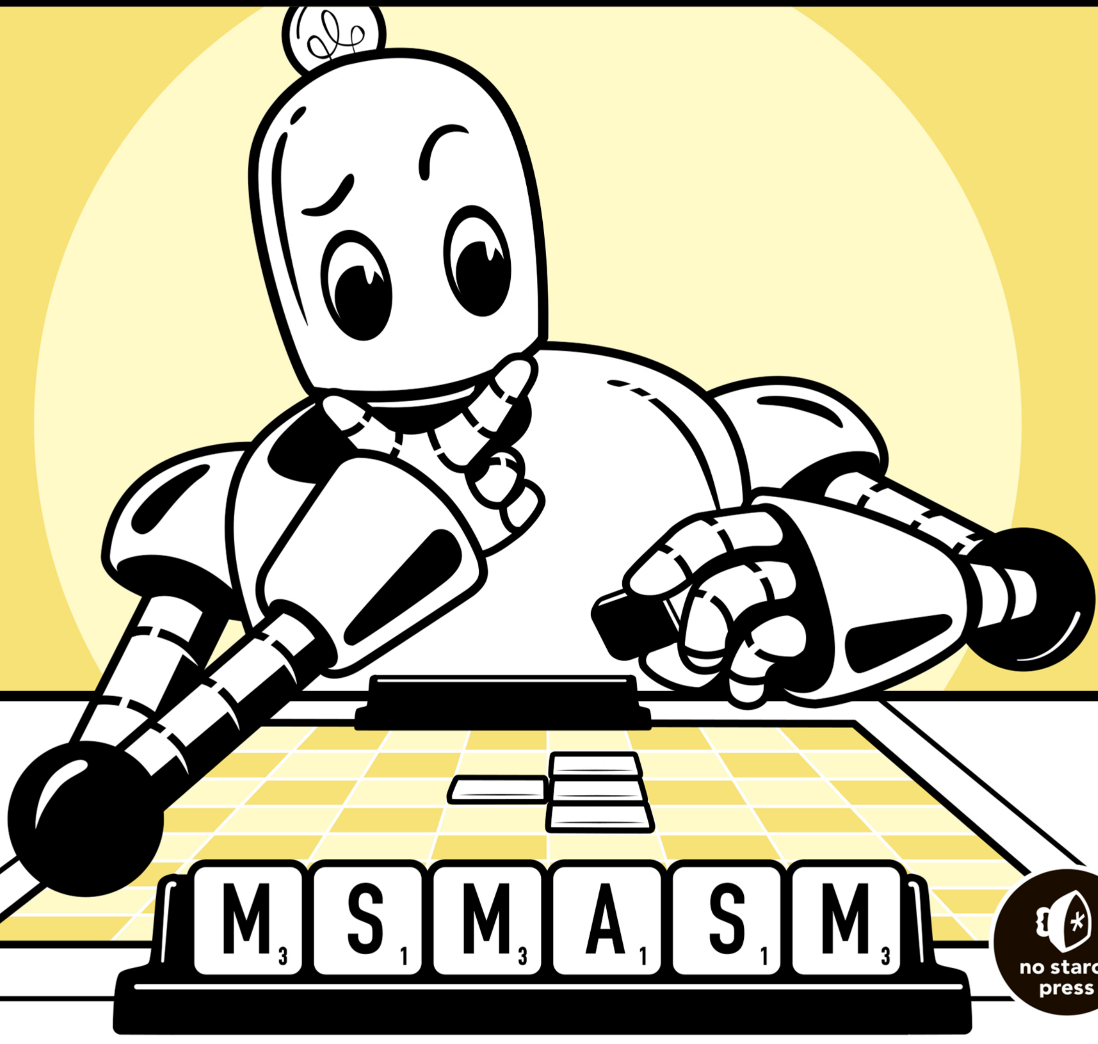
VOLUME 1

THE ART OF 64-BIT ASSEMBLY

X86-64 MACHINE ORGANIZATION AND PROGRAMMING

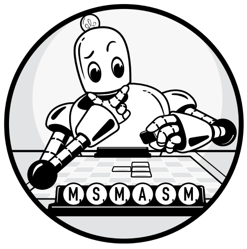
RANDALL HYDE



no starch press

1

HELLO, WORLD OF ASSEMBLY LANGUAGE



This chapter is a "quick-start" chapter that lets you begin writing basic assembly language programs as rapidly as possible. By the conclusion of this chapter, you should understand the basic syntax of a Microsoft Macro Assembler (MASM) program and the prerequisites for learning new assembly language features in the chapters that follow.

NOTE *This book uses the MASM running under Windows because that is, by far, the most commonly used assembler for writing x86-64 assembly language programs. Furthermore, the Intel documentation typically uses assembly language examples that are syntax-compatible with MASM. If you encounter x86 source code in the real world, it will likely be written using MASM. That being said, many other popular x86-64 assemblers are out there, including the GNU Assembler (gas), Netwide Assembler (NASM), Flat Assembler (FASM), and others. These assemblers employ a different syntax from MASM (gas being the one most radically different). At some point, if you work in assembly language much, you'll probably encounter source code written with one of these other assemblers. Don't fret; learning the syntactical differences isn't that hard once you've mastered x86-64 assembly language using MASM.*

This chapter covers the following:

- Basic syntax of a MASM program
- The Intel central processing unit (CPU) architecture
- Setting aside memory for variables
- Using machine instructions to control the CPU
- Linking a MASM program with C/C++ code so you can call routines in the C Standard Library
- Writing some simple assembly language programs

1.1 What You'll Need

You'll need a few prerequisites to learn assembly language programming with MASM: a 64-bit version of MASM, plus a text editor (for creating and modifying MASM source files), a linker, various library files, and a C++ compiler.

Today's software engineers drop down into assembly language only when their C++, C#, Java, Swift, or Python code is running too slow and they need to improve the performance of certain modules (or functions) in their code. Because you'll typically be interfacing assembly language with C++, or other high-level language (HLL) code, when using assembly in the real world, we'll do so in this book as well.

Another reason to use C++ is for the C Standard Library. While different individuals have created several useful libraries for MASM (see *http://www.masm32.com/* for a good example), there is no universally accepted standard set of libraries. To make the C Standard Library immediately accessible to MASM programs, this book presents examples with a short C/C++ main function that calls a single external function written in assembly language using MASM. Compiling the C++ main program along with the MASM source file will produce a single executable file that you can run and test.

Do you need to know C++ to learn assembly language? Not really. This book will spoon-feed you the C++ you'll need to run the example programs. Nevertheless, assembly language isn't the best choice for your first language, so this book assumes that you have some experience in a language such as C/C++, Pascal (or Delphi), Java, Swift, Rust, BASIC, Python, or any other imperative or object-oriented programming language.

1.2 Setting Up MASM on Your Machine

MASM is a Microsoft product that is part of the Visual Studio suite of developer tools. Because it's Microsoft's tool set, you need to be running some variant of Windows (as I write this, Windows 10 is the latest version; however, any later version of Windows will likely work as well). Appendix C provides a complete description of how to install Visual Studio Community (the "no-cost" version, which includes MASM and the Visual C++ compiler, plus other tools you will need). Please refer to that appendix for more details.

1.3 Setting Up a Text Editor on Your Machine

Visual Studio includes a text editor that you can use to create and edit MASM and C++ programs. Because you have to install the Visual Studio package to obtain MASM, you automatically get a production-quality programmer's text editor you can use for your assembly language source files.

However, you can use any editor that works with straight ASCII files (UTF-8 is also fine) to create MASM and C++ source files, such as Notepad++ or the text editor available from *https://www.masm32.com/*. Word processing programs, such as Microsoft Word, are not appropriate for editing program source files.

1.4 The Anatomy of a MASM Program

A typical (stand-alone) MASM program looks like Listing 1-1.

```
; Comments consist of all text from a semicolon character
; to the end of the line.

; The ".code" directive tells MASM that the statements following
; this directive go in the section of memory reserved for machine
; instructions (code).

        .code

; Here is the "main" function. (This example assumes that the
; assembly language program is a stand-alone program with its
; own main function.)

main    PROC

Machine instructions go here

        ret    ; Returns to caller

main    ENDP

; The END directive marks the end of the source file.

        END
```

Listing 1-1: Trivial shell program

A typical MASM program contains one or more *sections* representing the type of data appearing in memory. These sections begin with a MASM statement such as `.code` or `.data`. Variables and other memory values appear in a *data* section. Machine instructions appear in procedures that appear within a *code* section. And so on. The individual sections appearing in an assembly language source file are optional, so not every type of section will appear in a particular source file. For example, Listing 1-1 contains only a single code section.

The `.code` statement is an example of an assembler *directive*—a statement that tells MASM something about the program but is not an actual x86-64 machine instruction. In particular, the `.code` directive tells MASM to group the statements following it into a special section of memory reserved for machine instructions.

1.5 Running Your First MASM Program

A traditional first program people write, popularized by Brian Kernighan and Dennis Ritchie's *The C Programming Language* (Prentice Hall, 1978) is the "Hello, world!" program. The whole purpose of this program is to provide a simple example that someone learning a new programming language can use to figure out how to use the tools needed to compile and run programs in that language.

Unfortunately, writing something as simple as a "Hello, world!" program is a major production in assembly language. You have to learn several machine instruction and assembler directives, not to mention Windows system calls, to print the string "Hello, world!" At this point in the game, that's too much to ask from a beginning assembly language programmer (for those who want to blast on ahead, take a look at the sample program in Appendix C).

However, the program shell in Listing 1-1 is actually a complete assembly language program. You can compile (*assemble*) and run it. It doesn't produce any output. It simply returns back to Windows immediately after you start it. However, it does run, and it will serve as the mechanism for showing you how to assemble, link, and run an assembly language source file.

MASM is a traditional *command line assembler*, which means you need to run it from a Windows *command line prompt* (available by running the *cmd.exe* program). To do so, enter something like the following into the command line prompt or shell window:

```
C:\>ml64 programShell.asm /link /subsystem:console /entry:main
```

This command tells MASM to assemble the *programShell.asm* program (where I've saved Listing 1-1) to an executable file, link the result to produce a console application (one that you can run from the command line), and begin execution at the label `main` in the assembly language source file. Assuming that no errors occur, you can run the resulting program by typing the following command into your command prompt window:

```
C:\>programShell
```

Windows should immediately respond with a new command line prompt (as the `programShell` application simply returns control back to Windows after it starts running).

1.6 Running Your First MASM/C++ Hybrid Program

This book commonly combines an assembly language module (containing one or more functions written in assembly language) with a C/C++ main program that calls those functions. Because the compilation and execution process is slightly different from a stand-alone MASM program, this section demonstrates how to create, compile, and run a hybrid assembly/C++ program. Listing 1-2 provides the main C++ program that calls the assembly language module.

```
// Listing 1-2

// A simple C++ program that calls an assembly language function.
// Need to include stdio.h so this program can call "printf()".

#include <stdio.h>

// extern "C" namespace prevents "name mangling" by the C++
// compiler.

extern "C"
{
    // Here's the external function, written in assembly
    // language, that this program will call:

    void asmFunc(void);
};

int main(void)
{
    printf("Calling asmMain:\n");
    asmFunc();
    printf("Returned from asmMain\n");
}
```

Listing 1-2: A sample C/C++ program, listing1-2.cpp*, that calls an assembly language function*

Listing 1-3 is a slight modification of the stand-alone MASM program that contains the `asmFunc()` function that the C++ program calls.

```
; Listing 1-3

; A simple MASM module that contains an empty function to be
; called by the C++ code in Listing 1-2.

        .CODE

; (See text concerning option directive.)

        option  casemap:none

; Here is the "asmFunc" function.

        public  asmFunc
asmFunc PROC
```

```
; Empty function just returns to C++ code.

        ret    ; Returns to caller

asmFunc ENDP
        END
```

Listing 1-3: A MASM program, listing1-3.asm*, that the C++ program in Listing 1-2 calls*

Listing 1-3 has three changes from the original *programShell.asm* source file. First, there are two new statements: the `option` statement and the `public` statement.

The `option` statement tells MASM to make all symbols case-sensitive. This is necessary because MASM, by default, is case-insensitive and maps all identifiers to uppercase (so `asmFunc()` would become `ASMFUNC()`). C++ is a case-sensitive language and treats `asmFunc()` and `ASMFUNC()` as two different identifiers. Therefore, it's important to tell MASM to respect the case of the identifiers so as not to confuse the C++ program.

NOTE *MASM identifiers may begin with a dollar sign (*`$`*), underscore (*`_`*), or an alphabetic character and may be followed by zero or more alphanumeric, dollar sign, or underscore characters. An identifier may not consist of a* `$` *character by itself (this has a special meaning to MASM).*

The `public` statement declares that the `asmFunc()` identifier will be visible outside the MASM source/object file. Without this statement, `asmFunc()` would be accessible only within the MASM module, and the C++ compilation would complain that `asmFunc()` is an undefined identifier.

The third difference between Listing 1-3 and Listing 1-1 is that the function's name was changed from `main()` to `asmFunc()`. The C++ compiler and linker would get confused if the assembly code used the name `main()`, as that's also the name of the C++ `main()` function.

To compile and run these source files, you use the following commands:

```
C:\>ml64 /c listing1-3.asm
Microsoft (R) Macro Assembler (x64) Version 14.15.26730.0
Copyright (C) Microsoft Corporation.  All rights reserved.

 Assembling: listing1-3.asm

C:\>cl listing1-2.cpp listing1-3.obj
Microsoft (R) C/C++ Optimizing Compiler Version 19.15.26730 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

listing1-2.cpp
Microsoft (R) Incremental Linker Version 14.15.26730.0
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:listing1-2.exe
listing1-2.obj
listing1-3.obj
```

```
C:\>listing1-2
Calling asmFunc:
Returned from asmFunc
```

The `ml64` command uses the `/c` option, which stands for *compile-only*, and does not attempt to run the linker (which would fail because *listing1-3.asm* is not a stand-alone program). The output from MASM is an object code file (*listing1-3.obj*), which serves as input to the Microsoft Visual C++ (MSVC) compiler in the next command.

The `cl` command runs the MSVC compiler on the *listing1-2.cpp* file and links in the assembled code (*listing1-3.obj*). The output from the MSVC compiler is the *listing1-2.exe* executable file. Executing that program from the command line produces the output we expect.

1.7 An Introduction to the Intel x86-64 CPU Family

Thus far, you've seen a single MASM program that will actually compile and run. However, the program does nothing more than return control to Windows. Before you can progress any further and learn some real assembly language, a detour is necessary: unless you understand the basic structure of the Intel x86-64 CPU family, the machine instructions will make little sense.

The Intel CPU family is generally classified as a *von Neumann architecture machine*. Von Neumann computer systems contain three main building blocks: the *central processing unit (CPU)*, *memory*, and *input/output (I/O) devices*. These three components are interconnected via the *system bus* (consisting of the address, data, and control buses). The block diagram in Figure 1-1 shows these relationships.

The CPU communicates with memory and I/O devices by placing a numeric value on the address bus to select one of the memory locations or I/O device port locations, each of which has a unique numeric *address*. Then the CPU, memory, and I/O devices pass data among themselves by placing the data on the data bus. The control bus contains signals that determine the direction of the data transfer (to/from memory and to/from an I/O device).

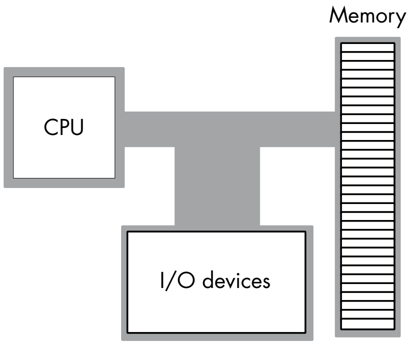


Figure 1-1: Von Neumann computer system block diagram

Within the CPU, special locations known as *registers* are used to manipulate data. The x86-64 CPU registers can be broken into four categories: general-purpose registers, special-purpose application-accessible registers, segment registers, and special-purpose kernel-mode registers. Because the segment registers aren't used much in modern 64-bit operating systems (such as Windows), there is little need to discuss them in this book. The special-purpose kernel-mode registers are intended for writing operating systems, debuggers, and other system-level tools. Such software construction is well beyond the scope of this text.

The x86-64 (Intel family) CPUs provide several *general-purpose registers* for application use. These include the following:

- Sixteen 64-bit registers that have the following names: RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8, R9, R10, R11, R12, R13, R14, and R15
- Sixteen 32-bit registers: EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, R8D, R9D, R10D, R11D, R12D, R13D, R14D, and R15D
- Sixteen 16-bit registers: AX, BX, CX, DX, SI, DI, BP, SP, R8W, R9W, R10W, R11W, R12W, R13W, R14W, and R15W
- Twenty 8-bit registers: AL, AH, BL, BH, CL, CH, DL, DH, DIL, SIL, BPL, SPL, R8B, R9B, R10B, R11B, R12B, R13B, R14B, and R15B

Unfortunately, these are not 68 independent registers; instead, the x86-64 overlays the 64-bit registers over the 32-bit registers, the 32-bit registers over the 16-bit registers, and the 16-bit registers over the 8-bit registers. Table 1-1 shows these relationships.

Because the general-purpose registers are not independent, modifying one register may modify as many as three other registers. For example, modifying the EAX register may very well modify the AL, AH, AX, and RAX registers. This fact cannot be overemphasized. A common mistake in programs written by beginning assembly language programmers is register value corruption due to the programmer not completely understanding the ramifications of the relationships shown in Table 1-1.

Table 1-1: General-Purpose Registers on the x86-64

Bits 0–63	Bits 0–31	Bits 0–15	Bits 8–15	Bits 0–7
RAX	EAX	AX	AH	AL
RBX	EBX	BX	BH	BL
RCX	ECX	CX	CH	CL
RDX	EDX	DX	DH	DL
RSI	ESI	SI		SIL
RDI	EDI	DI		DIL
RBP	EBP	BP		BPL
RSP	ESP	SP		SPL
R8	R8D	R8W		R8B

Bits 0–63	Bits 0–31	Bits 0–15	Bits 8–15	Bits 0–7
R9	R9D	R9W		R9B
R10	R10D	R10W		R10B
R11	R11D	R11W		R11B
R12	R12D	R12W		R12B
R13	R13D	R13W		R13B
R14	R14D	R14W		R14B
R15	R15D	R15W		R15B

In addition to the general-purpose registers, the x86-64 provides special-purpose registers, including eight *floating-point registers* implemented in the x87 *floating-point unit (FPU)*. Intel named these registers ST(0) to ST(7). Unlike with the general-purpose registers, an application program cannot directly access these. Instead, a program treats the floating-point register file as an eight-entry-deep stack and accesses only the top one or two entries (see "Floating-Point Arithmetic" in Chapter 6 for more details).

Each floating-point register is 80 bits wide, holding an extended-precision real value (hereafter just *extended precision*). Although Intel added other floating-point registers to the x86-64 CPUs over the years, the FPU registers still find common use in code because they support this 80-bit floating-point format.

In the 1990s, Intel introduced the MMX register set and instructions to support *single instruction, multiple data (SIMD)* operations. The *MMX register set* is a group of eight 64-bit registers that overlay the ST(0) to ST(7) registers on the FPU. Intel chose to overlay the FPU registers because this made the MMX registers immediately compatible with multitasking operating systems (such as Windows) without any code changes to those OSs. Unfortunately, this choice meant that an application could not simultaneously use the FPU and MMX instructions.

Intel corrected this issue in later revisions of the x86-64 by adding the *XMM register set*. For that reason, you rarely see modern applications using the MMX registers and instruction set. They are available if you really want to use them, but it is almost always better to use the XMM registers (and instruction set) and leave the registers in FPU mode.

To overcome the limitations of the MMX/FPU register conflicts, AMD/Intel added sixteen 128-bit XMM registers (XMM0 to XMM15) and the SSE/SSE2 instruction set. Each register can be configured as four 32-bit floating-point registers; two 64-bit double-precision floating-point registers; or sixteen 8-bit, eight 16-bit, four 32-bit, two 64-bit, or one 128-bit integer registers. In later variants of the x86-64 CPU family, AMD/Intel doubled the size of the registers to 256 bits each (renaming them YMM0 to YMM15) to support eight 32-bit floating-point values or four 64-bit double-precision floating-point values (integer operations were still limited to 128 bits).

The *RFLAGS* (or just *FLAGS*) register is a 64-bit register that encapsulates several single-bit Boolean (true/false) values.[1] Most of the bits in the RFLAGS register are either reserved for kernel mode (operating system) functions or are of little interest to the application programmer. Eight of these bits (or *flags*) are of interest to application programmers writing assembly language programs: the overflow, direction, interrupt disable,[2] sign, zero, auxiliary carry, parity, and carry flags. Figure 1-2 shows the layout of the flags within the lower 16 bits of the RFLAGS register.

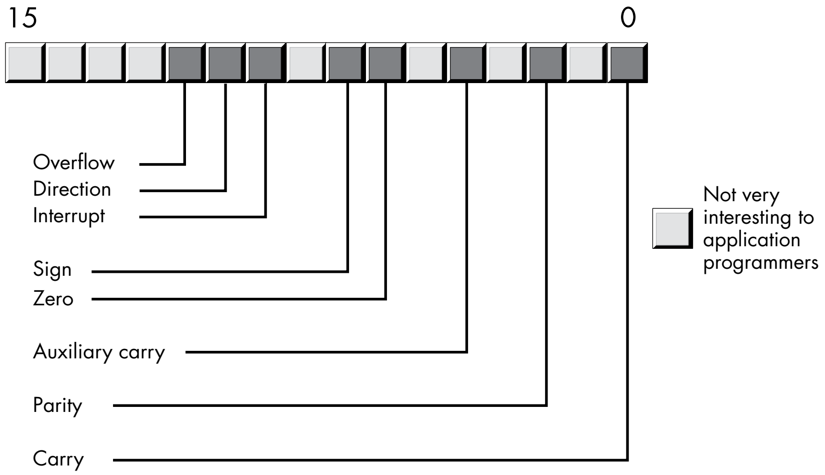


Figure 1-2: Layout of the FLAGS register (lower 16 bits of RFLAGS)

Four flags in particular are extremely valuable: the overflow, carry, sign, and zero flags, collectively called the *condition codes*.[3] The state of these flags lets you test the result of previous computations. For example, after comparing two values, the condition code flags will tell you whether one value is less than, equal to, or greater than a second value.

One important fact that comes as a surprise to those just learning assembly language is that almost all calculations on the x86-64 CPU involve a register. For example, to add two variables together and store the sum into a third variable, you must load one of the variables into a register, add the second operand to the value in the register, and then store the register away in the destination variable. Registers are a middleman in nearly every calculation.

You should also be aware that, although the registers are called *general-purpose*, you cannot use any register for any purpose. All the x86-64 registers have their own special purposes that limit their use in certain contexts. The RSP register, for example, has a very special purpose that effectively prevents

1. Technically, the I/O privilege level (IOPL) is 2 bits, but these bits are not accessible from user-mode programs, so this book ignores this field.

2. Application programs cannot modify the interrupt flag, but we'll look at this flag in Chapter 2; hence the discussion of this flag here.

3. Technically, the parity flag is also a condition code, but we will not use that flag in this text.

you from using it for anything else (it's the *stack pointer*). Likewise, the RBP register has a special purpose that limits its usefulness as a general-purpose register. For the time being, avoid the use of the RSP and RBP registers for generic calculations; also, keep in mind that the remaining registers are not completely interchangeable in your programs.

1.8 The Memory Subsystem

The *memory subsystem* holds data such as program variables, constants, machine instructions, and other information. Memory is organized into cells, each of which holds a small piece of information. The system can combine the information from these small cells (or *memory locations*) to form larger pieces of information.

The x86-64 supports *byte-addressable memory*, which means the basic memory unit is a byte, sufficient to hold a single character or a (very) small integer value (we'll talk more about that in Chapter 2).

Think of memory as a linear array of bytes. The address of the first byte is 0, and the address of the last byte is $2^{32} - 1$. For an x86 processor with 4GB memory installed,[4] the following pseudo-Pascal array declaration is a good approximation of memory:

```
Memory: array [0..4294967295] of byte;
```

C/C++ and Java users might prefer the following syntax:

```
byte Memory[4294967296];
```

For example, to execute the equivalent of the Pascal statement `Memory [125] := 0;`, the CPU places the value `0` on the data bus, places the address `125` on the address bus, and asserts the write line (this generally involves setting that line to `0`), as shown in Figure 1-3.

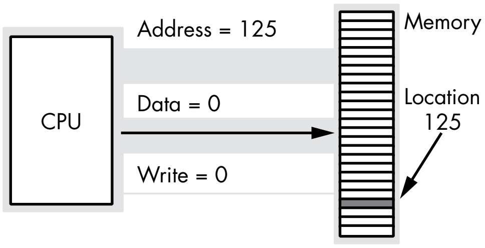


Figure 1-3: Memory write operation

To execute the equivalent of `CPU := Memory [125];`, the CPU places the address `125` on the address bus, asserts the read line (because the CPU is reading data from memory), and then reads the resulting data from the data bus (see Figure 1-4).

4. The following discussion will use the 4GB address space of the older 32-bit x86-64 processors. A typical x86-64 processor running a modern 64-bit OS can access a maximum of 2^{48} memory locations, or just over 256TB.

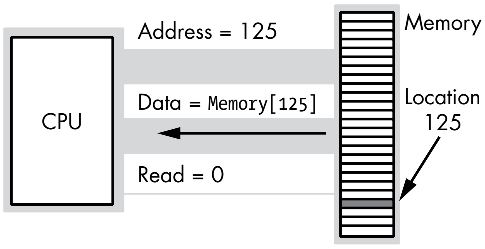


Figure 1-4: Memory read operation

To store larger values, the x86 uses a sequence of consecutive memory locations. Figure 1-5 shows how the x86 stores bytes, *words* (2 bytes), and *double words* (4 bytes) in memory. The memory address of each object is the address of the first byte of each object (that is, the lowest address).

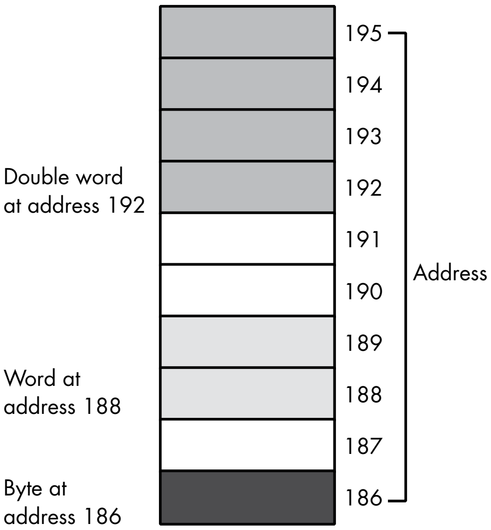


Figure 1-5: Byte, word, and double-word storage in memory

1.9 Declaring Memory Variables in MASM

Although it is possible to reference memory by using numeric addresses in assembly language, doing so is painful and error-prone. Rather than having your program state, "Give me the 32-bit value held in memory location 192 and the 16-bit value held in memory location 188," it's much nicer to state, "Give me the contents of `elementCount` and `portNumber`." Using variable names, rather than memory addresses, makes your program much easier to write, read, and maintain.

To create (writable) data variables, you have to put them in a data section of the MASM source file, defined using the `.data` directive. This directive tells MASM that all following statements (up to the next `.code` or other section-defining directive) will define data declarations to be grouped into a read/write section of memory.

Within a `.data` section, MASM allows you to declare variable objects by using a set of data declaration directives. The basic form of a data declaration directive is

```
label  directive ?
```

where *`label`* is a legal MASM identifier and *`directive`* is one of the directives appearing in Table 1-2.

Table 1-2: MASM Data Declaration Directives

Directive	Meaning
byte (or db)	Byte (unsigned 8-bit) value
sbyte	Signed 8-bit integer value
word (or dw)	Unsigned 16-bit (word) value
sword	Signed 16-bit integer value
dword (or dd)	Unsigned 32-bit (double-word) value
sdword	Signed 32-bit integer value
qword (or dq)	Unsigned 64-bit (quad-word) value
sqword	Signed 64-bit integer value
tbyte (or dt)	Unsigned 80-bit (10-byte) value
oword	128-bit (octal-word) value
real4	Single-precision (32-bit) floating-point value
real8	Double-precision (64-bit) floating-point value
real10	Extended-precision (80-bit) floating-point value

The question mark (`?`) operand tells MASM that the object will not have an explicit value when the program loads into memory (the default initialization is zero). If you would like to initialize the variable with an explicit value, replace the `?` with the initial value; for example:

```
hasInitialValue  sdword   -1
```

Some of the data declaration directives in Table 1-2 have a signed version (the directives with the `s` prefix). For the most part, MASM ignores this prefix. It is the machine instructions you write that differentiate between signed and unsigned operations; MASM itself usually doesn't care whether a variable holds a signed or an unsigned value. Indeed, MASM allows both of the following:

```
     .data
u8   byte    -1    ; Negative initializer is okay
i8   sbyte   250   ; even though +128 is maximum signed byte
```

All MASM cares about is whether the initial value will fit into a byte. The `-1`, even though it is not an unsigned value, will fit into a byte in memory. Even though `250` is too large to fit into a signed 8-bit integer (see

"Signed and Unsigned Numbers" in Chapter 2), MASM will happily accept this because `250` will fit into a byte variable (as an unsigned number).

It is possible to reserve storage for multiple data values in a single data declaration directive. The string multi-valued data type is critical to this chapter (later chapters discuss other types, such as arrays in Chapter 4). You can create a null-terminated string of characters in memory by using the `byte` directive as follows:

```
; Zero-terminated C/C++ string.
strVarName  byte 'String of characters', 0
```

Notice the `, 0` that appears after the string of characters. In any data declaration (not just byte declarations), you can place multiple data values in the operand field, separated by commas, and MASM will emit an object of the specified size and value for each operand. For string values (surrounded by apostrophes in this example), MASM emits a byte for each character in the string (plus a zero byte for the `, 0` operand at the end of the string). MASM allows you to define strings by using either apostrophes or quotes; you must terminate the string of characters with the same delimiter that begins the string (quote or apostrophe).

1.9.1 Associating Memory Addresses with Variables

One of the nice things about using an assembler/compiler like MASM is that you don't have to worry about numeric memory addresses. All you need to do is declare a variable in MASM, and MASM associates that variable with a unique set of memory addresses. For example, say you have the following declaration section:

```
     .data
i8   sbyte   ?
i16  sword   ?
i32  sdword  ?
i64  sqword  ?
```

MASM will find an unused 8-bit byte in memory and associate it with the `i8` variable; it will find a pair of consecutive unused bytes and associate them with `i16`; it will find four consecutive locations and associate them with `i32`; finally, MASM will find 8 consecutive unused bytes and associate them with `i64`. You'll always refer to these variables by their name. You generally don't have to concern yourself with their numeric address. Still, you should be aware that MASM is doing this for you.

When MASM is processing declarations in a `.data` section, it assigns consecutive memory locations to each variable.[5] Assuming `i8` (in the previous declarations) as a memory address of 101, MASM will assign the addresses appearing in Table 1-3 to `i8`, `i16`, `i32`, and `i64`.

5. Technically, MASM assigns offsets into the `.data` section to variables. Windows converts these offsets to physical memory addresses when it loads the program into memory at runtime.

Table 1-3: Variable Address Assignment

Variable	Memory address
`i8`	101
`i16`	102 (address of `i8` plus 1)
`i32`	104 (address of `i16` plus 2)
`i64`	108 (address of `i32` plus 4)

Whenever you have multiple operands in a data declaration statement, MASM will emit the values to sequential memory locations in the order they appear in the operand field. The label associated with the data declaration (if one is present) is associated with the address of the first (leftmost) operand's value. See Chapter 4 for more details.

1.9.2 Associating Data Types with Variables

During assembly, MASM associates a data type with every label you define, including variables. This is rather advanced for an assembly language (most assemblers simply associate a value or an address with an identifier).

For the most part, MASM uses the variable's size (in bytes) as its type (see Table 1-4).

Table 1-4: MASM Data Types

Type	Size	Description
`byte` (`db`)	1	1-byte memory operand, unsigned (generic integer)
`sbyte`	1	1-byte memory operand, signed integer
`word` (`dw`)	2	2-byte memory operand, unsigned (generic integer)
`sword`	2	2-byte memory operand, signed integer
`dword` (`dd`)	4	4-byte memory operand, unsigned (generic integer)
`sdword`	4	4-byte memory operand, signed integer
`qword` (`dq`)	8	8-byte memory operand, unsigned (generic integer)
`sqword`	8	8-byte memory operand, signed integer
`tbyte` (`dt`)	10	10-byte memory operand, unsigned (generic integer or BCD)
`oword`	16	16-byte memory operand, unsigned (generic integer)
`real4`	4	4-byte single-precision floating-point memory operand
`real8`	8	8-byte double-precision floating-point memory operand
`real10`	10	10-byte extended-precision floating-point memory operand
`proc`	N/A	Procedure label (associated with `PROC` directive)
`label`:	N/A	Statement label (any identifier immediately followed by a `:`)
`constant`	Varies	Constant declaration (equate) using `=` or `EQU` directive
`text`	N/A	Textual substitution using macro or `TEXTEQU` directive

Later sections and chapters fully describe the `proc`, *`label`*, *`constant`*, and *`text`* types.