1:
    ; Check if eltType is DWORD
    IFIDNI <eltType>, <DWORD>
        mov edx, eltType PTR[ebx + esi*(TYPE eltType)]
    ELSE
        ; Assuming eltType is BYTE or WORD
        movzx edx, eltType PTR[ebx + esi*(TYPE eltType)]
    ENDIF

    add eax, edx          ; Add to accumulator
    inc esi               ; Move to the next column

    loop L1

    ; Restore registers
    pop esi
    pop ecx
    pop ebx

ENDM

```

```
.data
; Define data arrays
tableB BYTE 10h, 20h, 30h, 40h, 50h
RowSizeB = ($ - tableB)
tableW WORD 10h, 20h, 30h, 40h, 50h
RowSizeW = ($ - tableW)
tableD DWORD 10h, 20h, 30h, 40h, 50h
RowSizeD = ($ - tableD)
index DWORD ?

.code
; Call the mCalc_row_sum macro for different arrays and data types
mCalc_row_sum index, OFFSET tableB, RowSizeB, BYTE
mCalc_row_sum index, OFFSET tableW, RowSizeW, WORD
mCalc_row_sum index, OFFSET tableD, RowSizeD, DWORD
```

Certainly, here's an explanation of the provided code:

The code introduces a macro called mCalc_row_sum, which is designed to calculate the sum of a row in a two-dimensional array. This macro takes four parameters: index, arrayOffset, rowSize, and eltType.

Inside the macro, registers ebx, ecx, and esi are pushed onto the stack to ensure they are preserved and not affected by the macro's operations.

The index parameter represents the row index, arrayOffset is the offset of the array, rowSize indicates the number of bytes in each table row, and eltType specifies the array type, which can be BYTE, WORD, or DWORD.

The row offset is calculated by multiplying the rowSize with the index and adding it to arrayOffset. This is done to find the starting address of the row within the array.

To determine the number of elements in a row (whether they are bytes, words, or double words), the macro uses the eltType.

If it's DWORD, no scaling is required. If it's BYTE or WORD, the ecx register is shifted to the right by 1 or 2 bits, respectively, to adjust it to the number of elements in a row.

The accumulator (eax) and the column index (esi) are initialized to 0, as the macro iterates over the row.

Within a loop labeled L1, the macro loads an element from memory based on the element type (eltType). If the element type is DWORD, it uses a simple mov instruction.

If the element type is BYTE or WORD, it uses movzx to zero-extend the value.

The element value is added to the accumulator (eax), and the column index is incremented (esi) to move to the next element in the row.

The loop continues until all elements in the row have been processed.

After the loop, the registers are popped to restore their original values.

The .data section defines three different arrays (tableB, tableW, and tableD) with different data types (BYTE, WORD, and DWORD) and calculates the size of a row for each array.

The .code section demonstrates how to use the mCalc_row_sum macro with these arrays, specifying the appropriate data type for each call.

Overall, this macro allows you to easily calculate the sum of a row in a 2D array with different data types, making your code more versatile and readable.

SPECIAL OPERATORS FOR MACROS

In assembly language macros, there are four special operators that enhance their flexibility and usability:

Substitution Operator (&):

The substitution operator (&) is a valuable tool in macros. It helps resolve ambiguous references to parameter names within a macro.

For example, consider the mShowRegister macro, which displays the name and hexadecimal contents of a 32-bit register.

When calling this macro with a register name, like ECX, the macro produces output that includes the register name and its value.

```
1880 .code
1881 mShowRegister ECX
```

Creating a String Variable: If you want to create a string variable inside a macro that includes the register name, using just regName within the string won't work as expected. The preprocessor might treat it as a regular string and not replace it with the argument passed to the macro.

Using the Substitution Operator (&): To force the preprocessor to insert the macro argument (e.g., ECX) into the string literal, you can use the substitution operator &. This operator ensures that the macro argument is correctly incorporated into the string. Here's an example of how you can define the tempStr variable with the & operator:

```
1885 mShowRegister MACRO regName
1886 .data
1887 tempStr BYTE " &regName=",0
```

In summary, the substitution operator & is a powerful tool for resolving parameter references within macros, making them more versatile and efficient in handling various inputs.

Expansion Operator (%) in Macros

In assembly language, the expansion operator (%) plays a vital role in macros. It can be used in several ways to evaluate expressions and expand text macros:

Evaluating Expressions: When used with TEXTEQU, the % operator evaluates a constant expression and converts the result to an integer.

For example, if you have a variable count = 10, you can use % to calculate (5 + count) and get the integer result, which is then represented as text:

```
1895 count = 10
1896 sumVal TEXTEQU %(5 + count) ; This results in "15"
```

Flexibility for Passing Arguments: The % operator offers flexibility in passing arguments to macros. If a macro expects a constant integer argument, you can use the % operator to pass an integer expression. For example:

```
mGotoxyConst %(5 * 10), %(3 + 4)
```

In this case, the expressions within %(...) are evaluated to their integer values, which are then passed to the macro.

Expanding Macros on a Line: When the expansion operator (%) is the first character on a source code line, it instructs the preprocessor to expand all text macros and macro functions found on that line.

This can be useful for creating dynamic text during assembly. For example, to display the size of an array, you can use TEXTEQU to create a text macro, and then expand it on the next line:

```
1905 TempStr TEXTEQU %(SIZEOF array)
1906 %
1907 ECHO The array contains TempStr bytes
```

This approach allows for dynamic text generation during assembly.

Displaying Line Numbers: In some cases, macros can display the line number from which they were called to help with debugging.

For instance, the LINENUM text macro references **@LINE**, a predefined assembler operator that returns the current source code line number.

When an error condition is detected in the macro, the **line number can be displayed in an error message,** making it easier to identify and fix issues in the source code.

In summary, the expansion operator (%) is a versatile tool in macros, enabling the evaluation of expressions, dynamic text generation, and enhanced debugging by displaying line numbers in error messages.

Literal-Text Operator (< >)

The literal-text operator (< >) is a tool that allows you to group characters and symbols into a single text literal.

Its main purpose is to prevent the preprocessor from treating these characters as separate arguments or operators.

This is particularly useful when you have a string that contains special characters like commas, percent signs, ampersands, or semicolons.

These special characters could otherwise be misinterpreted by the preprocessor.

For example, consider the mWrite macro, which expects a string literal as its argument.

If you pass it the following string without using the literal-text operator:

```
mWrite "Line three", 0dh, 0ah
```

The preprocessor would consider this as three separate arguments. In this case, text after the first comma would be discarded because the macro expects only one argument. To prevent this, you can surround the string with the literal-text operator:

```
mWrite <"Line three", 0dh, 0ah>
```

By doing this, the preprocessor treats all text enclosed within the brackets as a single macro argument.

Literal-Character Operator (!)

The literal-character operator (!) serves a similar purpose to the literal-text operator.

It's used to instruct the preprocessor to treat a predefined operator as a regular character.

This is useful when you need to include special characters within a text literal without them being misinterpreted by the preprocessor.

For example, consider the definition of the BadYValue symbol:

```
BadYValue TEXTEQU <Warning: Y-coordinate is !> 24>
```

Here, the ! operator is used to prevent the > symbol from being treated as a text delimiter. It ensures that the entire text within the < > brackets is preserved as a single text literal.

Example: Using %, &, and ! Operators

To illustrate these operators, let's consider an example. Suppose you have a symbol called BadYValue, and you want to create a macro called ShowWarning.

This macro takes a text argument, encloses it in quotes, and then passes it to the mWrite macro. You can achieve this using the substitution operator (&) as follows:

```
1930 ShowWarning MACRO message
1931 mWrite "&message"
1932 ENDM
```

Now, when you invoke ShowWarning and pass it the expression %BadYValue, the % operator evaluates (dereferences) BadYValue, turning it into its string representation. The program then displays the warning message correctly:

```
ShowWarning %BadYValue
```

As a result, the program runs and displays the warning message as intended: "Warning: Y-coordinate is > 24."

In summary, the literal-text operator and the literal-character operator are tools to control how the preprocessor interprets and handles special characters within your assembly code, allowing you to maintain the desired structure and functionality of your macros and text literals.

Macro Functions

A macro function is similar to a regular macro procedure, but with a key difference: it always returns a constant value, either an integer or a string, using the EXITM directive. Let's look at an example to understand this better:

```
1940 IsDefined MACRO symbol
1941     IFDEF symbol
1942         EXITM <-1> ;; True
1943     ELSE
1944         EXITM <0> ;; False
1945     ENDIF
1946 ENDM
```

In this example, the IsDefined macro function checks whether a given symbol has been defined. If the symbol is defined, it returns true (represented by -1); otherwise, it returns false (0).

Invoking a Macro Function:

When you want to use a macro function, you need to enclose its argument list in parentheses.

For instance, let's call the IsDefined macro and pass it the symbol RealMode, which may or may not have been defined:

```
1950 IF IsDefined(RealMode)
1951     mov ax, @data
1952     mov ds, ax
1953 ENDIF
```

If the assembler has already encountered a definition of RealMode before this point in the assembly process, it will assemble the two instructions as shown.

You can also use the macro function within other macros, like this Startup macro:

```
1970 Startup MACRO
1971     IF IsDefined(RealMode)
1972         mov ax, @data
1973         mov ds, ax
1974     ENDIF
1975 ENDM
```

The IsDefined macro can be valuable when you're designing programs for different memory models. For example, you can use it to determine which include file to use based on whether Real Mode is defined.

Defining the RealMode Symbol

To use the IsDefined macro effectively, you need to define the RealMode symbol appropriately. There are a couple of ways to do this. One is to add the following line at the beginning of your program:

```
RealMode = 1
```

Alternatively, you can define symbols using the assembler's command-line options. For example, this command defines the RealMode symbol with a value of 1:

```
ML -c -DRealMode=1 myProg.asm
```

This allows you to control whether the RealMode symbol is defined or not when assembling your program.

HelloNew Program:

The HelloNew.asm program provided is an example that demonstrates the usage of the macros described.

It displays a message on the screen and chooses the appropriate include file based on whether RealMode is defined.

The program is adaptable to both 16-bit Real Mode and 32-bit Protected Mode.

In summary, macro functions return constant values, either integers or strings, and are useful for conditional assembly based on the existence of defined symbols, making your assembly code more versatile and flexible.

Questions

What is the purpose of the IFB directive?

The purpose of the IFB directive is to check if a macro argument is blank (empty). It returns true if the argument is empty and false if it contains any content. It's often used to handle cases where a macro expects specific arguments and needs to respond differently when arguments are missing.

What is the purpose of the IFIDN directive?

The purpose of the IFIDN directive is to perform a case-sensitive match between two symbols (or macro parameter names) and determine if they are equal. It returns true if the symbols are the same and false if they are different. This directive is helpful for ensuring that certain conditions are met within a macro.

Which directive stops all further expansion of a macro?

The directive that stops all further expansion of a macro is the EXITM directive. When EXITM is encountered within a macro, it halts the macro's execution, preventing any more macro expansion or code generation.

How is IFIDNI different from IFIDN?

The key difference between IFIDNI and IFIDN is their case sensitivity. IFIDNI performs a case-insensitive match, meaning it treats symbols or names as equal regardless of letter case. In contrast, IFIDN is case-sensitive and only returns true if the symbols or names match exactly, including their letter case.

What is the purpose of the IFDEF directive?

The purpose of the IFDEF directive is to check whether a particular symbol (usually a macro or a variable) has been defined earlier in the code. It returns true if the symbol is

defined and false if it is not. IFDEF is commonly used to conditionally include or exclude code blocks based on the existence of specific symbols in the assembly program.

REPEAT BLOCKS

MASM provides several looping directives for generating repeated blocks of statements: WHILE, REPEAT, FOR, and FORC.

These directives operate at assembly time and use constant values for loop conditions and counters.

WHILE Directive:

The WHILE directive repeats a block of statements as long as a specific constant expression remains true. It has the following syntax:

```
1987 WHILE constExpression
1988   statements
1989 ENDM
```

For example, you can use the WHILE directive to generate Fibonacci numbers within a specific range, like so:

```
1992 .data
1993     val1 = 1
1994     val2 = 1
1995     DWORD val1 ; First two values
1996     DWORD val2
1997     val3 = val1 + val2
1998
1999 WHILE val3 LT 0F0000000h
2000   DWORD val3
2001   val1 = val2
2002   val2 = val3
2003   val3 = val1 + val2
2004 ENDM
```

This code generates Fibonacci numbers and stores them as assembly-time constants until the value exceeds 0F0000000h.

REPEAT Directive:

The REPEAT directive repeats a statement block a fixed number of times at assembly time, based on an unsigned constant integer expression.

```
REPEAT constExpression
  statements
ENDM
```

It's used when you need to repeat a block of code a predetermined number of times, similar to the DUP directive.

REPEAT Directive Example: Creating an Array

In this example, we use the REPEAT directive to create an array of WeatherReadings. Each WeatherReadings struct contains a location string and arrays for rainfall and humidity readings. The loop repeats for a total of WEEKS_PER_YEAR times:

```
WEEKS_PER_YEAR = 52

WeatherReadings STRUCT
    location BYTE 50 DUP(0)
    REPEAT WEEKS_PER_YEAR
        LOCAL rainfall, humidity
        rainfall DWORD ?
        humidity DWORD ?
    ENDM
WeatherReadings ENDS
```

This code defines a structured array for recording weather readings over the course of a year.

FOR Directive:

The FOR directive repeats a statement block by iterating over a comma-delimited list of symbols. Each symbol in the list represents one iteration of the loop.

```
FOR parameter, <arg1, arg2, arg3, ...>
  statements
ENDM
```

It's useful when you want to perform a set of operations for each item in a list of symbols.

FOR Directive Example: Student Enrollment

In this example, the FOR directive is used to create multiple SEMESTER objects for student enrollment in different semesters. The loop iterates over a list of semester names and generates corresponding SEMESTER objects:

```
2065 .data
2066 FOR semName, <Fall2013, Spring2014, Summer2014, Fall2014>
2067     semName SEMESTER <>
2068 ENDM
```

This code generates SEMESTER objects with different names for each semester.

FORC Directive:

The FORC directive repeats a statement block by iterating over a string of characters. Each character in the string represents one iteration of the loop.

```
2020 FORC parameter, <string>
2021   statements
2022 ENDM
```

It's handy when you need to process a block of code for each character in a string.

Student Enrollment:

You can use the FOR directive to create multiple SEMESTER objects, each with a different name from a list of symbols. This can be useful for managing student enrollments over multiple semesters.

Character Lookup Table:

The FORC directive can be used to generate a character lookup table. In this example, a table of non-alphabetic characters is created by iterating through a string of special characters. These looping directives offer flexibility and structure for generating repetitive code in assembly language programs, making it easier to manage and control complex operations.

FORC Directive Example: Character Lookup Table

In this example, the FORC directive is used to create a character lookup table for non-alphabetic characters. Each character in the string is processed to generate a corresponding entry in the lookup table:

```
2070 Delimiters LABEL BYTE
2071 FORC code, <@#$%^&*!<!>>
2072     BYTE "&code"
2073 ENDM
```

This code generates a lookup table containing the ASCII values and corresponding characters for various special symbols.

In this example, we create a linked list data structure using the ListNode structure, which contains a data area (NodeData) and a pointer to the next node (NextPtr).

The program defines and populates multiple instances of ListNode objects within a loop to create a linked list.

Here's the revised and expanded code with added explanations:

```
2077 INCLUDE Irvine32.inc
2078
2079 ; Define the ListNode structure
2080 ListNode STRUCT
2081     NodeData DWORD ?
2082     NextPtr DWORD ?
2083 ListNode ENDS
2084
2085 TotalNodeCount = 15
2086 NULL = 0
2087 Counter = 0
2088
2089 .data
2090 LinkedList LABEL PTR ListNode
2091
2092 ; Use the REPEAT directive to create a linked list
2093 REPEAT TotalNodeCount
2094     Counter = Counter + 1
2095     ; Create a new ListNode with data and link to the next node
2096     ListNode <Counter, ($ + Counter * SIZEOF ListNode)>
2097 ENDM
2098
2099 ; Create a tail node to mark the end of the list
2100 ListNode <0, 0>
2101
2102 .code
2103 main PROC
2104     mov esi, OFFSET LinkedList  ; Initialize the pointer to the start of the list
```

```
2105
2106 NextNode:
2107     ; Check for the tail node (end of the list).
2108     mov eax, (ListNode PTR [esi]).NextPtr
2109     cmp eax, NULL
2110     je quit  ; If NextPtr is NULL, exit the loop
2111
2112     ; Display the node data.
2113     mov eax, (ListNode PTR [esi]).NodeData
2114     call WriteDec  ; Display the integer value
2115     call Crlf      ; Move to the next line for the next value
2116
2117     ; Get the pointer to the next node.
2118     mov esi, (ListNode PTR [esi]).NextPtr
2119     jmp NextNode
2120
2121 quit:
2122     exit
2123
2124 main ENDP
2125
2126 END main
```

Program to illustrate what we have learnt above:

```
2132 INCLUDE Irvine32.inc
2133
2134 ; Define a macro that generates bytes based on a list of values
2135 mGenerateBytes MACRO values
2136     LOCAL L1
2137     FOR val, <values>
2138         BYTE val
2139     ENDM
2140 ENDM
2141
2142 .data
2143 byteArray BYTE 0, 0, 0, 0
2144
2145 .code
2146 main PROC
2147     ; Question 7 - Macro Expansion
2148     ; a
2149     mGenerateBytes <100, 20, 30>
2150     mov eax, OFFSET byteArray
2151     call DisplayByteArray
2152
2153     ; b
2154     mGenerateBytes <AL, 20>
2155     mov eax, OFFSET byteArray
2156     call DisplayByteArray
2157
```

```
    ; c
    byteVal = 42
    countVal = 5
    mGenerateBytes <byteVal, countVal>
    mov eax, OFFSET byteArray
    call DisplayByteArray

    ; Question 8 - Linked List Scenario
    TotalNodeCount = 15
    NULL = 0
    Counter = 0
    LinkedList LABEL PTR ListNode

    REPEAT TotalNodeCount
        Counter = Counter + 1
        ListNode <Counter, ($ + SIZEOF ListNode)>
    ENDM

    ; Traverse and display the linked list
    mov esi, OFFSET LinkedList

```

```
2179 NextNode:
2180     mov eax, (ListNode PTR [esi]).NextPtr
2181     cmp eax, NULL
2182     je quit
2183
2184     mov eax, (ListNode PTR [esi]).NodeData
2185     call WriteDec
2186     call Crlf
2187
2188     mov esi, (ListNode PTR [esi]).NextPtr
2189     jmp NextNode
2190
2191 quit:
2192     exit
2193
2194 main ENDP
2195
2196 ; Helper function to display the contents of the byteArray
2197 DisplayByteArray PROC
2198     mov ecx, LENGTHOF byteArray
2199     mov esi, 0
2200     mov edx, OFFSET byteArray
2201
2202 DisplayLoop:
2203     mov al, [edx + esi]
2204     call WriteHex
2205     call Crlf
2206     inc esi
2207     loop DisplayLoop
2208     ret
2209 DisplayByteArray ENDP
```

2210 It's gonna be **END main** you know...👊