CHAPTER 1

Understanding Global Concepts

With the development of the Microsoft Macro Assembler (MASM) version 6.1, you now have more options available to you for approaching a programming task. This chapter explains the general concepts of programming in assembly language, beginning with the environment and a review of the components you need to work in the assembler environment. Even if you are familiar with previous versions of MASM, you should examine this chapter for information on new terms and features.

The first section of this chapter reviews available processors and operating systems and how they work together. The section also discusses segmented architecture and how it affects a protected-mode operating environment such as Windows.

The second section describes some of the language components of MASM that are common to most programs, such as reserved words, constant expressions, operators, and registers. The remainder of this book was written with the assumption that you understand the information presented in this section.

The last section summarizes the assembly process, from assembling a program through running it. You can affect this process by the way you develop your code. Finally, this section explores how you can change the assembly process with the **OPTION** directive and conditional assembly.

The Processing Environment

The processing environment for MASM 6.1 includes the processor on which your programs run, the operating system your programs use, and the aspects of the segmented architecture that influence the choice of programming models. This section summarizes these elements of the environment and how they affect your programming choices.

8086-Based Processors

The 8086 "family" of processors uses segments to control data and code. The later 8086-based processors have larger instruction sets and more memory capacity, but they still support the same segmented architecture. Knowing the differences between the various 8086-based processors can help you select the appropriate target processor for your programs.

The instruction set of the 8086 processor is upwardly compatible with its successors. To write code that runs on the widest number of machines, select the 8086 instruction set. By using the instruction set of a more advanced processor, you increase the capabilities and efficiency of your program, but you also reduce the number of systems on which the program can run.

Table 1.1 lists modes, memory, and segment size of processors on which your application may need to run. Each processor is discussed in more detail following.

Table 1.1 8086 Family of Processors

Processor	Available Modes	Addressable Memory	Segment Size
8086/8088	Real	1 megabyte	16 bits
80186/80188	Real	1 megabyte	16 bits
80286	Real and Protected	16 megabytes	16 bits
80386	Real and Protected	4 gigabytes	16 or 32 bits
80486	Real and Protected	4 gigabytes	16 or 32 bits

Processor Modes

Real mode allows only one process to run at a time. The mode gets its name from the fact that addresses in real mode always correspond to real locations in memory. The MS-DOS operating system runs in real mode.

Windows 3.1 operates only in protected mode, but runs MS-DOS programs in real mode or in a simulation of real mode called virtual-86 mode. In protected mode, more than one process can be active at any one time. The operating system protects memory belonging to one process from access by another process; hence the name protected mode.

Protected-mode addresses do not correspond directly to physical memory. Under protected-mode operating systems, the processor allocates and manages memory dynamically. Additional privileged instructions initialize protected mode and control multiple processes. For more information, see "Operating Systems," following.

8086 and 8088

The 8086 is faster than the 8088 because of its 16-bit data bus; the 8088 has only an 8-bit data bus. The 16-bit data bus allows you to use **EVEN** and **ALIGN** on an 8086 processor to word-align data and thus improve data-handling efficiency. Memory addresses on the 8086 and 8088 refer to actual physical addresses.

80186 and 80188

These two processors are identical to the 8086 and 8088 except that new instructions have been added and several old instructions have been optimized. These processors run significantly faster than the 8086.

80286

The 80286 processor adds some instructions to control protected mode, and it runs faster. It also provides protected mode services, allowing the operating system to run multiple processes at the same time. The 80286 is the minimum for running Windows 3.1 and 16-bit versions of OS/2 ®.

80386

Unlike its predecessors, the 80386 processor can handle both 16-bit and 32-bit data. It supports the entire instruction set of the 80286, and adds several new instructions as well. Software written for the 80286 runs unchanged on the 80386, but is faster because the chip operates at higher speeds.

The 80386 implements many new hardware-level features, including paged memory, multiple virtual 8086 processes, addressing of up to 4 gigabytes of memory, and specialized debugging registers. Thirty-two–bit operating systems such as Windows NT and OS/2 2.0 can run only on an 80386 or higher processor.

80486

The 80486 processor is an enhanced version of the 80386, with instruction “pipelining” that executes many instructions two to three times faster. The chip incorporates both a math coprocessor and an 8K (kilobyte) memory cache. (The math coprocessor is disabled on a variation of the chip called the 80486SX.) The 80486 includes new instructions and is fully compatible with 80386 software.

8087, 80287, and 80387

These math coprocessors work concurrently with the 8086 family of processors. Performing floating-point calculations with math coprocessors is up to 100 times faster than emulating the calculations with integer instructions. Although there are technical and performance differences among the three coprocessors, the main difference to the applications programmer is that the 80287 and 80387 can

operate in protected mode. The 80387 also has several new instructions. The 80486 does not use any of these coprocessors; its floating-point processor is built in and is functionally equivalent to the 80387.

Operating Systems

With MASM, you can create programs that run under MS-DOS, Windows, or Windows NT—or all three, in some cases. For example, ML.EXE can produce executable files that run in any of the target environments, regardless of the programmer's environment. For information on building programs for different environments, see "Building and Running Programs" in Help for PWB.

MS-DOS and Windows 3.1 provide different processing modes. MS-DOS runs in the single-process real mode. Windows 3.1 operates in protected mode, allowing multiple processes to run simultaneously.

Although Windows requires another operating system for loading and file services, it provides many functions normally associated with an operating system. When an application requests an MS-DOS service, Windows often provides the service without invoking MS-DOS. For consistency, this book refers to Windows as an operating system.

MS-DOS and Windows (in protected mode) differ primarily in system access methods, size of addressable memory, and segment selection. Table 1.2 summarizes these differences.

Table 1.2 The MS-DOS and Windows Operating Systems Compared

Operating System	System Access	Available Active Processes	Addressable Memory	Contents of Segment Register	Word Length
MS-DOS and Windows real mode	Direct to hardware and OS call	One	1 megabyte	Actual address	16 bits
Windows virtual-86 mode	Operating system call	Multiple	1 megabyte	Segment selectors	16 bits
Windows protected mode	Operating system call	Multiple	16 megabytes	Segment selectors	16 bits
Windows NT	Operating system call	Multiple	512 megabytes	Segment selectors	32 bits

MS-DOS

In real-mode programming, you can access system functions by calling MS-DOS, calling the basic input/output system (BIOS), or directly addressing hardware. Access is through MS-DOS Interrupt 21h.

Windows

As you can see in Table 1.2, protected mode allows for much larger data structures than real mode, since addressable memory extends to 16 megabytes. In protected mode, segment registers contain selector values rather than actual segment addresses. These selectors cannot be calculated by the program; they must be obtained by calling the operating system. Programs that attempt to calculate segment values or to address memory directly do not work in protected mode.

Protected mode uses privilege levels to maintain system integrity and security. Programs cannot access data or code that is in a higher privilege level. Some instructions that directly access ports or affect interrupts (such as **CLI**, **STI**, **IN**, and **OUT**) are available at privilege levels normally used only by systems programmers.

Windows protected mode provides each application with up to 16 megabytes of "virtual memory," even on computers that have less physical memory. The term virtual memory refers to the operating system's ability to use a swap area on the hard disk as an extension of real memory. When a Windows application requires more memory than is available, Windows writes sections of occupied memory to the swap area, thus freeing those sections for other use. It then provides the memory to the application that made the memory request. When the owner of the swapped data regains control, Windows restores the data from disk to memory, swapping out other memory if required.

Windows NT

Windows NT uses the so-called "flat model" of 80386/486 processors. This model places the processor's entire address space within one 32-bit segment. The section "Defining Basic Attributes with .MODEL" in Chapter 2 explains how to use the flat model. In flat model, your program can (in theory) access up to 4 gigabytes of virtual memory. Since code, data, and stack reside in the same segment, each segment register can hold the same value, which need never change.

Segmented Architecture

The 8086 family of processors employs a segmented architecture—that is, each address is represented as a segment and an offset. Segmented addresses affect many aspects of assembly-language programming, especially addresses and pointers.

Segmented architecture was originally designed to enable a 16-bit processor to access an address space larger than 64K. (The section "Segmented Addressing," later in this chapter, explains how the processor uses both the segment and offset to create addresses larger than 64K.) MS-DOS is an example of an operating system that uses segmented architecture on a 16-bit processor.

With the advent of protected-mode processors such as the 80286, segmented architecture gained a second purpose. Segments can separate different blocks of code and data to protect them from undesirable interactions. Windows takes advantage of the protection features of the 16-bit segments on the 80286.

Segmented architecture went through another significant change with the release of 32-bit processors, starting with the 80386. These processors are compatible with the older 16-bit processors, but allow flat model 32-bit offset values up to 4 gigabytes. Offset values of this magnitude remove the memory limitations of segmented architecture. The Windows NT operating system uses 32-bit addressing.

Segment Protection

Segmented architecture is an important part of the Windows memory-protection scheme. In a "multitasking" operating system in which numerous programs can run simultaneously, programs cannot access the code and data of another process without permission.

In MS-DOS, the data and code segments are usually allocated adjacent to each other, as shown in Figure 1.1. In Windows, the data and code segments can be anywhere in memory. The programmer knows nothing about, and has no control over, their location. The operating system can even move the segments to a new memory location or to disk while the program is running.

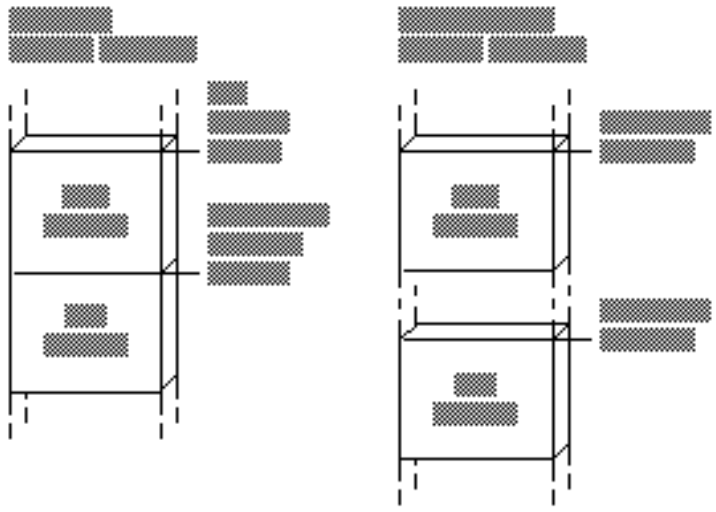

Figure 1.1 Segment Allocation

Segment protection makes software development easier and more reliable in Windows than in MS-DOS, because Windows immediately detects illegal

memory accesses. The operating system intercepts illegal memory accesses, terminates the program, and displays a message. This makes it easier for you to track down and fix the bug.

Because it runs in real mode, MS-DOS contains no mechanism for detecting an improper memory access. A program that overwrites data not belonging to it may continue to run and even terminate correctly. The error may not surface until later, when MS-DOS or another program reads the corrupted memory.

Segmented Addressing

Segmented addressing refers to the internal mechanism that combines a segment value and an offset value to form a complete memory address. The two parts of an address are represented as

segment:*offset*

The *segment* portion always consists of a 16-bit value. The *offset* portion is a 16-bit value in 16-bit mode or a 32-bit value in 32-bit mode.

In real mode, the segment value is a physical address that has an arithmetic relationship to the offset value. The segment and offset together create a 20-bit physical address (explained in the next section). Although 20-bit addresses can access up to 1 megabyte of memory, the BIOS and operating system on International Standard Architecture (IBM PC/AT and compatible) computers use part of this memory, leaving the remainder available for programs.

Segment Arithmetic

Manipulating segment and offset addresses directly in real-mode programming is called “segment arithmetic.” Programs that perform segment arithmetic are not portable to protected-mode operating systems, in which addresses do not correspond to a known segment and offset.

To perform segment arithmetic successfully, it helps to understand how the processor combines a 16-bit segment and a 16-bit offset to form a 20-bit linear address. In effect, the segment selects a 64K region of memory, and the offset selects the byte within that region. Here’s how it works:

1. The processor shifts the segment address to the left by four binary places, producing a 20-bit address ending in four zeros. This operation has the effect of multiplying the segment address by 16.
2. The processor adds this 20-bit segment address to the 16-bit offset address. The offset address is not shifted.
3. The processor uses the resulting 20-bit address, called the “physical address,” to access an actual location in the 1-megabyte address space.

Figure 1.2 illustrates this process.

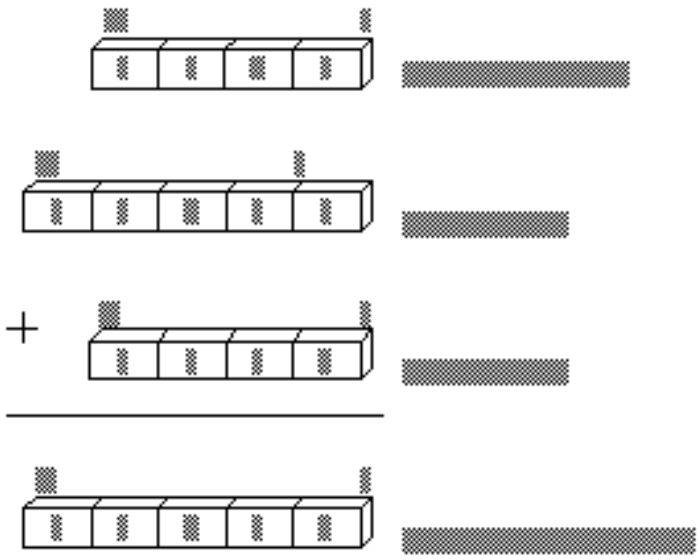

Figure 1.2 Calculating Physical Addresses

A 20-bit physical address may actually be specified by 4,096 equivalent *segment*:*offset* addresses. For example, the addresses 0000:F800, 0F00:0800, and 0F80:0000 all refer to the same physical address 0F800.

Language Components of MASM

Programming with MASM requires that you understand the MASM concepts of reserved words, identifiers, predefined symbols, constants, expressions, operators, data types, registers, and statements. This section defines important terms and provides lists that summarize these topics. For detailed information, see Help or the *Reference*.

Reserved Words

A reserved word has a special meaning fixed by the language. You can use it only under certain conditions. Reserved words in MASM include:

- Instructions, which correspond to operations the processor can execute.
- Directives, which give commands to the assembler.
- Attributes, which provide a value for a field, such as segment alignment.
- Operators, which are used in expressions.

- Predefined symbols, which return information to your program.

MASM reserved words are not case sensitive except for predefined symbols (see “Predefined Symbols,” later in this chapter).

The assembler generates an error if you use a reserved word as a variable, code label, or other identifier within your source code. However, if you need to use a reserved word for another purpose, the **OPTION NOKEYWORD** directive can selectively disable a word’s status as a reserved word.

For example, to remove the **STR** instruction, the **MASK** operator, and the **NAME** directive from the set of words MASM recognizes as reserved, use this statement in the code segment of your program before the first reference to **STR**, **MASK**, or **NAME**:

```
OPTION NOKEYWORD:<STR MASK NAME>
```

The section “Using the OPTION Directive,” later in this chapter, discusses the **OPTION** directive. Appendix D provides a complete list of MASM reserved words.

With the /Zm command-line option or **OPTION M510** in effect, MASM does not reserve any operators or instructions that do not apply to the current CPU mode. For example, you can use the symbol `ENTER` when assembling under the default CPU mode but not under **.286** mode, since the 80186/486 processors recognize `ENTER` as an instruction. The **USE32**, **FLAT**, **FAR32**, and **NEAR32** segment types and the 80386/486 register names are not keywords with processors other than the 80386/486.

Identifiers

An identifier is a name that you invent and attach to a definition. Identifiers can be symbols representing variables, constants, procedure names, code labels, segment names, and user-defined data types such as structures, unions, records, and types defined with **TYPEDEF**. Identifiers longer than 247 characters generate an error.

Certain restrictions limit the names you can use for identifiers. Follow these rules to define a name for an identifier:

- The first character of the identifier can be an alphabetic character (A–Z) or any of these four characters: @ _ $?
- The other characters in the identifier can be any of the characters listed above or a decimal digit (0–9).

Avoid starting an identifier with the at sign (@), because MASM 6.1 predefines some special symbols starting with @ (see “Predefined Symbols,” following).

Beginning an identifier with @ may also cause conflicts with future versions of the Macro Assembler.

The symbol—and thus the identifier—is visible as long as it remains within scope. (For more information about visibility and scope, see "Sharing Symbols with Include Files" in Chapter 8.)

Predefined Symbols

The assembler includes a number of predefined symbols (also called predefined equates). You can use these symbol names at any point in your code to represent the equate value. For example, the predefined equate **@FileName** represents the base name of the current file. If the current source file is TASK.ASM, the value of **@FileName** is TASK. The MASM predefined symbols are listed according to the kinds of information they provide. Case is important only if the /Cp option is used. (For additional details, see Help on ML command-line options.)

The predefined symbols for segment information include:

Symbol	Description
@code	Returns the name of the code segment.
@CodeSize	Returns an integer representing the default code distance.
@CurSeg	Returns the name of the current segment.
@data	Expands to DGROUP.
@DataSize	Returns an integer representing the default data distance.
@fardata	Returns the name of the segment defined by the **.FARDATA** directive.
@fardata?	Returns the name of the segment defined by the **.FARDATA?** directive.
@Model	Returns the selected memory model.
@stack	Expands to DGROUP for near stacks or STACK for far stacks. (See "Creating a Stack" in Chapter 2.)
@WordSize	Provides the size attribute of the current segment.

The predefined symbols for environment information include:

Symbol	Description
@Cpu	Contains a bit mask specifying the processor mode.
@Environ	Returns values of environment variables during assembly.
@Interface	Contains information about the language parameters.
@Version	Represents the text equivalent of the MASM version number. In MASM 6.1, this expands to 610.

The predefined symbols for date and time information include:

Symbol	Description
@Date	Supplies the current system date during assembly.
@Time	Supplies the current system time during assembly.

The predefined symbols for file information include:

Symbol	Description
@FileCur	Names the current file (base and suffix).
@FileName	Names the base name of the main file being assembled as it appears on the command line.
@Line	Gives the source line number in the current file.

The predefined symbols for macro string manipulation include:

Symbol	Description
@CatStr	Returns concatenation of two strings.
@InStr	Returns the starting position of a string within another string.
@SizeStr	Returns the length of a given string.
@SubStr	Returns substring from a given string.

Integer Constants and Constant Expressions

An integer constant is a series of one or more numerals followed by an optional radix specifier. For example, in these statements

```
mov     ax, 25
mov     bx, 0B3h
```

the numbers `25` and `0B3h` are integer constants. The `h` appended to `0B3` is a radix specifier. The specifiers are:

- `y` for binary (or `b` if the default radix is not hexadecimal)
- `o` or `q` for octal
- `t` for decimal (or `d` if the default radix is not hexadecimal)
- `h` for hexadecimal

Radix specifiers can be either uppercase or lowercase letters; sample code in this book is in lowercase. If you do not specify a radix, the assembler interprets the integer according to the current radix. The default radix is decimal, but you can change the default with the **.RADIX** directive.

Hexadecimal numbers must always start with a decimal digit (0–9). If necessary, add a leading zero to distinguish between symbols and hexadecimal numbers that start with a letter. For example, MASM interprets `ABCh` as an identifier. The hexadecimal digits A through F can be either uppercase or lowercase letters. Sample code in this book is in uppercase letters.

Constant expressions contain integer constants and (optionally) operators such as shift, logical, and arithmetic operators. The assembler evaluates constant expressions at assembly time. (In addition to constants, expressions can contain labels, types, registers, and their attributes.) Constant expressions do not change value during program execution.

Symbolic Integer Constants

You can define symbolic integer constants with either of the data assignment directives, **EQU** or the equal sign (=). These directives assign values to symbols during assembly, not during program execution. Symbolic constants are used to assign names to constant values. You can use a symbol with an assigned value in place of an immediate operand. For example, instead of referring in your code to keyboard scan codes with numbers such as 30 or 48, you can create more recognizable symbols:

```
SCAN_A  EQU    30
SCAN_B  EQU    48
```

then use the appropriate symbol in your program rather than the number. Using symbolic constants instead of undescriptive numbers makes your code more readable and easier to maintain. The assembler does not allocate data storage when you use either **EQU** or =. It simply replaces each occurrence of the symbol with the value of the expression.

The directives **EQU** and = have slightly different purposes. Integers defined with the = directive can be redefined with another value in your source code, but those defined with **EQU** cannot. Once you've defined a symbolic constant with the **EQU** directive, attempting to redefine it generates an error. The syntax is:

symbol **EQU** *expression*

The *symbol* is a unique name of your choice, except for words reserved by MASM. The *expression* can be an integer, a constant expression, a one- or two-character string constant (four-character on the 80386/486), or an expression that evaluates to an address. Symbolic constants let you change a constant value used throughout your source code by merely altering *expression* in the definition. This removes the potential for error and saves you the inconvenience of having to find and replace each occurrence of the constant in your program.

The following example shows the correct use of **EQU** to define symbolic integers.

```
column  EQU    80               ; Constant -    80
row     EQU    25               ; Constant -    25
screen  EQU    column * row     ; Constant  - 2000
line    EQU    row              ; Constant -    25

        .DATA

        .CODE
        .
        .
        .
        mov     cx, column
        mov     bx, line
```

The value of a symbol defined with the = directive can be different at different places in the source code. However, a constant value is assigned during assembly for each use, and that value does not change at run time.

The syntax for the = directive is:

symbol = expression

Size of Constants

The default word size for MASM 6.1 expressions is 32 bits. This behavior can be modified using **OPTION EXPR16** or **OPTION M510**. Both of these options set the expression word size to 16 bits, but **OPTION M510** affects other assembler behavior as well (see Appendix A).

It is illegal to change the expression word size once it has been set with **OPTION M510**, **OPTION EXPR16**, or **OPTION EXPR32**. However, you can repeat the same directive in your source code as often as you wish. You can place the same directive in every include file, for example.

Operators

Operators are used in expressions. The value of the expression is determined at assembly time and does not change when the program runs.

Operators should not be confused with processor instructions. The reserved word **ADD** is an instruction; the plus sign (+) is an operator. For example, `Amount+2` illustrates a valid use of the plus operator (+). It tells the assembler to add 2 to the constant value `Amount`, which might be a value or an address. Contrast this operation, which occurs at assembly time, with the processor's **ADD** instruction. **ADD** tells the processor at run time to add two numbers and store the result.

The assembler evaluates expressions that contain more than one operator according to the following rules:

- Operations in parentheses are performed before adjacent operations.
- Binary operations of highest precedence are performed first.
- Operations of equal precedence are performed from left to right.
- Unary operations of equal precedence are performed right to left.

Table 1.3 lists the order of precedence for all operators. Operators on the same line have equal precedence.

Table 1.3 Operator Precedence

Precedence	Operators
1	**( ), [ ]**
2	**LENGTH, SIZE, WIDTH, MASK, LENGTHOF, SIZEOF**
3	**.** (structure-field-name operator)
4	**:** (segment-override operator), **PTR**
5	**LROFFSET, OFFSET, SEG, THIS, TYPE**
6	**HIGH, HIGHWORD, LOW, LOWWORD**
7	+ ,– (unary)
8	*, /, **MOD, SHL, SHR**
9	+, – (binary)
10	**EQ, NE, LT, LE, GT, GE**
11	**NOT**
12	**AND**
13	**OR, XOR**
14	**OPATTR, SHORT, .TYPE**

Data Types

A "data type" describes a set of values. A variable of a given type can have any of a set of values within the range specified for that type.

The intrinsic types for MASM 6.1 are **BYTE**, **SBYTE**, **WORD**, **SWORD**, **DWORD**, **SDWORD**, **FWORD**, **QWORD**, and **TBYTE**. These types define integers and binary coded decimals (BCDs), as discussed in Chapter 6. The signed data types **SBYTE**, **SWORD**, and **SDWORD** work in conjunction with directives such as **INVOKE** (for calling procedures) and **.IF** (introduced in Chapter 7). The **REAL4**, **REAL8**, and **REAL10** directives define floating-point types. (See Chapter 6.)

Versions of MASM prior to 6.0 had separate directives for types and initializers. For example, **BYTE** is a type and **DB** is the corresponding initializer. The distinction does not apply in MASM 6.1. You can use any type (intrinsic or user-defined) as an initializer.

MASM does not have specific types for arrays and strings. However, you can treat a sequence of data units as arrays, and character or byte sequences as strings. (See "Arrays and Strings" in Chapter 5.)

Types can also have attributes such as *langtype* and *distance* (**NEAR** and **FAR**). For information on these attributes, see "Declaring Parameters with the PROC Directive" in Chapter 7.

You can also define your own types with **STRUCT**, **UNION**, and **RECORD**. The types have fields that contain string or numeric data, or records that contain bits. These data types are similar to the user-defined data types in high-level languages such as C, Pascal, and FORTRAN. (See Chapter 5, "Defining and Using Complex Data Types.")

You can define new types, including pointer types, with the **TYPEDEF** directive. **TYPEDEF** assigns a *qualifiedtype* (explained in the following) to a *typename* of your choice. This lets you build new types with descriptive names of your choosing, making your programs more readable. For example, the following statement makes the symbol `CHAR` a synonym for the intrinsic type **BYTE**:

```
CHAR    TYPEDEF BYTE
```

The *qualifiedtype* is any type or pointer to a type of the form:

[[*distance*]] **PTR** [[*qualifiedtype*]]

where *distance* is **NEAR, FAR**, or any distance modifier. (For more information on *distance*, see "Declaring Parameters with the PROC Directive" in Chapter 7.)

The *qualifiedtype* can also be any type previously defined with **TYPEDEF**. For example, if you use **TYPEDEF** to create an alias for **BYTE**—say, `CHAR` as in the preceding example—you can use `CHAR` as a *qualifiedtype* when defining the pointer type `PCHAR`, like this:

```
CHAR    TYPEDEF BYTE
PCHAR   TYPEDEF PTR CHAR
```

The typename `CHAR` in the first line becomes a *qualifiedtype* in the second line. Use of the **TYPEDEF** directive to define pointers is explained in "Accessing Data with Pointers and Addresses" in Chapter 3.

Since *distance* and *qualifiedtype* are optional syntax elements, you can use variables of type **PTR** or **FAR PTR**. You can also define procedure prototypes with *qualifiedtype*. For more information about procedure prototypes, see "Declaring Procedure Prototypes" in Chapter 7.

These rules govern the use of *qualifiedtype*:

- The only component of a *qualifiedtype* definition that can be forward-referenced is a structure or union type identifier.
- If you do not specify *distance*, the assembler assumes a distance that corresponds to the memory model. The assumed distance is **NEAR** for tiny, small, and medium models, and **FAR** for other models.
- If you do not specify a memory model with **.MODEL**, the assembler assumes **SMALL** model (and therefore **NEAR** pointers).

You can use a *qualifiedtype* in seven places:

Use	Example
In procedure arguments	`proc1 PROC pMsg:PTR BYTE`
In prototype arguments	`proc2 PROTO pMsg:FAR PTR WORD`
With local variables declared inside procedures	`LOCAL pMsg:PTR`
With the **LABEL** directive	`TempMsg LABEL PTR WORD`
With the **EXTERN** and **EXTERNDEF** directives	`EXTERN pMsg:FAR PTR BYTE` `EXTERNDEF MyProc:PROTO`
With the **COMM** directive	`COMM var1:WORD:3`
With the **TYPEDEF** directive	`PBYTE TYPEDEF PTR BYTE` `PFUNC TYPEDEF PROTO MyProc`

"Defining Pointer Types with TYPEDEF" in Chapter 3 shows ways to write a **TYPEDEF** type for a *qualifiedtype*. Attributes such as **NEAR** and **FAR** can also apply to a *qualifiedtype*.

You can determine an accurate definition for **TYPEDEF** and *qualifiedtype* from the BNF grammar definitions given in Appendix B. The BNF grammar defines each component of the syntax for any directive, showing the recursive properties of components such as *qualifiedtype*.

Registers

The 8086 family of processors have the same base set of 16-bit registers. Each processor can treat certain registers as two separate 8-bit registers. The 80386/486 processors have extended 32-bit registers. To maintain compatibility

with their predecessors, 80386/486 processors can access their registers as 16-bit or, where appropriate, as 8-bit values.

Figure 1.3 shows the registers common to all the 8086-based processors. Each register has its own special uses and limitations.

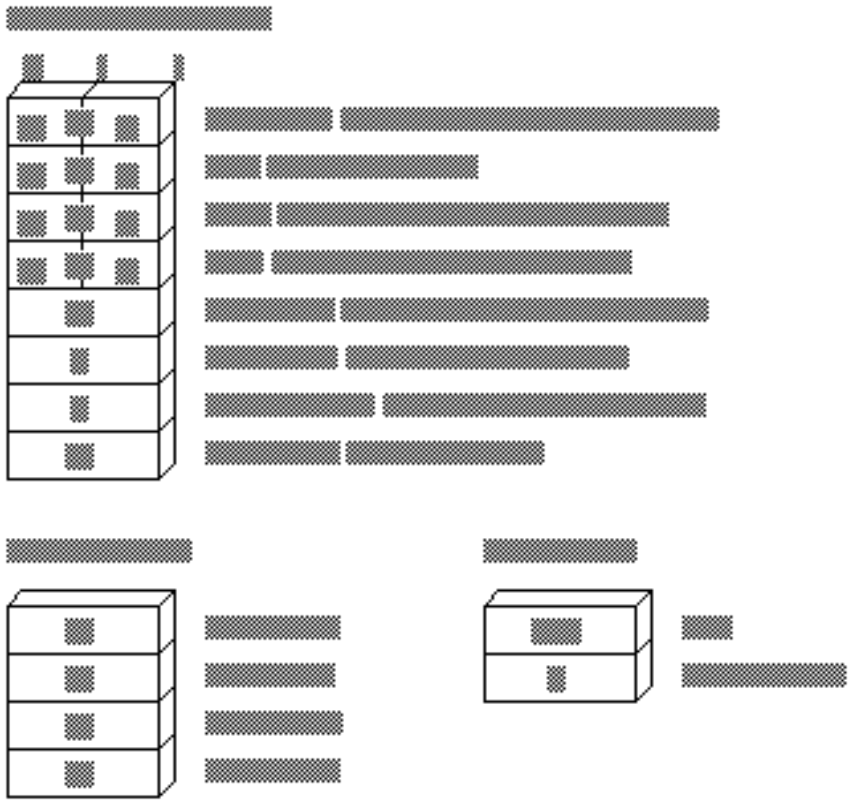

Figure 1.3 Registers for 8088–80286 Processors

80386/486 Only

The 80386/486 processors use the same 8-bit and 16-bit registers used by the rest of the 8086 family. All of these registers can be further extended to 32 bits, except segment registers, which always occupy 16 bits. The extended register names begin with the letter "E." For example, the 32-bit extension of AX is EAX. The 80386/486 processors have two additional segment registers, FS and GS. Figure 1.4 shows the extended registers of the 80386/486.

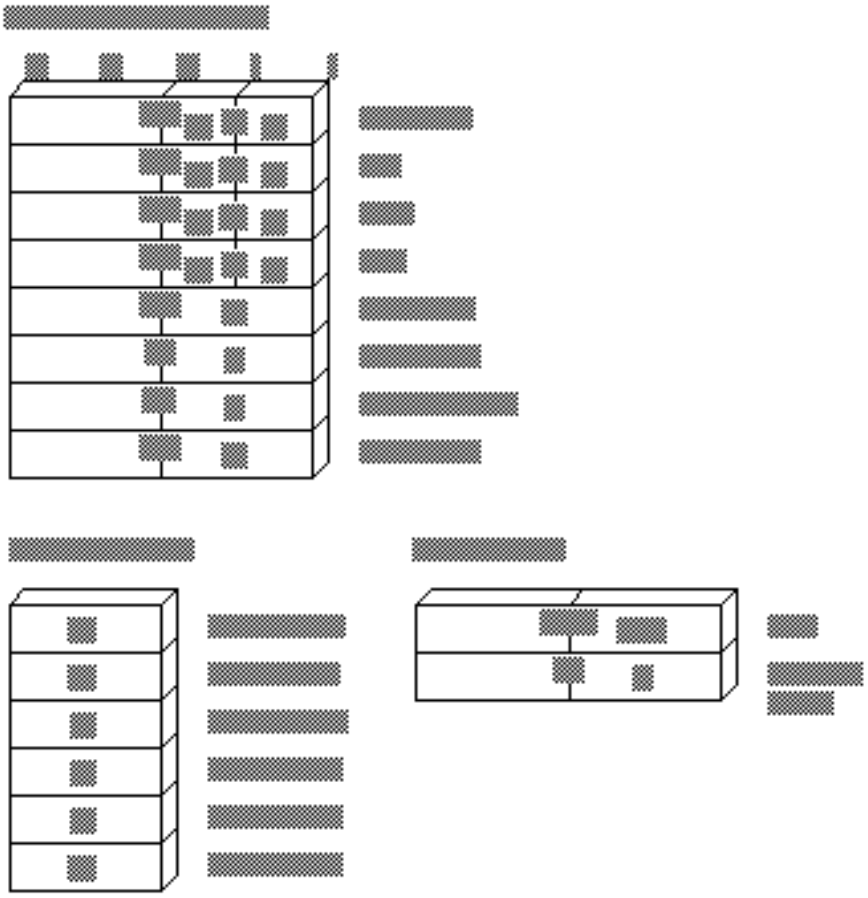

Figure 1.4 Extended Registers for the 80386/486 Processors

Segment Registers

At run time, all addresses are relative to one of four segment registers: CS, DS, SS, or ES. (The 80386/486 processors add two more: FS and GS.) These registers, their segments, and their purposes include:

Register and Segment	Purpose
CS (Code Segment)	Contains processor instructions and their immediate operands.
DS (Data Segment)	Normally contains data allocated by the program.
SS (Stack Segment)	Contains the program stack for use by **PUSH**, **POP**, **CALL**, and **RET**.

Register and Segment	Purpose
ES (Extra Segment)	References secondary data segment. Used by string instructions.
FS, GS	Provides extra segments on the 80386/486.

General-Purpose Registers

The AX, DX, CX, BX, BP, DI, and SI registers are 16-bit general-purpose registers, used for temporary data storage. Since the processor accesses registers more quickly than it accesses memory, you can make your programs run faster by keeping the most-frequently used data in registers.

The 8086-based processors do not perform memory-to-memory operations. For example, the processor cannot directly copy a variable from one location in memory to another. You must first copy from memory to a register, then from the register to the new memory location. Similarly, to add two variables in memory, you must first copy one variable to a register, then add the contents of the register to the other variable in memory.

The processor can access four of the general registers—AX, DX, CX, and BX—either as two 8-bit registers or as a single 16-bit register. The AH, DH, CH, and BH registers represent the high-order 8 bits of the corresponding registers. Similarly, AL, DL, CL, and BL represent the low-order 8 bits of the registers.

The 80386/486 processors can extend all the general registers to 32 bits, though as Figure 1.4 shows, you cannot treat the upper 16 bits as a separate register as you can the lower 16 bits. To use EAX as an example, you can directly reference the low byte as AL, the next lowest byte as AH, and the low word as AX. To access the high word of EAX, however, you must first shift the upper 16 bits into the lower 16 bits.

Special-Purpose Registers

The 8086 family of processors has two additional registers, SP and IP, whose values are changed automatically by the processor.

SP (Stack Pointer)

The SP register points to the current location within the stack segment. Pushing a value onto the stack decreases the value of SP by two; popping from the stack increases the value of SP by two. Thirty-two–bit operands on 80386/486 processors increase or decrease SP by four instead of two. The **CALL** and **INT** instructions store the return address on the stack and reduce SP accordingly. Return instructions retrieve the stored address from the stack and reset SP to its value before the call. SP can also be adjusted with instructions such as **ADD**. The program stack is described in detail in Chapter 3.

IP (Instruction Pointer)

The IP register always contains the address of the next instruction to be executed. You cannot directly access or change the instruction pointer. However, instructions that control program flow (such as calls, jumps, loops, and interrupts) automatically change the instruction pointer.

Flags Register

The 16 bits in the flags register control the execution of certain instructions and reflect the current status of the processor. In 80386/486 processors, the flags register is extended to 32 bits. Some bits are undefined, so there are actually 9 flags for real mode, 11 flags (including a 2-bit flag) for 80286 protected mode, 13 for the 80386, and 14 for the 80486. The extended flags register of the 80386/486 is sometimes called "Eflags."

Figure 1.5 shows the bits of the 32-bit flags register for the 80386/486. Earlier 8086-family processors use only the lower word. The unmarked bits are reserved for processor use, and should not be modified.

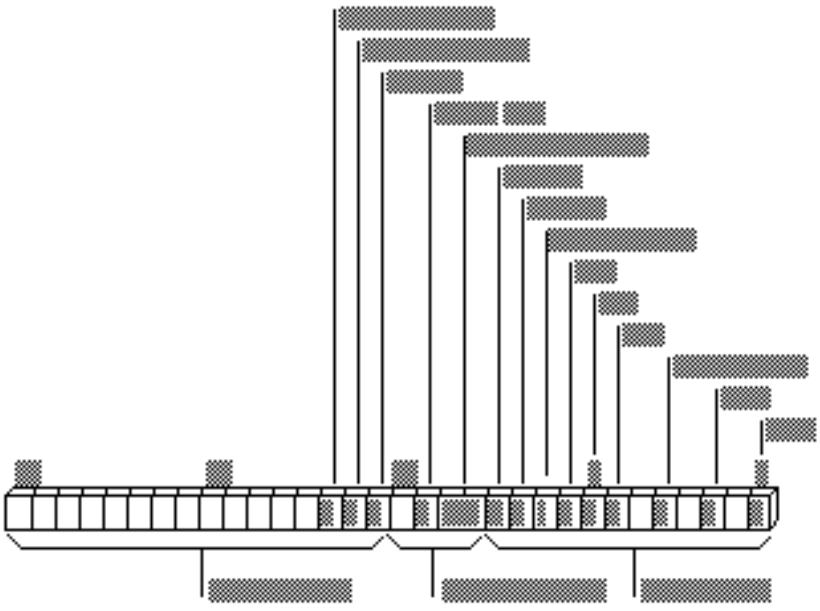

Figure 1.5 Flags for 8088-80486 Processors

In the following descriptions and throughout this book, "set" means a bit value of 1, and "cleared" means the bit value is 0. The nine flags common to all 8086-family processors, starting with the low-order flags, include:

Flag	Description
Carry	Set if an operation generates a carry to or a borrow from a destination operand.
Parity	Set if the low-order bits of the result of an operation contain an even number of set bits.
Auxiliary Carry	Set if an operation generates a carry to or a borrow from the low-order 4 bits of an operand. This flag is used for binary coded decimal (BCD) arithmetic.
Zero	Set if the result of an operation is 0.
Sign	Equal to the high-order bit of the result of an operation (0 is positive, 1 is negative).
Trap	If set, the processor generates a single-step interrupt after each instruction. A debugging program can use this feature to execute a program one instruction at a time.
Interrupt Enable	If set, interrupts are recognized and acted on as they are received. The bit can be cleared to turn off interrupt processing temporarily.
Direction	If set, string operations process down from high addresses to low addresses. If cleared, string operations process up from low addresses to high addresses.
Overflow	Set if the result of an operation is too large or small to fit in the destination operand.

Although all flags serve a purpose, most programs require only the carry, zero, sign, and direction flags.

Statements

Statements are the line-by-line components of source files. Each MASM statement specifies an instruction or directive for the assembler. Statements have up to four fields, as shown here:

[[*name***:**]] [[*operation*]] [[*operands*]] [[**;***comment*]]

The following list explains each field:

Field	Purpose
name	Labels the statement, so that instructions elsewhere in the program can refer to the statement by name. The *name* field can label a variable, type, segment, or code location.
operation	Defines the action of the statement. This field contains either an instruction or an assembler directive.
operands	Lists one or more items on which the instruction or directive operates.
comment	Provides a comment for the programmer. Comments are for documentation only; they are ignored by the assembler.

The following line contains all four fields:

```
mainlp: mov     ax, 7   ; Load AX with the value 7
```

Here, `mainlp` is the label, `mov` is the operation, and `ax` and `7` are the operands, separated by a comma. The comment follows the semicolon.

All fields are optional, although certain directives and instructions require an entry in the name or operand field. Some instructions and directives place restrictions on the choice of operands. By default, MASM is not case sensitive.

Each field (except the comment field) must be separated from other fields by white-space characters (spaces or tabs). MASM also requires code labels to be followed by a colon, operands to be separated by commas, and comments to be preceded by a semicolon.

A logical line can contain up to 512 characters and occupy one or more physical lines. To extend a logical line into two or more physical lines, put the backslash character (\) as the last non-whitespace character before the comment or end of the line. You can place a comment after the backslash as shown in this example:

```
.IF     (x > 0)    \  ; X must be positive
&&      (ax > x)   \  ; Result from function must be > x
&&      (cx == 0)     ; Check loop counter, too
mov     dx, 20h
.ENDIF
```

Multiline comments can also be specified with the **COMMENT** directive. The assembler ignores all text and code between the delimiters or on the same line as the delimiters. This example illustrates the use of **COMMENT**.

```
COMMENT ^               The assembler
                        ignores this text
^       mov     ax, 1   and this code
```

The Assembly Process

Creating and running an executable file involves four steps:

1. Assembling the source code into an object file
2. Linking the object file with other modules or libraries into an executable program
3. Loading the program into memory
4. Running the program

Once you have written your assembly-language program, MASM provides several options for assembling it. The **OPTION** directive has several different arguments that let you control the way MASM assembles your programs.

Conditional assembly allows you to create one source file that can generate a variety of programs, depending on the status of various conditional-assembly statements.

Generating and Running Executable Programs

This section briefly lists all the actions that take place during each of the assembly steps. You can change the behavior of some of these actions in various ways, such as using macros instead of procedures, or using the **OPTION** directive or conditional assembly. The other chapters in this book include specific programming methods; this section simply gives you an overview.

Assembling

The ML.EXE program does two things to create an executable program. First, it assembles the source code into an intermediate object file. Second, it calls the linker, LINK.EXE, which links the object files and libraries into an executable program.

At assembly time, the assembler:

- Evaluates conditional-assembly directives, assembling if the conditions are true.
- Expands macros and macro functions.
- Evaluates constant expressions such as `MYFLAG AND 80H`, substituting the calculated value for the expression.
- Encodes instructions and nonaddress operands. For example, `mov cx, 13` can be encoded at assembly time because the instruction does not access memory.
- Saves memory offsets as offsets from their segments.
- Places segments and segment attributes in the object file.
- Saves placeholders for offsets and segments (relocatable addresses).
- Outputs a listing if requested.
- Passes messages (such as **INCLUDELIB** and **.DOSSEG**) directly to the linker.

For information about conditional assembly, see “Conditional Directives” in this chapter; for macros, see Chapter 9. Further details about segments and offsets are included in Chapters 2 and 3. Assembly listings are explained in Appendix C.

Linking

Once your source code is assembled, the resulting object file is passed to the linker. At this point, the linker may combine several object files into an executable program. The linker:

- Combines segments according to the instructions in the object files, rearranging the positions of segments that share the same class or group.
- Fills in placeholders for offsets (relocatable addresses).
- Writes relocations for segments into the header of .EXE files (but not .COM files).
- Writes the result as an executable program file.

Classes and groups are defined in "Defining Segment Groups" in Chapter 2. Segments and offsets are explained in Chapter 3, "Using Addresses and Pointers."

Loading

After loading the executable file into memory, the operating system:

- Creates the program segment prefix (PSP) header in memory.
- Allocates memory for the program, based on the values in the PSP.
- Loads the program.
- Calculates the correct values for absolute addresses from the relocation table.
- Loads the segment registers SS, CS, DS, and ES with values that point to the proper areas of memory.

For information about segment registers, the instruction pointer (IP), and the stack pointer (SP), see "Registers" earlier in this chapter. For more information on the PSP see Help or an MS-DOS reference.

Running

To run your program, MS-DOS jumps to the program's first instruction. Some program operations, such as resolving indirect memory operands, cannot be handled until the program runs. For a description of indirect references, see "Indirect Operands" in Chapter 7.

Using the OPTION Directive

The **OPTION** directive lets you modify global aspects of the assembly process. With **OPTION**, you can change command-line options and default arguments. These changes affect only statements that follow the **OPTION** keyword.

For example, you may have MASM code in which the first character of a variable, macro, structure, or field name is a dot (.). Since a leading dot causes MASM 6.1 to generate an error, you can use this statement in your program:

```
OPTION DOTNAME
```

This enables the use of the dot for the first character.

Changes made with **OPTION** override any corresponding command-line option. For example, suppose you compile a module with this command line (which enables M510 compatibility):

```
ML /Zm TEST.ASM
```

The assembler disables M510 compatibility options for all code following this statement:

```
OPTION NOM510
```

The following lists explain each of the arguments for the **OPTION** directive. Where appropriate, an underline identifies the default argument. If you wish to place more than one **OPTION** statement on a line, separate them by commas.

Options for M510 compatibility include:

Argument	Description
CASEMAP: *maptype*	**CASEMAP:NONE** (or /Cx) causes internal symbol recognition to be case sensitive and causes the case of identifiers in the .OBJ file to be the same as specified in the **EXTERNDEF**, **PUBLIC**, or **COMM** statement. The default is **CASEMAP:NOTPUBLIC** (or /Cp). It specifies case insensitivity for internal symbol recognition and the same behavior as **CASEMAP:NONE** for case of identifiers in .OBJ files. **CASEMAP:ALL** (/Cu) specifies case insensitivity for identifiers and converts all identifier names to uppercase.
DOTNAME | **NODOTNAME**	Enables the use of the dot (.) as the leading character in variable, macro, structure, union, and member names.
M510 | **NOM510**	Sets all features to be compatible with MASM version 5.1, disabling the **SCOPED** argument and enabling **OLDMACROS**, **DOTNAME**, and, **OLDSTRUCTS**. **OPTION M510** conditionally sets other arguments for the **OPTION** directive. For more information on using **OPTION M510**, see Appendix A.

Argument	Description
OLDMACROS | **NOOLDMACROS**	Enables the version 5.1 treatment of macros. MASM 6.1 treats macros differently.
OLDSTRUCTS | **NOOLDSTRUCTS**	Enables compatibility with MASM 5.1 for treatment of structure members. See Chapter 5 for information on structures.
SCOPED | **NOSCOPED**	Guarantees that all labels inside procedures are local to the procedure when **SCOPED** (the default) is enabled.
SETIF2: TRUE | **FALSE**	If TRUE, **.ERR2** statements and **IF2** and **ELSEIF2** conditional blocks are evaluated on every pass. If FALSE, they are not evaluated. If **SETIF2** is not specified (or implied), **.ERR2**, **IF2**, and **ELSEIF2** expressions cause an error. Both the /Zm command-line argument and **OPTION M510** imply **SETIF2:TRUE**.

Options for procedure use include:

Argument	Description
LANGUAGE: *langtype*	Specifies the default language type (**C**, **PASCAL**, **FORTRAN**, **BASIC**, **SYSCALL**, or **STDCALL**) to be used with **PROC**, **EXTERN**, and **PUBLIC**. This use of the **OPTION** directive overrides the **.MODEL** directive but is normally used when **.MODEL** is not given.
EPILOGUE: *macroname*	Instructs the assembler to call the *macroname* to generate a user-defined epilogue instead of the standard epilogue code when a **RET** instruction is encountered. See Chapter 7.
PROLOGUE: *macroname*	Instructs the assembler to call *macroname* to generate a user-defined prologue instead of generating the standard prologue code. See Chapter 7.
PROC: *visibility*	Lets you explicitly set the default visibility as **PUBLIC**, **EXPORT**, or **PRIVATE.**

Other options include:

Argument	Description
EXPR16 | **EXPR32**	Sets the expression word size to 16 or 32 bits. The default is 32 bits. The **M510** argument to the **OPTION** directive sets the word size to 16 bits. Once set with the **OPTION** directive, the expression word size cannot be changed.

Argument	Description
EMULATOR | **NOEMULATOR**	Controls the generation of floating-point instructions.The **NOEMULATOR** option generates the coprocessor instructions directly. The **EMULATOR** option generates instructions with special fixup records for the linker so that the Microsoft floating-point emulator, supplied with other Microsoft languages, can be used. It produces the same result as setting the /Fpi command-line option. You can set this option only once per module.
LJMP | **NOLJMP**	Enables automatic conditional-jump lengthening. For information about conditional-jump lengthening, see Chapter 7.
NOKEYWORD:<*keywordlist*>	Disables the specified reserved words. For an example of the syntax for this argument, see "Reserved Words" in this chapter.
NOSIGNEXTEND	Overrides the default sign-extended opcodes for the **AND**, **OR**, and **XOR** instructions and generates the larger non-sign-extended forms of these instructions. Provided for compatibility with NEC V25 and NEC V35 controllers.
OFFSET: *offsettype*	Determines the result of **OFFSET** operator fixups. **SEGMENT** sets the defaults for fixups to be segment-relative (compatible with MASM 5.1). **GROUP**, the default, generates fixups relative to the group (if the label is in a group). **FLAT** causes fixups to be relative to a flat frame. (The **.386** mode must be enabled to use **FLAT**.) See Appendix A.
READONLY | **NOREADONLY**	Enables checking for instructions that modify code segments, thereby guaranteeing that read-only code segments are not modified. Same as the /p command-line option of MASM 5.1, except that it affects only segments with at least one assembly instruction, not all segments. The argument is useful for protected mode programs, where code segments must remain read-only.
SEGMENT: *segSize*	Allows global default segment size to be set. Also determines the default address size for external symbols defined outside any segment. The *segSize* can be **USE16**, **USE32**, or **FLAT**.

Conditional Directives

MASM 6.1 provides conditional-assembly directives and conditional-error directives. Conditional-assembly directives let you test for a specified condition and assemble a block of statements if the condition is true. Conditional-error directives allow you to test for a specified condition and generate an assembly error if the condition is true.

Both kinds of conditional directives test assembly-time conditions, not run-time conditions. You can test only expressions that evaluate to constants during assembly. For a list of the predefined symbols often used in conditional assembly, see "Predefined Symbols," earlier in this chapter.

Conditional-Assembly Directives

The **IF** and **ENDIF** directives enclose the conditional statements. The optional **ELSEIF** and **ELSE** blocks follow the **IF** directive. There are many forms of the **IF** and **ELSE** directives. Help provides a complete list.

The following statements show the syntax for the **IF** directives. The syntax for other condition-assembly directives follow the same form.

IF *expression1*
ifstatements
[[**ELSEIF** *expression2*
elseifstatements]]
[[**ELSE**
elsestatements]]
ENDIF

The *statements* within an **IF** block can be any valid instructions, including other conditional blocks, which in turn can contain any number of **ELSEIF** blocks. **ENDIF** ends the block.

MASM assembles the statements following the **IF** directive only if the corresponding condition is true. If the condition is not true and the block contains an **ELSEIF** directive, the assembler checks to see if the corresponding condition is true. If so, it assembles the statements following the **ELSEIF** directive. If no **IF** or **ELSEIF** conditions are satisfied, the assembler processes only the statements following the **ELSE** directive.

For example, you may want to assemble a line of code only if your program defines a particular variable. In this example,

```
IFDEF   buffer
buff    BYTE    buffer DUP(?)
ENDIF
```

the assembler allocates `buff` only if `buffer` has been previously defined.

MASM 6.1 provides the directives **IF1**, **IF2**, **ELSEIF1**, and **ELSIF2** to grant assembly only on pass one or pass two. To use these directives, you must either enable 5.1 compatibility (with the /Zm command-line switch or **OPTION M510**) or set **OPTION SETIF2:TRUE**, as described in the previous section.

The following list summarizes the conditional-assembly directives:

The Directive	Grants Assembly If
IF *expression*	*expression* is true (nonzero)
IFE *expression*	*expression* is false (zero)
IFDEF *name*	*name* has been previously defined
IFNDEF *name*	*name* has not been previously defined
IFB *argument**	*argument* is blank
IFNB *argument**	*argument* is not blank
IFIDN[I] *arg1*, *arg2**	*arg1* equals *arg2*
IFDIF[I] *arg1*, *arg2**	*arg1* does not equal *arg2*
	The optional **I** suffix (**IFIDNI** and **IFDIFI**) makes comparisons insensitive to differences in case.

* Used only in macros.

Conditional-Error Directives

You can use conditional-error directives to debug programs and check for assembly-time errors. By inserting a conditional-error directive at a key point in your code, you can test assembly-time conditions at that point. You can also use conditional-error directives to test for boundary conditions in macros.

Like other severe errors, those generated by conditional-error directives cause the assembler to return a nonzero exit code. If MASM encounters a severe error during assembly, it does not generate the object module.

For example, the **.ERRNDEF** directive produces an error if the program has not defined a given label. In the following example, **.ERRNDEF** makes sure a label called `publevel` actually exists.

```
.ERRNDEF    publevel
IF          publevel LE 2
PUBLIC      var1, var2
ELSE
PUBLIC      var1, var2, var3
ENDIF
```

The conditional-error directives use the syntax given in the previous section. The following list summarizes the conditional-error directives. Note their close correspondence with the previous list of conditional-assembly directives.

The Directive	Generates an Error
.ERR	Unconditionally where it occurs in the source file. Usually placed within a conditional-assembly block.
.ERRE *expression*	If *expression* is false (zero).
.ERRNZ *expression*	If *expression* is true (nonzero).
.ERRDEF *name*	If *name* has been defined.
.ERRNDEF *name*	If *name* has not been defined.
.ERRB *argument**	If *argument* is blank.
.ERRNB *argument**	If *argument* is not blank.
.ERRIDN[I] *arg1, arg2**	If *arg1* equals *arg2*.
.ERRDIF[I] *arg1, arg2**	If *arg1* does not equal *arg2*.
	The optional **I** suffix (**.ERRIDNI** and **.ERRDIFI**) makes comparisons insensitive to case.

* Used only in macros

Two special conditional-error directives, **.ERR1** and **.ERR2**, generate an error only on pass one or pass two. To use these directives, you must either enable 5.1 compatibility (with the /Zm command-line switch or **OPTION M510**) or set **OPTION SETIF2:TRUE**, as described in the previous section.