Contents

[STRUCTURES 2](#_Toc220526585)

[INDIRECT AND INDEXED OPERANDS STRUCTS 20](#_Toc220526586)

[PERFORMANCE OF ALIGNED AND MISALIGNED STRUCTS 27](#_Toc220526587)

[STRUCTURES CONTAINING STRUCTURES 31](#_Toc220526588)

[UNIONS IN ASSEMBLY 38](#_Toc220526589)

[MACROS IN ASSEMBLY 40](#_Toc220526590)

[ECHO AND LOCAL DIRECTIVES 42](#_Toc220526591)

[CONDITIONAL ASSEMBLY DIRECTIVES 56](#_Toc220526592)

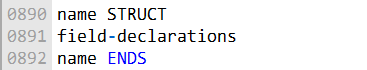
[SPECIAL OPERATORS FOR MACROS 68](#_Toc220526593)

[REPEAT BLOCKS 74](#_Toc220526594)

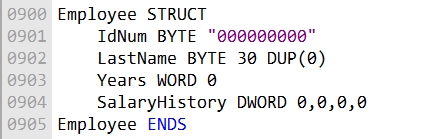
STRUCTURES

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**



The following structure defines an employee structure:



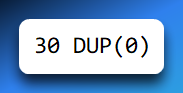
Structure Field Description

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

Note on DUP Operator

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

Syntax example:



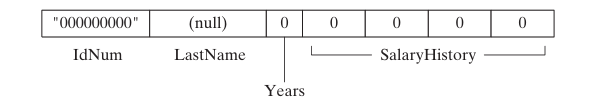
This creates an array of **30 bytes**, each initialized to **0**.

DUP is commonly used to:

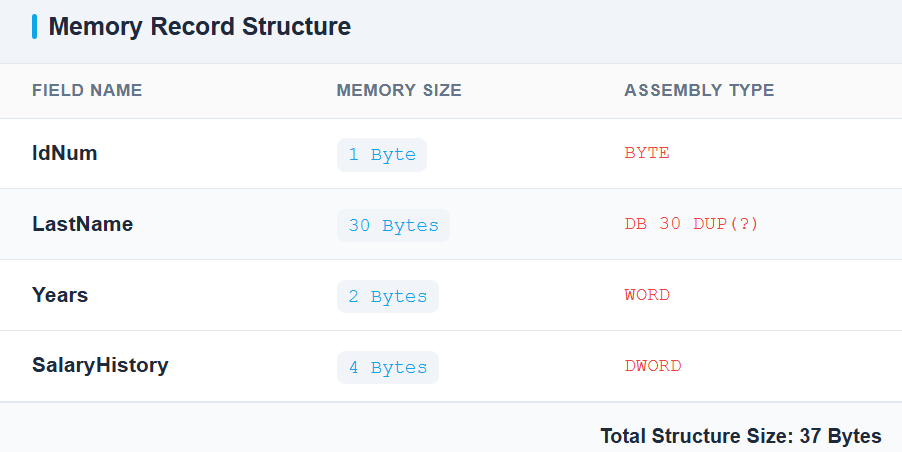
* Zero-initialize strings
* Initialize arrays inside structures

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:



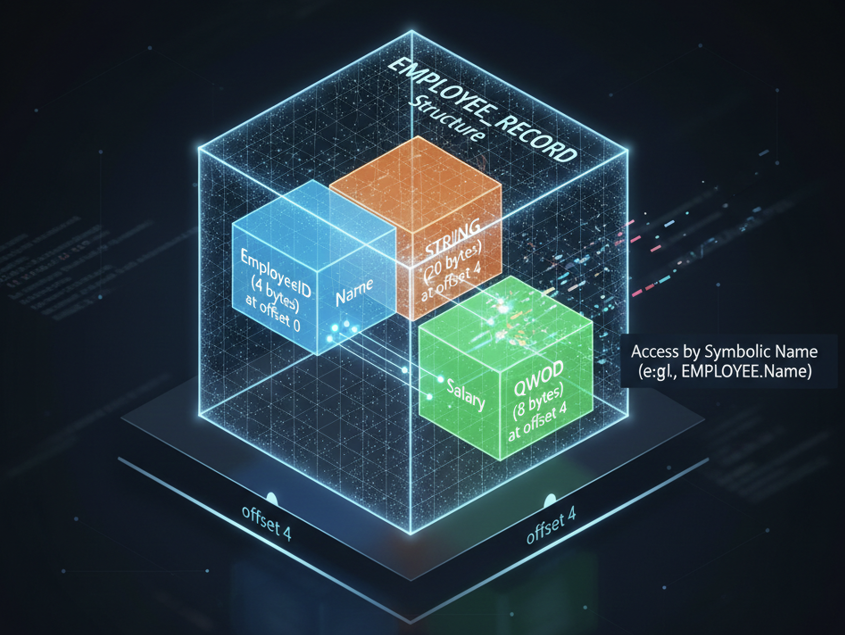
Using Structures in Assembly Language (MASM)

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.

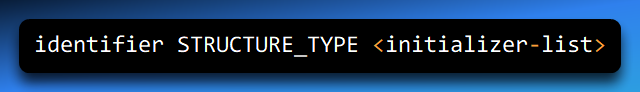
Structures are mainly used to:

* 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



Components Explained

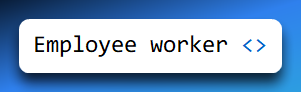
* **identifier**  
  Name of the structure variable (instance)
* **STRUCTURE\_TYPE**  
  Name of the previously defined structure
* **initializer-list (optional)**
  + Values used to initialize structure fields
  + Must follow the **same order** as fields in the structure definition
  + Enclosed in angle 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



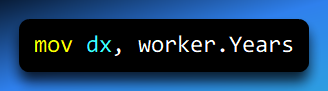
Explanation:

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

III. Referencing a Structure Variable

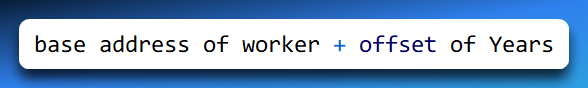
Once a structure variable is declared, it behaves like a block of memory.

**Example**



Accesses the Years field inside the worker structure

The assembler translates this into a memory access using:



IV. Referencing Structure Members (Dot Operator)

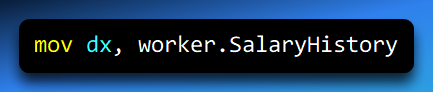
Syntax



Explanation

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

Example

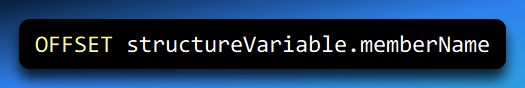


* Reads the SalaryHistory field from the worker structure
* This is the **preferred and safest** way to access structure members

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



Example



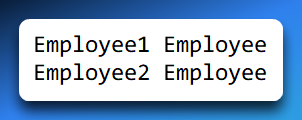
**What this does:**

* 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

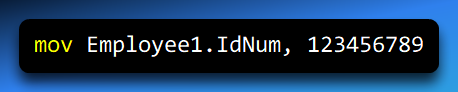


Explanation:

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

VII. Accessing Structure Fields (Dot Operator – Recommended)

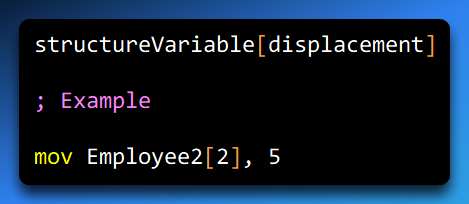
Example



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

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

Syntax



Explanation

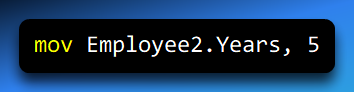
* The displacement is measured **in bytes**, not fields
* It is calculated from the **start of the structure**
* In this example: Offset 2 happens to correspond to the Years field

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

Why This Is Dangerous

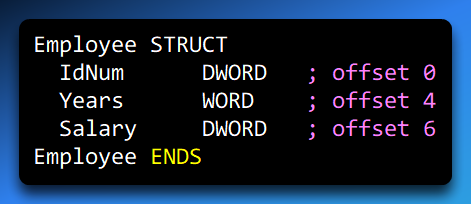
* Breaks if:
  + Field sizes change
  + Fields are reordered
* Hard to read and maintain
* Easy to introduce bugs

Preferred approach:

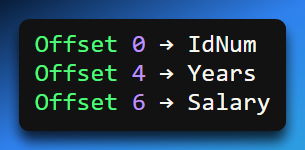


IX. Memory Layout Insight (Conceptual)

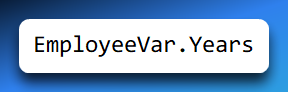
Example structure:



Visual Layout



The assembler computes these offsets automatically when you use:



X. Common Pitfalls

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

XI. Key Takeaways

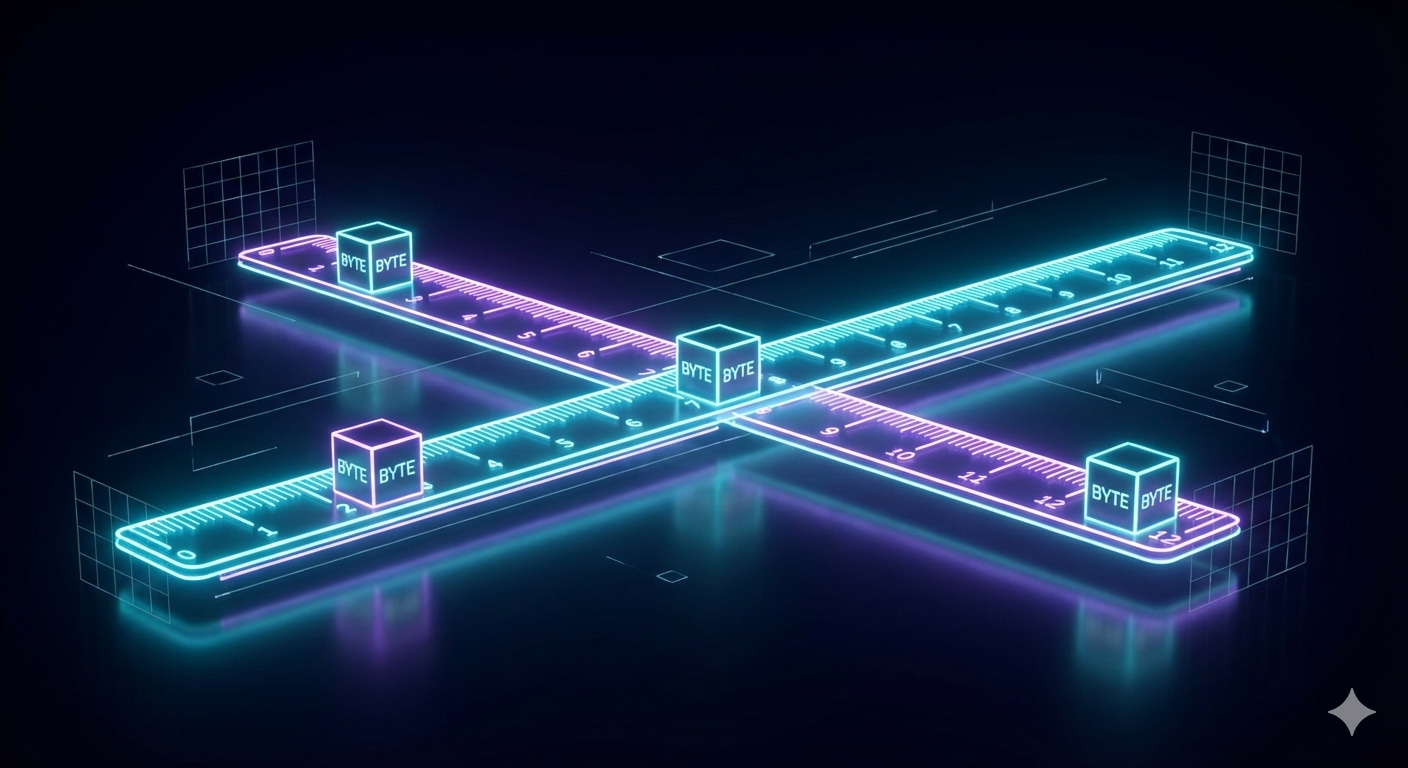
* Structures group related data into a single memory block
* Use **dot notation** for clarity and safety
* OFFSET is used when you need addresses
* Displacement access is possible but discouraged
* Assemblers compute field offsets — let them do it

Aligning Structure Fields

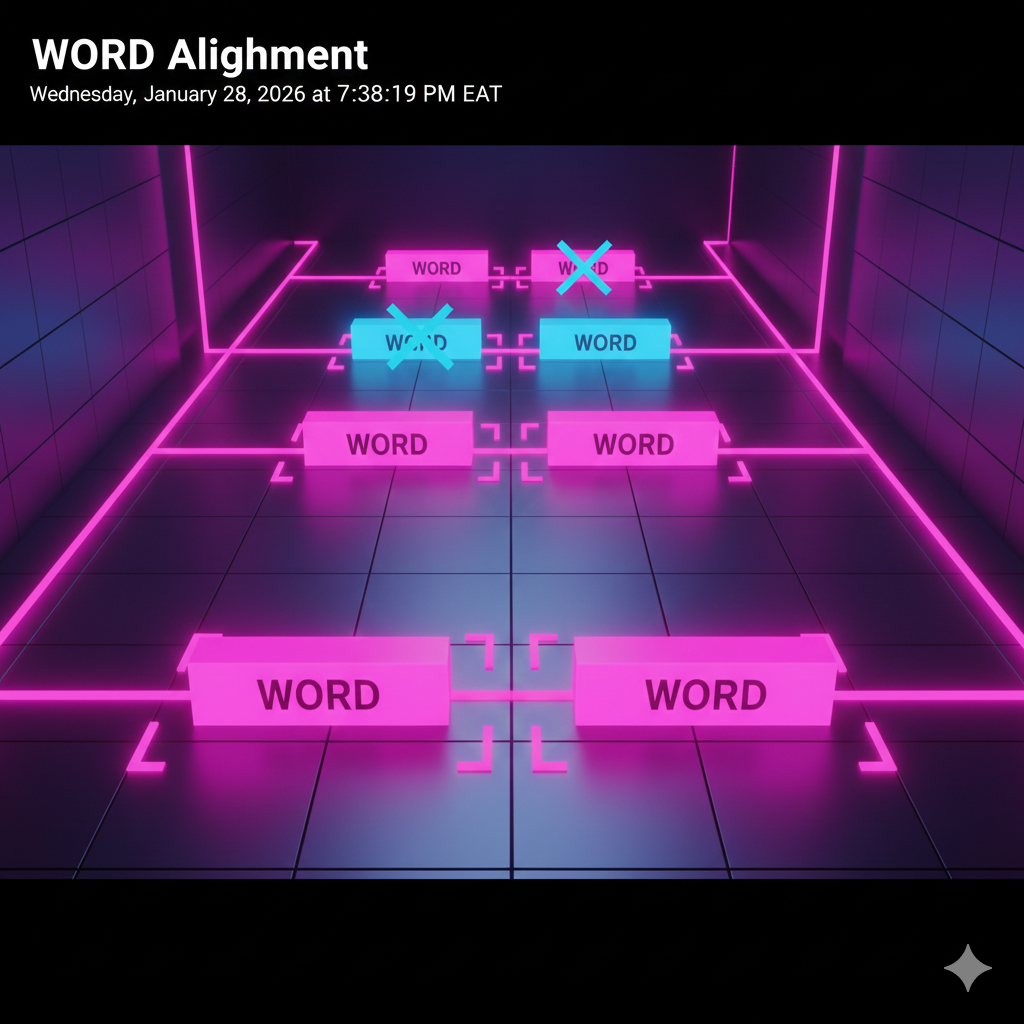
To achieve **optimal memory I/O performance**, structure members should be **aligned according to their data types**.

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

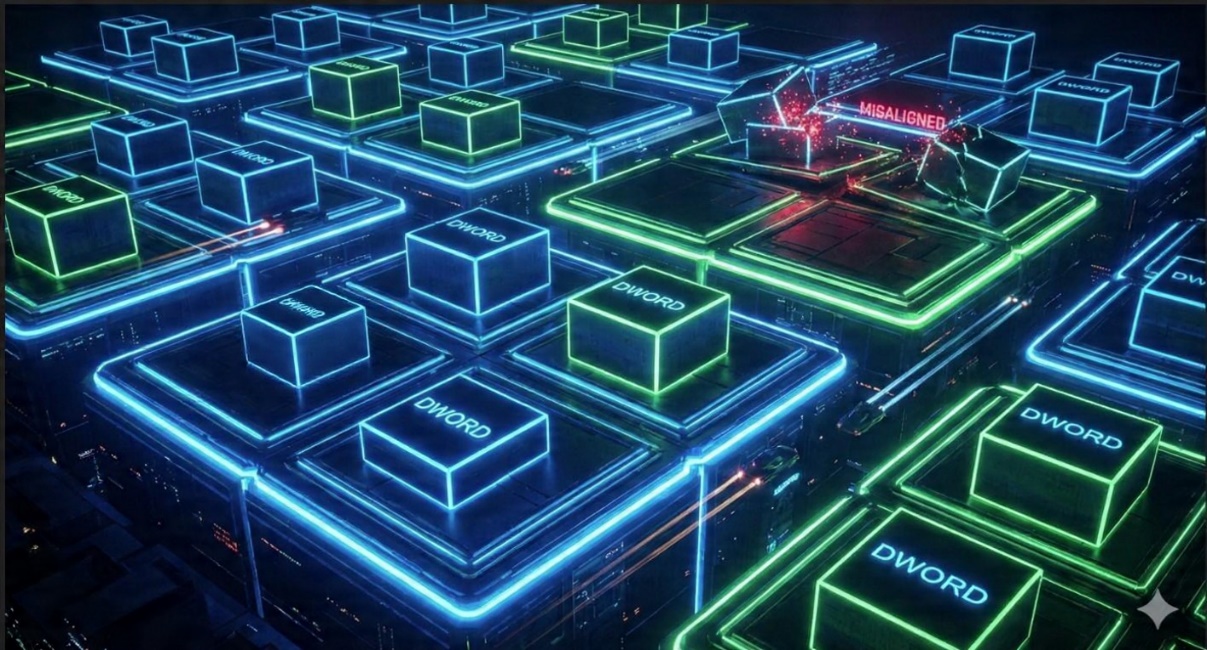
A **BYTE** member should be aligned on a **byte boundary**.



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



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

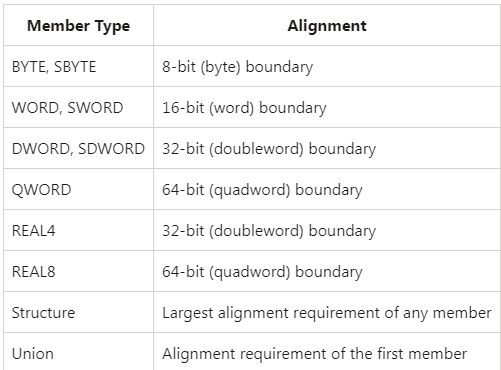


Proper alignment allows the CPU to **access data in fewer memory cycles**.

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

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

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



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:

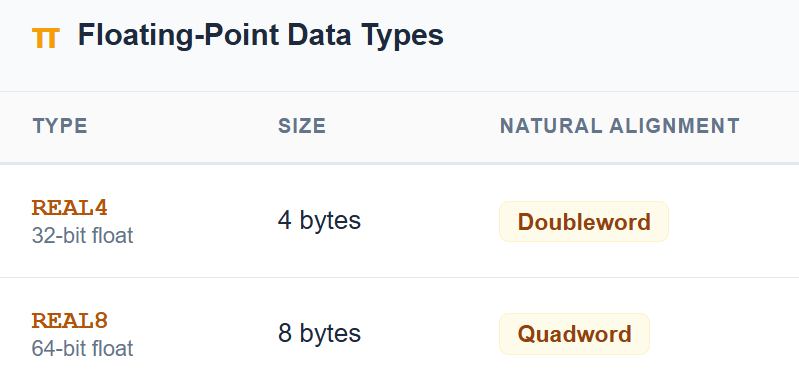
* A WORD (2 bytes) is best placed at an **even address**
* A DWORD (4 bytes) is best placed at an address divisible by **4**

II. Key Alignment Rules (x86)

Natural Alignment of Basic Data Types



**Floating-Point Alignment**



**Structure and Union Alignment**

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

The assembler automatically calculates padding unless explicitly overridden.

3. Why Alignment Matters

**Performance Impact**

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

**Practical Impact -** Small performance loss for single variables

**Large performance degradation** when:

* Iterating over arrays of structures
* Processing large datasets

4. Automatic vs Manual Alignment

**Automatic Alignment (Default)**

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

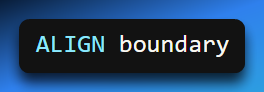
**Manual Alignment (When Needed)**

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

5. ALIGN Directive

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

Syntax:

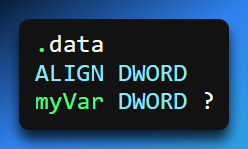


Where boundary can be:

* BYTE
* WORD
* DWORD
* QWORD

6. Aligning Standalone Variables

Example



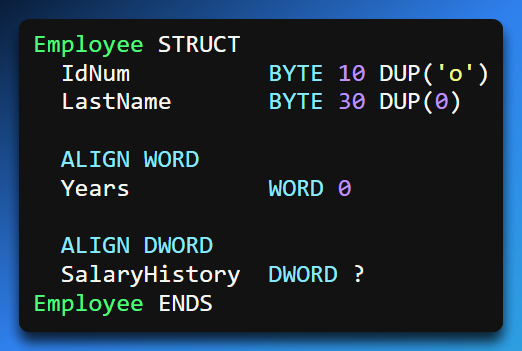
Explanation

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

7. Aligning Structure Fields

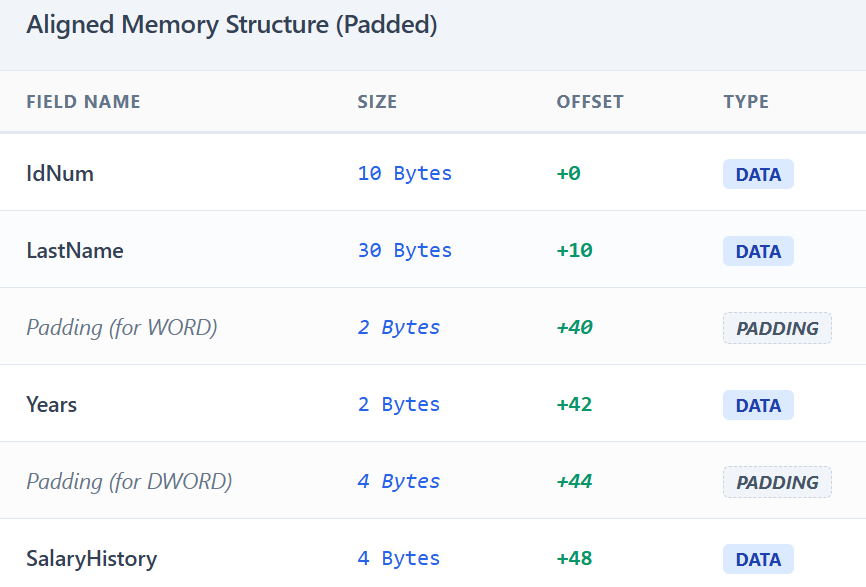
Problem Without Alignment - Fields may start at suboptimal addresses depending on preceding fields’ sizes.

Correctly Aligned Structure Example:



8. How Padding Works (Memory Layout)

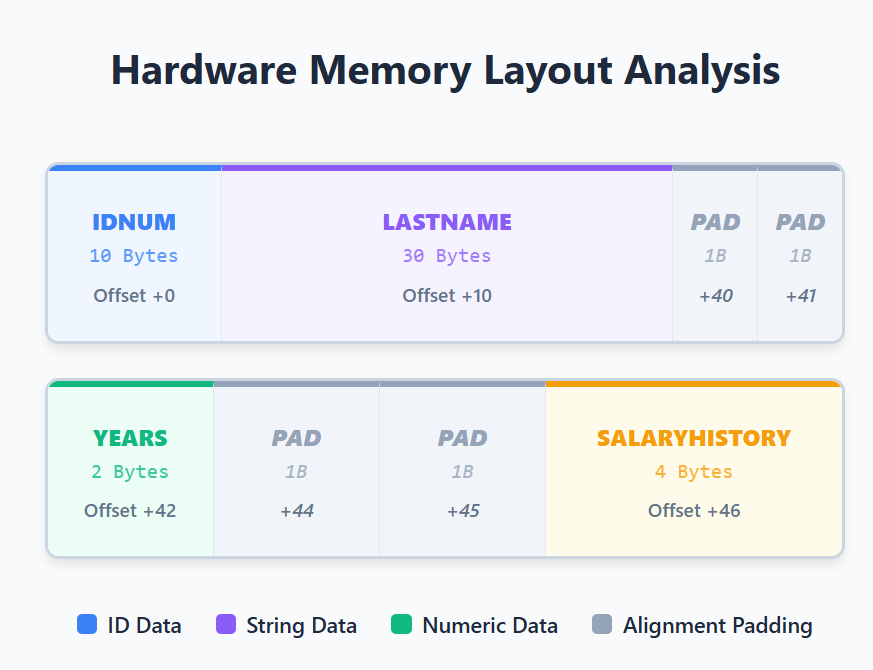
Field Sizes & Offsets



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

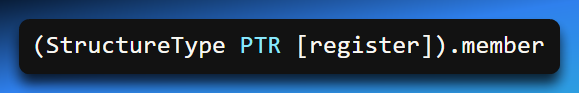
PTR Operator (Why It Is Needed)

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



Example: Accessing a Structure via Indirection



Step-by-Step Explanation

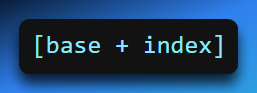
1. OFFSET worker loads the address of the structure into ESI
2. [ESI] dereferences that address
3. Employee PTR tells MASM the memory layout
4. .Years accesses the correct offset inside the structure

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

2. Indexed Operands (Register + Offset Addressing)

**What Is an Indexed Operand?**

An **indexed operand** accesses memory using:



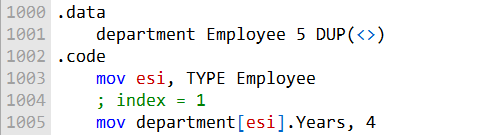
This is especially useful for **arrays of structures**.

**Typical Use Case**

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

3. Arrays of Structures

**Declaring an Array of Structures**



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

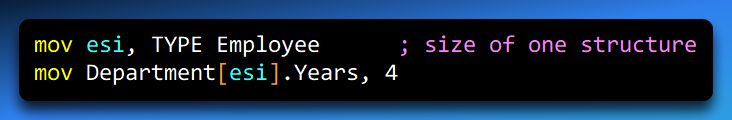
4. Accessing an Array Element Using Indexing

**Correct Concept**

To access element i:



Corrected Example (Fixing 2022 Notes)

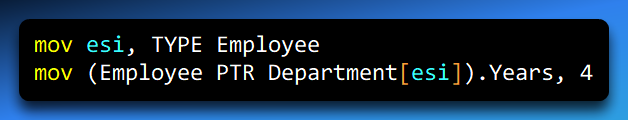


⚠ **Important Correction:**

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

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

**Preferred & Clearer Form**



This ensures:

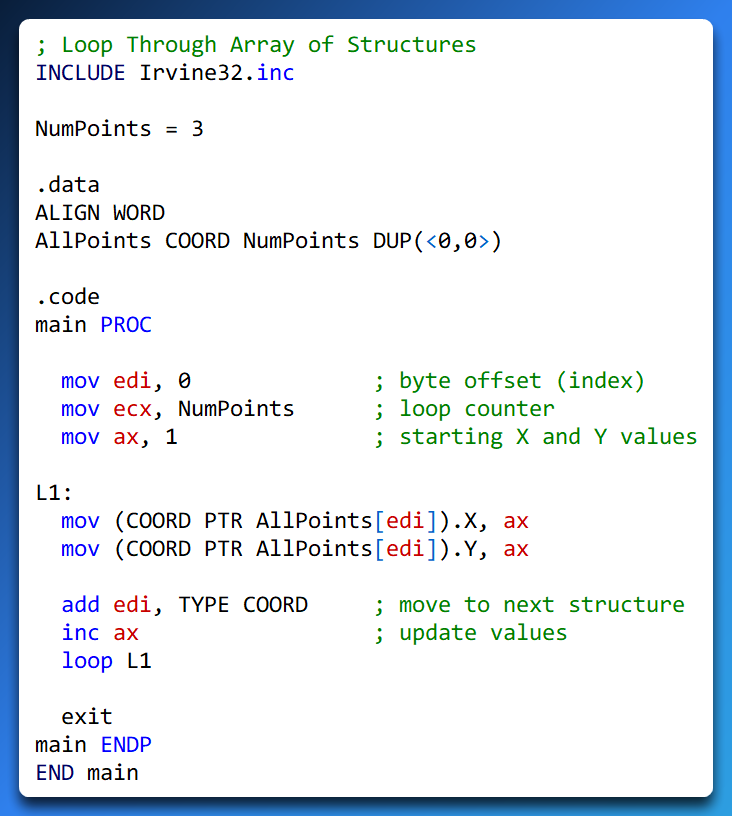
* Correct type interpretation
* Safe member access

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:



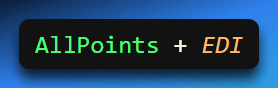
7. How This Loop Works (Step-by-Step)

**Initialization**

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

**Inside the Loop**

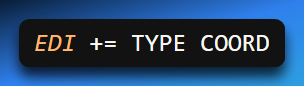
Access current structure:



Write to X field

Write to Y field

Advance to next element:

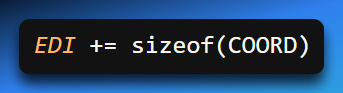


1. Increment coordinate value
2. Loop until ECX = 0

8. Memory Layout Visualization (Recommended)



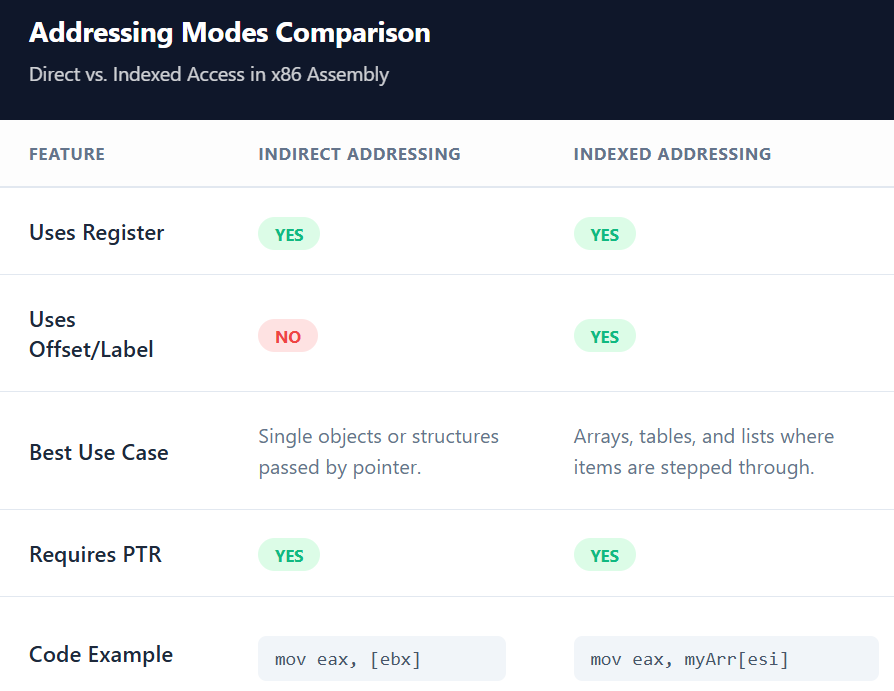
After each iteration:



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



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.

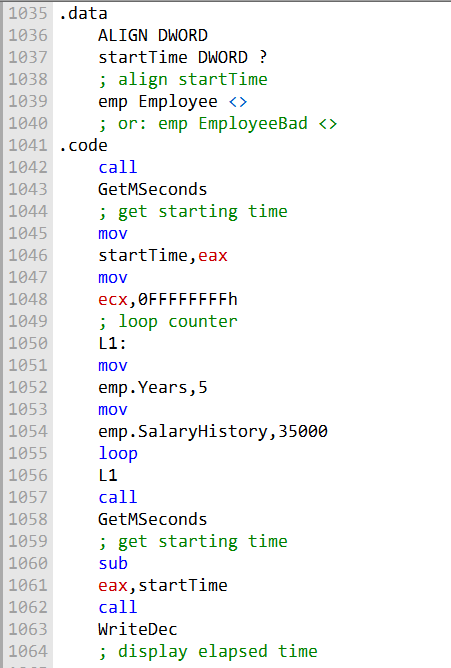
The **performance impact** of misalignment depends on:

* 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.

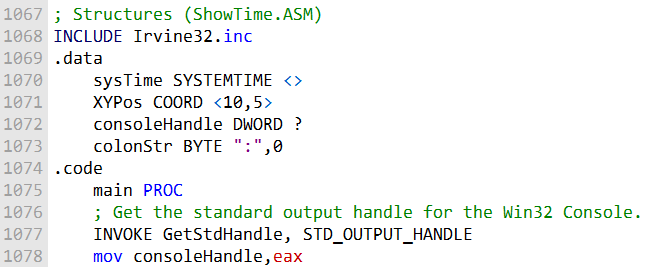


Example: Performance Impact of Aligned vs. Misaligned Structures

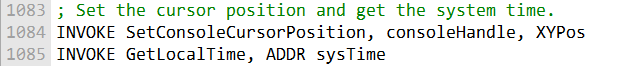
* The code example:
  + Retrieves the **system time**.
  + Executes a **loop that accesses structure fields**.
  + Calculates the **elapsed time**.
* The variable emp can be declared as either:
  + **Employee** → aligned structure
  + **EmployeeBad** → misaligned structure
* Observed results:
  + Using **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:



* 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.



Setting Cursor Position and Retrieving System Time

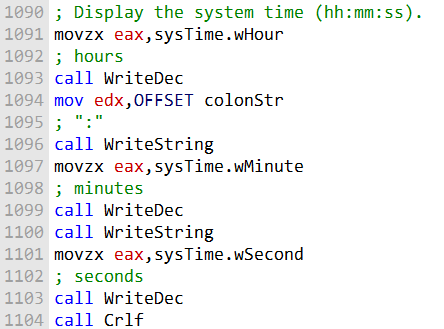
The program **sets the cursor position** and **retrieves the system time**.

**SetConsoleCursorPosition** function:

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

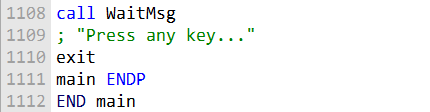
**GetLocalTime** function:

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



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.



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:

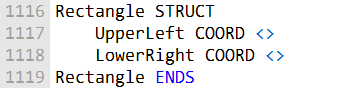
* Code readability
* Maintainability

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

STRUCTURES CONTAINING STRUCTURES

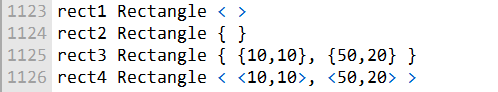
Structures can contain instances of other structures. This is called a nested structure.

For example, a Rectangle structure can be defined in terms of its upper-left and lower-right corners, both COORD structures:



This means that a **Rectangle structure** will contain **two COORD structures**, one for the upper-left corner and one for the lower-right corner.

Rectangle variables can be declared without overrides or by overriding individual COORD fields. The following examples show how to declare Rectangle variables:



The first two declarations, rect1 and rect2, will create Rectangle variables with the default values for the UpperLeft and LowerRight fields.

The third declaration, rect3, will create a Rectangle variable with the specified values for the UpperLeft and LowerRight fields.

The fourth declaration, rect4, is an alternative way to declare a Rectangle variable with the pecified values for the UpperLeft and LowerRight fields.

Once a Rectangle variable has been declared, you can access its fields using the dot notation.

For example, the following code moves the value 10 to the X coordinate of the upper-left corner of the rect1 variable:



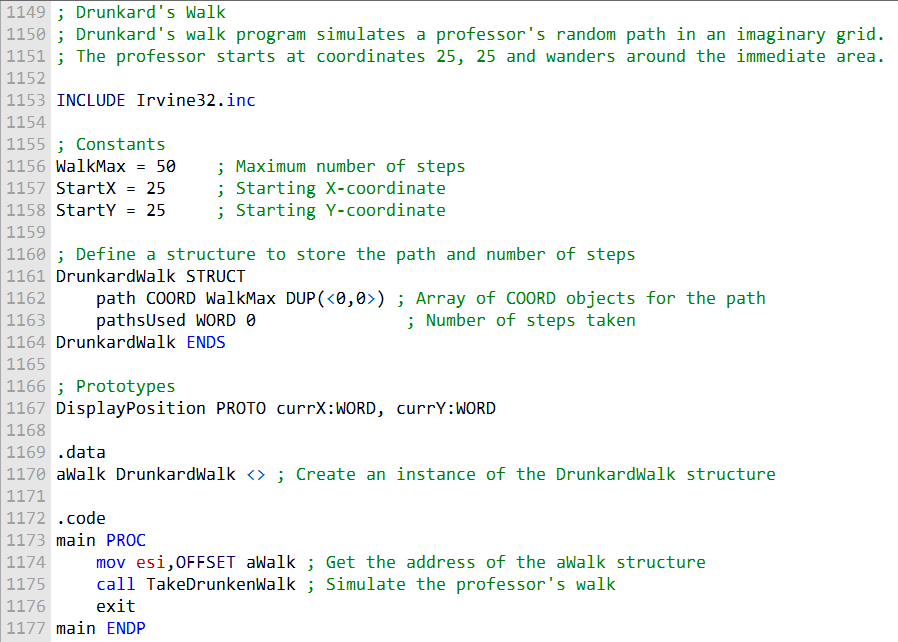
You can also access a structure field using an indirect operand. For example, the following code moves the value 10 to the Y coordinate of the upper-left corner of the structure pointed to by the esi register:

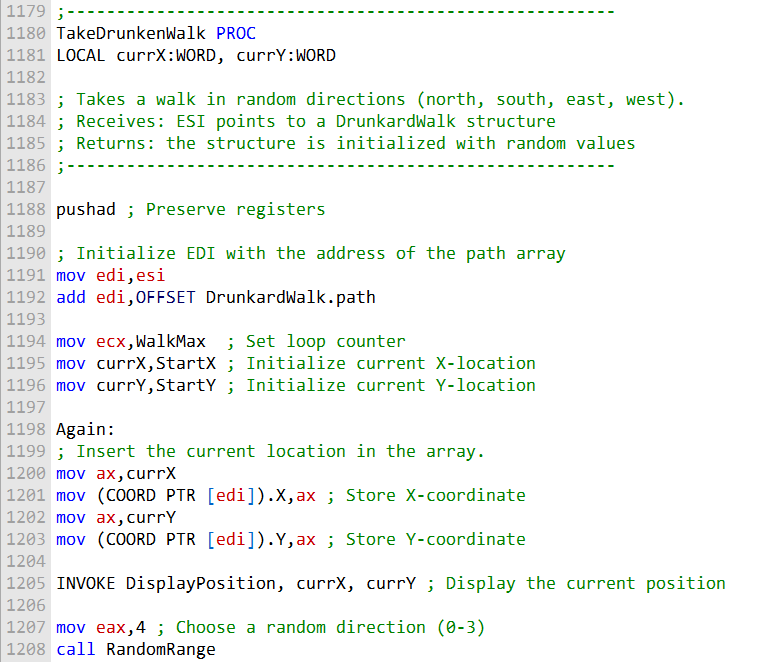


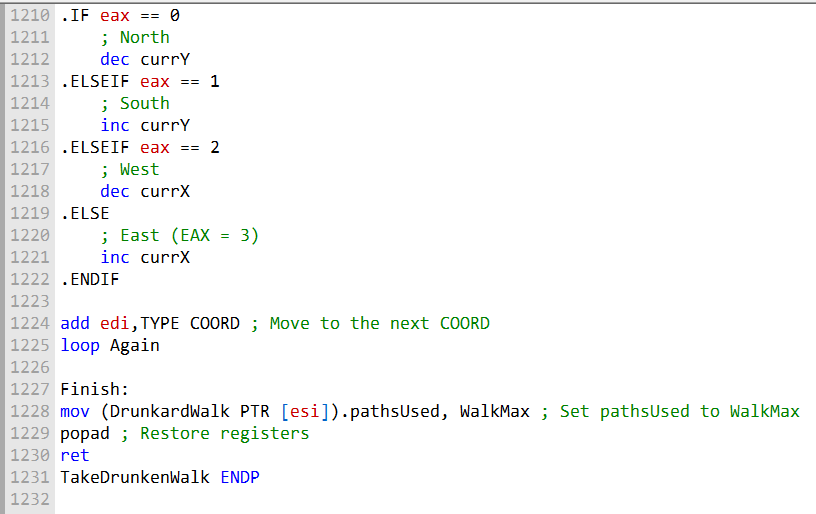
The OFFSET operator can be used to return pointers to individual structure fields, including nested fields. For example, the following code moves the value 50 to the X coordinate of the lower-right corner of the rect2 variable:

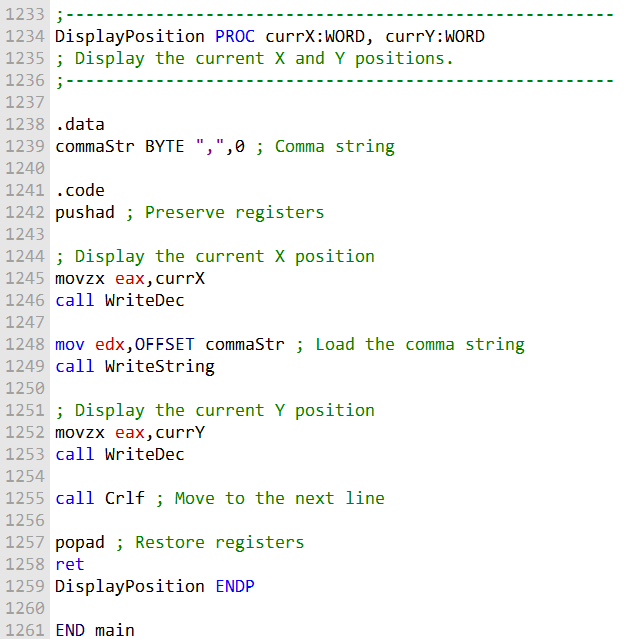


**Example program:**









I'll provide an explanation of the "Drunkard's Walk" program, which simulates a professor's random path in an imaginary grid:

***Include Directives:***

The program begins with an INCLUDE directive to include the Irvine32.inc library, which provides functions for console I/O and other useful features. Constants:

Constants like WalkMax, StartX, and StartY are defined at the beginning. These constants determine the maximum number of steps in the walk and the starting coordinates of the professor.

***Structure Definition:***

The program defines a structure called DrunkardWalk using the STRUCT and ENDS directives. This structure contains two components: path: An array of COORD objects, which will store the professor's path. pathsUsed: A word to keep track of the number of steps taken.

***Prototypes:***

The DisplayPosition function's prototype is declared. It's used to display the current position (X and Y coordinates) of the professor. Data Section:

The .data section is used to define data, including a variable aWalk, which is an instance of the DrunkardWalk structure.

***Code Section:***

The .code section contains the program's main logic. main Procedure:

The main procedure is the program's entry point. It initializes esi with the address of the aWalk structure. Then, it calls the TakeDrunkenWalk procedure to simulate the professor's walk.

***TakeDrunkenWalk Procedure:***

This procedure simulates the professor's walk. It initializes local variables currX and currY to the starting coordinates and uses a loop to simulate the walk. Inside the loop: The current coordinates are inserted into the path array.

The DisplayPosition function is called to display the current position. A random direction (north, south, east, or west) is chosen for the next step. The loop continues until the maximum number of steps (WalkMax) is reached.

***DisplayPosition Procedure:***

This procedure displays the current X and Y positions with proper formatting. It uses WriteDec to display the coordinates and WriteString to add a comma between X and Y. It also adds a line break with Crlf to format the output.

***Program Termination:***

After the professor's walk is completed, the program sets the pathsUsed field of the aWalk structure to WalkMax to indicate the number of steps taken.

***End of Program:***

The END main statement marks the end of the program. In summary, this program uses a structure to simulate a random walk for a professor. It keeps track of the professor's path and displays the X and Y coordinates at each step.

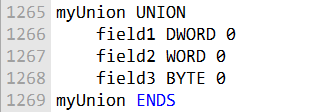
The professor starts at a specified position, and the program randomly selects the next step's direction until the maximum number of steps is reached. The structure helps organize and manage the data related to the walk.

UNIONS IN ASSEMBLY

In assembly language, unions are used to group fields together at the same offset.

The size of a union is determined by its largest field.

To declare a union, you use the **UNION and ENDS directives**, like this:



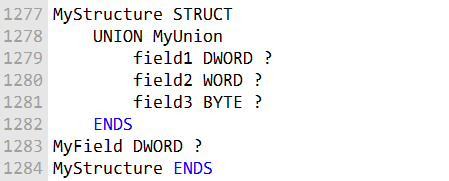
You don't have to indent.

Fields in a union follow the same rules as structures, but each field can have only one initializer.

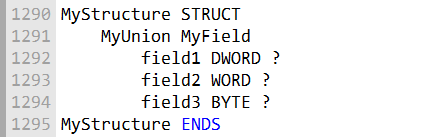
**Nesting Unions in Structures**

You can nest a union inside a structure in two ways:

Declare the union inside the structure:



Use the union's name in a declaration inside the structure:



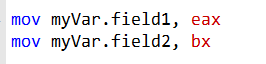
**Declaring and Using Union Variables:**

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

However, you can only have one initializer:



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



**Unions Can Contain Structures**

Unions can also contain structures.

For instance, you can have a union that can hold different types of structures based on the situation. Example - KEY\_EVENT\_RECORD:

In the Windows API, there's a structure called KEY\_EVENT\_RECORD that contains a union named uChar. It's used to handle keyboard input events.

Depending on the situation, it can contain either a Unicode character or an ASCII character.

The union lets you use the appropriate character based on the event type.

Unions and structures are helpful for organizing data in assembly language programs.

Unions allow you to store data of different sizes in the same memory space, which can be useful in various situations.

MACROS IN ASSEMBLY

Certainly, I'll simplify and clarify the information about macros in assembly language.

Overview of Macros:

A macro is a named block of assembly language statements. It can be called multiple times in a program.

When you invoke a macro, a copy of its code is inserted directly into the program where it was invoked.

This is known as inline expansion, and there's no actual CALL instruction involved. Placement of Macros:

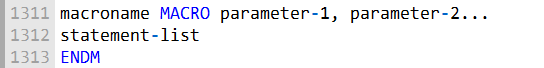
Macro definitions are usually placed at the beginning of the program's source code or in a separate file included with an INCLUDE directive.

Macros are expanded during the assembler's preprocessing step.

The assembler replaces every invocation of the macro with a copy of the macro's source code. If a program defines a macro but never calls it, the macro code does not appear in the compiled program. Defining Macros:

A macro is defined using the **MACRO and ENDM directives.**

The syntax is as follows:



Parameters are named placeholders for arguments passed to the macro.

Parameters can be any text, integers, variable names, or other values. The preprocessor treats them as text.

Parameters don't have type information; type checking occurs during assembly.

**Example - mPutchar Macro:**

The mPutchar macro takes a single parameter named char. It pushes eax, moves the character char into al, and calls the WriteChar procedure.

Finally, it pops eax to restore the original value.

When you use this macro, it replaces the macro invocation with these statements, and the character is displayed on the console.

In essence, macros are a way to create reusable blocks of code that can be inserted directly into your program. They make your code more modular and easier to maintain.



***Invoking Macros:***

To use a macro, insert its name in your program, and you can provide arguments.

The syntax for invoking a macro is: **macroname argument-1, argument-2, ...** macroname must be a previously defined macro name in your source code.

Each argument replaces a parameter in the macro. The order of arguments must match the order of parameters.

You can pass a different number of arguments than the number of parameters in the macro. If you pass too many arguments, the assembler warns you.

If you pass too few, the unfilled parameters remain empty.

For example, if you have a macro called mPutchar that displays characters on the console, you can invoke it like this: mPutchar 'A'.

The macro call is expanded to the code that displays 'A' on the console.

***Debugging Macros:***

Debugging programs with macros can be challenging.

Check the listing file (.LST) after assembling to ensure that each macro is expanded as intended.

In the Visual Studio debugger, you can view the disassembly to see how each macro call is expanded into actual code. This can help with debugging.

***Additional Macro Features:***

You can use the REQ qualifier to specify that a macro parameter is required.

If the macro is invoked without an argument for a required parameter, the assembler shows an error message.

For example, in the mPutchar macro, you can specify that the char parameter is required:

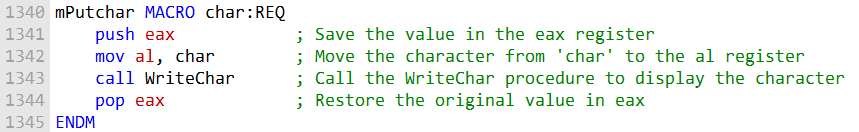


To exclude comments from appearing in macro expansions, use double semicolons (;;).

This way, comments within a macro definition won't show up when the macro is expanded. For example, you can add comments like this: **;; reminder: char must contain 8 bits** within your macro, and they won't appear in the expanded code.

In summary, invoking macros is like using predefined functions in your program. You provide arguments to the macro, which then replaces them with its predefined instructions.

Debugging macros can be done by checking the listing file and using the debugger. Additionally, you can specify required parameters and control which comments appear in macro expansions.



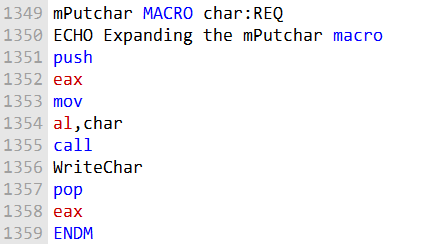
This macro essentially saves the value in eax, displays the character specified by the char argument, and then restores eax to its original value. Remember that the REQ qualifier indicates that the char parameter is required when invoking the macro.

ECHO AND LOCAL DIRECTIVES

***ECHO directive***

The ECHO directive writes a string to standard output as the program is assembled. This can be useful for debugging or for printing informative messages to the user.

Example:



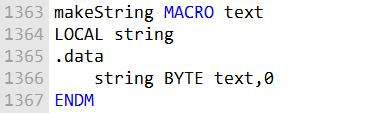
When this macro is expanded, the assembler will print the message "Expanding the mPutchar macro" to standard output.

***LOCAL Directive***

Macros can often contain labels. If a macro is invoked multiple times in a program, the labels in the macro will be redefined each time. This can lead to errors if the same label is used in multiple places in the program.

The LOCAL directive can be used to avoid this problem. When a label is declared as LOCAL, the assembler will generate a unique name for the label each time the macro is expanded.

Example:



In this example, the makeString macro declares a variable named string. If the macro is invoked multiple times, the assembler will generate a unique name for the string variable each time.

To use the LOCAL directive, simply add the LOCAL keyword before the label declaration. For example:

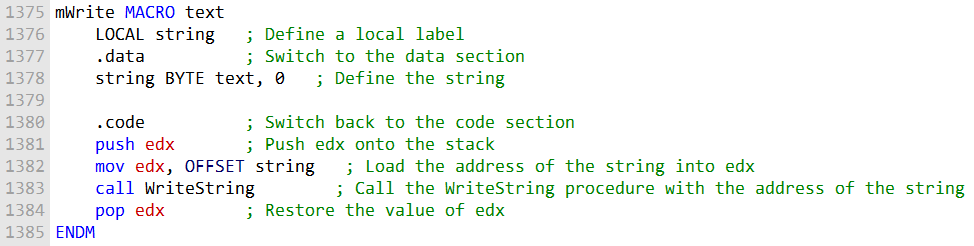


The LOCAL keyword can be used for any type of label, including code labels and data labels.

The ECHO and LOCAL directives can be useful for debugging macros and avoiding label redefinition errors.

***Macros Containing Code and Data***

Macros can contain both code and data. This means that a macro can define both instructions and variables. Example:



This macro defines a function called mWrite() that displays a string on the console. The macro takes a single parameter, text, which is the string to display.

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

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

***Nested Macros***

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

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



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

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

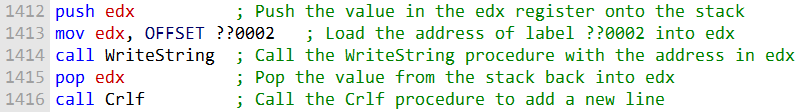
The text parameter is passed directly to the mWrite() macro. Example:



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



This code will then be expanded to the following instructions:



***Tips for Creating Macros***

:

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

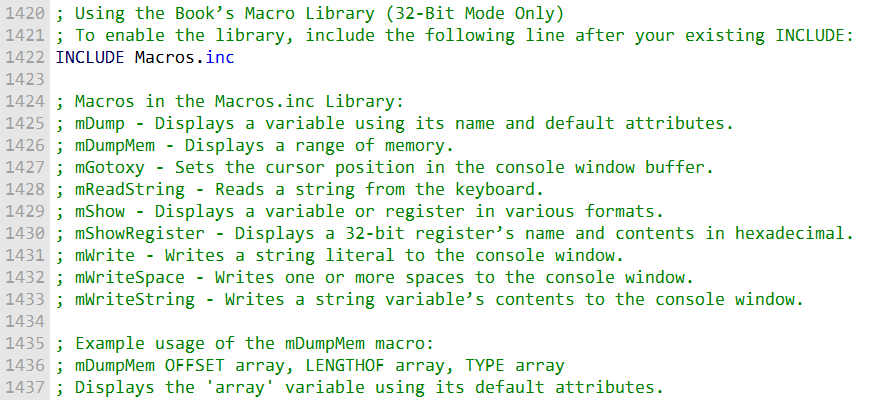
Use a modular approach when creating macros. This means breaking down complex macros into smaller, simpler macros. This will make the macros more reusable and flexible.

Use the LOCAL directive to avoid label redefinition errors.

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

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

**Irvine32/64 library and Macros:**



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

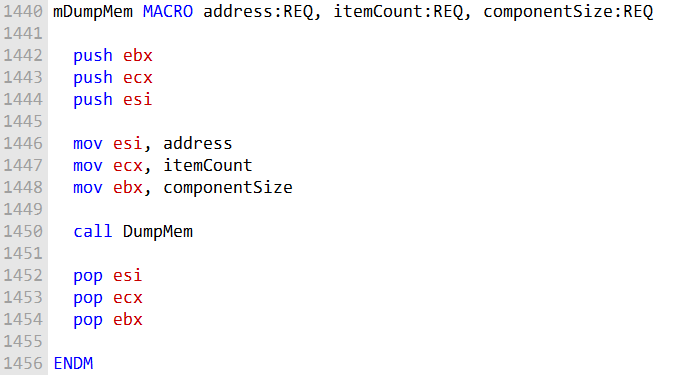
You can enable this library by including it in your program, and then you can use the provided macros to streamline your code.

***=====================================================***

Let's expand on them a bit:

**mDumpMem macro**

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



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

It takes 3 parameters:

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

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

Example usage:



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

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

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

***=====================================================***

**mDump macro**

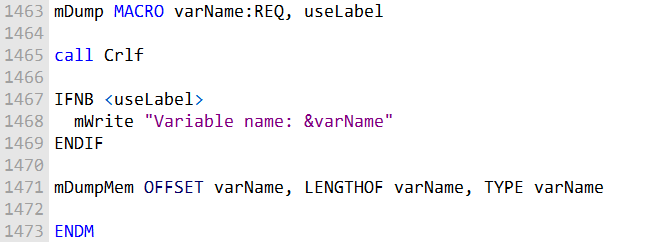
The mDump macro displays the address and hexadecimal contents of a variable.

It takes two parameters:

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

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

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



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

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

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

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

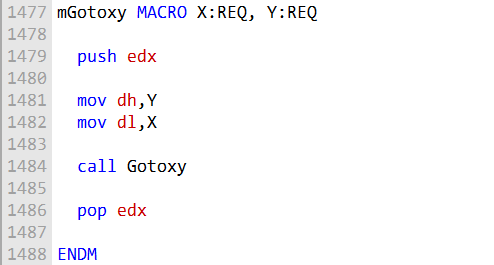
***=====================================================***

**mGotoxy macro**

The mGotoxy macro positions the cursor in the console window.

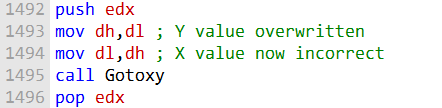
It takes two BYTE parameters for the X and Y coordinates.

Avoid passing DH and DL to prevent register conflicts.



**Register conflict example:**

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



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

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

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

***=====================================================***

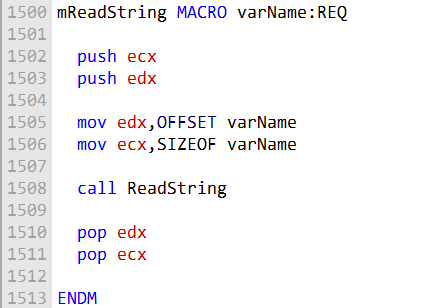
**mReadString macro**

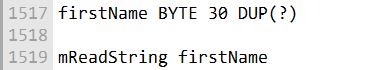
The mReadString macro reads keyboard input into a buffer.

It takes one parameter:

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

Avoids using ECX and EDX internally to prevent register conflicts.





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

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

***=====================================================***

**mShow macro**

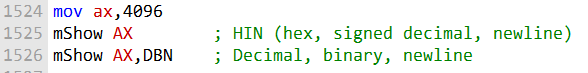
The mShow macro displays a register or variable's name and contents in different formats. It is useful for debugging.

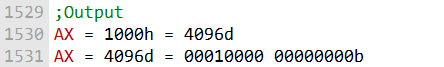
mShow takes a register/variable name followed by format specifiers:

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

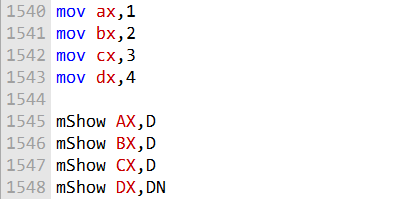
You can combine multiple formats like "HDB" and add multiple newlines. The default is "HIN".

Display AX in multiple formats:



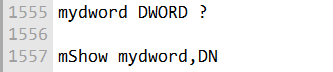


Display multiple registers on one line:





Display variable in decimal with newline:



***=====================================================***

**mShowRegister macro**

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

It takes two parameters:

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



**Displays:**

EBX=7FFD9000

**Implementation**

The macro does the following:

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

Some key points:

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

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

***=====================================================***

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

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

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

**mWriteSpace**

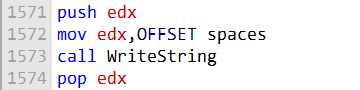
The mWriteSpace macro defines a local label named spaces. This label is used to identify a string of spaces in the .data section of the program.

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

Example:



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



This code will then be expanded to the following instructions:



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

***=====================================================***

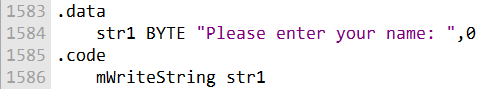
**mWriteString**

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

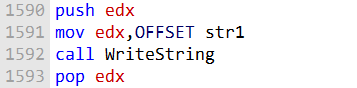
The macro first saves the EDX register on the stack. This is because the WriteString() function uses the EDX register to store the address of the string to write.

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

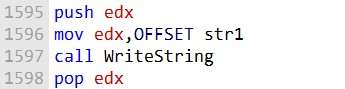
Example:



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



This code will then be expanded to the following instructions:



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

**Conclusion**

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



can be rewritten using the macros as follows:



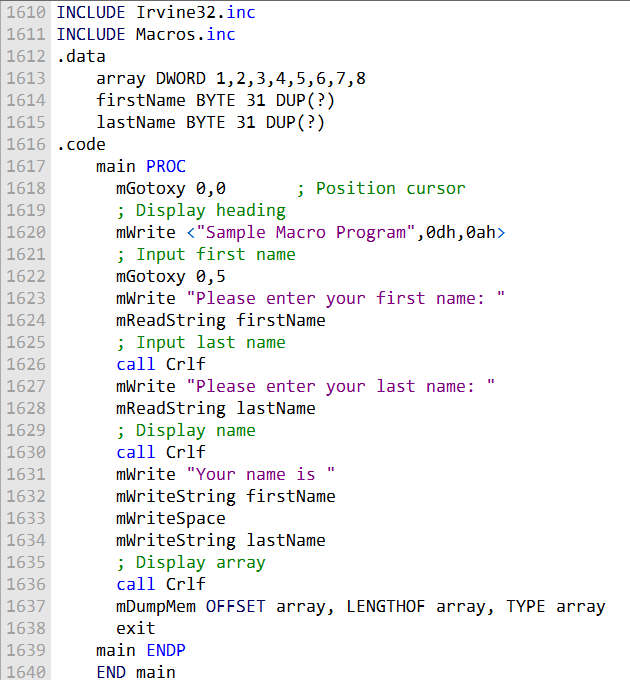
Here is the Wraps.asm program explained with the code rewritten for clarity:

**Purpose:**

Demonstrates use of wrapper macros for common procedures.

**Macros used:**

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



**Here are the review questions rewritten:**

1. When a macro is invoked, does the assembler automatically insert CALL and RET instructions in the generated code? Explain.
2. Where is macro expansion handled - at assembly time or runtime?
3. What is the main advantage of using macros with parameters compared to macros without parameters?
4. Can a macro definition appear before or after the code that invokes it, as long as it is in the code segment? Explain.
5. How does replacing a procedure call with a macro invocation typically affect code size if the macro is called multiple times? Explain.
6. Can a macro contain data definitions like DW and DB? Explain.

**Here are answers to the review questions:**

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

CONDITIONAL ASSEMBLY DIRECTIVES

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

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

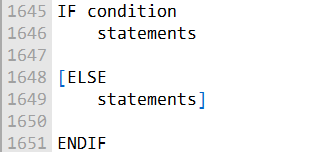
These allow macros to contain conditional logic, repetition, local symbols, early exit, and other advanced logic.

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

**Conditional-assembly directives**

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

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



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

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

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

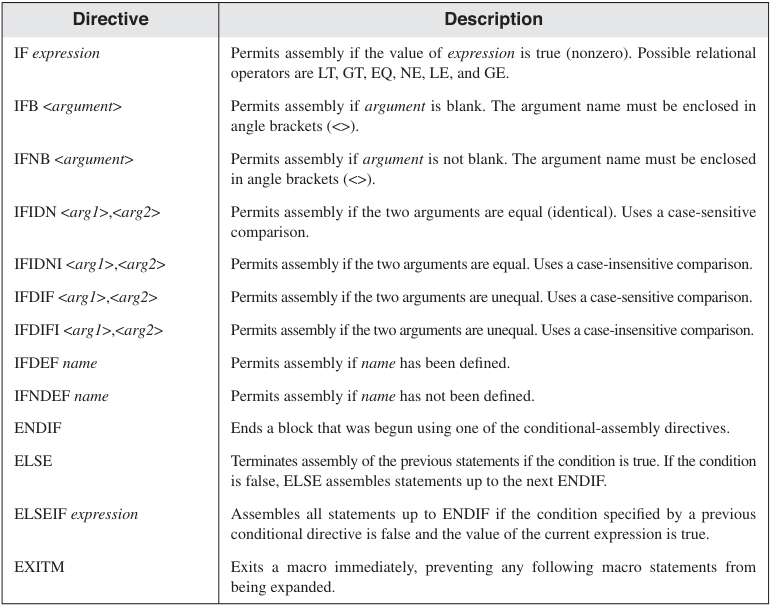
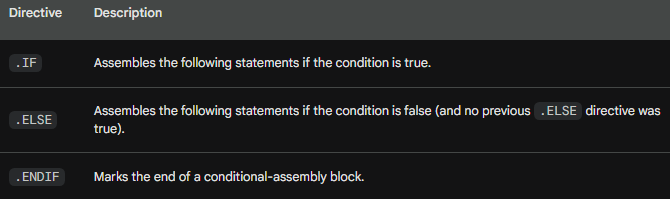
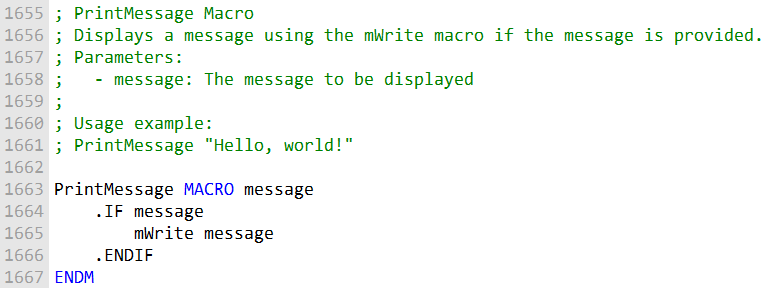


Table 2:



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:



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:



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

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

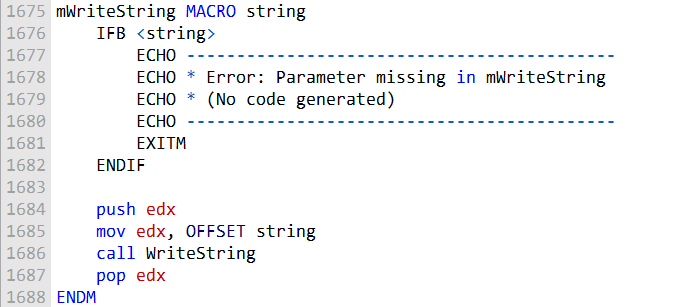
For example, the PrintMessage macro could be modified to support different types of messages, such as error messages, warning messages, and informational messages.

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

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

**Checking for Missing Arguments**

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



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

To address this issue, you can use conditional directives:

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

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

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

**Default Argument Initializers**

Macros can have default argument initializers.

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

The syntax for a default argument initializer is as follows:



Spaces before and after the operators are optional.

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



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

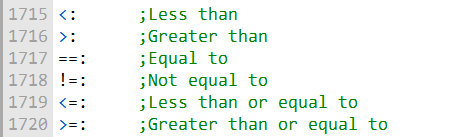
Then, it calls the mWrite macro to display the text and adds a line break by invoking the Crlf procedure.

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

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

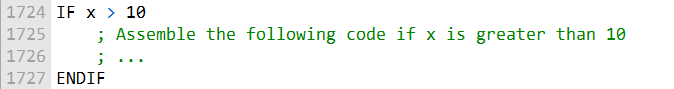
**Boolean Expressions**

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



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:



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

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

**IF, ELSE, and ENDIF directives**

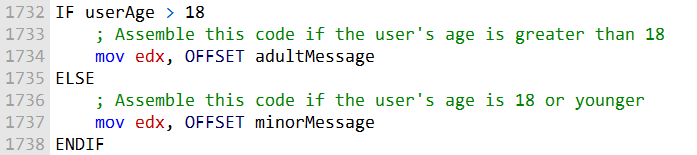
The **IF, ELSE, and ENDIF directives** are used to control the assembly of code based on certain conditions.

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

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

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

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



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

**mGotoxyConst macro**

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

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

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

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

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

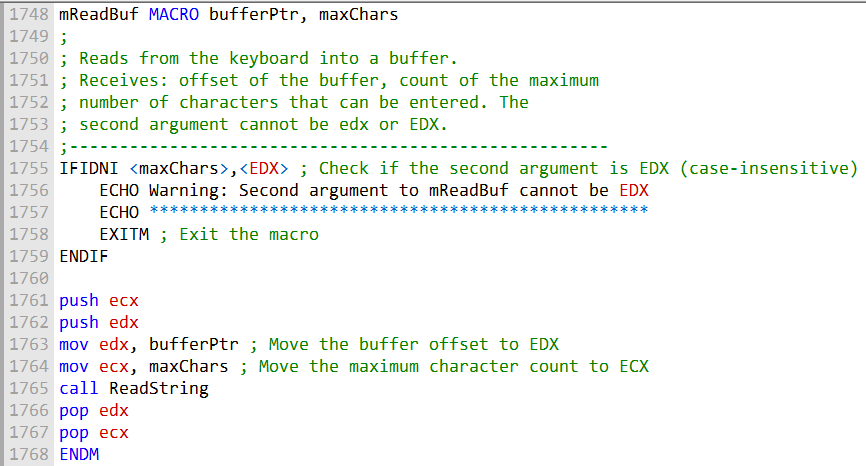
The following is an example of how to use the mGotoxyConst macro:



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

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

**The IFIDN and IFIDNI Directives**



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

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

**IFIDNI:**

IFIDNI is a conditional assembly directive.

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

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

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

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

**ECHO:**

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

**E**

**XITM:**

EXITM is used to exit a macro.

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

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

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

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

**mov:**

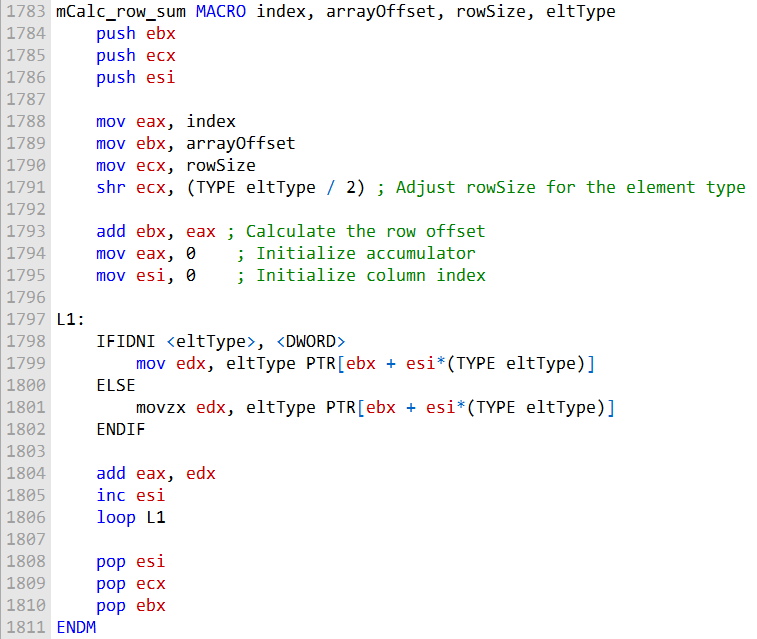
The mov instruction is used to move data between registers and memory locations.

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

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

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

**Summing a matrix row**



**Here's what the macro does:**

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

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

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

It calculates the row offset and initializes the accumulator.

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

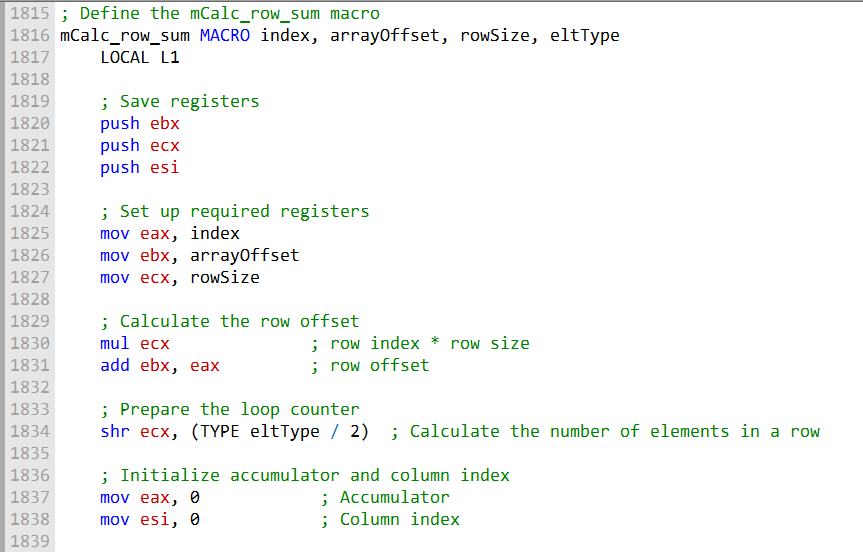
Depending on this condition, it reads the elements from memory correctly.

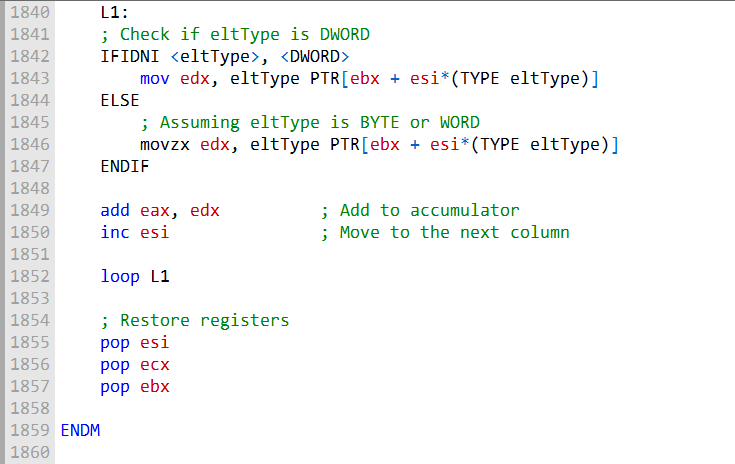
It accumulates the elements into eax and increments the column index.

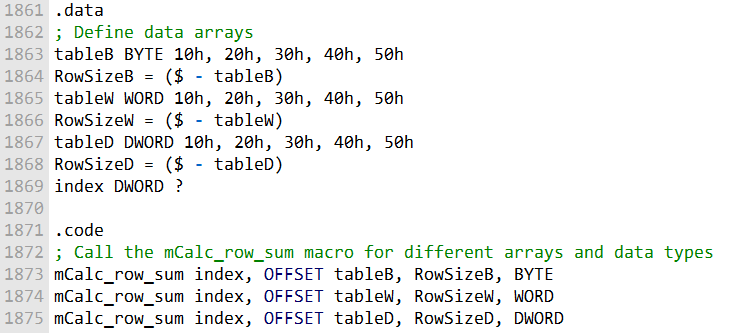
The loop continues until all elements are processed.

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

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







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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

SPECIAL OPERATORS FOR MACROS

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

**Substitution Operator (&):**

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

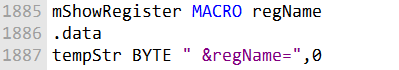
For example, consider the mShowRegister macro, which displays the name and hexadecimal contents of a 32-bit register.

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



**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:



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:



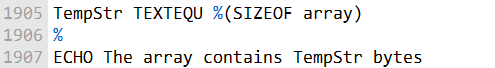
**Flexibility for Passing Arguments:** The % operator offers flexibility in passing arguments to macros. If a macro expects a constant integer argument, you can use the % operator to pass an integer expression. For example:



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:



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:



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:



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:

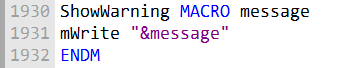


Here, the ! operator is used to prevent the > symbol from being treated as a text delimiter. It ensures that the entire text within the < > brackets is preserved as a single text literal.

**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:



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:

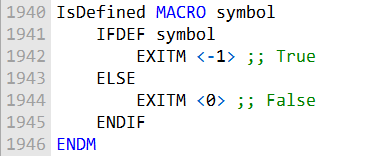


As a result, the program runs and displays the warning message as intended: "Warning: Y-coordinate is > 24."

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:

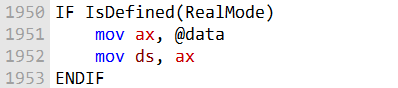


In this example, the IsDefined macro function checks whether a given symbol has been defined. If the symbol is defined, it returns true (represented by -1); otherwise, it returns false (0).

**Invoking a Macro Function**:

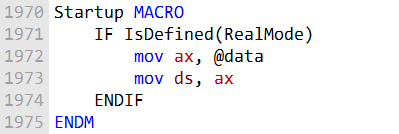
When you want to use a macro function, you need to enclose its argument list in parentheses.

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



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:



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:



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:



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.

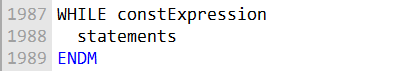
REPEAT BLOCKS

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

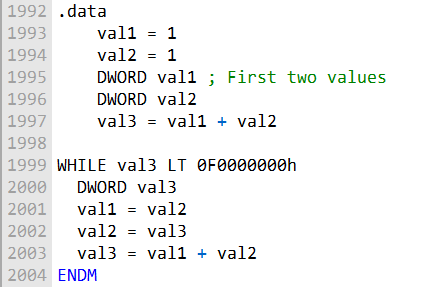
These directives operate at assembly time and use constant values for loop conditions and counters.

**WHILE Directive:**

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



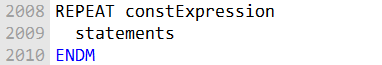
For example, you can use the WHILE directive to generate Fibonacci numbers within a specific range, like so:



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

**REPEAT Directive:**

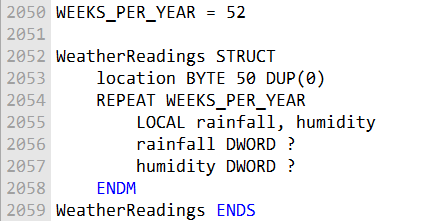
The REPEAT directive repeats a statement block a fixed number of times at assembly time, based on an unsigned constant integer expression.



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**

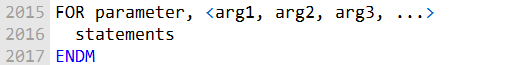
In this example, we use the REPEAT directive to create an array of WeatherReadings. Each WeatherReadings struct contains a location string and arrays for rainfall and humidity readings. The loop repeats for a total of WEEKS\_PER\_YEAR times:



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

**FOR Directive:**

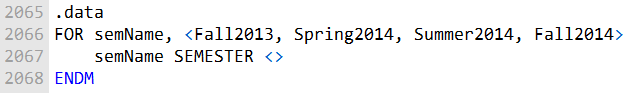
The FOR directive repeats a statement block by iterating over a comma-delimited list of symbols. Each symbol in the list represents one iteration of the loop.



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**

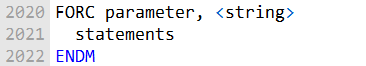
In this example, the FOR directive is used to create multiple SEMESTER objects for student enrollment in different semesters. The loop iterates over a list of semester names and generates corresponding SEMESTER objects:



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.



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

**Student Enrollment:**

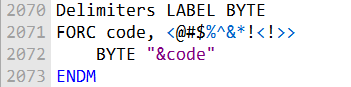
You can use the FOR directive to create multiple SEMESTER objects, each with a different name from a list of symbols. This can be useful for managing student enrollments over multiple semesters.

**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:



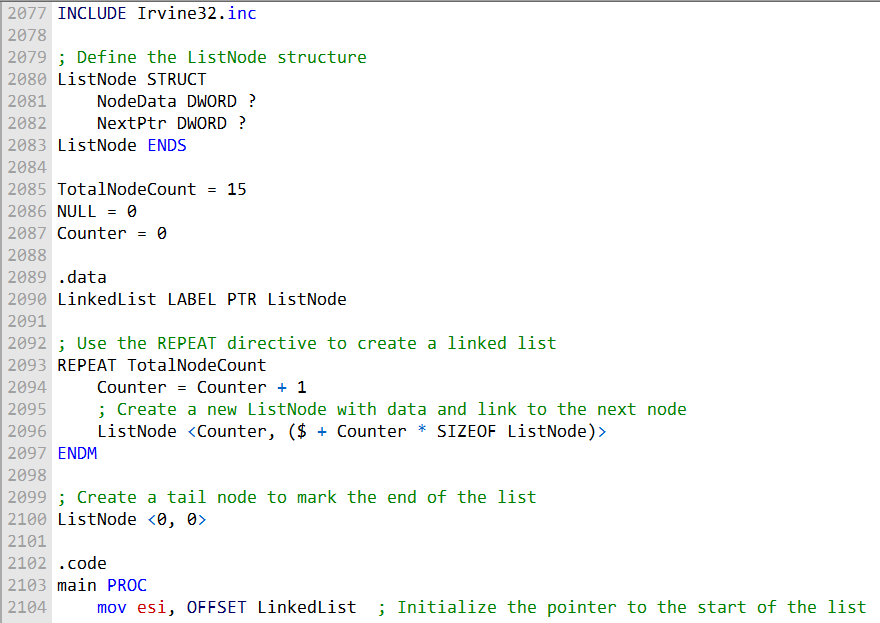
This code generates a lookup table containing the ASCII values and corresponding characters for various special symbols.

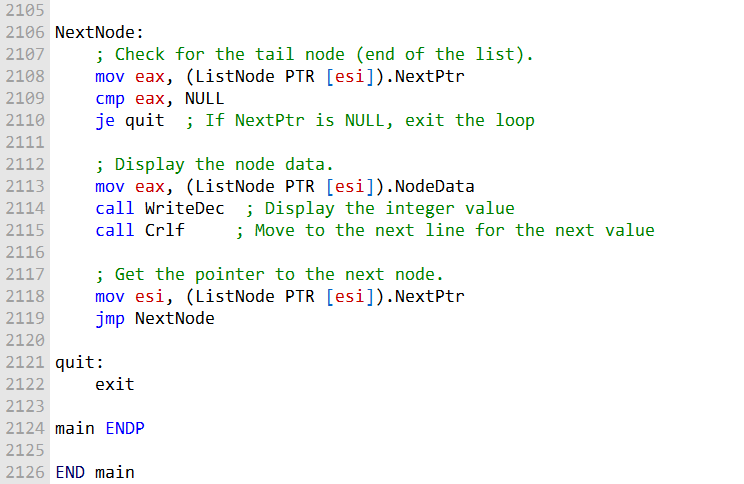
--------------------------------------

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).

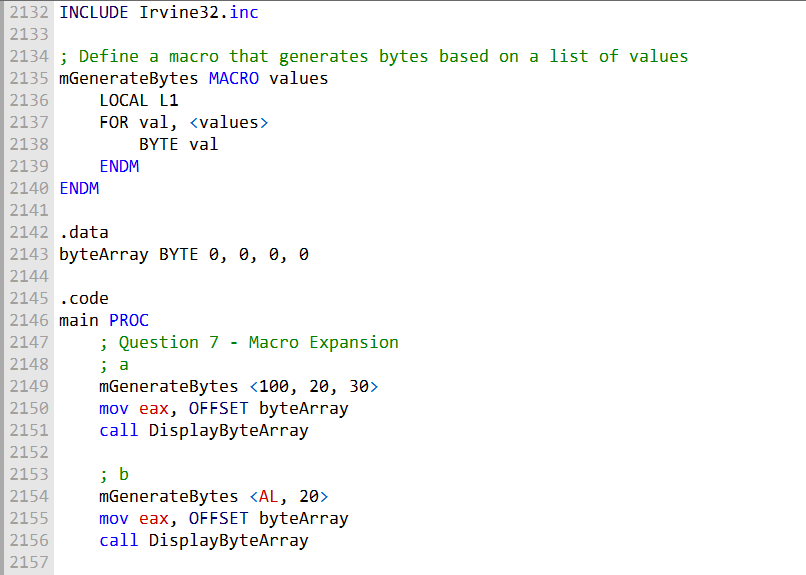
The program defines and populates multiple instances of ListNode objects within a loop to create a linked list.

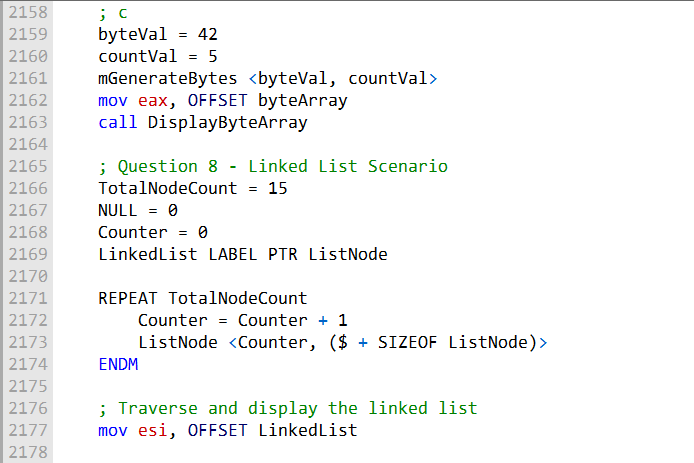
Here's the revised and expanded code with added explanations:

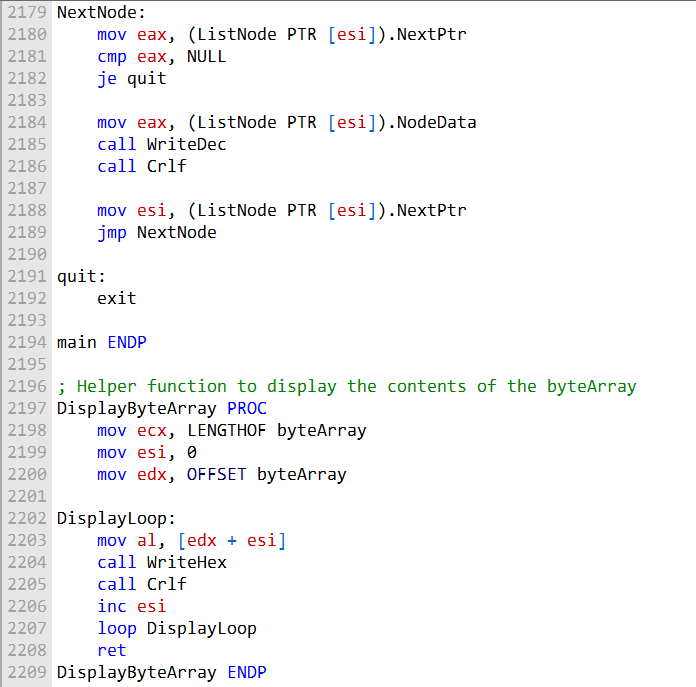




Program to illustrate what we have learnt above:







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