INTRINSIC DATA TYPES

What does “intrinsic data types” mean?

Intrinsic data types are the **built-in data sizes** that the assembler understands.

They answer three simple questions:

1. **How big is the data?** (8 bits, 16 bits, 32 bits, etc.)
2. **Is it signed or unsigned?** (can it be negative?)
3. **Is it an integer or a real (floating-point) number?**

That’s it. No magic.

What the assembler actually cares about

Here’s the key idea:

The assembler mainly cares about **size**.

It needs to know:

- how many bytes to reserve
- how many bytes an instruction will read or write

The assembler **does NOT strongly enforce**:

- signed vs unsigned

That distinction is mostly **for humans**.

Signed vs Unsigned (Important but subtle)

- DWORD → 32-bit **unsigned**
- SDWORD → 32-bit **signed**

Both:

- are **32 bits**
- take up **4 bytes**
- look identical in memory

The only difference is **how *you* interpret the bits**

That's why programmers often use SDWORD:

- not because the assembler demands it
- but because it makes intent clear

Why intrinsic data types matter

Intrinsic data types help you:

- choose the correct **operand size**
- avoid reading or writing the wrong number of bytes
- understand how values are stored in memory

If you get the size wrong, the CPU will still execute —
but your result may be **wrong or corrupted**.

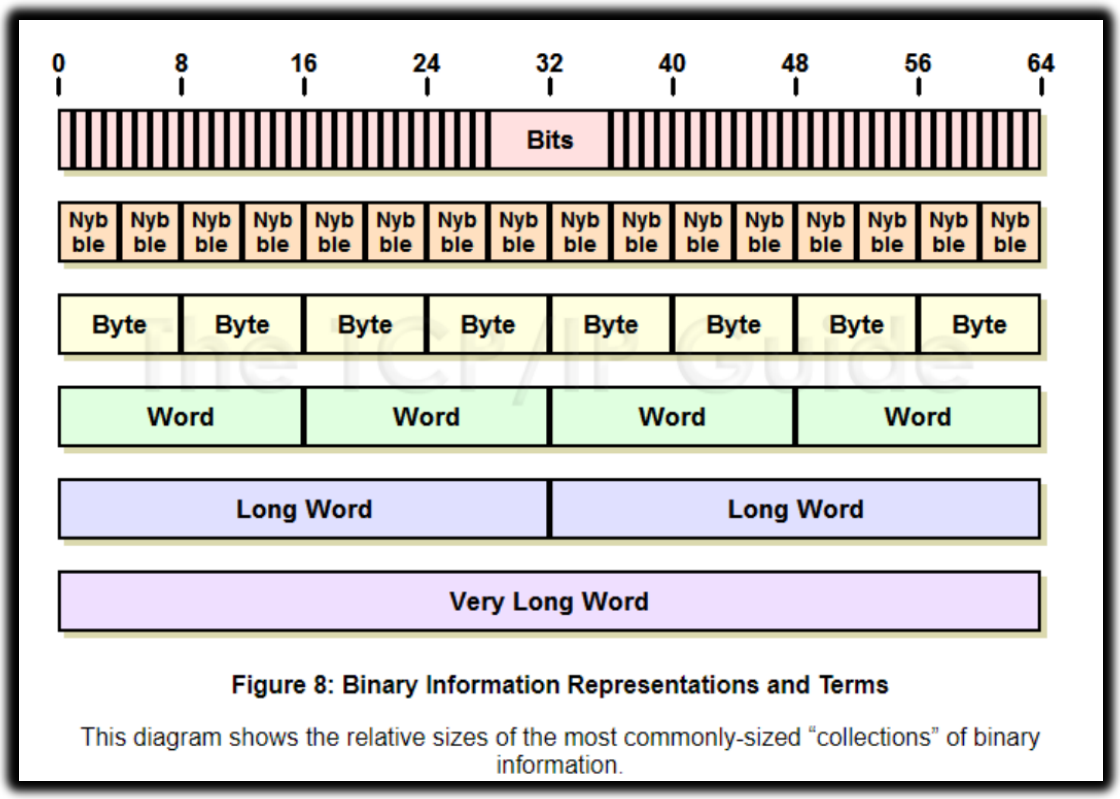


Figure 8: Binary Information Representations and Terms

This diagram shows the relative sizes of the most commonly-sized "collections" of binary information.

Key Takeaways

Intrinsic data types describe the **size**, **signed/unsigned nature**, and whether the value is an **integer or real number**.

The assembler cares about **operand size**, but does **not enforce signed vs unsigned**.

Programmers often use SDWORD to indicate signedness, but it is **not required**.

Intrinsic data types help explain how data is stored and used in assembly.

About overlapping types (Very important concept)

Some types overlap in functionality.

Example:

- DWORD → 32-bit unsigned
- SDWORD → 32-bit signed

Same size. Same memory.
Different **meaning**.

The assembler sees "32 bits".
The programmer sees "signed" or "unsigned".

So when I say "intrinsic data types"...

Yes — you mean **the ones in that image**.

These are the **basic building blocks** of all data in a computer.

Let's walk through them naturally.

Bit-Level Building Blocks (From smallest to bigger)

- **Bit**
  A single 0 or 1. The smallest possible unit of data.
- **Nibble (4 bits)**
  Half a byte. One hexadecimal digit fits here.
- **Byte (8 bits)**
  Stores:
  - ✓ a character
  - ✓ a small number
    This is the most common basic unit.
- **Word (16 bits)**
  Twice a byte. Used for larger numbers.
- **Double Word (32 bits)**
  Four bytes. Very common in 32-bit programs.
- **Quad Word (64 bits)**
  Eight bytes. Used for very large numbers.

Everything else is built from these.

Intrinsic Data Types in Assembly

Integer types

- **BYTE**
  8-bit **unsigned** integer
  Range: 0 to 255
- **SBYTE**
  8-bit **signed** integer
  Range: –128 to 127
- **WORD**
  16-bit **unsigned** integer
  Range: 0 to 65,535
- **SWORD**
  16-bit **signed** integer
  Range: –32,768 to 32,767

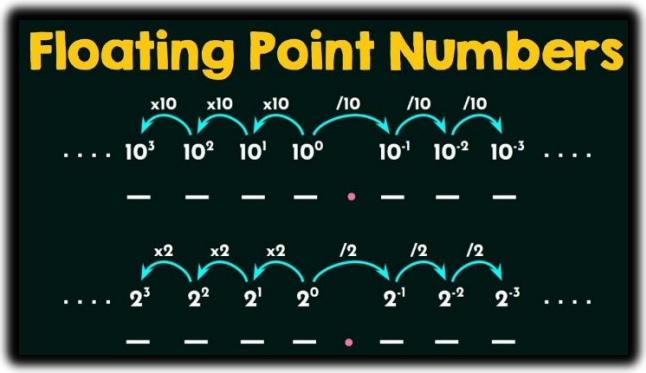
- **DWORD**
  32-bit **unsigned** integer
  Range: 0 to 4,294,967,295
- **SDWORD**
  32-bit **signed** integer
  Range: –2,147,483,648 to 2,147,483,647

Larger / special integer types

- **FWORD (48 bits)**
  Used mainly for **far pointers** (old protected-mode stuff)
- **QWORD (64 bits)**
  Very large integers
- **TBYTE (80 bits)**
  Rarely used
  Mostly related to the floating-point unit

Floating-point (real numbers)

- **REAL4**
  32-bit floating-point
  Common for basic decimal values
- **REAL8**
  64-bit floating-point
  Higher precision
- **REAL10**
  80-bit floating-point
  Very high precision, rarely used



Final idea

The assembler cares about **how many bytes**.
The programmer cares about **what those bytes mean**.

That's why intrinsic data types exist.

DATA DEFINITIONS (ASSEMBLY VARIABLES)

A **data definition** in assembly is how you create a variable.

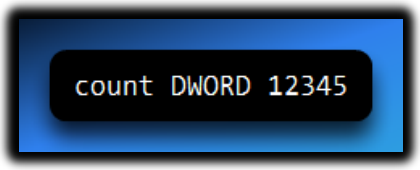
It answers two questions:

1. **How much memory do I need?**
2. **What value should it start with?**

General syntax

- **label** → the variable name (optional, but almost always used)
- **directive** → the data type / size
- **value** → the initial value

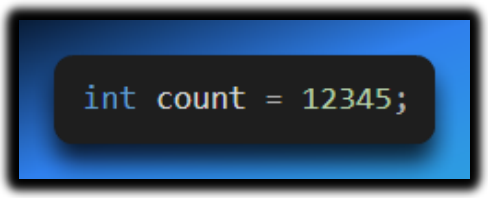
Example



This means:

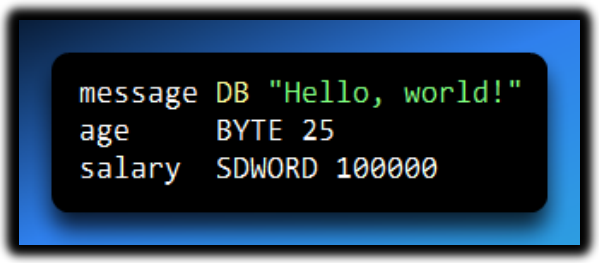
- create a variable named count
- reserve 4 bytes (32 bits)
- store the value 12345 in it

Equivalent C code:



Same idea, different language.

More examples



What's happening here:

- message
  - ✓ DB reserves **1 byte per character**
  - ✓ "Hello, world!" takes **13 bytes**
- age
  - ✓ BYTE reserves **1 byte**
  - ✓ stores the value 25
- salary
  - ✓ SDWORD reserves **4 bytes**
  - ✓ stores a signed integer value

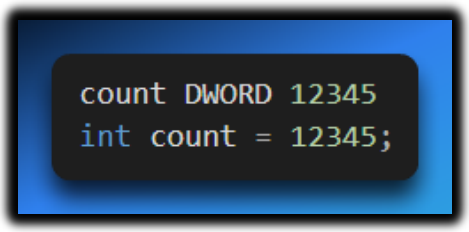
Why the data type matters

The assembler **must know the size** of the variable:

- how many bytes to reserve
- how many bytes instructions should read or write

If you don't specify the type, the assembler has no idea what to do.

Assembly vs C (Same concept)



Both:

- reserve memory
- assign an initial value
- give the memory a name

Assembly just makes the size explicit.

Short forms (Just aliases)

These are **short names**, not new types:

- BYTE → DB
- WORD → DW
- DWORD → DD
- QWORD → DQ
- TBYTE → DT

They all do the same job: **reserve memory**.

Legacy Data Directives (Still used in 2026?)

Yes — **absolutely still used**.

Directives like DB, DW, DD, DQ, and DT are:

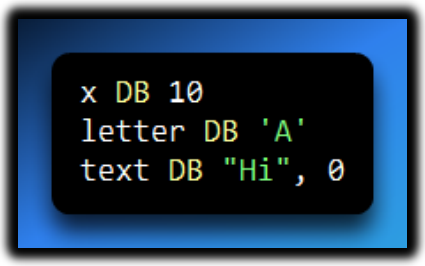
- still supported
- still common
- still the standard way to define data in MASM

They are called "legacy" only because they've been around forever — not because they're obsolete.

The Core Data Directives (Explained Clearly)

1. DB — Declare Byte (8 bits)

Reserves **1 byte** per value.



Common uses:

- characters
- small numbers
- strings (byte-by-byte)

2. DW — Declare Word (16 bits)

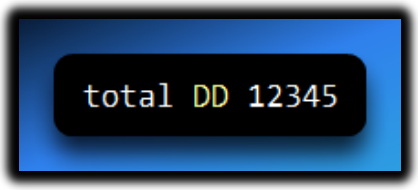
Reserves **2 bytes**.

Used for:

- 16-bit values
- older or compact data

3. DD — Declare Doubleword (32 bits)

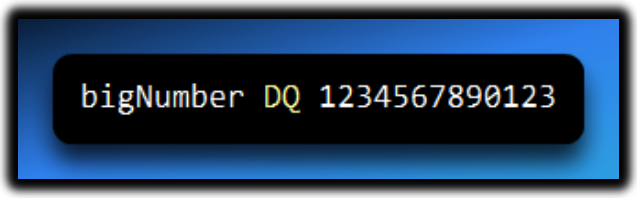
Reserves **4 bytes**.



This is one of the **most common** directives in 32-bit programs.

4. DQ — Declare Quadword (64 bits)

Reserves **8 bytes**.

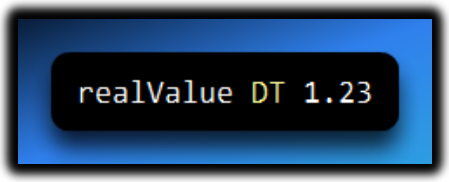


Used for:

- large integers
- 64-bit values

5. DT — Declare Ten Bytes (80 bits)

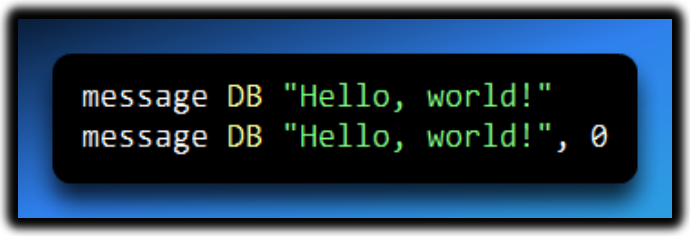
Reserves **10 bytes**.



Used for:

- extended precision floating-point (FPU)
- rare, but valid

About strings and null terminators



Both are valid.

The second one:

- adds a **null terminator**
- is better when interacting with C-style functions

MASM does **not** automatically add 0 for you.

Big Idea to Remember

Data definition directives:

- reserve memory
- define size
- optionally initialize values

The assembler:

- assigns addresses
- tracks them in the symbol table
- replaces variable names with real memory locations

You write **names**.
The assembler handles **addresses**.

Data definitions are how assembly creates variables — by explicitly stating how many bytes to reserve and what value to store in them.

Defining Data Types (Part 1 – Beginner Explanation)

Big Picture: What This Section Is About

This section explains:

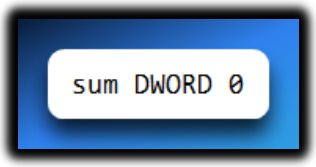
- How variables are **defined** in assembly
- How variables are **initialized**
- What happens if variables are **not initialized**
- How different **byte-sized data types** work

Main Rules for Data Definitions

1. At Least One Initializer Is Required

When you define a variable, the assembler expects **a value**.

Example:



- DWORD → data type (4 bytes)
- 0 → initializer

Even zero counts as a valid initializer.

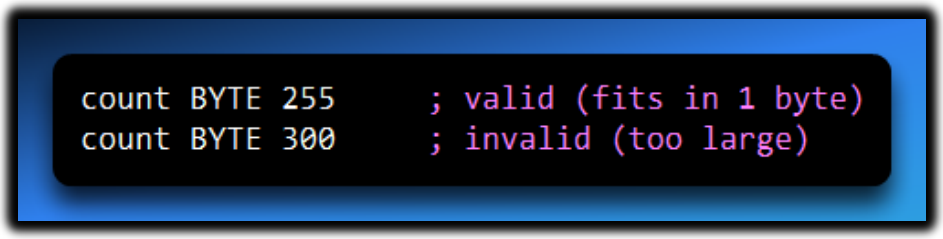
2. Multiple Initializers Use Commas

You can define **multiple values** at once by separating them with commas. Example:

This creates **four bytes** in memory.

3. Integer Initializers Must Match the Data Size

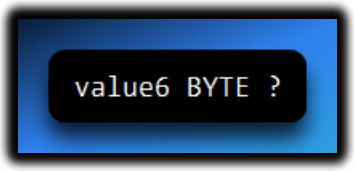
For integer data types, the value must **fit in the size** of the variable. Example:



4. Leaving a Variable Uninitialized (?)

If you want to reserve memory **without giving it a value**, use ?.

Example:



This means:

- Memory is reserved
- The value is unknown (garbage) at program start

⚠ **Important:** Uninitialized variables **must not be used** before assigning a value.

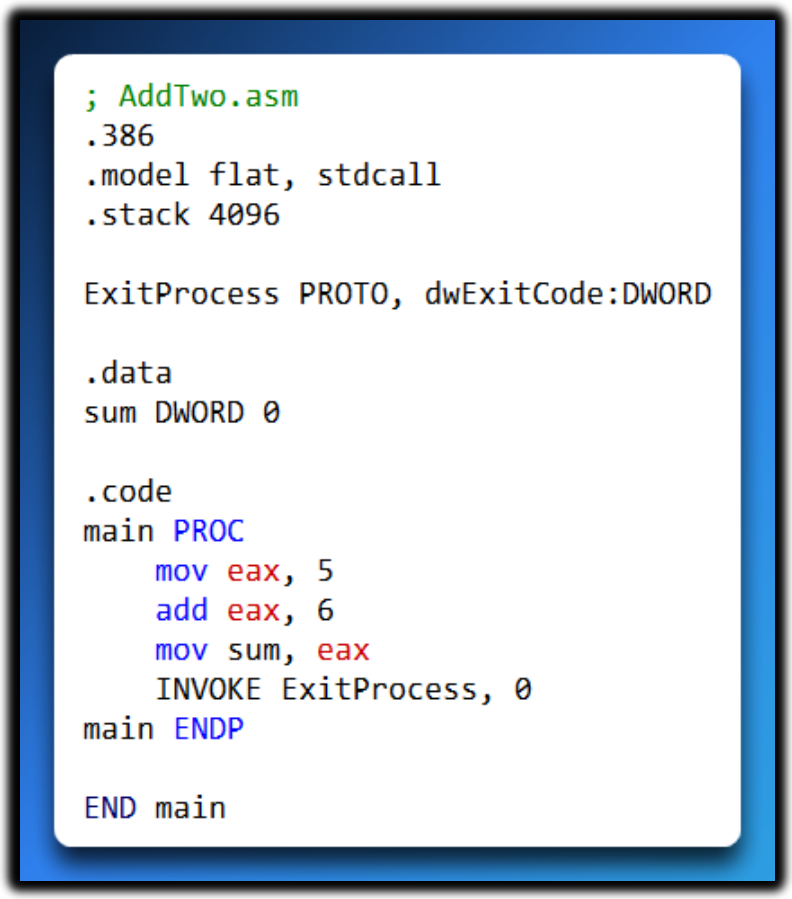
5. Everything Becomes Binary

No matter how you write an initializer:

- Decimal
- Hex
- Character literal

👉 The assembler converts it into **binary** before storing it in memory.

6. Worked Example: Adding Two Numbers



```
; AddTwo.asm
.386
.model flat, stdcall
.stack 4096

ExitProcess PROTO, dwExitCode:DWORD

.data
sum DWORD 0

.code
main PROC
    mov eax, 5
    add eax, 6
    mov sum, eax
    INVOKE ExitProcess, 0
main ENDP

END main
```

Defines a variable: **sum DWORD 0**

sum is a 4-byte integer initialized to 0; the program loads 5 into eax, adds 6 to it so eax becomes 11, and then stores the result: **mov sum, eax**

The program exits and final value is 11.

7. Debugging Tip

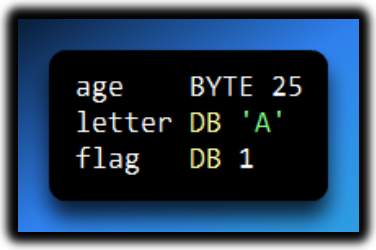
To observe the variable, set a breakpoint after mov sum, eax, step through the instructions, and watch sum in the debugger to see the memory value change in real time.

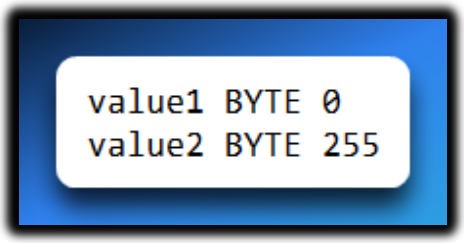
BYTE-SIZED DATA TYPES (Very Important)

BYTE / DB (Unsigned, 8 bits)

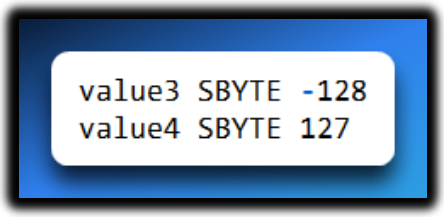
- Size: **1 byte (8 bits)**
- Range: **0 to 255**
- Used for: small numbers, characters, raw data

Examples:





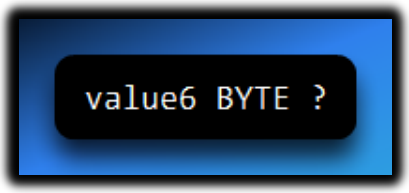
SBYTE is a signed 8-bit data type that occupies 1 byte of memory, can store values from −128 to +127, and is commonly used for small numbers that may be negative (for example: temp SBYTE -10 or change SBYTE 5).



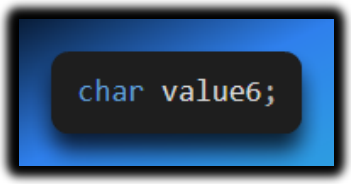
Signed vs Unsigned

- **Unsigned** → only positive values (and zero)
- **Signed** → positive **and** negative values

Uninitialized Variables (Important Warning)



Reserves 1 byte of memory but does not initialize it, so the value stored is random garbage just like



...in C language, which is why you must always initialize variables before using them.

Data Definition Directives

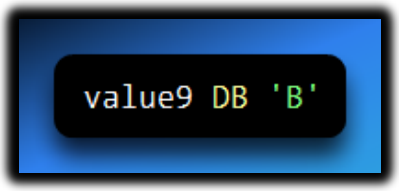
DIRECTIVE	MEANING	CAPACITY
DB	**Define Byte** (Legacy MASM style)	8-bit
BYTE	**Unsigned Byte** (Modern MASM style)	8-bit (0 to 255)
SBYTE	**Signed Byte**	8-bit (-128 to +127)

```
; These are functionally identical
value9 DB 'B' ; Allocated 1 byte
value9 BYTE 'B' ; Same allocation

; Range difference
val1 BYTE 255 ; Valid
val2 SBYTE -1 ; Valid (same bit pattern as 255)
```

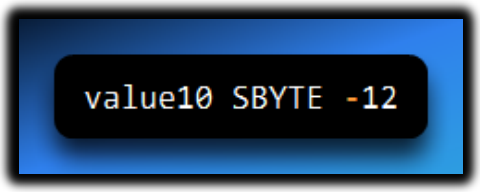
> **Pro Tip:** Use SBYTE when the value represents a temperature, coordinate, or any number that can be negative. Use BYTE for raw data or characters.

Character Initialization Example



- 'B' is a character
- ASCII value of 'B' = **66**
- Stored as **one byte**

Signed Byte Example



- Stores -12
- Uses signed representation
- Can hold negative values

Key Takeaways (Exam-Ready)

- Variables must have an initializer (or ?)
- ? means uninitialized (garbage value)
- BYTE / DB = unsigned 8-bit
- SBYTE = signed 8-bit
- Character literals are stored as ASCII values
- All data becomes binary in memory

💡 **Defining a variable means reserving memory and deciding how the bits should be interpreted.**

DATA DEFINITION PART 2: ARRAYS & SIZES

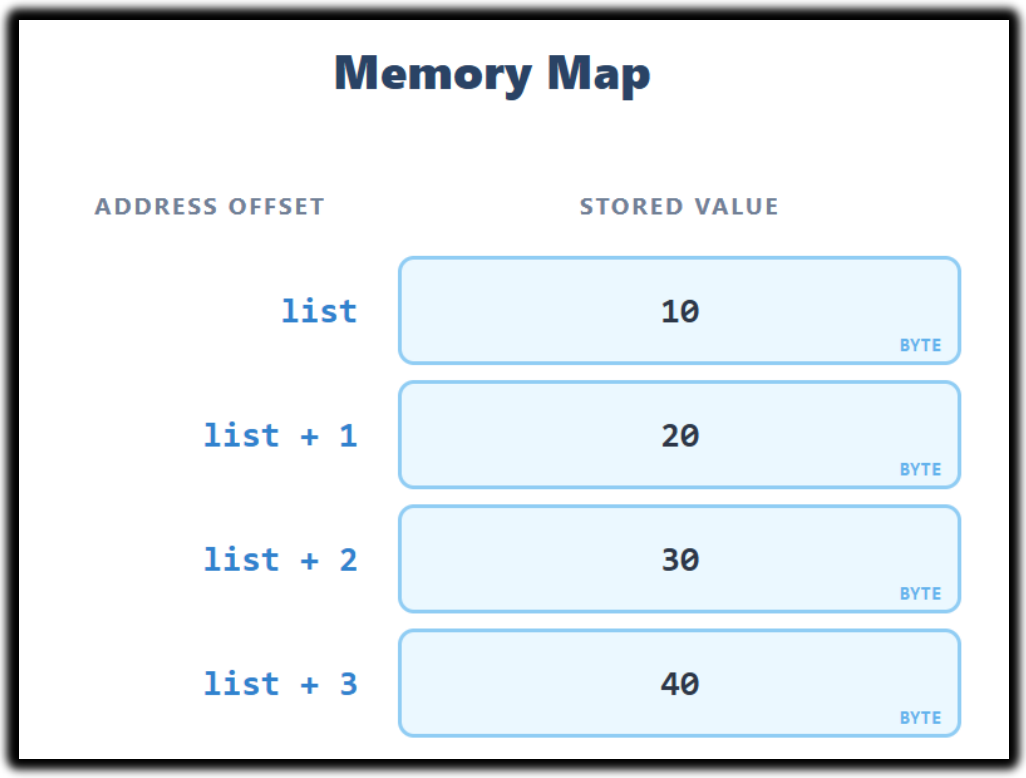
In high-level languages like C++ or Python, you create an array with brackets []. In Assembly, you just list values one after another.

Creating Arrays (The "Label" Trick)

When you define multiple values under one name, you are creating an array.

```
list BYTE 10, 20, 30, 40
```

You are creating **4 bytes in memory**:



- The **label list only points to the first value**, which is 10.
- The assembler doesn't automatically give names to the other values (20, 30, 40).
- To access them, you have to calculate their position relative to list.

For example:

- list → gives you 10
- list + 1 → gives you 20
- list + 2 → gives you 30
- list + 3 → gives you 40

So, the **label is like the "starting address" of your array**, and the other elements are reached by adding an offset in bytes.

The Memory Map:

If list starts at memory offset **0000**:

OFFSET (HEX)	VALUE (DEC)	DESCRIPTION
0000	10	This is where the label **list** points.
0001	20	This is **list + 1**.
0002	30	This is **list + 2**.
0003	40	This is **list + 3**.

Contiguous Memory

When you write:

```
list BYTE 10, 20, 30, 40
     BYTE 50, 60, 70, 80
     BYTE 81, 82, 83, 84
```

here's what's happening:

- The assembler **doesn't care about line breaks**.
- As long as you **don't give a new label**, it just keeps placing the numbers **right after the previous ones in memory**.
- So all 12 numbers are stored **one after another** in memory.

Memory layout looks like this:

Offset	Value
0	10
1	20
2	30
3	40
4	50
5	60
6	70
7	80
8	81
9	82
10	83
11	84

- The **label list points only to the first number (10 at offset 0)**.
- To access the others, you use **offsets**: list + 1 → 20, list + 4 → 50, etc.
- To the computer, this is **just one long strip of memory**, like a long row of boxes.

BYTE vs INTEGER Confusion 😵

Many beginners get confused because:

- In **C++/Java**, int is always **4 bytes (32 bits)**.
- In **Assembly**, numbers don't have a fixed size by default. They are stored in a **container (data type) you choose**.

Think of it like **boxes**:

Data Storage Capacities

BOX TYPE	SIZE	CAPACITY (RANGE)
BYTE	**1 Byte** 8 bits	U: 0 to 255 S: -128 to +127
DWORD	**4 Bytes** 32 bits	U: 0 to 4,294,967,295 S: -2,147,483,648 to +2,147,483,647

- **Number 10** fits easily in a BYTE (8-bit box).
- You **don't need a DWORD (4-byte box)** for such a small number.
- U is unsigned, S is signed.

Why use BYTE instead of DWORD?

1. **Memory efficiency**:
   - ✓ 1,000 small numbers (like ages 0–100) → 1,000 bytes with BYTE, but 4,000 bytes with DWORD.
   - ✓ Saving 75% of memory!
2. **Compatibility**:
   - ✓ Some old hardware or file formats expect data to be **in bytes**.

⚠ The Catch

- If you try to put a number bigger than 255 into a BYTE:
  - ✓ The assembler will **give an error**, or
  - ✓ It might **silently chop off the extra bits**, giving you the **wrong value**.

✅ In short:

- You can spread your data across multiple lines; the assembler just packs them in a row.
- BYTE is just a small container—use it when the number is small.
- Integers in assembly are **as big as you declare** (BYTE, WORD, DWORD, etc.), unlike high-level languages.

MIXING RADIXES (THE "SALAD BOWL")

Assembly doesn't care how you write the number.

You can mix Hex, Decimal, Binary, and Character literals in the same list.

They all get converted to binary in the end.

```
; All of these are valid in the same list
myList BYTE 10,      ; Decimal
            20h,     ; Hexadecimal
            'A',     ; Character (ASCII value 65)
            001010b  ; Binary
```

Big Idea to Remember

- **Labels point to the start:** list is just the address of the first item. To get the rest, you add to the address (Offset).
- **Contiguous Memory:** Data defined sequentially sits sequentially in RAM.
- **Size matters, not type:** You can store an "integer" in a BYTE as long as it fits (0-255). You don't always need a DWORD.

STRINGS

Strings Are Just Arrays of Bytes

In assembly, there is **no "string type"** like in high-level languages (C, Python, etc.).

- A **string is just a sequence of bytes**.
- Each **character** in the string is stored in **one byte**.
- The byte holds the **ASCII value** of the character.

Example:

```
greeting1 BYTE "Good afternoon", 0
```

MEMORY DUMP: 0x0000 - 0x000E LITTLE ENDIAN

ADDR	HEX	CHAR
0	47	G
1	6F	o
2	6F	o
3	64	d
4	20	(sp)
5	61	a
6	66	f
7	74	t
8	65	e
9	72	r
10	6E	n
11	6F	o
12	6F	o
13	6E	n
14	00	*NULL*

String Result: "Good afternoon"

✅ Notice that:

- Each **character takes 1 byte**.
- The **null terminator (0)** is also a **single byte** marking the end of the string.

Labels Are Just Starting Addresses

```
names1 DB "Learning Assembly then WinAPI", 0
names2 DB "Learning Reverse Engineering then C#", 0
names3 DB "Learning to be good as C programming", 0
```

- names1, names2, names3 are **labels**.
- A label is **just a pointer to the first byte** of the string in memory.
- The computer uses the label as a **starting reference**, but it doesn't know the length of the string unless you tell it.
- Everything after the first byte is **contiguous memory** (like we discussed with list BYTE 10,20…).

Why We Use BYTE

- We write BYTE because each **character fits in 1 byte**.
- Strings are really **arrays of bytes**, not a special datatype.

Think of it like this:

```
String "Hi" →  H   i   0
Bytes          48  69  00
```

- Each character is **stored in one box** (byte).
- The **null byte** (0) is the **stop signal** for string functions, like printf in C or WinAPI string routines.

Multi-line Strings & Special Characters

You can split strings across multiple lines or add special characters:

```
greeting BYTE "Welcome to the program", 0Dh, 0Ah, \
       "Created by ChatGPT", 0
```

- 0Dh = **carriage return** (CR) → moves cursor to start of line
- 0Ah = **line feed** (LF) → moves cursor down a line
- \ → line continuation character (lets you break one string across multiple lines)

Memory layout is still **just a sequence of bytes**, now including CR/LF:

```
"W", "e", "l", "c", "o", ..., 0Dh, 0Ah, "C", "r", "e", ... , 0
```

Everything remains a **byte**.

Putting It All Together

1. Each string is a **contiguous sequence of bytes** in memory.
2. The **label points to the first byte**.
3. Each **character = 1 byte (ASCII code)**.
4. **Null terminator (0) = 1 byte** marks the end.
5. Multi-line strings or special characters like CR/LF are just **additional bytes** in the same array.

So even the biggest sentence like "Learning Reverse Engineering then C#" is just **a row of bytes**:

Byte 0	Byte 1	...	Byte n	Byte n+1
'L'	'e'	...	'#'	0

✅ Key Insight

Strings in assembly are **not magical objects**.

- They are **arrays of bytes**.
- The **label is the pointer**.
- The **assembler only cares about memory**.
- Null terminators allow functions to **know where the string ends**.

DUP Operator (Duplicate Made Easy)

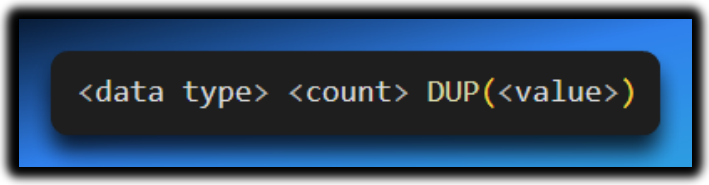
The **DUP operator** in assembly is all about **making copies**—it lets you allocate multiple pieces of memory and optionally initialize them with the same value.

Think of it as a "memory copy machine" for variables, arrays, strings, or even structures.

How it works:

- **Count:** How many times you want to repeat something.
- **Value:** What you want to repeat (it can be a number, a string, or even an uninitialized placeholder).

The syntax looks like this:



- <data type> could be BYTE, WORD, DWORD, etc.
- <count> is how many times you want to repeat.
- <value> is what you want to fill each slot with. If you leave it as ?, the memory is just reserved but contains random "garbage" values until you set it.

Examples:

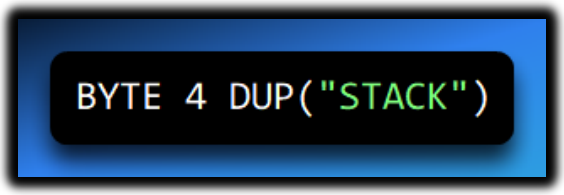
Allocate 20 bytes, all zero:

This creates a block of **20 bytes**, each containing 0.

Allocate 20 bytes, uninitialized:

Memory is reserved for 20 bytes, but the values are **undefined**. Think of it like an empty box—you can fill it later.

Create a repeated string:



This repeats the sequence "STACK" four times in memory, effectively making "STACKSTACKSTACKSTACK".

Allocate an array of 10 integers, initialized to zero:

Here, you get **10 integers**, each 4 bytes, all set to 0.

Allocate an array of structures:

```
STRUC MyStructure
    DWORD integer
    BYTE 4 string
ENDSTRUC

MyStructure 10 DUP(0)
```

This reserves space for **10 structures**, each containing a 4-byte integer and a 4-byte string.

Key idea:

Yes, **DUP literally means "duplicate"**. It's your way to **repeat a value or pattern** efficiently in memory without writing it out multiple times.

Whether you're filling arrays, initializing strings, or creating structures, **DUP saves time, space, and effort**.

Think of it like telling the assembler: *"Hey, make 10 of this, or 20 of that, all lined up in memory, and set them to this value—or leave them blank for now."*

WORD and SWORD

In assembly language, **WORD** and **SWORD** are used to work with **16-bit numbers**. Each 16-bit number takes **2 bytes** of memory.

WORD (Unsigned 16-bit Integer)

- **WORD** is for **unsigned numbers**, meaning only positive numbers from 0 to 65535.
- Each WORD reserves **2 bytes** in memory.

```
word1 WORD 65535   ; reserves 2 bytes and stores the largest unsigned 16-bit value
word3 WORD ?       ; reserves 2 bytes but leaves it uninitialized (random value)
```

SWORD (Signed 16-bit Integer)

- **SWORD** is for **signed numbers**, meaning it can store **negative and positive numbers** from -32768 to 32767.
- Each SWORD also takes **2 bytes** in memory.
- Example:

```
word2 SWORD -32768  ; reserves 2 bytes and stores the smallest signed 16-bit value
```

Key Idea:

Think of **WORD as a box that only holds positive numbers**, and **SWORD as a box that can hold negative numbers too**. Both boxes are **16 bits** (2 bytes) wide, so the memory size is the same, only the interpretation changes.

WORD Arrays

You can create **arrays of 16-bit numbers** in assembly, just like arrays in C, using either **explicit listing** or the **DUP operator**.

- **Memory layout:** Each 16-bit element occupies **2 bytes**. So if your array starts at memory offset 0000, the next element is at 0002, then 0004, and so on.
- **Example with explicit listing:**

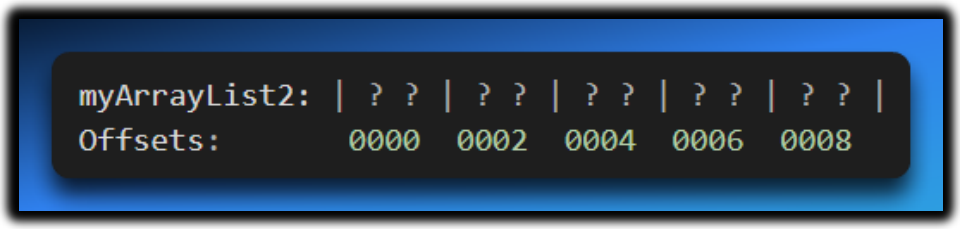
```
myList WORD 1000, 2000, 3000  ; 3 elements, each 2 bytes
```

Example with DUP (uninitialized array):

```
myArrayList2 WORD 5 DUP(?)  ; reserves 5 elements, each 2 bytes, values undefined
```

Here, ? means the elements are **uninitialized**. They have random "garbage" values until your code sets them.

Visualizing Memory (Conceptual):



Each element takes **2 bytes**, so to access the next element, you **increment the offset by 2**.

Summary:

- **WORD:** Unsigned 16-bit number (0 to 65535)
- **SWORD:** Signed 16-bit number (-32768 to 32767)
- Arrays: Use **listing or DUP** to store multiple words, remembering each takes 2 bytes in memory.

Offset	Value
0000:	1
0002:	2
0004:	3
0006:	4
0008:	5

DWORD and SDWORD

In assembly language, **DWORD** and **SDWORD** are used to work with **32-bit integers**. Each 32-bit number takes **4 bytes** of memory.

I. DWORD (Unsigned 32-bit Integer)

- **DWORD** is for **unsigned numbers**, meaning only positive numbers from 0 to 4,294,967,295.
- Each DWORD reserves **4 bytes** in memory.

```
vall DWORD 12345678h    ; reserves 4 bytes and stores the unsigned number 0x12345678
va13 DWORD 20 DUP(?)    ; reserves 20 DWORDs (80 bytes), all uninitialized
```

Usage Tip: You can also use **DD** (Define Doubleword) as a legacy directive. It works the same as DWORD:

```
vall DD 12345678h
```

II. SDWORD (Signed 32-bit Integer)

- **SDWORD** is for **signed numbers**, meaning it can store **negative and positive numbers** from -2,147,483,648 to 2,147,483,647.
- Each SDWORD also takes **4 bytes** in memory.

```
va12 SDWORD -12324352   ; reserves 4 bytes for a signed 32-bit integer
```

III. Arrays of 32-bit Numbers

You can create arrays of DWORDs or SDWORDs either by listing values explicitly or using the **DUP operator**:

Explicit initialization:

```
my32BitArray DWORD 10, 203, 482, 505  ; 4 elements, each 4 bytes
```

Uninitialized array using DUP:

```
myArrayList DWORD 5 DUP(?)   ; 5 DWORDs, each 4 bytes, values undefined
```

Memory layout concept:

- Each element occupies **4 bytes**, so if the first element is at offset 0000, the next is at 0004, then 0008, and so on.
- Arrays let you easily store multiple 32-bit numbers in **contiguous memory**.

IV. Extra Tip: DWORD for Offsets

You can also use **DWORD** to store the **32-bit memory offset of another variable**:

```
pval DWORD va13   ; pval now contains the offset (address) of va13
```

This is useful for pointers or referencing other variables in memory.

V. Summary:

- **DWORD:** Unsigned 32-bit integer, 4 bytes, 0 → 4,294,967,295
- **SDWORD:** Signed 32-bit integer, 4 bytes, -2,147,483,648 → 2,147,483,647
- Arrays: Use listing or **DUP** to store multiple DWORDs
- Legacy DD directive works the same as DWORD

QWORD (Quadword)

The **QWORD directive** in assembly language is used to allocate storage for **64-bit values**, meaning each QWORD takes **8 bytes** of memory.

Think of it as a really big box that can hold very large numbers.

1. Syntax and Usage

You can define QWORD values in two ways:

Standard directive:

```
quad1 QWORD 1234567812345678h  ; allocate 8 bytes and store the 64-bit value
```

Short form (DQ – Define Quadword):

```
quad2 DQ 1234567812345678h     ; same as above, just shorthand
```

Tip: The value must fit in 64 bits, otherwise the assembler will throw an error.

2. Memory Organization

Each QWORD takes **8 bytes**, so if you define multiple QWORDs in an array, memory offsets increase by 8 each time:

This is just like how **DWORD arrays** worked, but each element is double the size.

3. Arrays of QWORDs

Just like with BYTE or DWORD, you can use the **DUP operator** to define multiple QWORDs at once:

```
myArray QWORD 10 DUP(0)  ; allocate 10 QWORDs, each initialized to 0
```

Each element is **8 bytes**, so this reserves **80 bytes total** (10 × 8).

Using ? instead of 0 leaves them uninitialized:

```
myArray QWORD 10 DUP(?)  ; 10 QWORDs, uninitialized
```

4. QWORD and Registers

- In **32-bit mode**, your registers like **EAX** are 32 bits, so storing a 64-bit QWORD might need **two 32-bit registers** or special memory instructions.
- In **64-bit mode**, the **RAX** register can hold a full QWORD directly.

Example:

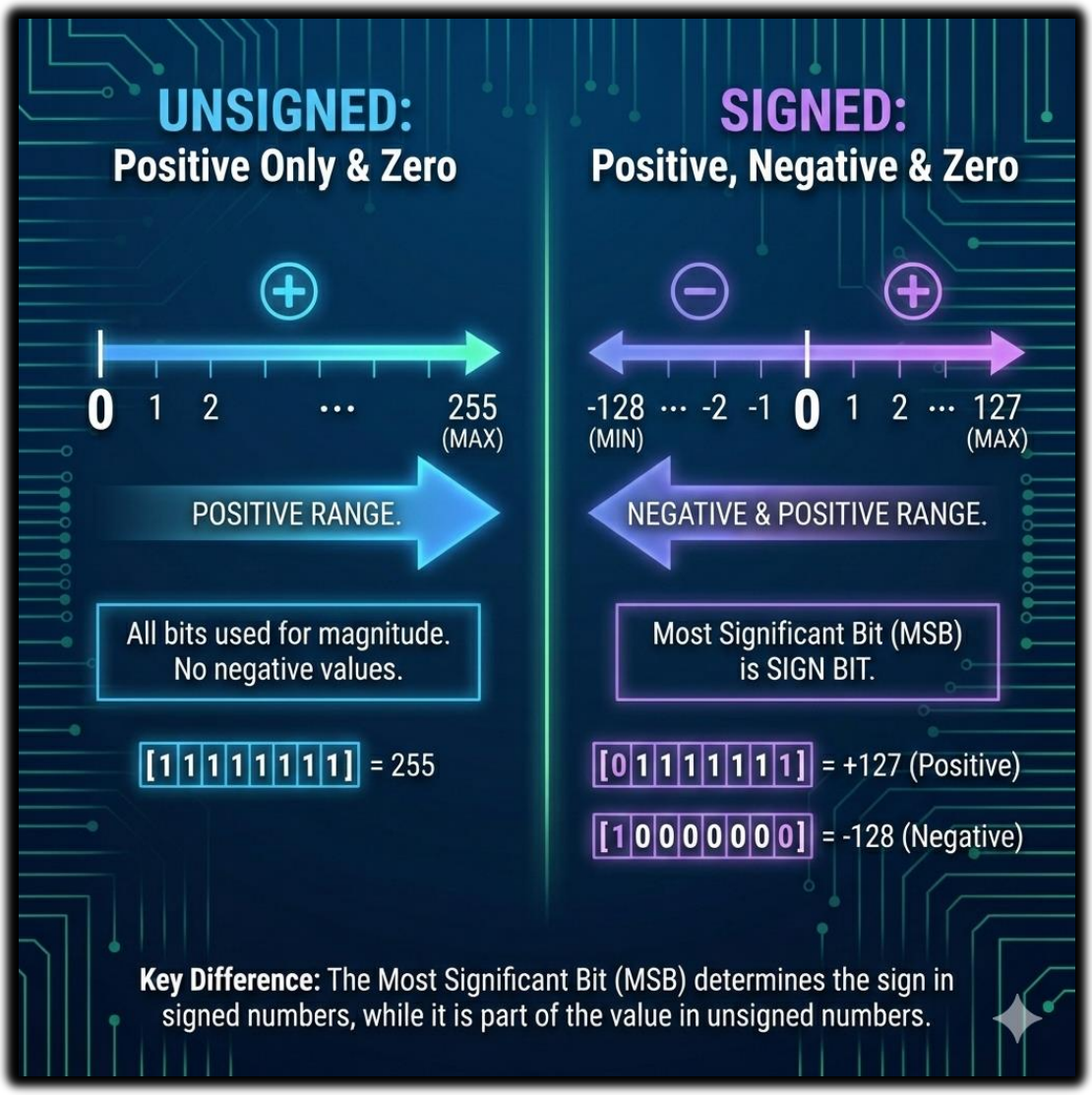
- This is important if you start working with large numbers, addresses, or high-precision calculations.

5. Summary Notes

- QWORD = 64 bits = 8 bytes
- QWORD can be initialized directly or with DUP
- Short form DQ is equivalent to QWORD
- Arrays increment in memory by 8 bytes per element
- 32-bit registers can't hold QWORDs directly; use 64-bit registers or split into two 32-bit halves

💡 **Memory efficiency tip:**
Use QWORD only when you need numbers bigger than 32 bits, otherwise DWORD is enough and takes half the memory.

Never forget this concept in Assembly:



Let's continue….

PACKED BCD AND TBYTE

Packed BCD (Binary Coded Decimal) is a **special way to represent decimal numbers in binary**, designed for **efficiency and precision**, especially in financial or scientific applications.

1. What is Packed BCD?

Packed BCD stores decimal digits in **pairs**, with **two decimal digits per byte**.

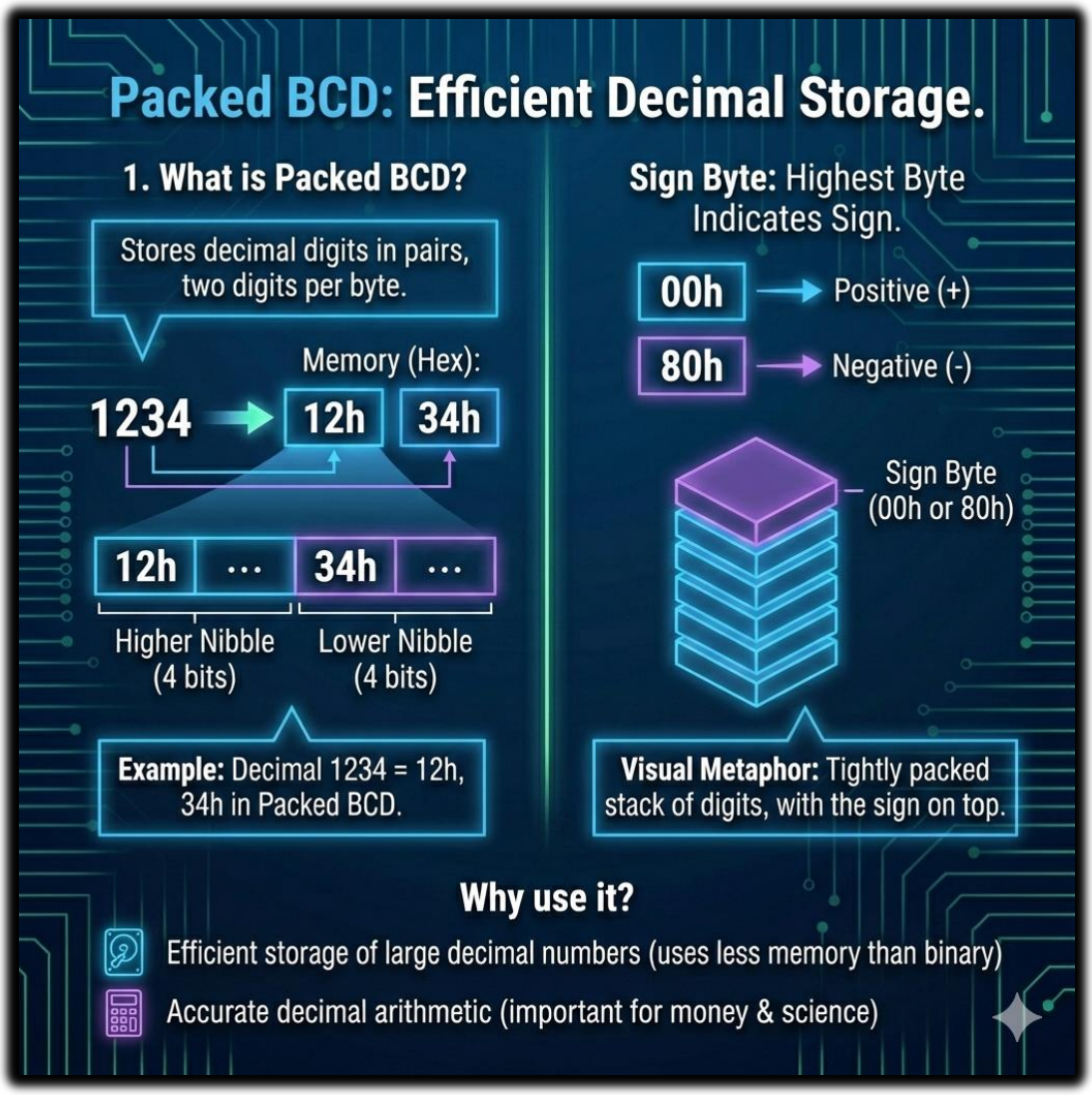
Example: The decimal number 1234 in packed BCD is stored as **34 12** (hex representation in memory).

- The **lower nibble** of a byte stores one digit.
- The **higher nibble** stores the next digit.

Sign byte: The highest byte of a packed BCD variable indicates the sign.

- 00h → Positive
- 80h → Negative

Think of it like a **tightly packed stack of digits**, with the sign sitting on top.



Why use it?

- Efficient storage of large decimal numbers (takes less memory than converting to binary integers).
- Accurate decimal arithmetic — important for **money calculations**, **scientific data**, and some **embedded systems**.

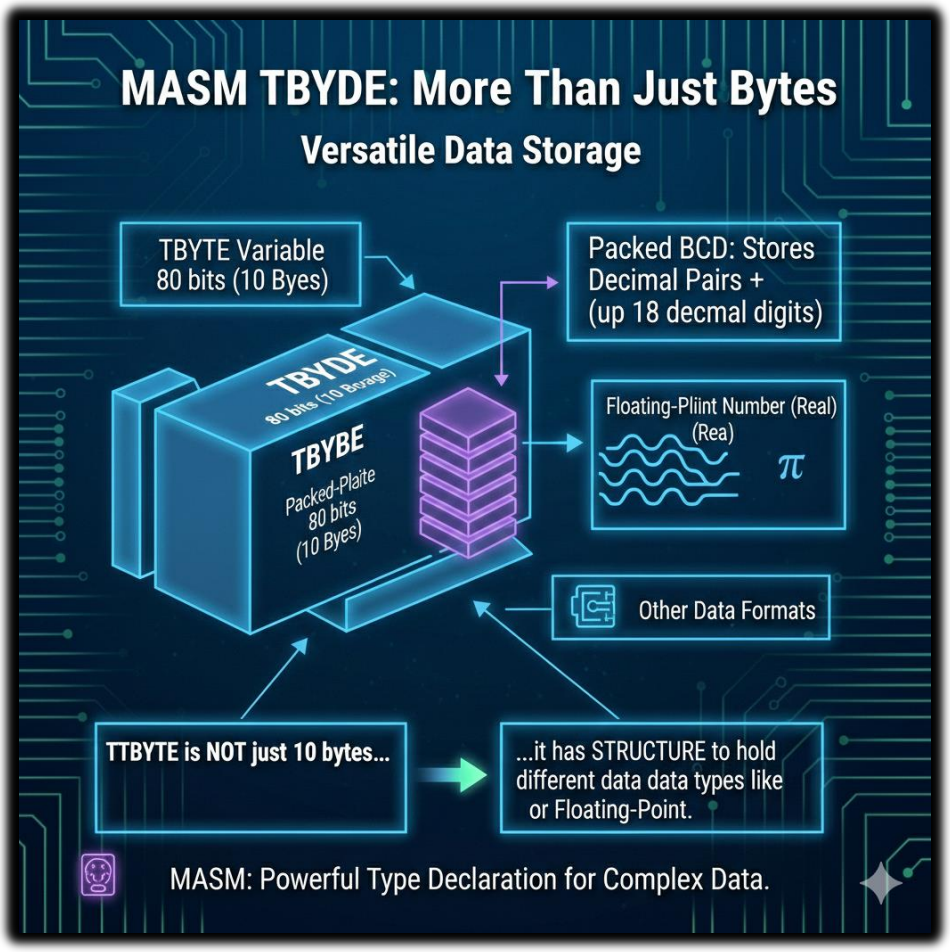
2. The TBYTE Directive

In MASM, **TBYTE** is used to declare variables that can store **packed BCD data**.

Even though TBYTE is **80 bits (10 bytes)**, it isn't just "10 bytes of storage" — it can also hold **floating-point numbers** or other data formats.

Memory layout for a TBYTE BCD number:

- **1st byte:** Sign
- **Next 9 bytes:** Decimal digits, 2 digits per byte



Example: Declaring a packed BCD variable

```
my_bcd_variable TBYTE 1234h   ; NOT correct, needs hex BCD
```

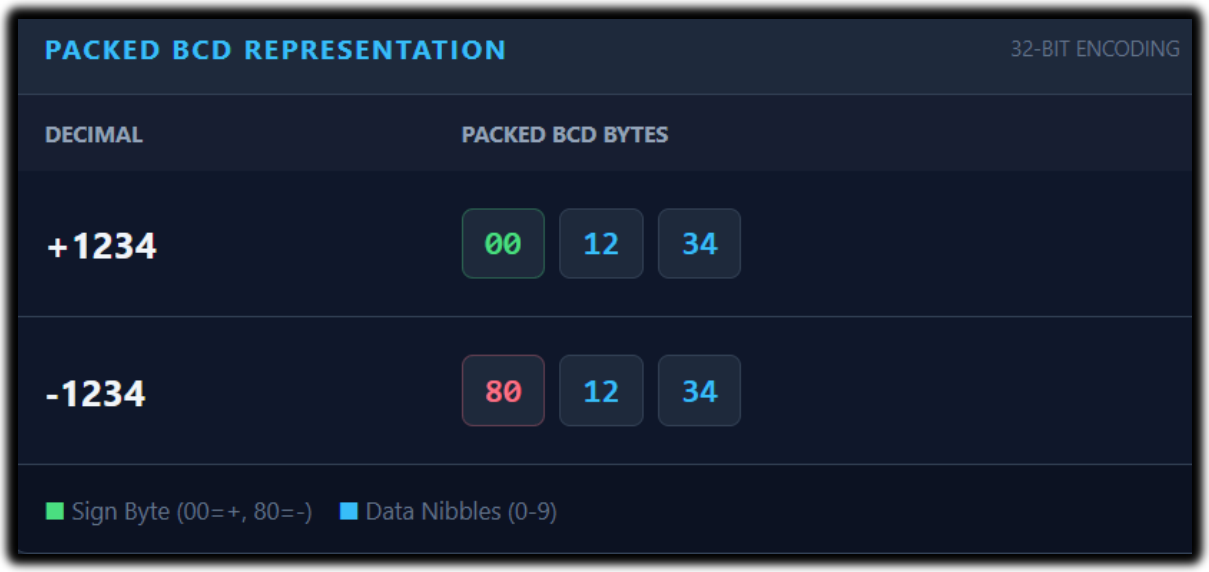
The correct way:

```
my_bcd_variable TBYTE 001234h  ; Positive 1234 in packed BCD
my_bcd_variable TBYTE 801234h  ; Negative 1234 in packed BCD
```

Important: MASM does **not** automatically convert decimal numbers to BCD. You must write them in **hexadecimal BCD form**.

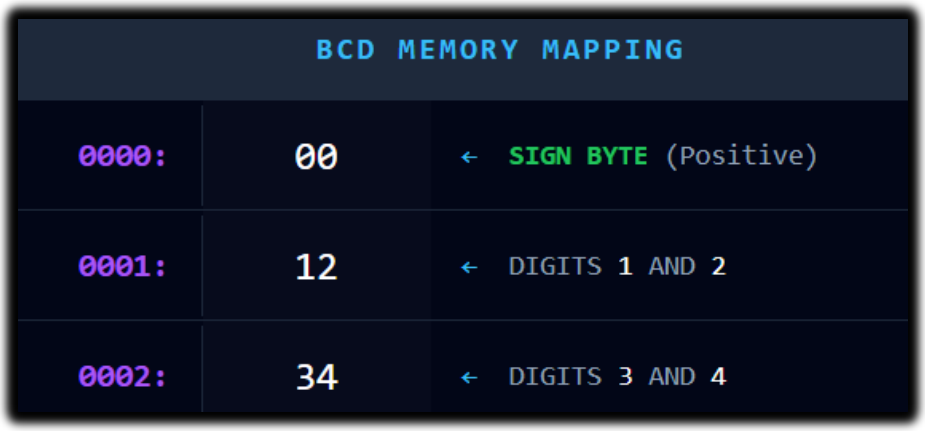
3. Packed BCD in Memory

Let's look at **1234** as an example:



- Each byte after the sign byte stores **two decimal digits**.
- Positive numbers start with 00h. Negative numbers start with 80h.

Visualizing storage:



If the number were larger, you'd continue storing 2 digits per byte.

4. Declaring Arrays of Packed BCD

You can use **DUP** with TBYTE too:

```
myBCDArray TBYTE 5 DUP(0)  ; 5 packed BCD variables, initialized to 0
```

- Each element takes **10 bytes**.
- Total memory: 5 × 10 = 50 bytes

Uninitialized array:

```
myBCDArray TBYTE 5 DUP(?)  ; 5 packed BCD variables, uninitialized
```

5. Converting Real Numbers to Packed BCD

Sometimes, you have **floating-point numbers** (REAL4, REAL8, etc.) and want them as packed BCD. This is done using **FPU instructions**:

```
posVal REAL4 1.5        ; a real number
bcdVal TBYTE ?          ; packed BCD variable

FLD posVal              ; load real number onto FPU stack
FBSTP bcdVal            ; convert to packed BCD, round to nearest integer
```

- FLD → Load floating-point number onto FPU stack
- FBSTP → Convert the value to **packed BCD** and store it in bcdVal

Example: If posVal = 1.5, then bcdVal will store 02 in packed BCD.

6. Why Packed BCD Matters

- **Efficiency:** Stores **two digits per byte** instead of wasting 8 bits for a single decimal digit.
- **Accuracy:** No rounding errors when doing **decimal math**, unlike floating-point binary.
- **Applications:** Financial apps, calculators, scientific measurements, embedded systems.

Analogy: Think of Packed BCD as **a neatly packed number stack**, where each box holds 2 digits, and the top box holds the sign. Computers can easily read, write, and calculate with these numbers without wasting memory.

7. Quick Reference

- **Directive:** TBYTE
- **Size:** 10 bytes (80 bits)
- **Sign byte:** First byte, 00h positive, 80h negative
- **Digits:** Next 9 bytes, 2 digits per byte
- **Initialization:** Must be in **hexadecimal**
- **Arrays:** Use DUP operator for multiple variables

✅ Example: Complete Packed BCD Declaration

```
; Positive 1234
myBCD1 TBYTE 001234h

; Negative 1234
myBCD2 TBYTE 801234h

; Array of 3 packed BCD numbers, uninitialized
myBCDArray TBYTE 3 DUP(?)
```

Memory usage:

- myBCD1 → 10 bytes
- myBCD2 → 10 bytes
- myBCDArray → 30 bytes (3 × 10)

Packed BCD is essentially **a super-efficient way to store decimal numbers** where every byte counts. The **TBYTE directive** is just your tool for declaring variables that can hold packed BCD or other 10-byte data types.

DEFINING FLOATING-POINT TYPES IN MASM

Floating-point numbers are used to represent **real numbers**, meaning numbers with fractional parts, like 1.23456789.

In MASM, there are **three main floating-point types**:

TYPE	SIZE	PRECISION	TYPICAL RANGE
REAL4	4 bytes	7 digits	±3.4E38 to ±1.2E-38
REAL8	8 bytes	15 digits	±1.7E308 to ±2.4E-308
REAL10	10 bytes	19 digits	±4.9E324 to ±1.1E-324

I. Single-Precision: REAL4

- **Size:** 4 bytes (32 bits)
- **Precision:** ~7 significant digits
- **Range:** $\pm 3.4 \times 10^{38}$ to $\pm 1.2 \times 10^{-38}$

Example:

```
rVal1 REAL4 -1.2    ; Declare a single-precision floating-point variable and initialize it to -1.2
```

- Memory usage: 4 bytes
- Good for general-purpose calculations where moderate precision is enough.

II. Double-Precision: REAL8

- **Size:** 8 bytes (64 bits)
- **Precision:** ~15 significant digits
- **Range:** $\pm 1.7 \times 10^{308}$ to $\pm 2.4 \times 10^{-308}$

Example:

```
rVal2 REAL8 3.2E-260  ; Declare a double-precision variable for extremely small values
```

- Memory usage: 8 bytes
- Use when **high precision** is required, such as scientific calculations or very small/large numbers.

III. Extended-Precision: REAL10

- **Size:** 10 bytes (80 bits)
- **Precision:** ~19 significant digits
- **Range:** $\pm 4.9 \times 10^{324}$ to $\pm 1.1 \times 10^{-324}$

Example:

```
rVal3 REAL10 4.6E+4096  ; Extended-precision variable for extremely large numbers
```

- Memory usage: 10 bytes
- Ideal for **high-precision math**, financial, or scientific computations where single- or double-precision isn't enough.

IV. Arrays of Floating-Point Numbers

You can use the **DUP operator** to declare arrays of floating-point variables:

```
ShortArray REAL4 20 DUP(0.0)  ; Array of 20 single-precision floats, all initialized to 0.0
```

- **Memory usage:** 20 × 4 = 80 bytes
- Efficient way to initialize large arrays of floating-point numbers.

V. Using DD, DQ, and DT Directives

MASM also allows you to declare floating-point numbers with **DD, DQ, and DT**, which are legacy equivalents:

DIRECTIVE	USE	SIZE
DD	Short Real (Single Precision)	4 bytes
DQ	Long Real (Double Precision)	8 bytes
DT	Extended Real	10 bytes

Note: DD stands for *Define Doubleword*, DQ for *Define Quadword*, and DT for *Define Ten-byte*.

Examples:

```
rVal1 DD -1.2        ; Single-precision
rVal2 DQ 3.2E-260    ; Double-precision
rVal3 DT 4.6E+4096   ; Extended-precision
```

This is equivalent to using **REAL4, REAL8, REAL10**. Use whichever style you prefer, but **REALx directives are clearer** for readability.

VI. Precision vs Range

- **Precision**: Number of significant digits the type can represent.
- **Range**: Maximum and minimum values it can store.

TYPE	SIGNIFICANT DIGITS	APPROXIMATE RANGE
REAL4	7	±3.4E38 to ±1.2E-38
REAL8	15	±1.7E308 to ±2.4E-308
REAL10	19	±4.9E324 to ±1.1E-324

Tip: Extended-precision is overkill for most applications but is useful for **scientific or financial computing**.

VII. Real Numbers vs Floating-Point Numbers

- **Real Numbers (math concept)**: Infinite precision and size. Can include fractions, irrational numbers (like π), etc.
- **Floating-Point Numbers (computer representation)**: Approximation of real numbers. Finite precision and range, limited by storage size (4, 8, or 10 bytes).

Key points:

- Floating-point types approximate real numbers.
- Precision is **limited**.
- They can represent extremely large or small numbers, but not perfectly.

✅ Quick Example Summary

```
; Single-precision float
rVal1 REAL4 -1.2

; Double-precision float
rVal2 REAL8 3.2E-260

; Extended-precision float
rVal3 REAL10 4.6E+4096

; Array of 20 single-precision floats
ShortArray REAL4 20 DUP(0.0)
```

- **Memory usage:** 4, 8, 10 bytes for each variable
- **Arrays**: multiply size by element count

Summary:

Floating-point types in MASM (REAL4, REAL8, REAL10) let you store real numbers of varying precision. Choose **REAL4** for normal calculations, **REAL8** for high-precision scientific data, and **REAL10** for extreme precision. Arrays can be initialized using **DUP**, and legacy directives **DD, DQ, DT** are equivalent but less readable.

ADD NUMBERS PROGRAM (ADDING INTEGER VARIABLES)

This program shows how to add **three 32-bit integers stored in memory** and store the result in a fourth variable.

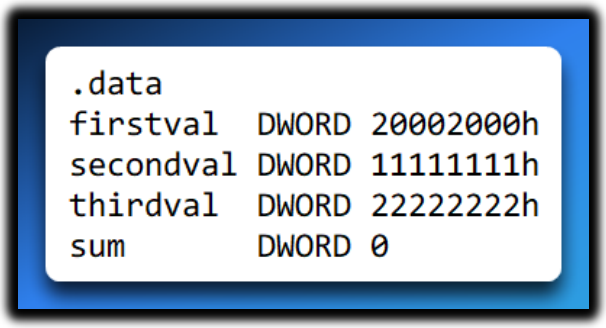
```
; AddVariables.asm
.386
.model flat, stdcall
.stack 4096

ExitProcess PROTO, dwExitCode:DWORD
```

Explanation:

- .386: Enables 80386 instructions, meaning we can use 32-bit registers like EAX.
- .model flat, stdcall: Flat memory model (all memory in one linear space) with the stdcall calling convention.
- .stack 4096: Reserves a 4 KB stack for function calls and local variables.
- ExitProcess PROTO: Declares a prototype for the Windows API function ExitProcess, which terminates the program.

Declaring Data (Variables)



- .data section: Where all global variables and constants are stored.
- DWORD: Each variable is **4 bytes (32 bits)**.
- Hexadecimal values like 20002000h are **base-16 numbers**, which the CPU stores as binary in memory.

Memory Layout (example):

VARIABLE	HEX VALUE	BINARY (32-BIT PATTERN)
firstval	20002000h	0010 0000 0000 0000 0010 0000 0000 0000
secondval	11111111h	0001 0001 0001 0001 0001 0001 0001 0001
thirdval	22222222h	0010 0010 0010 0010 0010 0010 0010 0010
sum	00000000h	0000 0000 0000 0000 0000 0000 0000 0000

Each value occupies 4 bytes in memory, stored consecutively unless alignment or padding is introduced.

Code Section (Adding the Variables)

```
.code
main PROC
    mov eax, firstval   ; Load firstval into the EAX register
    add eax, secondval  ; Add secondval to EAX
    add eax, thirdval   ; Add thirdval to EAX
    mov sum, eax        ; Store the final result in sum
    INVOKE ExitProcess, 0
main ENDP
END main
```

Step-by-Step Explanation:

1. **mov eax, firstval**
   - Moves the value of firstval (20002000h) into the **32-bit register EAX**.
2. **add eax, secondval**
   - Adds secondval (11111111h) to the current value in EAX.
   - EAX now holds 20002000h + 11111111h.
3. **add eax, thirdval**
   - Adds thirdval (22222222h) to EAX.
   - EAX now holds the **sum of the three variables**.
4. **mov sum, eax**
   - Moves the value in EAX into the variable sum in memory.
5. **INVOKE ExitProcess, 0**
   - Calls the Windows API function to terminate the program.
   - The program exits cleanly with return code 0.

Key Points to Understand

Registers vs Memory:

- Registers (EAX) are **fast temporary storage** inside the CPU.
- Memory (firstval, sum) is slower but permanent for the program.

Hexadecimal Representation:

- Each hex digit represents **4 bits**, so 8 digits = 32 bits = 1 DWORD.

Adding Variables:

- You **can't directly add memory-to-memory** in x86 assembly.
- You must move one value to a register (EAX), then add the others.

Storing Result:

- Once the calculation is done in the register, move it back to memory with mov sum, eax.

Optional Visual Memory Diagram

```
Memory Layout (.data section)
Address     Variable      Value (Hex)
0000        firstval      20002000
0004        secondval     11111111
0008        thirdval      22222222
000C        sum           00000000  ; Will store the result after addition
```

- After execution: sum = 20002000h + 11111111h + 22222222h = 53315333h

✅ Summary:

1. Define variables in .data using DWORD for 32-bit integers.
2. Use a register (EAX) to perform arithmetic.
3. Use mov and add instructions to manipulate and sum values.
4. Store the result back to memory.

This is the **memory + register way** to add numbers in assembly.

DATA ORGANIZATION: THE ENDIANNESS DEBATE

When you have a number bigger than one byte—like a 32-bit integer—it takes up multiple memory slots. This immediately raises a question:

Which byte goes first in memory?

Do we store the number left-to-right (like how we normally write numbers) or right-to-left? The answer depends on the system, and this is called **Byte Ordering**, or more formally, **Endianness**.

Little Endian (The Intel Way)

Most x86 processors (Intel, AMD) use **Little-Endian**. Here's what that means:

1. The Rule

- The "Little End," meaning the **Least Significant Byte (LSB)**, goes into the **lowest memory address**.
- The "Most Significant Byte (MSB)" is stored last, at a higher address.

2. Why Little-Endian?

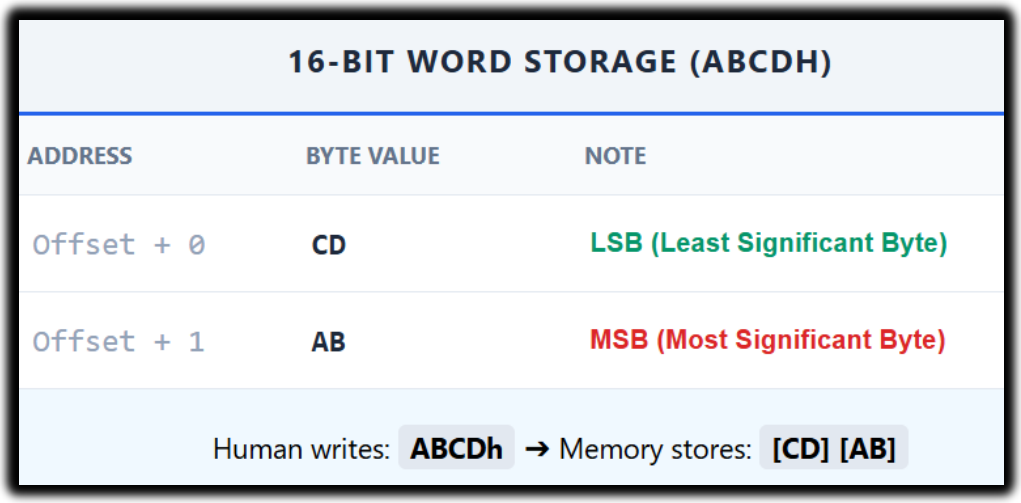
- The CPU stores the LSB first because it's the "least important" byte in the overall number.
- This ordering actually makes some **math operations more efficient**, since addition and other arithmetic start from the LSB.

Think of it like writing numbers on paper backward so you can start calculating immediately with the ones place.

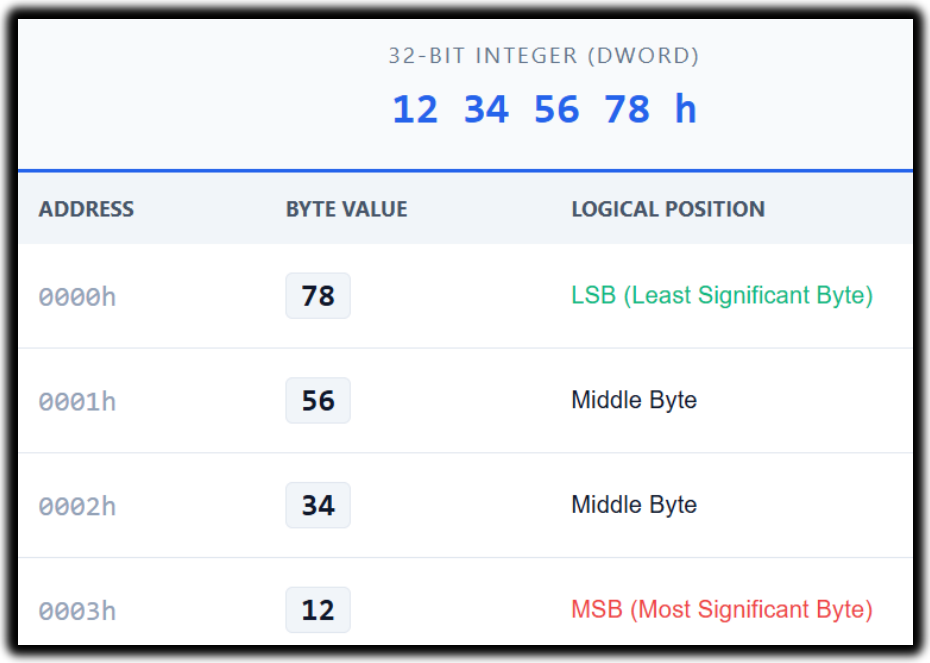
3. Example

Hex number: 12345678h

- 78 → LSB ("Little End")
- 12 → MSB ("Big End")



Memory layout in Little-Endian:



Notice how it looks reversed in memory—but **inside a CPU register**, it still looks normal (12345678h). Only the memory order is "scrambled."

Big Endian (The "Human" Way)

Other systems, like some ARM or MIPS processors, use **Big-Endian**.

1. The Rule

- The "Big End," meaning the **Most Significant Byte (MSB)**, goes into the **lowest memory address**.
- The LSB comes last.

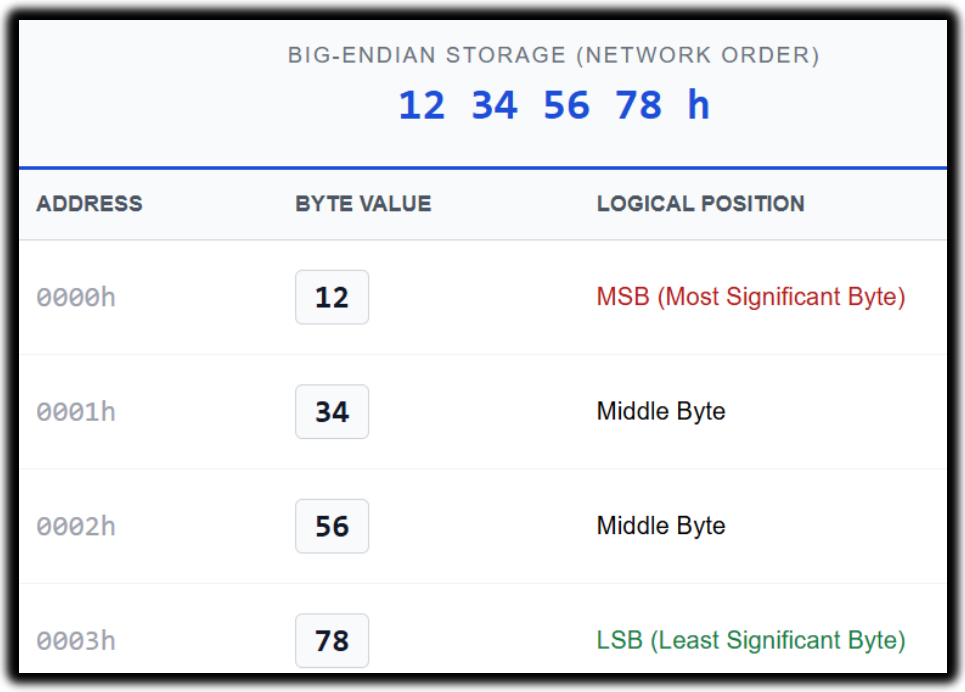
2. Why Big-Endian?

- This looks natural to humans, because the number is stored left-to-right, just like we read or write it on paper.

3. Example

Same number: 12345678h

Memory layout in Big-Endian:



Now memory mirrors the way we normally write numbers, but certain CPU operations may be slightly less convenient.

Why Endianess Matters

If you **save a file on a Little-Endian PC** and try to read it on a **Big-Endian machine** without converting it, your numbers can become completely scrambled.

- Example: Number 1 is stored in 32 bits: 00 00 00 01 (Little-Endian).
- A Big-Endian system might read it as 01 00 00 00, which equals **16,777,216** instead of 1.

Bottom line: Understanding endianness is crucial for:

- **Porting software** across platforms
- **Networking**, where different systems communicate binary data
- **File formats**, where byte order is explicitly defined

Big Ideas to Remember

1. **Little Endian (x86):**
    - LSB comes first (lowest memory address)
    - Useful for CPU arithmetic
2. **Big Endian:**
    - MSB comes first
    - Aligns with human reading/writing habits
3. **Memory vs. Register:**
    - Inside a register (like EAX), numbers always appear correctly (12345678h).
    - The "scrambling" only happens when stored in RAM.

Quick Memory Trick

- **Little-Endian:** Think of it like writing a number "backwards" in memory so the CPU can start calculating from the ones place immediately.
- **Big-Endian:** Think of it like writing numbers normally—humans like it, CPUs don't mind, but arithmetic can be less convenient.

Declaring Uninitialized Data

I. The .DATA? Directive: Ghost Storage

In earlier notes, we used .DATA to define variables with **specific values**, like:

```
count DWORD 10
```

This works when you know the exact value your variable should start with. But what happens if you need a **massive buffer**, like a temporary storage for an image or a large array, and you **don't know the values yet**?

That's where **.DATA?** comes in.

II. The Concept

.DATA? tells the assembler:

"I want to reserve memory for these variables, but I don't want to initialize them yet."

- Variables declared here **exist in memory at runtime**, but the **compiled executable doesn't store any actual values** for them.
- In other words, your program **asks the OS** to allocate RAM when it runs, instead of bloating the .exe file with zeros or other initial values.

III. The Magic Behind the Scenes

Here's what happens under the hood:

Disk vs RAM

- .DATA variables with initial values are **stored directly in the .exe file**. That means the file grows by the size of the initialized data.
- .DATA? variables **occupy no extra space on disk**. The program only tells the OS: "When you run me, please give me this much memory."

Runtime Allocation

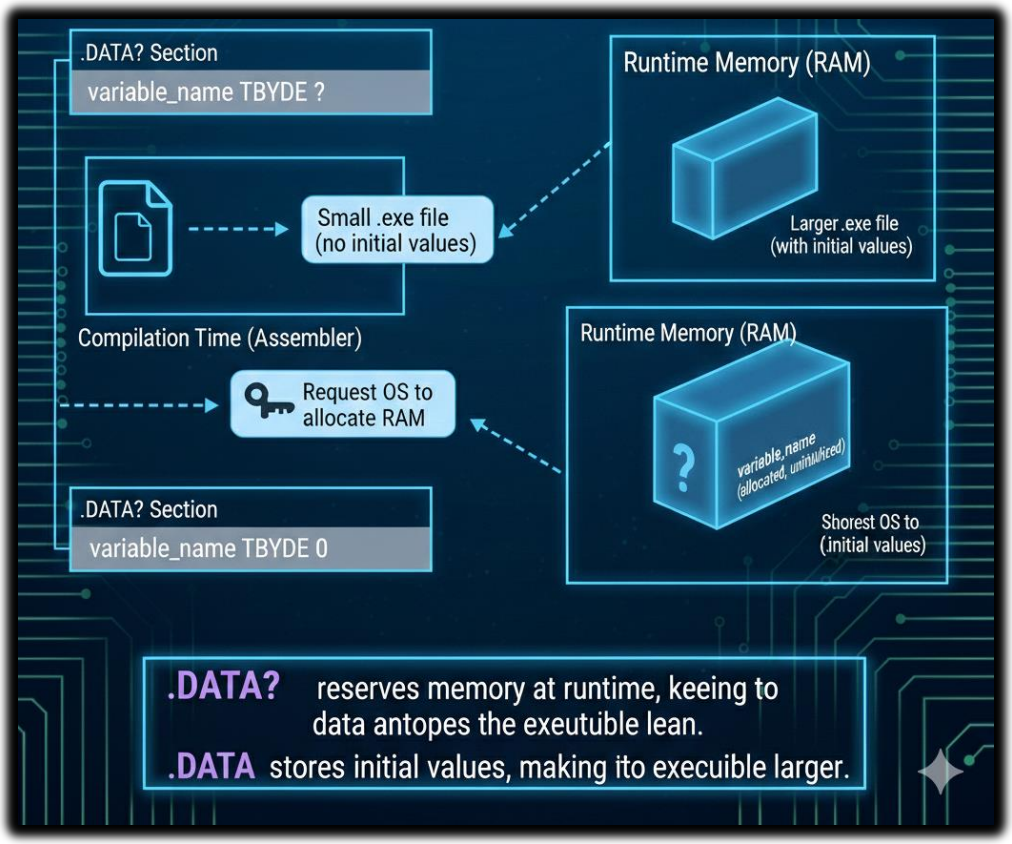
- The OS reserves the memory in your process's **Data Segment** (part of RAM) **when your program starts**.
- The bytes in .DATA? contain **garbage values**, meaning whatever bits were previously in that RAM location—so you **must initialize them before use**.

Why this matters

- Using .DATA? is a **disk-space optimization**. Large arrays can dramatically increase your executable's size if initialized with .DATA.
- It's also a **performance-friendly habit**, because loading huge initialized blocks from disk is slower than just letting the OS give you "empty" RAM.

3. Fat File vs Skinny File: An Example

Let's compare:




```
.data
smallArray DWORD 10 DUP(0)     ; 10 * 4 bytes = 40 bytes stored in .exe

.data?
bigArray DWORD 5000 DUP(?)     ; 5000 * 4 bytes = 20,000 bytes reserved
```

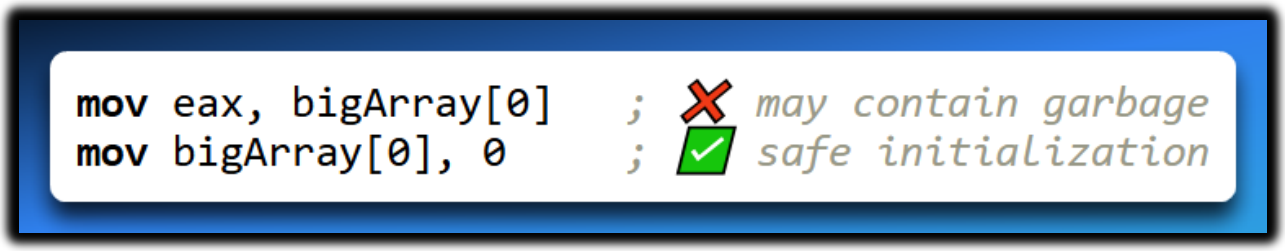
IMPACT ON EXECUTABLE (.EXE) SIZE

VARIABLE	.EXE GROWTH	NOTES
`smallArray`	**+40 bytes**	Defined in `.data` using `DUP(0)`. Because it is **initialized**, the zeros must be physically stored inside the file.
`bigArray`	**0 bytes**	Defined in `.data?` using `DUP(?)`. Because it is **uninitialized**, the OS is simply told to "reserve space" in RAM at runtime.

💡 **Key insight:** .DATA? is like putting a **sticky note in memory** that says: *"Set aside this much space, but I don't care what's in it yet."*

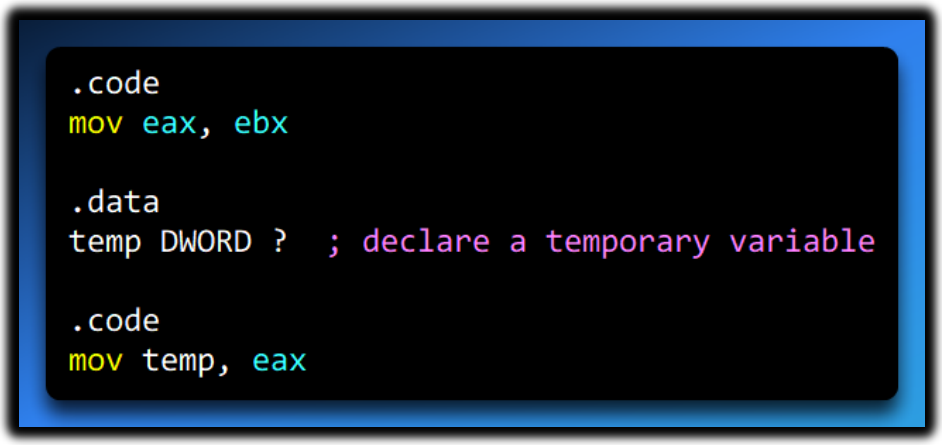
4. Garbage Alert ⚠

- The ? symbol doesn't zero out memory—it **leaves random bits** that were already in RAM.
- Always **initialize variables** from .DATA? **before reading them**, otherwise your program may behave unpredictably.



MIXING CODE AND DATA (THE "MESSY ROOM" METHOD)

MASM allows you to switch between code and data anywhere:



What really happens:

- The assembler **automatically moves temp to the Data Segment**, even if you declared it in the middle of your code.
- CPU execution remains **linear**, and registers store temporary values as expected.

Best Practice

- While possible, **don't scatter data declarations in code**.
- Keep .data and .code **separate** for clarity and maintainability.

DECLARING TYPES: THE CHEAT SHEET

When using .DATA?, you **still need to pick the right size**, even if the value is unknown.

INTEGER DATA TYPES (MASM)

TYPE	SIZE	CATEGORY	EXAMPLE DECLARATION
BYTE	8 bits	UNSIGNED	`var1 BYTE ?`
SBYTE	8 bits	SIGNED	`var2 SBYTE ?`
WORD	16 bits	UNSIGNED	`var3 WORD ?`
SWORD	16 bits	SIGNED	`var4 SWORD ?`
DWORD	32 bits	UNSIGNED	`var5 DWORD ?`
SDWORD	32 bits	SIGNED	`var6 SDWORD ?`
QWORD	64 bits	UNSIGNED	`var7 QWORD ?`

Remember: .DATA? = "Reserve this space at runtime."
? = "I don't care what's inside yet—initialize it before use!"

Readability Pro-Tips

Writing assembly is **hard enough** without messy formatting. Follow these rules:

1. **Capitalize directives** (.DATA?, .CODE) for clarity.
2. **Indent consistently**—makes loops and logic easier to scan.
3. **Use comments liberally**—Future-You will thank Present-You.
4. **Clear labels**—especially for jumps or memory offsets.

Big Idea to Remember

- .DATA? = **Runtime Allocation**: space is reserved **in RAM**, not on disk.
- ? = **Uninitialized**: the memory contains random garbage.
- ✅ Always **initialize** before reading.

💡 **Analogy:** .DATA is like packing boxes with items before shipping—takes space in the truck (.exe). .DATA? is like telling the warehouse: "Hold this space for me when I arrive"—the truck stays empty until runtime.

SYMBOLIC CONSTANTS (MAKING ASSEMBLY HUMAN)

Assembly is already low-level and unforgiving. One of the few tools you get to make it *readable* and *maintainable* is **symbolic constants**.

At their core:

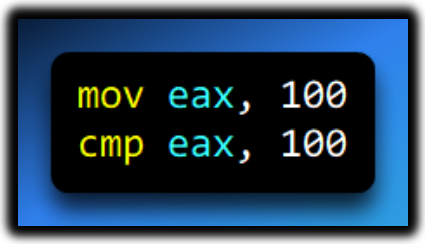
A symbolic constant is a name that represents a fixed value that never changes.

Instead of scattering raw numbers and strings all over your code, you give them names. That way:

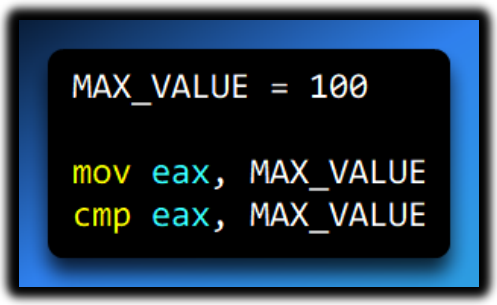
- Your code reads like intent, not math homework
- You change a value **once**, not everywhere
- You avoid "magic numbers" that nobody remembers later

I. THE BIG IDEA

Instead of this:



You do this:



Same machine code.
Much better for humans.

If you ever want MAX_VALUE to be 200 instead?
You change **one line**.

II. WHAT SYMBOLIC CONSTANTS ARE (AND ARE NOT)

✅ **What they are**

- **Compile-time substitutions**
- The assembler literally **replaces the name with the value**
- They do **not** exist in memory
- They do **not** take space in RAM or the .exe

❌ **What they are not**

- They are **not variables**
- You cannot change them at runtime
- You cannot take their address

Think of them as **smart text replacement**, not storage.

HOW MASM CREATES SYMBOLIC CONSTANTS

MASM gives you **three main ways** to define symbolic constants:

I) INTEGER EXPRESSIONS ONLY

```
MY_CONSTANT = 100
```

Used for **numbers**

Can include expressions

```
SCREEN_WIDTH  = 800
SCREEN_HEIGHT = 600
PIXELS        = SCREEN_WIDTH * SCREEN_HEIGHT
```

Anywhere you use PIXELS, MASM replaces it with 480000.

II) EQU (NUMBERS OR TEXT)

EQU is more flexible.

Numeric example:

```
BUFFER_SIZE EQU 4096
```

Text example:

```
MY_TEXT_CONSTANT EQU "Hello, world!"
```

What EQU really means:

"Whenever you see this name, replace it with exactly this text."

So, these two lines are **identical to the assembler**:

```
mov eax, "Hello, world!"
mov eax, MY_TEXT_CONSTANT
```

⚠ Important:
This does **not** create a string in memory.
It only replaces text **at assembly time**.

III. TEXTEQU (TEXT ONLY, LONGER TEXT)

```
LONG_TEXT TEXTEQU <This is a very long string that might exceed normal limits>
```

- Mostly used for **macros** or long text blocks
- Same idea as EQU, just specialized for text
- Rare in beginner code, but good to know it exists

IV. WHERE YOU CAN USE SYMBOLIC CONSTANTS

Symbolic constants can be used **anywhere MASM expects**:

- A number
- A memory address
- A text string (depending on the directive)

Example: This moves 100 into EAX.

```
mov eax, MY_CONSTANT
```

V. STRINGS: WHERE CONFUSION USUALLY STARTS 🔥

Let's clear this up **once and for all**, because this is where the original notes went sideways.

❌ WRONG IDEA (COMMON MYTH)

"You can store a string in a DWORD for performance."

❌ This is false.

A DWORD is **4 bytes**.
A string like "Hello, world!" is **13 bytes + null terminator**.

You physically **cannot** store a string inside a single DWORD.

VI. THE CORRECT WAY TO STORE STRINGS

✅ Strings are stored as byte arrays

```
.data
myMessage DB "Hello, world!", 0
```

- DB = Define Byte
- Each character = 1 byte
- 0 = null terminator (required by Windows APIs)

This creates **actual memory** holding the characters.

VII. WHY THIS FAILED CODE IS WRONG

```
MY_TEXT_CONSTANT EQU "Hello, world!"

.data
myMessage DWORD MY_TEXT_CONSTANT    ; ❌ wrong
myMessage DB MY_TEXT_CONSTANT       ; ❌ wrong
```

Why both are wrong:

1. MY_TEXT_CONSTANT is **text substitution**, not data
2. DWORD expects a 32-bit number, not characters
3. DB MY_TEXT_CONSTANT expands to:

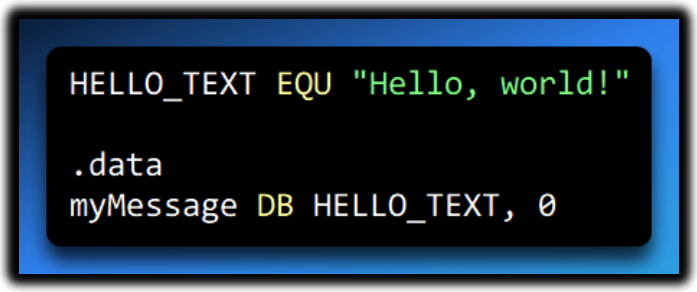
❌ MASM does not automatically add a null terminator

❌ Still not a real string unless written correctly

VIII. THE RIGHT WAY TO COMBINE CONSTANTS + STRINGS

Option 1: Direct string storage (most common)

Option 2: Use constants for readability



Now you get:

- Readable code
- Correct memory layout
- Proper null-terminated string

IX. THE BIG CONFUSION: DWORD + STRINGS

Here's the **truth bomb** that clears everything up:

🔥 **You cannot store a string in a DWORD — but you CAN store a POINTER to a string in a DWORD.**

✅ This is valid and common:

```
.data
myMessage DB "Hello, world!", 0
myMessagePtr DWORD OFFSET myMessage
```

- myMessage → actual string bytes
- myMessagePtr → address of the string (32-bit pointer)

Windows API calls expect **pointers**, not strings themselves.

X. WHY MessageBoxA WORKS

```
call MessageBoxA, NULL, myMessage, MB_OK
```

You are **not passing the string**.

You are passing:

- The **address** of the string
- That address fits in a DWORD (on 32-bit systems)

This is why the confusion happens.

XI. FINAL SUMMARY

✅ **Facts to remember**

- Symbolic constants:
  - ❖ Exist only at **assembly time**
  - ❖ Do **not** allocate memory
  - ❖ Improve readability and maintainability
- Strings:
  - ❖ Must be stored using DB
  - ❖ Must be null-terminated
  - ❖ Cannot fit in a DWORD
- DWORD + strings:
  - ❖ ❌ You cannot store characters in a DWORD
  - ❖ ✅ You can store a **pointer** to a string in a DWORD

XII. BIG IDEA TO REMEMBER 🧠

Constants replace text.
Variables reserve memory.
DWORDs hold numbers or addresses — not characters.

THE EQUAL-SIGN (=) DIRECTIVE: SMART REPLACEMENT 🧠

The **equal-sign directive (=)** is one of the simplest—and most powerful—tools in MASM.

At a high level:

The = directive associates a **symbol name** with an **integer expression**.

That's it.
No memory.
No runtime behavior.
Just **compile-time substitution**.

I. What the = Directive Actually Does

When you write COUNT = 500

You are telling the assembler:

"Whenever you see the word COUNT, replace it with the number 500."

That replacement happens during the **assembler's preprocessing step**, *before* machine code is generated.

This means:

- COUNT is **not a variable**
- It does **not exist in memory**
- The CPU will never know COUNT existed

Only the number 500 survives into the final machine code.

II. Step-by-Step: What MASM Really Sees

Let's say your source file starts like this: **COUNT = 500**

Ten lines later, you write: **MOV EAX, COUNT**

What *you* wrote: **mov eax, COUNT**

What MASM turns it into internally: **mov eax, 500**

By the time the assembler is done, the symbol COUNT is completely gone.

🔥 **Key insight:**
The assembler does *textual replacement*, not runtime evaluation.

III. What Can the Expression Be?

Although COUNT = 500 is the most common case, the expression can be more complex:

```
MAX_ITEMS   = 100
ITEM_SIZE   = 4
BUFFER_SIZE = MAX_ITEMS * ITEM_SIZE
```

MASM computes the math **at assembly time**, and replaces BUFFER_SIZE with 400.

This is incredibly useful for:

- Array sizes
- Offsets
- Limits
- Configuration values

IV. Why Use Symbols Instead of Literal Numbers?

You *could* write this: **mov eax, 500**

But now imagine this number appears:

- 12 times
- In loops
- In comparisons
- In array bounds

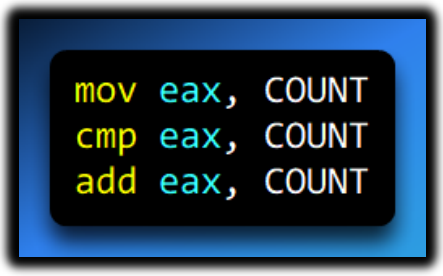
Six months later, you realize it should be 600.

Now you're hunting through code, hoping you don't miss one.

V. The Real Power: One Change, Everywhere

Using a symbol: **COUNT = 500**

Later:



Now, when requirements change: **COUNT = 600**

Reassemble the program.

✅ Every instance of COUNT is automatically replaced with 600
✅ No logic changes
✅ No missed updates
✅ No bugs from inconsistent values

This is **maintainability**, not convenience.

VI. Why This Matters in Assembly (More Than High-Level Languages)

In high-level languages, constants are common and expected.

In assembly:

- Numbers have no meaning on their own
- 500 could be:
  - ❖ A loop limit
  - ❖ A buffer size
  - ❖ A timeout
  - ❖ A magic value with special meaning

Using symbols gives **semantic meaning** to raw numbers.

Compare:

```
cmp eax, 500

;versus

cmp eax, MAX_RECORDS
```

Same machine code.
Very different readability.

VII. Important Limitations of =

The equal-sign directive has **rules**:

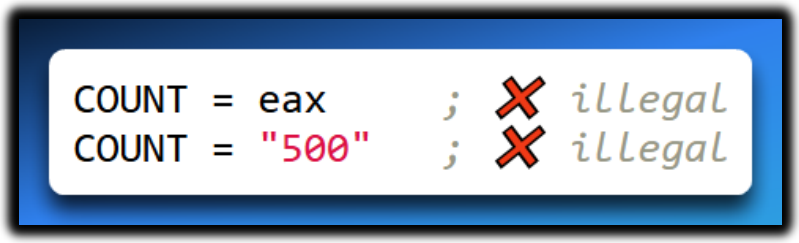
✅ **Allowed**

- Integer values
- Integer expressions
- Arithmetic with other symbols

❌ **Not allowed**

- Strings
- Memory definitions
- Runtime changes

This will **not** work:



Why?

Because = is resolved **before registers or memory even exist**.

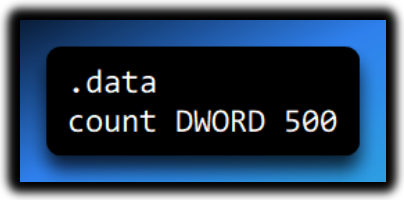
VIII. = vs Variables (Critical Distinction)

Let's compare a symbolic constant:

COUNT = 500

- Exists only at assembly time
- No memory
- Cannot change at runtime

To a variable:



- Stored in memory
- Can be modified by the program
- Takes space in RAM and the executable

🔥 **Rule of thumb:**
If the value never changes → use =
If the value changes → use .data

IX. BIG IDEA TO REMEMBER 🧠

The = directive is a promise to the assembler, not the CPU.

It says:
"Replace this name with this number everywhere, before the program even exists."

Or even shorter:

= makes your code readable.
The assembler does the boring work.

THE CURRENT LOCATION COUNTER $ (WHERE AM I RIGHT NOW?)

Assembly programs don't magically know where things live in memory.
Someone has to keep track of addresses as code and data are laid out.

That "someone" is the **assembler**, and the tool it uses is called the:

Current Location Counter (LC)
Also known as the **Assembly Pointer (AP)**
Represented by the symbol **$**

I. What the Current Location Counter Really Is

The **current location counter ($)** is a special symbol that always represents:

"The address in memory where the assembler is currently writing."

Not where the CPU is executing.
Not where the program is *running*.
But where the **assembler is placing bytes** while building your program.

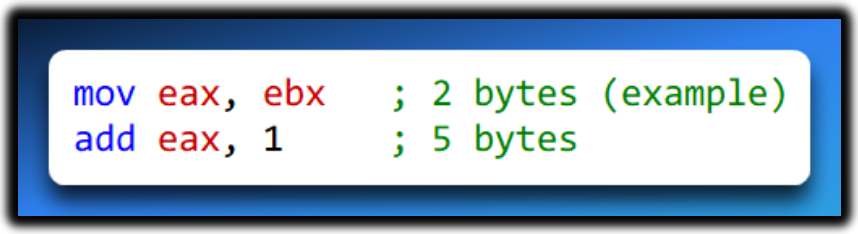
🔥 This is a *compile-time* concept, not a runtime one.

II. How the Assembler Uses $

When the assembler starts reading your source file:

- The location counter starts at **0**
- As instructions and data are processed:
  - $ increases
  - By exactly the number of bytes generated

Example:

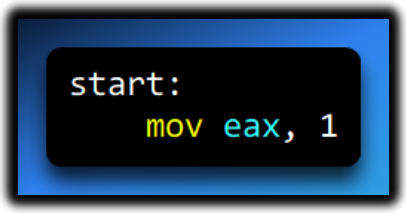


As each instruction is assembled:

- $ moves forward
- Just like a cursor writing bytes into memory

III. $ Is How Labels Get Their Addresses

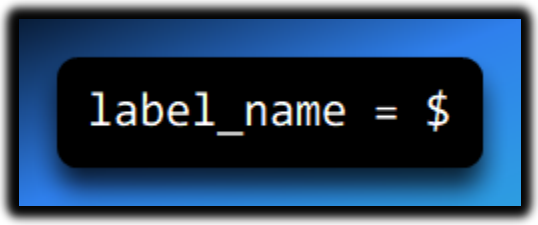
Every label you write is secretly tied to $.



What the assembler really does is:

"At this moment, $ equals some address.
Associate start with that address."

So, you can think of a label as:



...but automatically managed for you.

IV. Using $ Directly (The Self-Pointer Trick)

Now let's look at this line:

This is subtle, clever, and 100% legal.

What's happening step by step:

1. The assembler reaches selfPtr
2. $ currently points to the address where selfPtr will be stored
3. The value of $ is written *into* that memory location

Result:

selfPtr contains its own address

In plain English:

"Create a variable, and store the address of that variable inside itself."

Why this works

- DWORD → 4 bytes of storage
- $ → the address of the first byte of those 4 bytes

So selfPtr ends up holding a pointer to itself.

This is useful in:

- Low-level memory structures
- Tables of pointers
- Self-describing data layouts

V. Very Important Clarification ⚠

$ is **not a CPU register**.

- The CPU never sees $
- $ does not exist at runtime
- $ is resolved **during assembly**

By the time the program runs:

- $ is gone
- Only raw numbers (addresses) remain

VI. The Book Analogy (Why This Helps)

Imagine you're writing a book.

- The **location counter** is the current page number
- Every time you write more text, the page number increases
- When you say:

"See page 42"

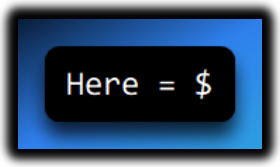
You're doing the same thing assembly does with $.

In assembly:

- Memory = book
- Addresses = page numbers
- $ = "the page I'm currently writing on"

VII. Symbolic Constants + $ (Power Combo)

You can combine $ with the = directive:



Now Here becomes a symbolic constant equal to the **current address**.

This is useful for:

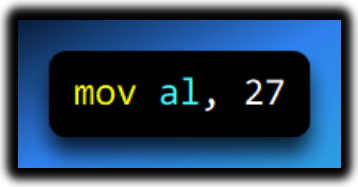
- Computing offsets
- Measuring sizes
- Building jump tables
- Aligning structures

VIII. Keyboard Definitions (Related but Different)

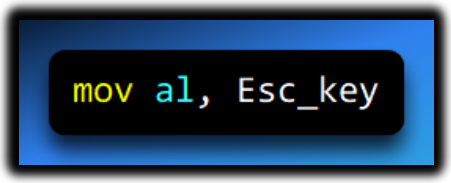
This example: **Esc_key = 27**

Is **not** related to $, but it uses the same idea of symbolic clarity.

Instead of writing:



You write:

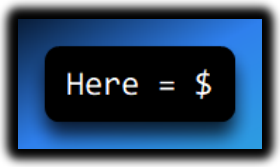


Same machine code.
Much clearer intent.

IX. $ Vs Variables (Critical Distinction)

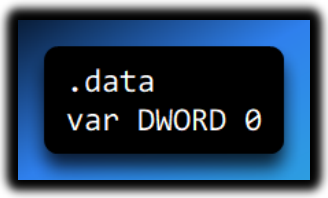
Compare these two:

Symbolic address:



- Exists only at assembly time
- No memory
- No storage

Variable:



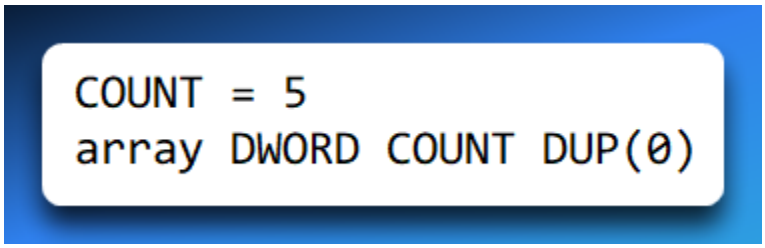
- Allocates memory
- Has a runtime address
- Can be modified

🔥 Rule of thumb:

$ tells you **where things are**
Variables hold **what things are**

X. DUP OPERATOR CONNECTION

The DUP operator uses **symbolic constants**, not $, but they often appear together.



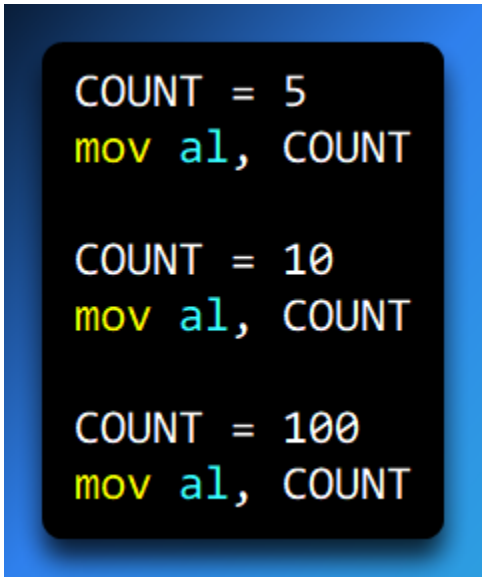
Here:

- COUNT is resolved first
- DUP allocates memory
- $ advances as each DWORD is written

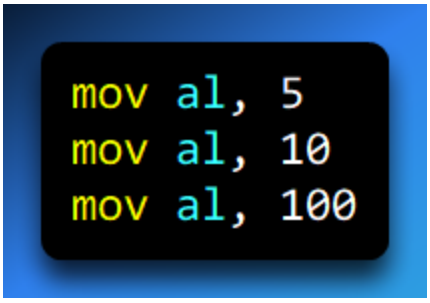
Everything still flows through the location counter.

XI. REDEFINING SYMBOLS (ASSEMBLER-TIME ONLY)

This part is crucial and often misunderstood:



What gets assembled is:



Why?

Because:

- The assembler reads the file **top to bottom**
- Symbols change value **as the assembler processes lines**
- Runtime execution order does not matter at all

🔥 **Assembler time ≠ Runtime**

XII. FINAL CLEAN SUMMARY (STICK THIS IN YOUR BRAIN)

The Current Location Counter $:

- Represents the **current memory address**
- Managed entirely by the assembler
- Changes as code/data are generated

- Can be used to initialize pointers

$ is:

- Compile-time only
- Address-focused
- Invisible at runtime

$ is NOT:

- A CPU register
- A variable
- Something you can modify directly

BIG IDEA TO REMEMBER 🧠

$ answers one question:

👉 *"Where is the assembler writing right now?"*

Once you understand that, everything else about labels, pointers, offsets, and layout starts to click into place.

Whatever the hell we read in that section😂😂😂
That's what I call real assembly language as a rite of passage,
meet the $

ARRAY SIZE CALCULATION WITH THE $ OPERATOR

What does "size of an array" mean in assembly?

When we talk about the *size* of an array in assembly, we're usually asking one of two things:

- **How many bytes does this array occupy in memory?**
- **How many elements does this array contain?**

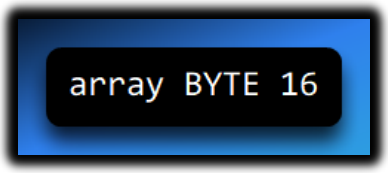
Assembly doesn't track "arrays" the way high-level languages do. To the assembler, an array is just a **block of consecutive bytes in memory**. It's your job to keep track of how big that block is and how you intend to use it.

There are two common ways to do this:

1. **Explicitly specify the size**
2. **Let the assembler calculate it using the $ operator**

Declaring an array by explicitly stating its size

Example:



This line means:

- array is a **label** (a name for a memory location)
- BYTE means each element is **1 byte**
- 16 means **reserve 16 bytes of memory**

That's it. Nothing more, nothing less.

Important clarification (this trips people up a lot)

array BYTE 16 **does NOT** mean:

- "The first element has the value 16"
- "The array contains the number 16"

It means:

"Allocate 16 bytes of storage starting at the label array."

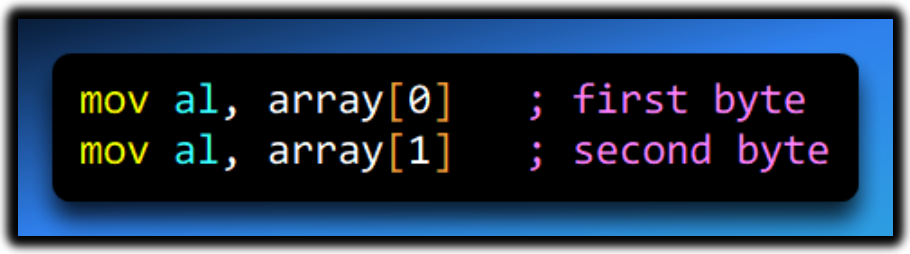
The contents of those 16 bytes are **uninitialized** unless your assembler or environment zero-fills memory (many don't).

Think of it like this:

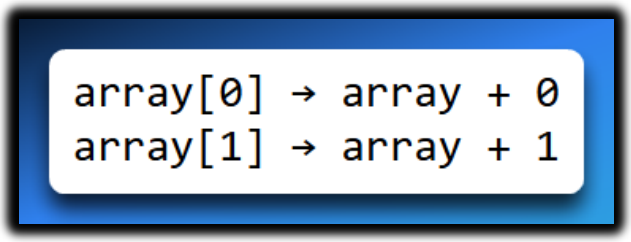
You just told the assembler:
"Please set aside 16 empty lockers and call the first one array."

Accessing elements in the array

Once declared, the label array acts as the **base address** of that block of memory.



The assembler calculates the correct address automatically:

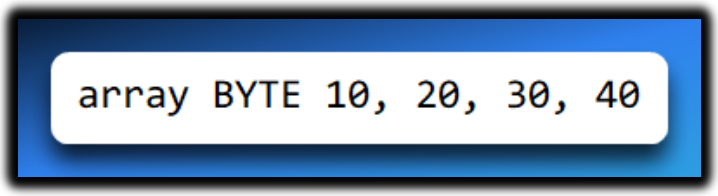


This works because:

- Each element is 1 byte (BYTE)
- Offsets are counted in bytes

Initializing an array with values

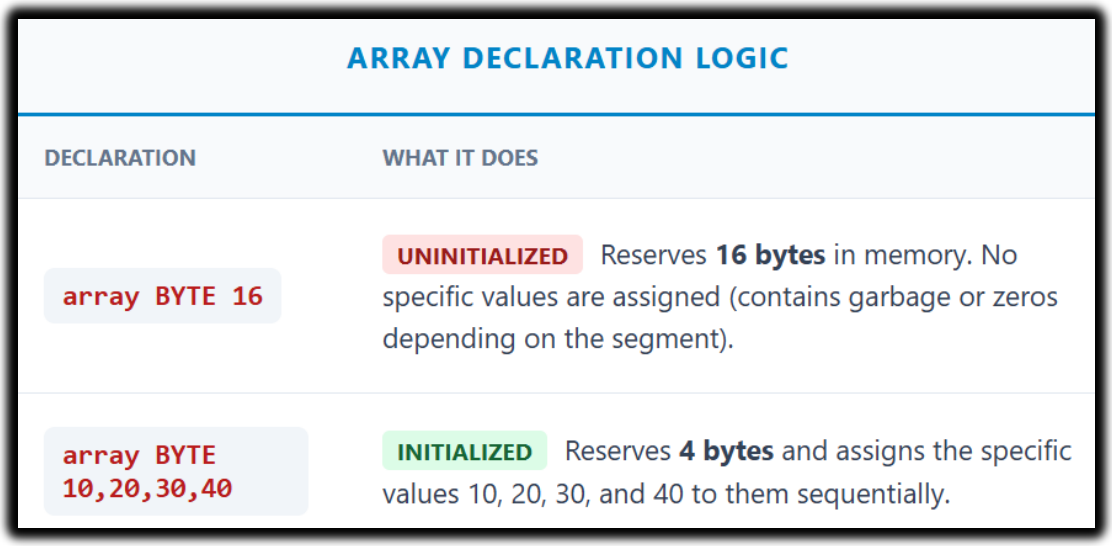
Now compare that with this:



This declaration means something *completely different*:

- Allocate **4 bytes**
- Initialize them with these values:
  - array[0] = 10
  - array[1] = 20
  - array[2] = 30
  - array[3] = 40

So:



This is one of the most important distinctions in assembly:
a single number can mean "size" or "data" depending on context.

Letting the assembler calculate array size using $

What is $?

The $ symbol represents the **current location counter (LC)**.

The LC is the assembler's internal pointer that always says:

"This is the address where the next byte will be placed."

As the assembler processes data declarations, $ increases automatically.

```
array BYTE 10, 20, 30, 40
array_size = ($ - array)
```

What happens here:

1. The assembler lays down 4 bytes for array
2. $ now points *just past* the last byte
3. Subtracting the starting address (array) gives:

```
array_size = 4
```

So array_size equals the **number of bytes** occupied by the array.

⚠ **Critical rule:**
array_size must be defined **immediately after** the array declaration.
If anything else is declared in between, $ will no longer reflect the end of the array.

Calculating the size of a string

Strings are just byte arrays.

```
string BYTE "This is a long string", 0
string_size = ($ - string)
```

This gives you:

- The total number of bytes in the string
- **Including the null terminator (0)**

That's often exactly what you want when working with string routines.

If your string spans multiple lines or declarations, the same rule applies — as long as string_size comes immediately after the final byte.

Arrays of WORDs and DWORDs

So far, everything has been in bytes. But what if your array elements are larger?

WORD arrays (2 bytes per element)

```
list WORD 1000h, 2000h, 3000h, 4000h
list_size = ($ - list) / 2
```

Why divide by 2?

- $ - list gives **total bytes**
- Each element is 2 bytes
- Dividing converts *bytes → elements*

Result: **list_size = 4**

DWORD arrays (4 bytes per element)

```
list DWORD 10000000h, 20000000h, 30000000h, 40000000h
list_size = ($ - list) / 4
```

Same idea:

- $ - list → total bytes
- Divide by 4 → number of elements

Key ideas to remember

- Assembly doesn't know what an "array" is — only **memory and labels**
- BYTE, WORD, DWORD tell the assembler:
  - ❖ How much space each element takes
- $ tells you:
  - ❖ Where the assembler is *right now*
- ($ - label) gives you:
  - ❖ Total space used since that label

Final summary

- array BYTE 16
  → Reserves **16 bytes**, no values assigned
- array BYTE 10,20,30,40
  → Reserves **4 bytes**, initialized with values
- $ - array
  → Calculates **total bytes used**
- Divide by element size
  → Converts **bytes → element count**

Once you internalize that $ is just the assembler saying *"here's where I am in memory right now"*, all of this becomes very natural.

THE EQU DIRECTIVE — GIVING NAMES TO VALUES (AND IDEAS)

The EQU directive is how you give a **name** to something in assembly—whether that "something" is a number, a calculation, or even a piece of text.

Think of EQU as a **label-maker for constants**. Once you define a name with EQU, the assembler treats that name as a stand-in for whatever you assigned to it.

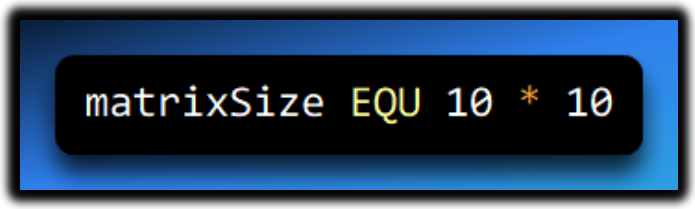
When the assembler later sees that name anywhere in your program, it **literally substitutes** the value or text you defined. There's no memory involved, no runtime work—this all happens *at assembly time*.

The Three Forms of EQU

There are three valid ways to use EQU, depending on what you want the symbol to represent.

1. name EQU expression

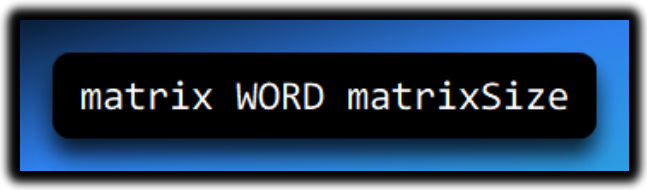
This is the most common form. The expression must evaluate to an **integer** at assembly time.



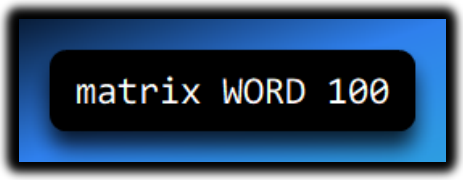
Here's what's happening:

- The assembler evaluates 10 * 10
- It gets 100
- Everywhere it sees matrixSize, it substitutes 100

So later code like this:



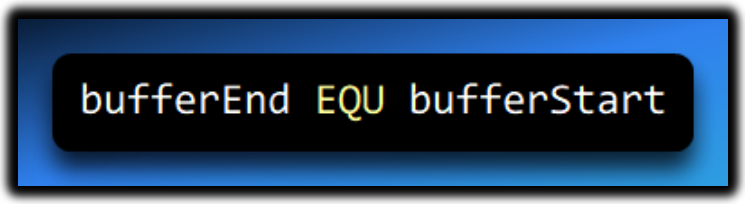
is treated as if you had written:



This makes your code clearer and easier to change later.

2. name EQU symbol

In this form, you're defining one symbol in terms of another.



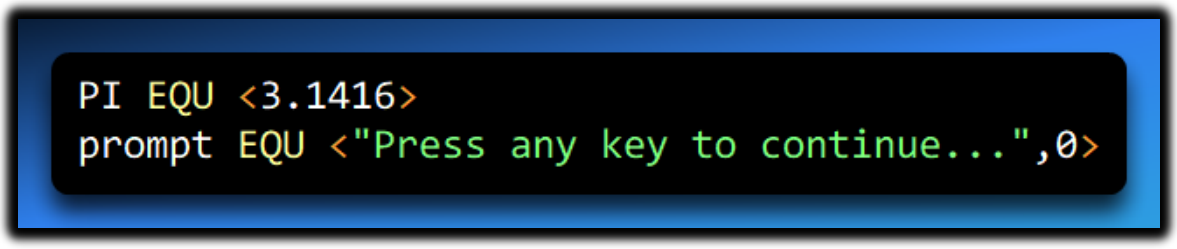
This is useful when:

- You want multiple names for the same value
- You're improving readability
- You're abstracting meaning (e.g., "what this address represents")

Again, substitution happens during assembly—not at runtime.

3. name EQU <text>

This form lets you associate a symbol with **arbitrary text**, not just numbers.



Important detail:

- The assembler does **not** evaluate this as a number
- It simply copies the text wherever the symbol appears

This is especially useful for:

- Real-number constants
- Strings
- Data definitions that don't evaluate to integers

What EQU Really Does (Behind the Scenes)

A key idea to understand:

EQU does not allocate memory.

It doesn't reserve space, and it doesn't create a variable.
It only tells the assembler:

"Whenever you see this name, replace it with this value or text."

So EQU is purely a **compile-time substitution tool**.

Using EQU for Complex or Tricky Calculations

One of the biggest strengths of EQU is that it lets you define expressions that would be annoying—or error-prone—to calculate by hand.

I. Example: Defining a Stack Address

```
stackStart EQU _end + 1024
```

What's going on here?

- _end is a symbol automatically provided by the assembler
- It marks the end of your program's code/data
- You add 1024 bytes to that address
- The result becomes the symbolic name stackStart

Now, instead of scattering magic numbers throughout your code, you have a clear, meaningful name that explains *why* that address exists.

II. One Very Important Rule: EQU Is Immutable

Once you define a symbol with EQU, **you cannot redefine it in the same source file**.

This is intentional.

Why?

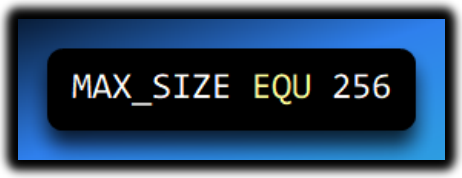
- It prevents accidental redefinitions
- It guarantees that a symbol always means the same thing
- It makes large programs safer and easier to maintain

When you see a name defined with EQU, you can trust that its value never changes.

III. EQU vs = (or ==) — A Crucial Difference

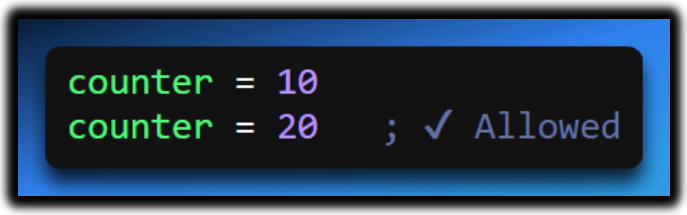
Assemblers often support another directive: = (or ==, depending on the assembler).

EQU: Constant, fixed, immutable



- Defined once
- Cannot change
- Best for true constants

=: Redefinable



- The symbol's value can change
- Useful for assembly-time calculations or conditional assembly
- More flexible, but also easier to misuse

IV. The Big Picture Difference

CONSTANT DEFINITION COMPARISON

DIRECTIVE	REDEFINABLE?	PURPOSE
EQU	✗ No	**Safe, permanent constants.** Once defined, the assembler will throw an error if you try to change its value. Best for fixed system limits or physical constants.
= / ==	✓ Yes	**Adjustable, temporary values.** Can be reassigned multiple times throughout the code. Useful for loop counters or calculated offsets during assembly.

If you want **stability and clarity**, use EQU.
If you need **flexibility during assembly**, use =.

V. Why EQU Matters in Real Programs

Using EQU properly:

- Makes code easier to read
- Eliminates “magic numbers”
- Reduces bugs caused by inconsistent values
- Makes large assembly programs manageable

In short:
EQU lets you name ideas, not just numbers.

And that’s one of the most powerful things you can do in assembly.

TEXTEQU DIRECTIVE

I. Big picture: what problem does TEXTEQU solve?

When you write assembly, you often repeat the same pieces of text:

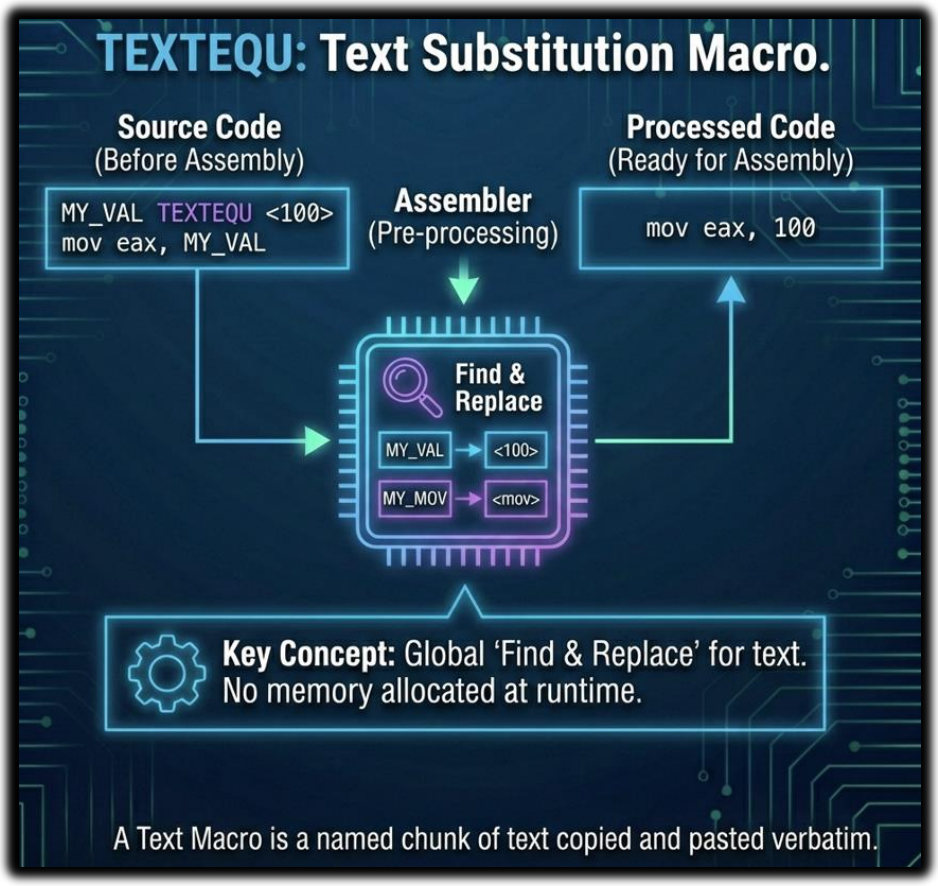
- instruction names (mov, add)
- operands (al, eax)
- constants
- short instruction sequences

TEXTEQU exists so you don’t have to keep retyping those pieces. Instead, you give them a **name**, and the assembler swaps that name with its associated text **before assembly actually happens**.

So, the key idea is this:

TEXTEQU creates text substitutions, not variables and not memory.

Nothing is stored at runtime. This is all about helping *you* and the assembler.



II. What exactly is a text macro?

A **text macro** is a named chunk of text that the assembler copies and pastes wherever the name appears.

Think of it like:

- a global "find and replace"
- or a nickname for a piece of assembly code

When the assembler sees the macro name, it replaces it with its text **verbatim** (after expression evaluation, if any).

The three forms of TEXTEQU

There are three common ways to define a text macro, depending on what you want the macro to represent.

I. Assigning literal text

```
name TEXTEQU <text>
```

Here, you're directly attaching a chunk of text to a name. The angle brackets < > tell the assembler:

"Treat everything inside here as literal text."

Example:

```
continueMsg TEXTEQU <"Do you wish to continue (Y/N)?">
```

What this means:

- continueMsg is now a text macro
- Wherever continueMsg appears later, the assembler replaces it with:

```
"Do you wish to continue (Y/N)?"
```

This is especially useful for:

- prompt messages
- repeated strings
- long instruction fragments

It makes your code easier to read and easier to change later.

II. Assigning one text macro to another

```
name TEXTEQU textmacro
```

Now move is simply another name for mov.

This might look pointless at first—but it becomes powerful when you start combining macros to build instructions dynamically.

III. Assigning a constant expression

```
name TEXTEQU %constExpr
```

The % is important. It tells the assembler:

"Evaluate this expression now, then convert the result into text."

So, this is still a **text macro**, but its value comes from a calculation.

Building text macros step by step (important example)

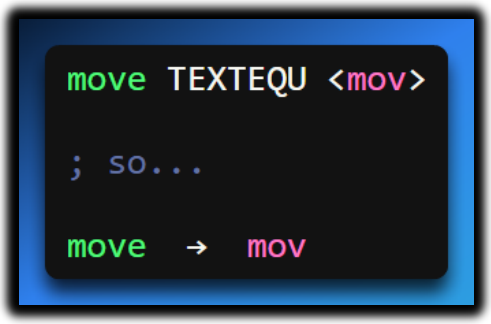
Let's walk slowly through this example and see what the assembler is actually doing.

```
rowSize = 5
count TEXTEQU %(rowSize * 2)
```

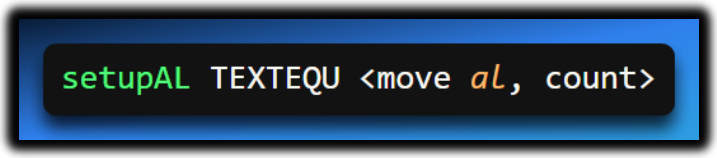
- rowSize is a numeric constant with value 5
- count becomes a text macro
- The expression (rowSize * 2) is evaluated immediately
- The result is 10
- So count is replaced with the text 10

At this point: **count → 10**

Now we add another layer:



Now we combine everything:



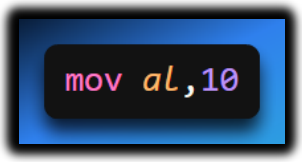
The assembler expands this in stages:

1. move → mov
2. count → 10

Final expansion:

NB: Am using **onecompiler.com** for these images together with **sharex** gradient mode.

So, whenever you write setupAL, the assembler literally sees:



This is a huge deal for readability and maintainability.

Why this matters in real code

Text macros let you:

- Avoid magic numbers scattered everywhere
- Centralize instruction patterns
- Build readable "mini-commands" out of raw assembly
- Change behavior in one place instead of dozens

They're especially useful in:

- loops
- setup code
- repeated register initialization
- macro-heavy assembly projects

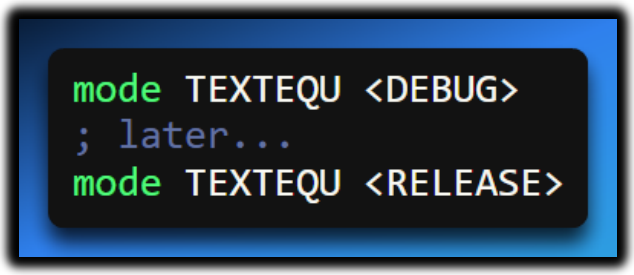
Redefining text macros (key difference from EQU)

One very important rule:

Text macros defined with TEXTEQU can be redefined.

This is *not* true for EQU.

That means you can do something like:



From that point forward, mode expands to RELEASE.

This makes TEXTEQU flexible, but it also means:

- You must be careful about scope and order
- Earlier code uses the old definition
- Later code uses the new one

What TEXTEQU is *not*

To avoid common confusion:

- ❌ It does **not** reserve memory
- ❌ It does **not** create a runtime variable
- ❌ It does **not** generate instructions by itself

It only tells the assembler how to **rewrite your source code text** before assembling it.

Mental model

If you remember one sentence, make it this:

TEXTEQU is a compile-time text substitution tool that helps you write clearer, reusable assembly code.

No runtime cost. No memory impact. Just smarter source code.