

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.....	70
SPECIAL OPERATORS FOR MACROS.....	86
REPEAT BLOCKS	92

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 "00000000"
0902     LastName BYTE 30 DUP(0)
0903     Years WORD 0
0904     SalaryHistory DWORD 0,0,0,0
0905 Employee ENDS
```

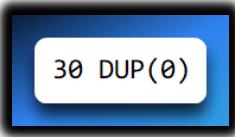
Structure Field Description

- **IdNum**
 - ⊕ A **byte-sized field**
 - ⊕ Initialized to the string **"00000000"**
 - ⊕ 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:



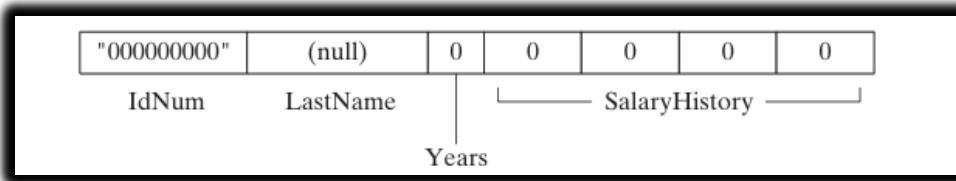
This creates an array of **30 bytes**, each initialized to **0**.

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)

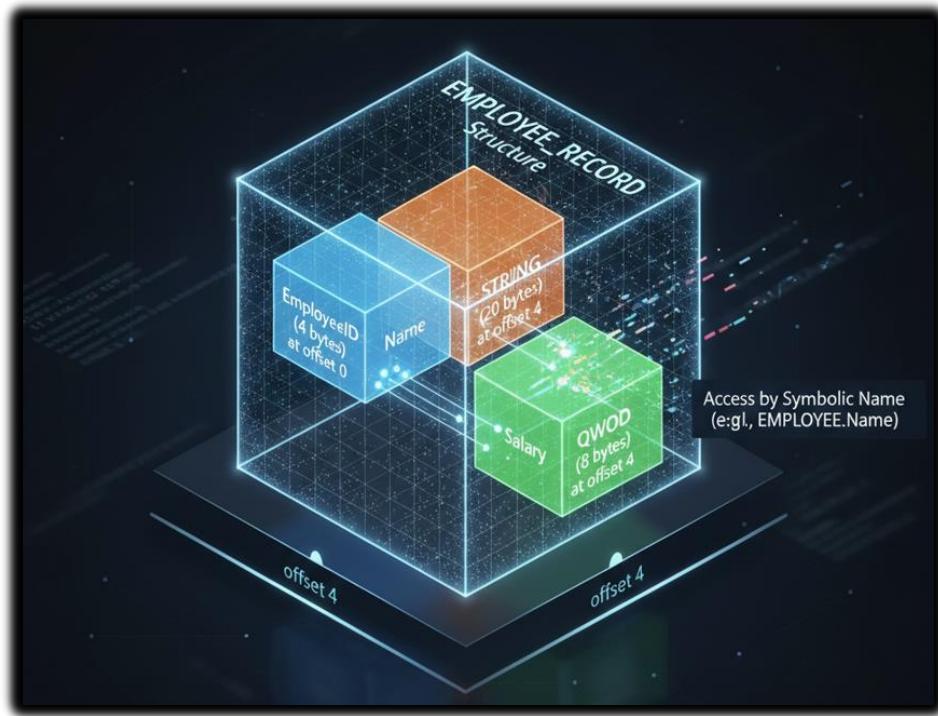
I. What Is a Structure?

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

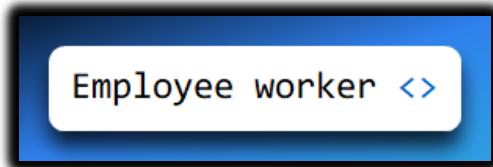
- **identifier**
Name of the structure variable (instance)
- **STRUCTURE_TYPE**
Name of the previously defined structure
- **initializer-list (optional)**
 - ⊕ Values used to initialize structure fields
 - ⊕ Must follow the **same order** as fields in the structure definition
 - ⊕ Enclosed in angle brackets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

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 (Δ 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

Δ 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)

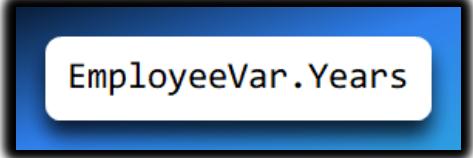
Example structure:

```
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:



EmployeeVar.Years

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

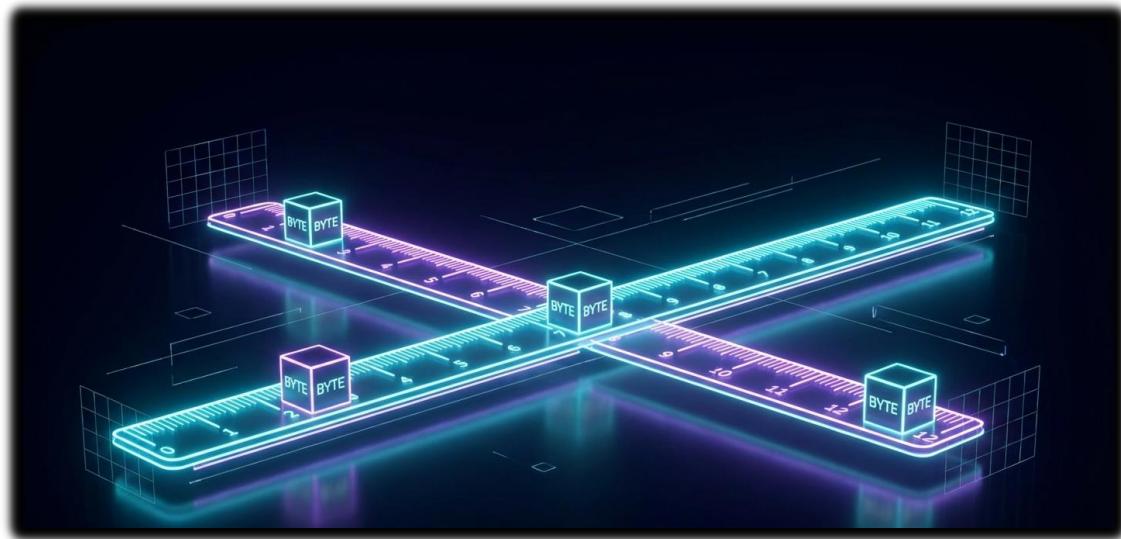
- Structures group related data into a single memory block
- Use **dot notation** for clarity and safety
- OFFSET is used when you need addresses
- Displacement access is possible but discouraged
- Assemblers compute field offsets — let them do it

Aligning Structure Fields

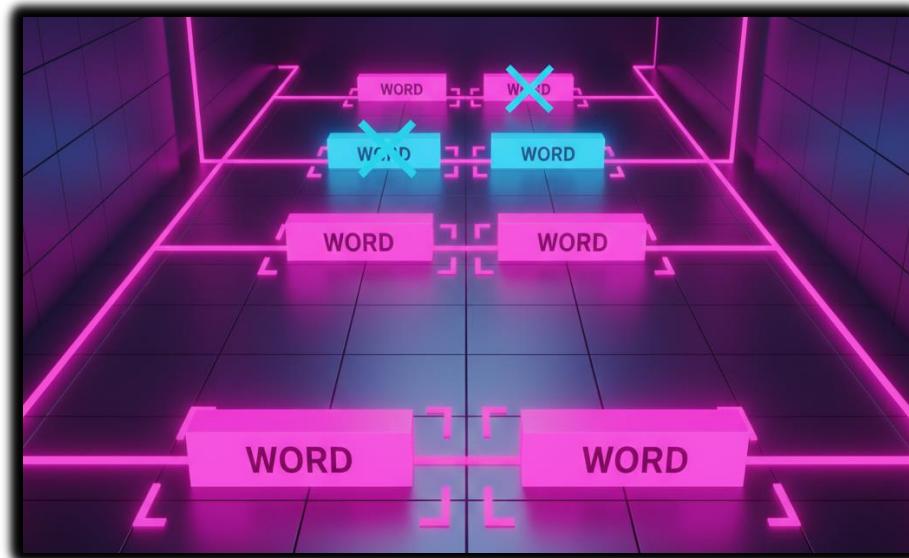
To achieve **optimal memory I/O performance**, structure members should be **aligned according to their data types**.

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

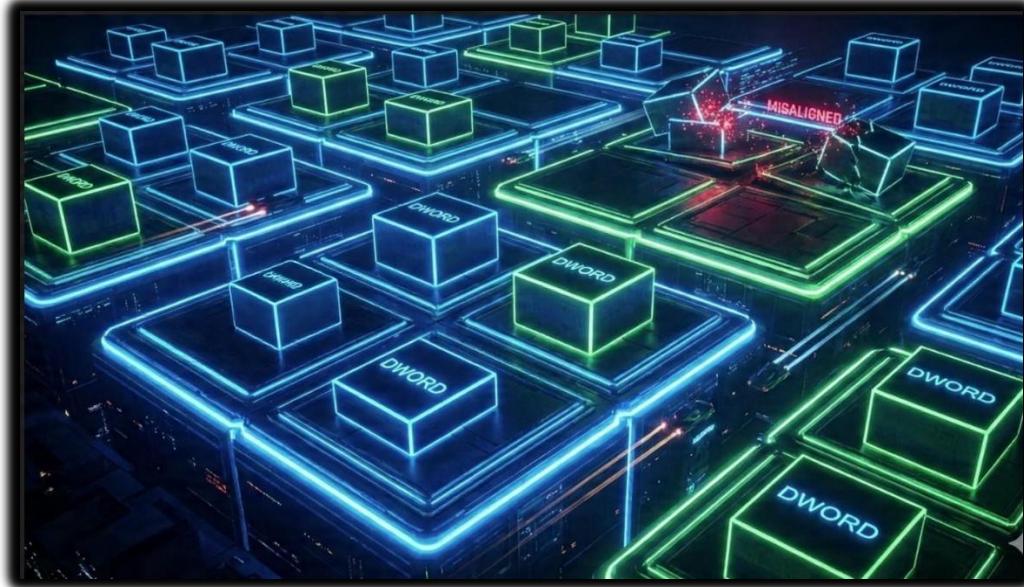
A **BYTE** member should be aligned on a **byte boundary**.



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

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

Floating-Point Alignment

II Floating-Point Data Types		
TYPE	SIZE	NATURAL ALIGNMENT
REAL4 32-bit float	4 bytes	Doubleword
REAL8 64-bit float	8 bytes	Quadword

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

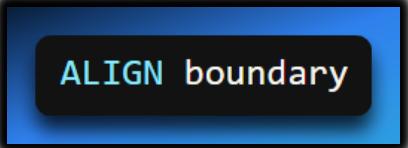
Manual Alignment (When Needed)

- 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:



ALIGN boundary

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)

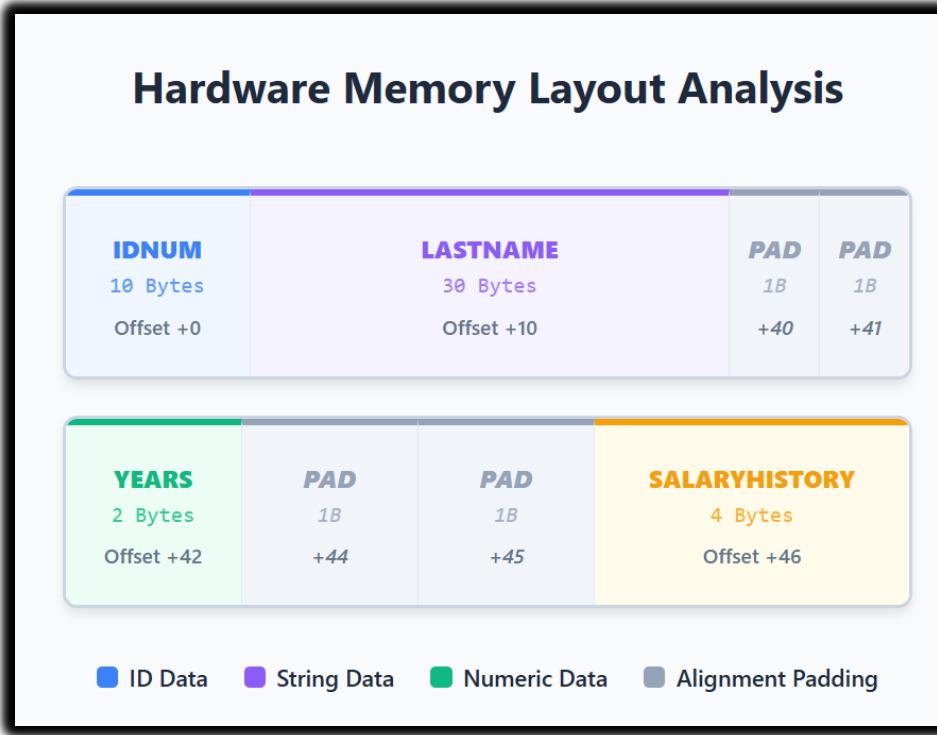
Field Sizes & Offsets

Aligned Memory Structure (Padded)			
FIELD NAME	SIZE	OFFSET	TYPE
IdNum	10 Bytes	+0	DATA
LastName	30 Bytes	+10	DATA
<i>Padding (for WORD)</i>	2 Bytes	+40	PADDING
Years	2 Bytes	+42	DATA
<i>Padding (for DWORD)</i>	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:

[base + index]

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:
 - ⊕ $\text{esi} = \text{TYPE Employee}$
 - ⊕ $\text{Offset} = 1 \times \text{structure size}$

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

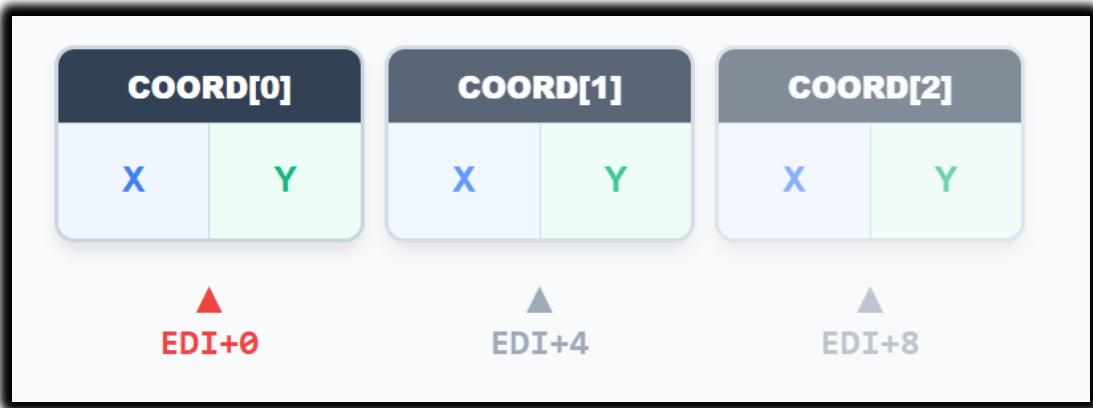
Write to Y field

Advance to next element:

```
EDI += sizeof(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		
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	<code>mov eax, [ebx]</code>	<code>mov eax, myArr[esi]</code>

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.

```
1035 .data
1036     ALIGN DWORD
1037     startTime DWORD ?
1038     ; align startTime
1039     emp Employee <>
1040     ; or: emp EmployeeBad <>
1041 .code
1042     call
1043     GetMSeconds
1044     ; get starting time
1045     mov
1046     startTime,eax
1047     mov
1048     ecx,0FFFFFFFh
1049     ; loop counter
1050     L1:
1051     mov
1052     emp.Years,5
1053     mov
1054     emp.SalaryHistory,35000
1055     loop
1056     L1
1057     call
1058     GetMSeconds
1059     ; get starting time
1060     sub
1061     eax,startTime
1062     call
1063     WriteDec
1064     ; 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

```
1149 ; Drunkard's Walk
1150 ; Drunkard's walk program simulates a professor's random path in an imaginary grid.
1151 ; The professor starts at coordinates 25, 25 and wanders around the immediate area.
1152
1153 INCLUDE Irvine32.inc
1154
1155 ; Constants
1156 WalkMax = 50      ; Maximum number of steps
1157 StartX = 25        ; Starting X-coordinate
1158 StartY = 25        ; Starting Y-coordinate
1159
1160 ; Define a structure to store the path and number of steps
1161 DrunkardWalk STRUCT
1162     path COORD WalkMax DUP(<0,0>) ; Array of COORD objects for the path
1163     pathsUsed WORD 0                ; Number of steps taken
1164 DrunkardWalk ENDS
1165
1166 ; Prototypes
1167 DisplayPosition PROTO currX:WORD, currY:WORD
1168
1169 .data
1170 aWalk DrunkardWalk <> ; Create an instance of the DrunkardWalk structure
1171
1172 .code
1173 main PROC
1174     mov esi,OFFSET aWalk ; Get the address of the aWalk structure
1175     call TakeDrunkenWalk ; Simulate the professor's walk
1176     exit
1177 main ENDP
1178 ;-----
1179 TakeDrunkenWalk PROC
1180 LOCAL currX:WORD, currY:WORD
1181
1182 ; Takes a walk in random directions (north, south, east, west).
1183 ; Receives: ESI points to a DrunkardWalk structure
1184 ; Returns: the structure is initialized with random values
1185 ; -----
1186
1187 pushad ; Preserve registers
1188
1189 ; Initialize EDI with the address of the path array
1190 mov edi,esi
1191 add edi,OFFSET DrunkardWalk.path
1192
1193 mov ecx,WalkMax ; Set loop counter
1194 mov currX,StartX ; Initialize current X-location
1195 mov currY,StartY ; Initialize current Y-location
1196
1197 Again:
1198 ; Insert the current location in the array.
1199 mov ax,currX
1200 mov (COORD PTR [edi]).X,ax ; Store X-coordinate
1201 mov ax,currY
1202 mov (COORD PTR [edi]).Y,ax ; Store Y-coordinate
1203
1204 INVOKE DisplayPosition, currX, currY ; Display the current position
1205
1206 mov eax,4 ; Choose a random direction (0-3)
1207 call RandomRange
1208
1209 .IF eax == 0
1210     ; North
1211     dec currY
1212 .ELSEIF eax == 1
1213     ; South
1214     inc currY
1215 .ELSEIF eax == 2
1216     ; West
1217     dec currX
1218 .ELSE
1219     ; East (EAX = 3)
1220     inc currX
1221 .ENDIF
1222
1223 add edi,TYPE COORD ; Move to the next COORD
1224 loop Again
1225
1226 Finish:
1227 mov (DrunkardWalk PTR [esi]).pathsUsed, WalkMax ; Set pathsUsed to WalkMax
1228 popad ; Restore registers
1229 ret
1230 TakeDrunkenWalk ENDP
1231
1232
```

```

1233 ;-----
1234 DisplayPosition PROC currX:WORD, currY:WORD
1235 ; Display the current X and Y positions.
1236 ;-----
1237
1238 .data
1239 commaStr BYTE ",",0 ; Comma string
1240
1241 .code
1242 pushad ; Preserve registers
1243
1244 ; Display the current X position
1245 movzx eax,currX
1246 call WriteDec
1247
1248 mov edx,OFFSET commaStr ; Load the comma string
1249 call WriteString
1250
1251 ; Display the current Y position
1252 movzx eax,currY
1253 call WriteDec
1254
1255 call Crlf ; Move to the next line
1256
1257 popad ; Restore registers
1258 ret
1259 DisplayPosition ENDP
1260
1261 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:



mPutchar 'A'

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

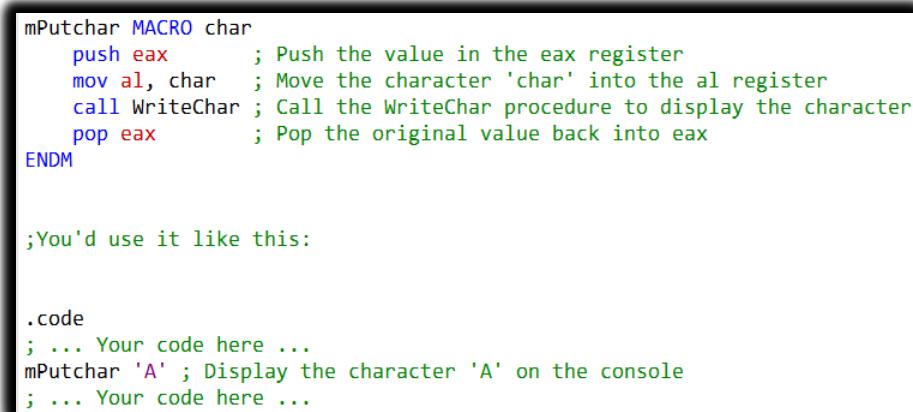
Key Advantages of Parameterized Macros:

- 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:

```
LOCAL string  
LOCAL loopStart
```

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

Example – makeString Macro:

```
makeString MACRO  
    LOCAL string  
    mov string, 0  
ENDM
```

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.

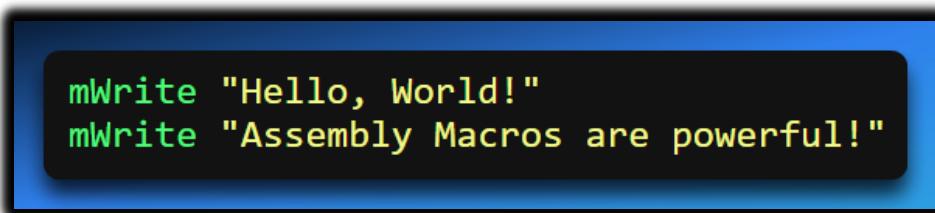
Code Section:

- 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:



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



Nested Macros in Assembly

Definition:

- A nested macro is a macro that is called from within another macro.
- When the assembler encounters a call to a nested macro, it expands the code in place.
- Parameters from the enclosing macro are passed directly to the nested macro.

Example – mWriteln Macro:

```
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
```

Simplifies writing a line of text by combining:

- mWrite (displays the string)
- Crlf (adds a newline)

Macro Invocation 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:

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

This code will then be expanded to the following instructions:

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

Then the mWrite macro expands to assembly instructions:

```
push edx
mov edx, OFFSET <string_label>
call WriteString
pop edx
call Crlf
```

How It Works:

1. **Pushes register** (edx) to save previous value.
2. **Loads string address** into the register.
3. **Calls WriteString** to display the text.
4. **Pops register** to restore original value.
5. **Calls Crlf** to append a newline.

Tips for Creating Effective Macros

- **Keep macros short and simple:**
 - ⊕ Easier to **read, understand, and maintain.**
- **Use a modular approach:**
 - ⊕ Break **complex operations** into smaller macros.
 - ⊕ Increases **reusability and flexibility.**
- **Use LOCAL directive:**
 - ⊕ Prevents **label redefinition errors** when macros are called multiple times.
- **Document macros clearly:**
 - ⊕ Helps other programmers **understand usage and purpose.**
- **Key Takeaway:**
 - ⊕ Macros are a **powerful tool** for simplifying repetitive code.
 - ⊕ Proper use ensures **clean, maintainable, and reusable assembly programs.**

Irvine32/64 library and Macros:

```
1420 ; Using the Book's Macro Library (32-Bit Mode Only)
1421 ; To enable the library, include the following line after your existing INCLUDE:
1422 INCLUDE Macros.inc
1423
1424 ; Macros in the Macros.inc Library:
1425 ; mDump - Displays a variable using its name and default attributes.
1426 ; mDumpMem - Displays a range of memory.
1427 ; mGotoxy - Sets the cursor position in the console window buffer.
1428 ; mReadString - Reads a string from the keyboard.
1429 ; mShow - Displays a variable or register in various formats.
1430 ; mShowRegister - Displays a 32-bit register's name and contents in hexadecimal.
1431 ; mWrite - Writes a string literal to the console window.
1432 ; mWriteSpace - Writes one or more spaces to the console window.
1433 ; mWriteString - Writes a string variable's contents to the console window.
1434
1435 ; Example usage of the mDumpMem macro:
1436 ; mDumpMem OFFSET array, LENGTHOF array, TYPE array
1437 ; Displays the 'array' variable using its default attributes.
```

Macros.inc Library – Overview

- **Purpose:**
 - ⊕ Provides a collection of **predefined macros** that simplify common assembly tasks.
 - ⊕ Helps programmers **avoid repetitive code** and work more efficiently.
- **Typical Macros Included:**
 - ⊕ **Display variables** on the console
 - ⊕ **Manipulate cursor positions**
 - ⊕ **Read input from the keyboard**
 - ⊕ **Dump memory blocks**
 - ⊕ And more—ready-to-use, modular macros for common tasks
- **How to Use:**
 - ⊕ Include the library in your program:



INCLUDE Macros.inc

mDumpMem Macro

Displays a **block of memory** in the console window in a readable format.

Useful for **debugging**, inspecting memory, or tracking variable contents.

Parameters:

- **Memory offset** – starting location of the memory block to display
- **Number of items** – how many items to show
- **Size of each item** – e.g., byte (1), word (2), or double word (4)

How It Works:

- Internally calls the **DumpMem procedure** from the library.
- Maps the macro parameters to **registers used by DumpMem**:
 - ⊕ ESI → memory offset
 - ⊕ ECX → number of items
 - ⊕ EBX → size of each item

Example Usage:

```
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
```

Or

```
mDumpMem ADDR array, 10, 4
```

Displays **10 DWORDs** starting from array in the console.

Benefits

- Provides a **quick, readable memory dump** without writing low-level assembly instructions.
- Makes **debugging and memory inspection** much easier.
- Avoids the risk of **register corruption**, as it internally manages EBX, ECX, and ESI.

Purpose

- Displays a **memory block** in the console using the **DumpMem procedure**.
- Shows both the **memory offsets** and **hex values** of the data.
- The output format depends on the **size of each memory component**.

Parameters

1. **address** – Offset of the memory block to dump.
2. **itemCount** – Number of items to display.
3. **componentSize** – Size in bytes of each component (1 = byte, 2 = word, 4 = DWORD).

How It Works

- Maps the macro arguments to **registers internally** for the DumpMem procedure:
 - ⊕ ESI ← address
 - ⊕ ECX ← itemCount
 - ⊕ EBX ← componentSize
- Displays a **formatted memory dump**, showing the **offset and hex values** of each component.
- Avoids modifying registers in your main program, so your code remains safe.

Example Usage

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

Displays the **contents of array**:

- ✚ OFFSET array → starting address
- ✚ LENGTHOF array → number of items
- ✚ TYPE array → size of each element (BYTE, WORD, DWORD)

Produces a **console-friendly hex dump** that helps you inspect the memory block easily.

mDumpMem is a **convenient, reusable macro** for debugging.

Automatically formats memory, handles registers safely, and saves you from writing repetitive code for memory inspection.

mDump macro

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

- **varName** – The name of the variable to dump.
- **useLabel** (*optional*) – A label to display alongside the output. If provided and non-blank, the macro prints the variable name.

Behavior:

- The output size and format automatically match the variable's declared type.
- Useful for debugging or inspecting variable contents at runtime.

```
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 output string.
- IFNB checks whether useLabel was passed a non-blank value.
- mDumpMem is called to print the hexadecimal dump using the variable's attributes.

Summary:

The mDump macro provides a simple and convenient way to inspect variables for debugging, automatically formatting output based on the variable's type and optionally labeling it for clarity.

mDump helps inspect variable values, sometimes you need precise control of the console display. That's where **mGotoxy** comes in.

mGotoxy Macro

The mGotoxy macro positions the cursor at a specified location in the console window.

Parameters:

- Two **BYTE** values representing the **X** (column) and **Y** (row) coordinates.

Usage Notes:

- Avoid passing **DH** and **DL** directly to prevent register conflicts.
- Useful for creating organized console output, menus, or formatted debugging displays.

```

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,d1 ; 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 user input from the keyboard and stores it in a specified buffer.

Parameter:

- **varName** – The name of the buffer where the input string will be stored.

Behavior and Notes:

- Safely reads a string from the keyboard and stores it in the buffer.
- Internally avoids using **ECX** and **EDX** registers to prevent conflicts with other code.
- Useful for interactive console programs that require text input.

```
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.

Summary:

mReadString provides a simple, safe way to capture keyboard input into a variable without interfering with other registers, making it ideal for debugging or console-based user input.

mShow macro

The mShow macro displays the name and contents of a register or variable in one or more selectable formats. It is primarily used for debugging and inspecting values quickly.

Parameters:

1. **Register or variable name** – The target value to display.
2. **Format specifiers (*optional*)** – A string defining how the value should be displayed:
 - **H** – Hexadecimal
 - **D** – Decimal (unsigned by default)
 - **I** – Signed decimal
 - **B** – Binary
 - **N** – Newline

Behavior:

- Multiple format specifiers can be combined, e.g., "HDB" will display the value in hex, decimal, and binary.
- Multiple Ns can be included to insert extra line breaks.
- If no format is specified, the default is "HIN" (hexadecimal, signed decimal, newline).

Example:

To display the contents of 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
```

mShow AX, "HDBN"

This will print AX's value in hexadecimal, decimal, and binary, followed by a newline.

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
```

Summary:

mShow provides a flexible, readable way to inspect registers or variables in multiple formats, making it an essential tool for debugging assembly programs.

mShowRegister macro

The mShowRegister macro displays a 32-bit register's name and its hexadecimal value. It is designed for debugging and inspection of register contents.

Parameters:

1. **regName** – The name of the register to display (string).
2. **regValue** – The 32-bit value of the register.

Example Output:



Implementation Details:

- Declares a local string variable tempStr to hold the label.
- Pushes **EAX** and **EDX** to preserve registers used internally.
- Builds the label string by combining the register name and '='.
- Calls WriteString to display the label.
- Pops **EDX** to restore its original value.
- Moves the register value into **EAX**.
- Calls WriteHex to display the hexadecimal value.
- Pops **EAX** to restore its original value.

Key Points:

- Local variables prevent interference with the caller's code.
- Register preservation avoids corruption of important registers.
- WriteString and WriteHex handle formatted output.
- Substitution operators ensure the correct regName and regValue are displayed.

Summary:

mShowRegister encapsulates all the steps needed to display a register's name and value. The caller only provides the name and value, making debugging simple and safe.

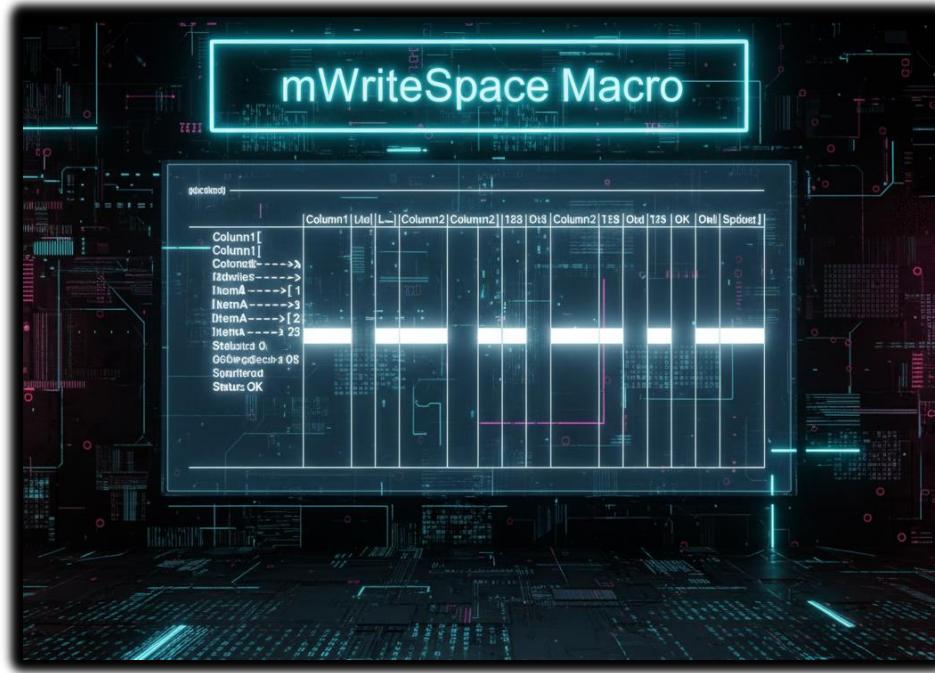
Transition to mWriteSpace and mWriteString

In addition to displaying variables and registers, sometimes you need to control console output directly. The next macros, mWriteSpace and mWriteString, handle formatted console output.

mWriteSpace Macro

- Writes one or more spaces to the console window.
- **Parameter (optional):** The number of spaces to write.
- **Default:** 1 space if no parameter is provided.

Use Case: Formatting output, aligning columns, or creating padding in the console.



✍ mWriteString Macro

- The **mWriteString macro** is like a shortcut for printing text to the console.
- Instead of writing all the low-level instructions every time, you just call this macro with the name of a string variable.
- It makes your code cleaner and less repetitive.

Parameter

- The macro takes **one parameter**: the **name of the string variable** you want to display.
- Example: If you have a string called msg, you can call the macro with msg to print it.

How It Works

- The macro relies on **low-level routines** (like WriteString) that actually handle the printing.
- When you invoke the macro:
 1. It pushes the address of the string variable onto the stack.
 2. Calls the WriteString routine to display the text.
 3. Pops the address off the stack to restore the program's state.
- This way, the macro acts like a wrapper around the printing routine.

Use Cases

- **Displaying messages:** Show instructions or prompts to the user.
- **Labels:** Print headings or identifiers in your program output.
- **Formatted text:** Align or organize console output neatly.
- **Debugging:** Print variable values or status messages while testing.

Benefits

- Saves time: No need to rewrite printing code.
- Cleaner code: Makes programs easier to read.
- Reusable: You can call it anywhere in your program.
- Safer: Reduces mistakes since the printing logic is centralized.

mWriteSpace Macro

The mWriteSpace macro writes one or more space characters to the console. It handles the details of preparing the string and calling the appropriate output routines.

How it Works:

1. Defines a **local label** named spaces in the .data section. This label points to a string of space characters.
2. Generates the code for the mWriteSpace() function:
 - o Pushes the address of the spaces string onto the stack.
 - o Calls the WriteString() function to display the spaces.
 - o Pops the address from the stack to restore the original state.
3. If a numeric parameter is passed (e.g., mWriteSpace 5), the macro ensures that exactly that number of spaces is written.

```
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
```

Macro expands to a call sequence. Low-level instructions executed internally:

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

Explanation:

- rep movsb copies the specified number of bytes (spaces) from the source to the destination.
- The source (EDX) increments after each byte copied, as does the destination (console output).
- Pushing and popping EDX ensures registers used internally are preserved.

Summary:

mWriteSpace provides a convenient and safe way to insert spaces into console output without manually writing loops or managing registers. You can specify the number of spaces, and the macro handles all underlying instructions for you.

mWriteString

The mWriteString macro writes the contents of a string variable to the console window. It abstracts the low-level instructions needed to pass the string to WriteString().

Parameter:

- **varName** – The name of the string variable to display.

How it Works:

1. Saves **EDX** on the stack to preserve the register, since WriteString() uses **EDX** to hold the string's address.
2. Loads the address of the string variable into **EDX**.
3. Calls the WriteString() function to display the string.
4. Restores **EDX** from the stack.

Example Usage:

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

Expanded macro code:

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
```

Instructions executed internally:

This code will then be expanded to the following instructions:

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

Summary:

mWriteString simplifies console output by wrapping the WriteString() call, preserving registers, and avoiding repetitive code. Using this macro makes programs cleaner and more readable.

Wraps.asm

Demonstrates use of wrapper macros for common procedures, making code easier to write and maintain.

```
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
```

Purpose:

- Simplify common console and debugging tasks.
- Make code more readable and maintainable.

Macros Used:

1. **Input a string:** mReadString
2. **Set cursor position:** mGotoxy
3. **Display formatted output:** mWrite
4. **Display a string:** mWriteString
5. **Hex dump memory:** mDumpMem

Example:

Instead of writing:

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

It can be rewritten using the macros as follows:

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

Or

```
mWriteString str1
```

This approach reduces boilerplate and ensures safe, consistent handling of registers and memory.

Key Takeaways from Wraps.asm:

- Wrapper macros encapsulate low-level details. They preserve registers and prevent conflicts.
- They allow combining multiple actions into simple, readable statements.
- Using these macros makes assembly programs cleaner, easier to debug, and more maintainable.

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

Assembly macros can include conditional logic, repetition, and local symbols using **conditional assembly directives**. These directives are evaluated **at assembly time**, not runtime, giving macros much more power than simple text substitution.

Common Directives:

- **IF, ELSE, ENDIF** – Conditionally assemble blocks of code based on a specified condition.
- **IRP, IRPC** – Repeat a block of code for each parameter value passed to the macro.
- **REPT** – Repeat a block a fixed number of times.
- **WHILE, ENDW** – Repeat a block while a condition evaluates as true.
- **EXITM** – Exit macro expansion early, skipping remaining code in the macro.
- **LOCAL** – Declare symbols that are local to the macro, preventing naming conflicts.

CONDITIONAL ASSEMBLY DIRECTIVES

Assembly macros can include conditional logic, repetition, and local symbols using conditional assembly directives. These directives are evaluated at assembly time, not runtime, giving macros much more power than simple text substitution.

COMMON DIRECTIVES:

- **IF, ELSE, ENDIF** – Conditionally assemble blocks of code based on a specified condition.
- **IRP, IRPC** – Repeat a block of code for each parameter value passed to the macro.
- **REPT** – Repeat a block a fixed number of times.
- **WHILE, ENDW** – Repeat a block while a condition evaluates as true.
- **EXITM** – Exit macro expansion early, skipping remaining code in the macro.
- **LOCAL** – Declare symbols that are local to the macro, preventing naming conflicts.

Key Points:

- These directives **operate at assembly time**, meaning the assembler decides which code to include or exclude in the final program.
- They allow macros to have **conditional logic, loops, early exits, and local symbols**, making macros far more versatile than simple text replacement.
- By combining these directives, complex, reusable, and safe macros can be created for a variety of programming tasks.

Conditional-assembly directives

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

Conditional-assembly directives are like "if-then" rules for your code. They tell the computer to only include certain parts of the code if a specific condition is true.

This makes your code more flexible because one set of instructions can change depending on what you need at that moment.

The Basic Structure

The general syntax follows this simple logic:

1. **Check a condition:** Is something true or false?
2. **Include code:** If it's true, the computer assembles that part of the code.
3. **Skip code:** If it's false, the computer ignores that part and moves on.

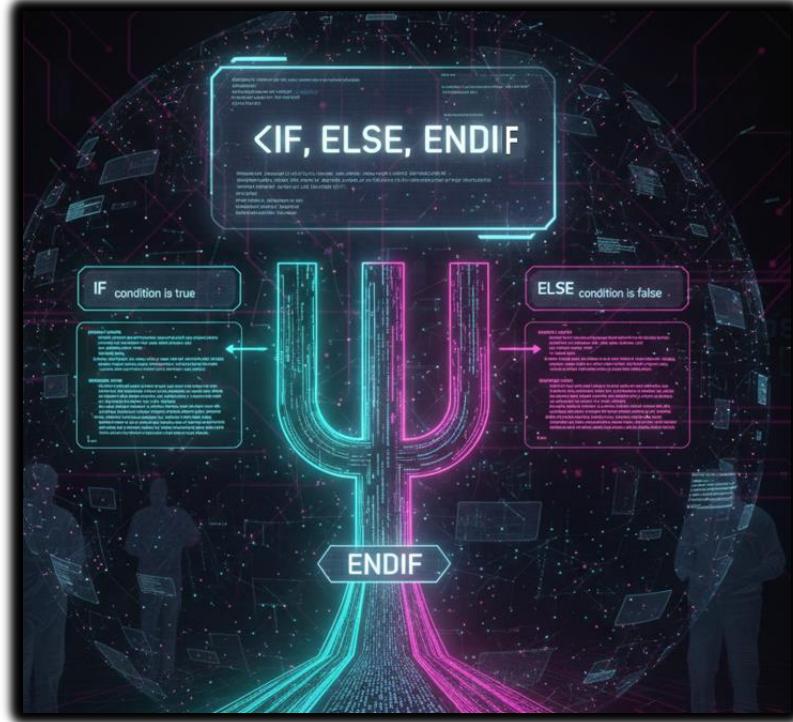
```
1645 IF condition
1646     statements
1647
1648 [ELSE
1649     statements]
1650
1651 ENDIF
```

Why use them?

- **Flexibility:** You can write one macro that works in many different situations.
- **Efficiency:** You only include the code you actually need, which keeps the final program clean.

IF directive

The IF directive is like a fork in the road for your code. It tells the computer which path to take based on a specific rule.



How it Works:

- **The Check:** The computer looks at the condition you gave it.
- **If True:** It assembles (includes) the code right after the IF.
- **If False:** It skips that part and assembles the code after the ELSE instead (if you provided one).

Here are the most common tools used to control these conditions:

Directive	Description
IF <i>expression</i>	Permits assembly if the value of <i>expression</i> is true (nonzero). Possible relational operators are LT, GT, EQ, NE, LE, and GE.
IFB < <i>argument</i> >	Permits assembly if <i>argument</i> is blank. The argument name must be enclosed in angle brackets (<>).
IFNB < <i>argument</i> >	Permits assembly if <i>argument</i> is not blank. The argument name must be enclosed in angle brackets (<>).
IFIDN < <i>arg1</i> >,< <i>arg2</i> >	Permits assembly if the two arguments are equal (identical). Uses a case-sensitive comparison.
IFIDNI < <i>arg1</i> >,< <i>arg2</i> >	Permits assembly if the two arguments are equal. Uses a case-insensitive comparison.
IFDIF < <i>arg1</i> >,< <i>arg2</i> >	Permits assembly if the two arguments are unequal. Uses a case-sensitive comparison.
IFDIFI < <i>arg1</i> >,< <i>arg2</i> >	Permits assembly if the two arguments are unequal. Uses a case-insensitive comparison.
IFDEF <i>name</i>	Permits assembly if <i>name</i> has been defined.
IFNDEF <i>name</i>	Permits assembly if <i>name</i> 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 <i>expression</i>	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:

```
1655 ; PrintMessage Macro
1656 ; Displays a message using the mWrite macro if the message is provided.
1657 ; Parameters:
1658 ; - message: The message to be displayed
1659 ;
1660 ; Usage example:
1661 ; PrintMessage "Hello, world!"
1662
1663 PrintMessage MACRO message
1664     .IF message
1665         mWrite message
1666     .ENDIF
1667 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 make macros **more flexible and adaptable**.

Example: A PrintMessage macro can use these directives to support **different message types** (e.g., error, warning, informational).

Conditional-assembly directives are **evaluated at assembly time**, not at runtime.

The **condition must be constant** and **cannot depend on runtime values**.

Checking for Missing Arguments

Purpose: Prevent errors caused by missing arguments in a macro.

How: Use conditional assembly directives:

- ⊕ **IFB** – Checks if an argument is blank (missing).
- ⊕ **IFNB** – Checks if an argument is not blank (provided).

Ensures macros handle optional or required parameters safely.

Example: Can wrap macro code in IFB/IFNB blocks to verify arguments before using them.

```
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
```

- Macro: mWriteString displays a string using the WriteString procedure.
- If the string argument is **missing**, the macro can:
 - ⊕ Display an **error message** during assembly.
 - ⊕ **Prevent code generation**, avoiding invalid instructions.

Benefit: Ensures macros are **used correctly** with the required arguments, improving safety and maintainability.

Default Argument Initializers

Macros can define **default values for arguments**, called **default argument initializers**.

This feature allows a macro to work even if the caller omits one or more arguments.

If an argument is **not provided** when the macro is called, the **default value** is automatically used.

This prevents errors due to missing arguments and allows macros to be more **flexible and reusable**.

Default arguments are evaluated at **assembly time**, just like all other macro logic.

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

Spaces before and after the operators are optional.

Or

```
MacroName Arg1:DefaultValue1, Arg2:DefaultValue2
```

- Arg1 and Arg2 are macro parameters.
- DefaultValue1 and DefaultValue2 are used if the caller does **not** provide values.

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

```
1697 ; mWriteln Macro
1698 ; This macro writes a string to the console followed by a carriage return and line feed (CRLF).
1699
1700 mWriteln MACRO text:=<" ">
1701     ; Check if a text argument is provided. If not, use a space as the default.
1702     IFB <text>
1703         mWrite " " ; Display a space if no text is provided
1704     ELSE
1705         mWrite text ; Display the provided text
1706     ENDIF
1707
1708     ; Call the Crlf procedure to move to the next line.
1709     call Crlf
1710 ENDM
```

Display text followed by a newline, with a default value if no text is provided.

- Checks if a **text argument** is provided.
- If missing, uses a **space " "** as the default value.
- Calls the mWrite macro to display the text.
- Calls the Crlf procedure to add a **line break**.

Calling mWriteln with no arguments prints a **single space** followed by an **end-of-line**.

Ensures the macro works even if no argument is provided, using a **safe default**.

Or another example:

```
; Define a macro with a default argument
PrintMessage Msg:"Hello, World!"

; Call with no arguments → uses default
PrintMessage

; Call with an argument → overrides default
PrintMessage Msg:"Custom message"
```

- In the first call, Msg is **not provided**, so "Hello, World!" is used.
- In the second call, the argument "Custom message" **overrides the default**.

Benefits:

- Makes macros more **robust** and **user-friendly**.
- Reduces the need for repetitive argument checking (IFB/IFNB) in simple cases.
- Supports creating **versatile macros** that work with minimal input.

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:

```
1724 IF x > 10
1725     ; Assemble the following code if x is greater than 10
1726     ; ...
1727 ENDIF
```

- Instructions within an IF block are **assembled only if the condition is true** (e.g., IF x > 10).
- If the condition is false, the code is **skipped during assembly**, not at runtime.

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** and an **ELSE** block is present, the ELSE statements are assembled instead.

The ENDIF directive marks the **end of the conditional 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
```

IF, ELSE, ENDIF are evaluated at **assembly time**, not runtime.

Example: Conditional Assembly in mGotoxyConst

Purpose: Move the cursor to specified coordinates while checking that the coordinates are valid.

Parameters: X and Y – must be **constant expressions**.

Behavior:

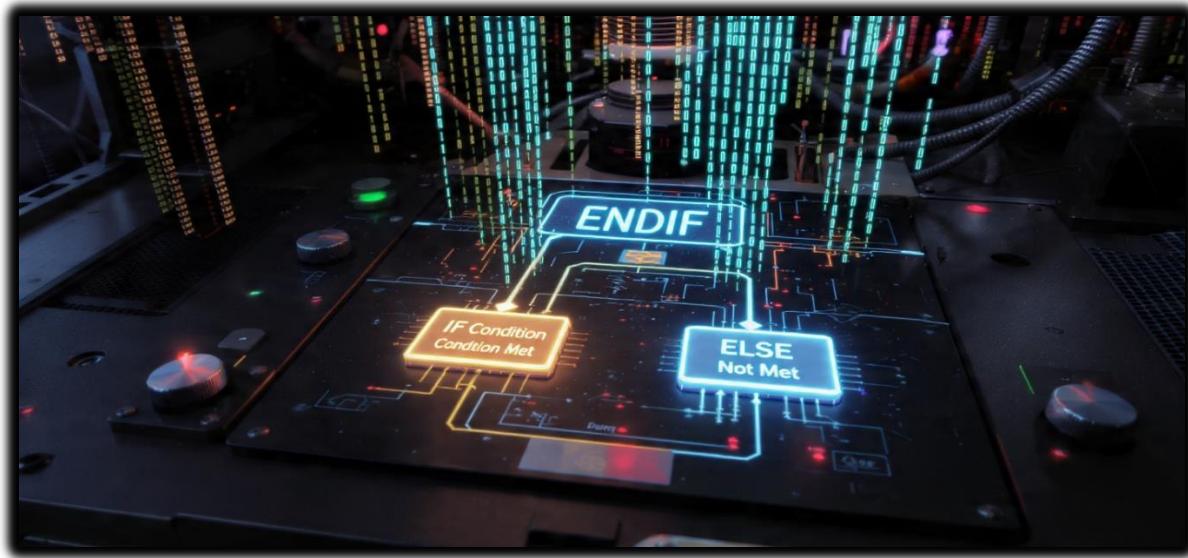
- Checks if X is between 0 and 79, and Y is between 0 and 24.
- If either value is out of range:
 - ⊕ Displays a **warning message**.
 - ⊕ Sets a **flag** and exits the macro.
- If values are valid:
 - ⊕ Assembles code to move the cursor to the specified coordinates.

Example Usage:

```
mGotoxyConst 10, 20 ; Moves cursor to column 10, row 20
```

Key Point:

Using IF, ELSE, and ENDIF inside a macro allows for flexible argument validation and conditional assembly of code, enabling more robust macros.



The IFIDN and IFIDNI Directives

mReadBuf Macro Explanation

```
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
```

Purpose:

- Reads keyboard input into a specified buffer.
- Checks that the second argument (maxChars) is **not the same as EDX**, preventing register conflicts.
- If the argument is invalid, warns the programmer and exits the macro.

How it Works:

1. IFIDNI – Case-insensitive comparison:

- ⊕ Compares maxChars with EDX.
- ⊕ If they are equal (ignoring case), the macro:
 - Displays a **warning message** using ECHO.
 - Exits the macro using EXITM.

2. ECHO – Assembly-time messaging:

- ⊕ Prints warnings or messages to the console **during assembly**, not at runtime.

3. EXITM – Early macro exit:

- ⊕ Stops macro expansion immediately if a problem is detected.

4. push and pop – Register preservation:

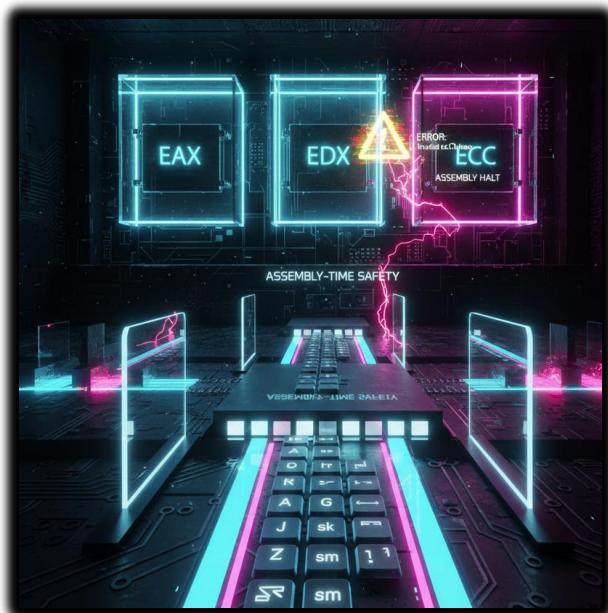
- ⊕ push ecx / push edx saves the original register values on the stack.
- ⊕ After using ECX and EDX internally, pop edx / pop ecx restores the original values.
- ⊕ Prevents **register corruption** in the calling code.

5. mov – Load registers:

- ⊕ mov edx, bufferPtr → loads the buffer address into EDX.
- ⊕ mov ecx, maxChars → loads the max number of characters into ECX.
- ⊕ These are used by the ReadString routine to read input.

Summary:

- The macro **ensures safety** by checking arguments and preserving registers.
- If maxChars is invalid (EDX), it **warns and exits** at assembly time.
- Otherwise, it **reads keyboard input** into the buffer safely.
- Combines **conditional assembly, stack manipulation, and register setup** for robust macro behavior.



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
```

- The macro takes **four parameters**: index, arrayOffset, rowSize, and eltType.
- rowSize is adjusted based on eltType so it represents the **number of elements per row**, not bytes.
- The macro initializes registers and sets up required variables.
- It calculates the **row offset** and initializes the accumulator (EAX).
- Inside loop **L1**, an IFIDNI conditional checks whether eltType is DWORD.
- Based on this check, elements are read from memory using the **correct size**.
- Each element is **added to EAX**, and the column index is incremented.
- The loop continues until **all elements in the row are processed**.
- After completion, the macro **restores registers** by popping them from the stack.
- **Overall:** The macro computes the **sum of a row** in an array while correctly handling **different element types and sizes**.

```

1815 ; Define the mCalc_row_sum macro
1816 mCalc_row_sum MACRO index, arrayOffset, rowSize, eltType
1817     LOCAL L1
1818
1819     ; Save registers
1820     push ebx
1821     push ecx
1822     push esi
1823
1824     ; Set up required registers
1825     mov eax, index
1826     mov ebx, arrayOffset
1827     mov ecx, rowSize
1828
1829     ; Calculate the row offset
1830     mul ecx          ; row index * row size
1831     add ebx, eax      ; row offset
1832
1833     ; Prepare the loop counter
1834     shr ecx, (TYPE eltType / 2) ; Calculate the number of elements in a row
1835
1836     ; Initialize accumulator and column index
1837     mov eax, 0          ; Accumulator
1838     mov esi, 0          ; Column index
1839
1840 L1:
1841     ; Check if eltType is DWORD
1842     IFIDNI <eltType>, <DWORD>
1843         mov edx, eltType PTR[ebx + esi*(TYPE eltType)]
1844     ELSE
1845         ; Assuming eltType is BYTE or WORD
1846         movzx edx, eltType PTR[ebx + esi*(TYPE eltType)]
1847     ENDIF
1848
1849     add eax, edx        ; Add to accumulator
1850     inc esi             ; Move to the next column
1851
1852     loop L1
1853
1854     ; Restore registers
1855     pop esi
1856     pop ecx
1857     pop ebx
1858
1859 ENDM
1860
1861 .data
1862 ; Define data arrays
1863 tableB BYTE 10h, 20h, 30h, 40h, 50h
1864 RowSizeB = ($ - tableB)
1865 tableW WORD 10h, 20h, 30h, 40h, 50h
1866 RowSizeW = ($ - tableW)
1867 tableD DWORD 10h, 20h, 30h, 40h, 50h
1868 RowSizeD = ($ - tableD)
1869 index DWORD ?
1870
1871 .code
1872 ; Call the mCalc_row_sum macro for different arrays and data types
1873 mCalc_row_sum index, OFFSET tableB, RowSizeB, BYTE
1874 mCalc_row_sum index, OFFSET tableW, RowSizeW, WORD
1875 mCalc_row_sum index, OFFSET tableD, RowSizeD, DWORD

```

mCalc_row_sum Macro Explanation

Introduces a macro named **mCalc_row_sum** that calculates the **sum of a row** in a two-dimensional array.

The macro takes **four parameters**:

- index – row index
- arrayOffset – starting address of the array
- rowSize – number of bytes per row
- eltType – element type (BYTE, WORD, or DWORD)

Registers **EBX, ECX, and ESI** are pushed onto the stack to preserve their original values.

The **row offset** is calculated by multiplying rowSize by index and adding it to arrayOffset, yielding the starting address of the target row.

The macro determines the **number of elements per row** based on eltType:

- DWORD → no scaling required
- WORD → shift ECX right by 1
- BYTE → shift ECX right by 2

The accumulator (**EAX**) and column index (**ESI**) are initialized to 0.

Inside loop **L1**:

- An element is loaded from memory based on eltType.
- DWORD elements use mov.
- BYTE and WORD elements use movzx to zero-extend the value.
- The value is added to **EAX**.
- The column index (**ESI**) is incremented.

The loop continues until **all elements in the row are processed**.

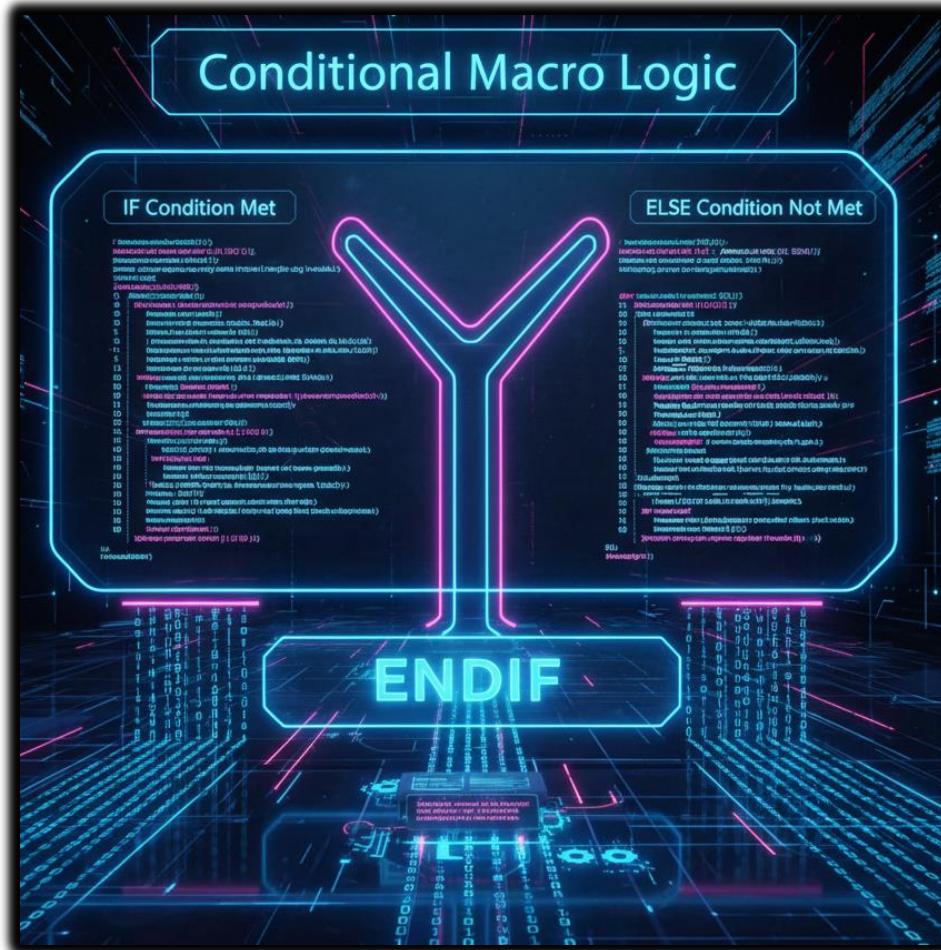
After the loop, registers are **popped** to restore their original values.

The **.data** section defines three arrays:

- tableB (BYTE)
 - tableW (WORD)
 - tableD (DWORD)
 - Each includes a calculated row size.

The **.code section** demonstrates calling mCalc_row_sum with each array and the appropriate element type.

The macro provides a **flexible and reusable way** to compute row sums in 2D arrays of varying data types while preserving register safety and readability.



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.

```
2008 REPEAT constExpression  
2009     statements  
2010 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:

```
2050 WEEKS_PER_YEAR = 52
2051
2052 WeatherReadings STRUCT
2053     location BYTE 50 DUP(0)
2054     REPEAT WEEKS_PER_YEAR
2055         LOCAL rainfall, humidity
2056         rainfall DWORD ?
2057         humidity DWORD ?
2058     ENDM
2059 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.

```
2015 FOR parameter, <arg1, arg2, arg3, ...>
2016     statements
2017 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
```

```
2158 ; c
2159 byteVal = 42
2160 countVal = 5
2161 mGenerateBytes <byteVal, countVal>
2162 mov eax, OFFSET byteArray
2163 call DisplayByteArray
2164
2165 ; Question 8 - Linked List Scenario
2166 TotalNodeCount = 15
2167 NULL = 0
2168 Counter = 0
2169 LinkedList LABEL PTR ListNode
2170
2171 REPEAT TotalNodeCount
2172     Counter = Counter + 1
2173     ListNode <Counter, ($ + SIZEOF ListNode)>
2174 ENDM
2175
2176 ; Traverse and display the linked list
2177 mov esi, OFFSET LinkedList
2178
```

```
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... 