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**.

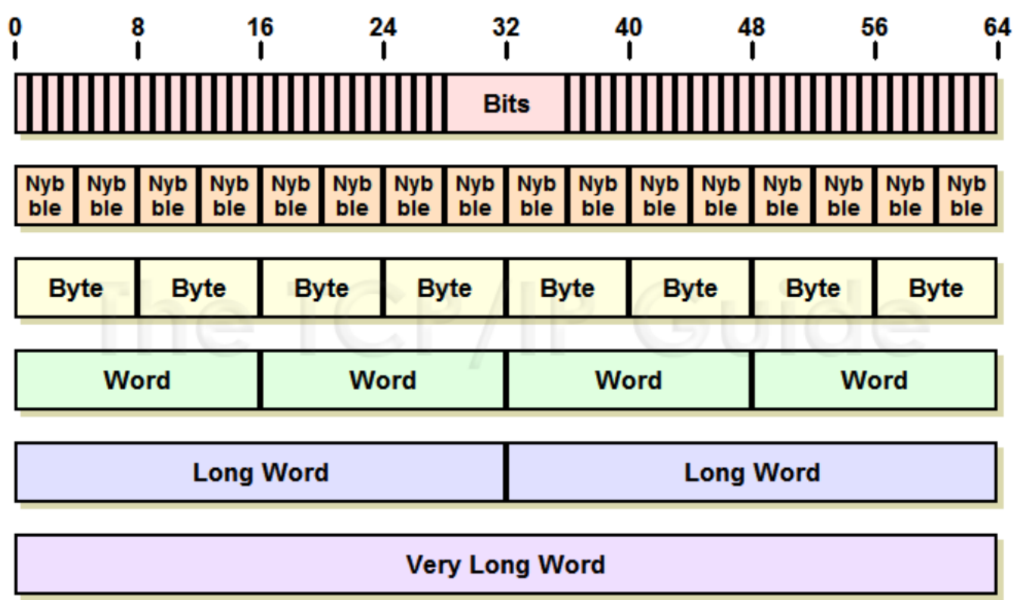


Figure 8: Binary Information Representations and Terms

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

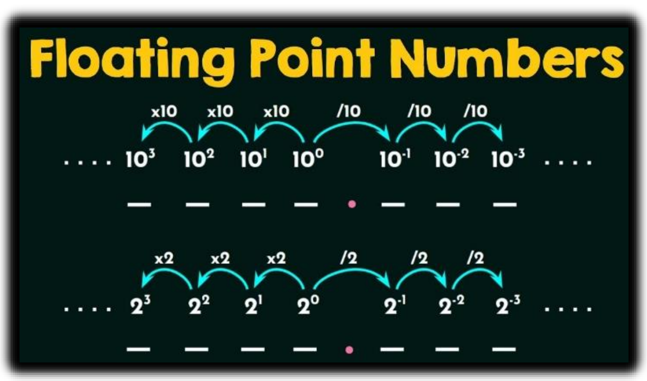
- **DWORD**
  32-bit **unsigned** integer
  Range: 0 to 4,294,967,295
- **SDWORD**
  32-bit **signed** integer
  Range: –2,147,483,648 to 2,147,483,647

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]  directive  value
```

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

Example

```
count DWORD 12345
```

This means:

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

Equivalent C code:

```c
int count = 12345;
```

Same idea, different language.

More examples

```
message DB "Hello, world!"
age     BYTE 25
salary  SDWORD 100000
```

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)

```
count DWORD 12345
int count = 12345;
```

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.

```
x DB 10
letter DB 'A'
text DB "Hi", 0
```

Common uses:

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

2. DW — Declare Word (16 bits)

Reserves **2 bytes**.

```
count DW 500
```

Used for:

- 16-bit values
- older or compact data

3. DD — Declare Doubleword (32 bits)

Reserves **4 bytes**.

```
total DD 12345
```

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

4. DQ — Declare Quadword (64 bits)

Reserves **8 bytes**.

```
bigNumber DQ 1234567890123
```

Used for:

- large integers
- 64-bit values

5. DT — Declare Ten Bytes (80 bits)

Reserves **10 bytes**.

```
realValue DT 1.23
```

Used for:

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

About strings and null terminators

```
message DB "Hello, world!"
message DB "Hello, world!", 0
```

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:

```
sum DWORD 0
```

- 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:

```
nums BYTE 1, 2, 3, 4
```

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:

```
count BYTE 255       ; valid (fits in 1 byte)
count BYTE 300       ; invalid (too large)
```

4. Leaving a Variable Uninitialized (?)

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

Example:

```
value6 BYTE ?
```

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:

```
age     BYTE 25
letter  DB 'A'
flag    DB 1
```

```
value1 BYTE 0
value2 BYTE 255
```

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).

```
value3 SBYTE -128
value4 SBYTE 127
```

Signed vs Unsigned

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

Uninitialized Variables (Important Warning)

```
value6 BYTE ?
```

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

```
char value6;
```

...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

```
value9 DB 'B'
```

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

Signed Byte Example

```
value10 SBYTE -12
```

- 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

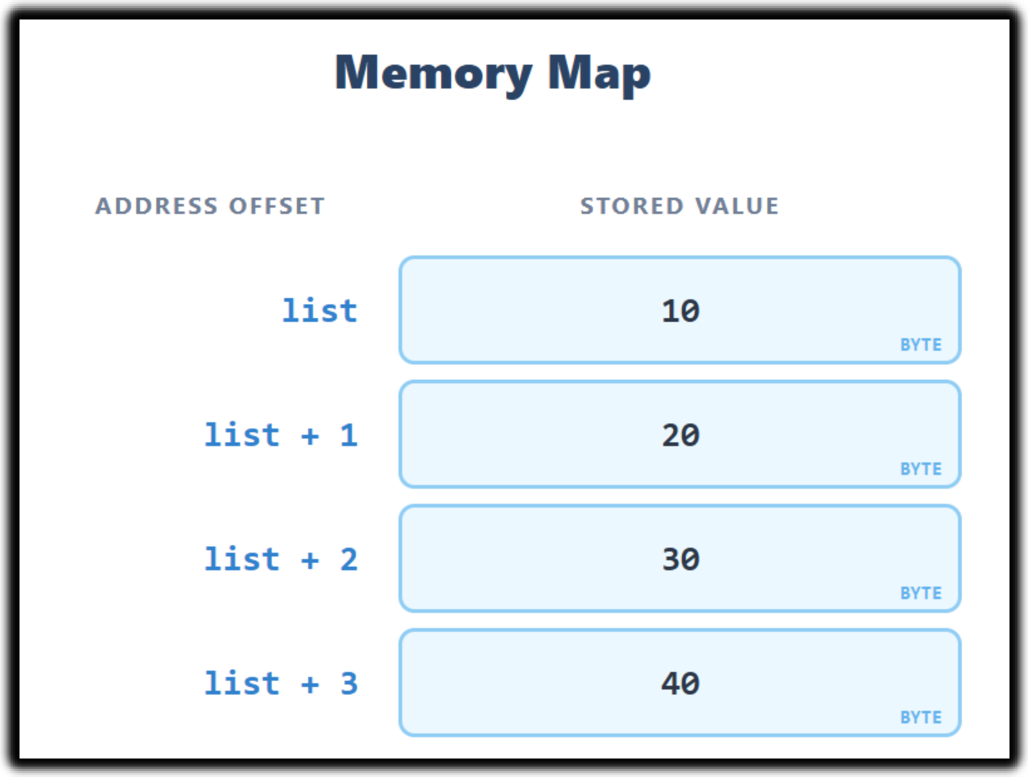
In high-level languages like C++ or Python, you create an array with brackets []. In Assembly, you just list values one after another.

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.