Microsoft® Macro Assembler

for the MS-DOS® Operating System

Programmer's Guide

Microsoft Corporation

Pre-release

If you have comments about the software, complete the Software Problem Report at the back of this manual and return it to Microsoft Corporation.

If you have comments about the software documentation, complete the Documentation Feedback reply card at the back of this manual and return it to Microsoft Corporation.

Document No. 510830012-100-O00-0587

Contents

Tables

Introduction

Welcome to the Microsoft® Macro Assembler. This package provides all the tools you need to create assembly-language programs.

The Macro Assembler (**MASM**) provides a logical programming syntax suited to the segmented architecture of the 8086, 8088, 80186, 80188, 80286, and 80386 microprocessors, and the 8087, 80287, and 80387 math coprocessors.

The assembler produces relocatable object modules from assembly-language source files. These object modules can be linked, using **LINK**, the Microsoft 8086 Overlay Linker, to create executable programs for the MS-DOS® operating system. Object modules created with **MASM** are compatible with many high-level-language object modules, including those created with the Microsoft C, FORTRAN, BASIC, and Pascal compilers.

MASM has a variety of features that make source development easier. It has a full set of macro directives, it allows selective assembly of portions of a source file, and it supports a wide range of operators for creating complex asssembly-time expressions. **MASM** carries out strict syntax checking of all instruction statements, including strong typing for memory operands.

New Features

This version of the assembler adds the following major new features:

- All instructions and addressing modes of the 80386 processor are now supported.
- The new CodeView debugger allows source-level debugging on assembly-language files. Multiple windows, mouse and keyboard versions of all commands, and variable watch statements are among the many CodeView features
- New segment directives allow simplified segment definitions for programmers who are linking with Microsoft high-level languages or who want to follow Microsoft segment conventions.

- Error messages have been clarified and enhanced.

In addition to these major features, there are numerous minor enhancements. If you are updating from a previous version of the Microsoft Macro Assembler, you may want to start by reading Version 5.0, and discusses compatibility issues.

System Requirements

In addition to a computer with one of the 8086-family processors, you must have Version 2.0 or later of the MS-DOS or PC-DOS operating system. Since these two operating systems are essentially the same, this manual uses the term DOS to include both. To run the assembler itself, your computer system must have about 192K of memory. However, the CodeView debugger requires at least 320K.

Although it is possible to operate the Macro Assembler with one double-sided floppy-disk drive, two floppy-disk drives are recommended as a minimum configuration. One floppy-disk drive and a hard disk make a more convenient development environment.

About This Manual and Other Assembler Documentation

This manual is intended as a reference manual for writing applications programs in assembly language. It is not intended as a tutorial for beginners, nor does it discusss systems programming or advanced techniques.

The topics that an assembly-language programmer might need to understand are listed below with references to where they are documented.

For Information on:	**See:**
How to set up the assembler software	Chapter 1, “Getting Started.” tells how to set up the assembler and utility software.

An overview of the program development process	Chapter 1, "Getting Started." describes the program development process and gives brief examples of each step.
How to use the assembler and the other programs provided with the Microsoft Macro Assembler package	Part 1, "Using Assembler Programs," describes the command lines, options, and output of **MASM** and **CREF**. Microsoft CodeView and Utilities manual describes the command lines, options, commands, and output of the CodeView debuggerm **LINK**, **LIB**, **MAKE**, and other utilities. Error messages are described in Appendix B of the respective manuals. The command-line syntax for all assembler programs is summarized in the *Microsoft Macro Assembler Reference.*
An overview of the format for assembly language source code	Chapter 1, "Getting Started," shows examples of assembly-language source files, while Chapter 4, "Writing Source Code," discusses basic concepts in a reference format.
How to program in the version of assembly language recognized by **MASM**	Part 2, "Using Directives to Control Assembly-Time Processing," explains the directives, operands, operators, expressions, and other language features understood by **MASM**. However, the manual is not designed to teach novice users how to program in assembly language. If you are new to assembly language, you will still need additional books or courses. Some tutorial books that may be helpful are listed in later this introduction.
An overview of the architecture of 8086-family processors	Chapter 13, "Understanding 8086-Family Processors," discusses the segments, memory use, registers, and other basic features of 8086-family processors.
How to use the instruction sets for the 8086/80186/80286/80386 microprocessors	

Part 3, "Using Instructions to Control Run-Time Processing," describes each of the instructions. The material is intended as a reference, not a tutorial. Beginners may need to study other books on assembly language.

Reference data on instructions	The *Microsoft Macro Assembler Reference* lists each instruction alphabetically and gives data on encoding and timing for each. This manual is particularly useful for programmers who wish to optimize assembly code.
How to use the instruction sets of the 8087/80287/80387 math coprocessors	Chapter 19, "Calculating with a Math Coprocessor," describes the coprocessor instructions and tells how to use the most important ones.
Information on DOS structure and fuctuin calls	This information may be useful to many programmers. However, it is beyond the scope of the documentation provided with the Microsoft Macro Assembler package. You can find information on DOS in the *Microsoft MS-DOS Programmer's Reference* and in many other books about DOS. Some of the books listed later in this introduction cover these topics.
Hardware features of your computer	For some assembly-language tasks, you may need to know about the basic input and output systems (BIOS) or other hardware features of the computers that will run your programs. Consult the technical reference manuals for your computer or one of the many books that describe hardware features. Some of the books listed in later in this introduction discuss hardware features of IBM and IBM-compatible computers.

IBM Compilers and Assemblers

Many IBM® languages are produced for IBM by Microsoft. IBM languages that are similar to corresponding Microsoft languages include:

IBM Personal Computer Macro Assembler, Versions 1.0 and 2.0

IBM Personal Computer FORTRAN, Version 3.*x*

IBM Personal Computer C, Version 1.0

IBM Personal Computer Pascal, Versions 1.0 to 3.*x*

IBM Personal Computer BASIC Compiler, Versions 1.0 and 2.0.

These languages are compatible with the Microsoft Macro Assembler Version 5.0 except as noted in Appendix B.

Books on Assembly Language

The following books may be useful in helping you learn how to program in assembly language:

Lafore, Robert, *Assembly Language Primer for the IBM PC & XT*. New York: Plume/Waite, 1984.

An introduction to assembly language, including some information on DOS function calls and IBM-type BIOS

Metcalf, Christopher D., and Sugiyama, Marc B., *COMPUTE!'s Beginner's Guide to Machine Language on the IBM PC & PCjr*. Greensboro, NC: COMPUTE! Publications, Inc., 1985.

Beginning discussion of assembly language, including information the instruction set and MS-DOS function calls

Microsoft MS-DOS Programmer's Reference Manual. Bellevue, WA: Microsoft Corporation.

Reference manual for MS-DOS

Morgan, Christopher and the Waite Group, *Bluebook of Assembly Routines for the IBM PC*. New York: New American Library, 1984.

Sample assembly routines that can be integrated into assembly or high-level-language programs

Norton, Peter, *The Peter Norton Programmer's Guide to the IBM PC*. Bellevue, WA: Microsoft Press, 1985.

Information on using IBM-type BIOS and MS-DOS function calls

Scanlon, Leo J., *IBM PC Assembly Language: A Guide for Programmers*. Bovie, MD: Robert J. Brady Co., 1983.

An introduction to assembly language, including information on DOS function calls

Schneider, Al, *Fundamentals of IBM PC Assembly Language*. Blue Ridge Summit, PA: Tab Books Inc., 1984.

An introduction to assembly language, including information on DOS function calls

iAPX 386 Programmer's Reference Manual. Santa Clara, CA: Intel Corporation, 1986.

Reference manual for 80386 processor and instruction set (manuals for previous processors are also available)

These books are listed for your convenience only. Microsoft Corporation does not endorse these books (with the exception of those published by Microsoft) or recommend them over others on the same subjects.

Notational Conventions

This manual uses the notation described in the following list:

Example of Convention	**Description of Convention**
`Examples`	The typeface shown in the left column is used to simulate the appearance of information that would be printed on your screen or by your printer. For example, the following

source line is printed in this special typeface:

```
        mov     ax,WORD PTR string[3]
```

When discussing this source line in text, items appearing on the line, such as `string[3]`, also appear in the special typeface.

KEYWORDS and **symbols**

Bold letters indicate command line options, assembly-language keywords or symbols, and the names of files that come with the Microsoft Macro Assembler package.

For example, the directive **ORG**, the instruction **mov**, the register **AX**, the option **/ZI**, and the file name **MASM** are always shown in bold when they appear in text or in syntax displays (but not in examples).

In syntax displays, bold type indicates any words, punctuation, or symbols (such as commas, parentheses, semicolons, hyphens, equal signs, and operators) that you must type exactly as shown.

For example, the syntax of the **IFDIF** directive is shown as:

IFDIF <*argument1***>,<***argument2***>**

The word **IFDIF**, the angle brackets, and the comma are all shown in bold. Therefore they must be typed exactly as shown.

placeholders

Words in italics are placeholders for variable information that you must supply. A file name is an example of this kind of information.

For example, the syntax of the **OFFSET** operator is shown below:

OFFSET *expression*

This indicates that any *expression* may be supplied following the **OFFSET** operator. When writing source code to match this syntax, you might type

```
OFFSET here+6
```

where `here+6` is the expression. The placeholder is shown in italics both in syntax displays and in descriptions explaining syntax displays.

⟦*optional items*⟧

Double brackets surround optional syntax elements. For example, the syntax of the index operator is shown as:

⟦*expression1*⟧[*expression2*]

This indicates that *expression1* is optional, since it is contained in double brackets, but *expression2* is required and it must be enclosed in brackets.

When writing code to match this syntax, you might type `[bx]`, leaving off the optional *expression1*, or you might type `test[5]`, using `test` as *expression1*.

{*choice1* | *choice2*}

Braces and vertical bars indicate that you have a choice between two or more items. Braces enclose the choices, and vertical bars separate the choices. You must choose one of the items.

For example, the **/W** (warning-level) option has the following syntax:

/W{0 | 1 | 2}

You can type `/W0`, `/W1`, or `/W2` to indicate the desired level of warning. However, typing `/W3` is illegal since 3 is not one of the choices enclosed in braces.

Repeating elements...

Three dots following an item indicate that more items having the same form may be entered.

For example, the syntax of the **PUBLIC** directive is shown below:

PUBLIC *name* ⟦**,***name*...⟧

The dots following the second name indicate that you can enter as many *names* as you like as long as each is preceded by a comma.

However, since the first *name* is not in brackets, you must enter at least one *name*.

```
Program
   .
   .
   .
Fragment
```

A column of dots in syntax lines and program examples shows that a portion of the program has been omitted.

For instance, in the following program fragment, only the opening lines and the closing lines of a macro are shown. The internal lines are omitted since they are not relevant to the concept being illustrated.

```
work  MACRO   realarg,testarg
      .ERRB  <realarg>  ;; Too few
      .ERRNB <testarg>  ;; Too many
      .                 ;; Just right
      .
      .
      ENDM
```

Defined terms and "Prompts"

Quotation marks set off terms defined in the text. For example, the term "bit splicing" appears in quotation marks the first time it is defined.

Quotation marks also set off command-line prompts in text. For example, **LINK** prompts you for the names of object files; this prompt is called the "Object Modules" prompt.

KEY NAMES

Small capital letters are used for the names of keys and key sequences, which you must press. Examples include ENTER and CONTROL-C.

■ Example

The following example shows how this manual's notational conventions are used to indicate the syntax of the **MASM** command line:

MASM ⟦*options*⟧ *sourcefile* ⟦,⟦*objectfile*⟧ ⟦,⟦*listingfile*⟧ ⟦,⟦*crossreferencefile*⟧⟧⟧⟧ ⟦;⟧

This syntax shows that you must first type the program name, **MASM**. You can then enter any number of *options*. You must enter a *sourcefile*.

You can enter an *objectfile* preceded by a comma. You can enter a *listingfile*, but if you do, you precede it with the commas associated with the *sourcefile* and *objectfile*. Similarly, you can enter a *crossreferencefile*, but if you do, you must precede it with the commas associated with the other files. You can also enter a semicolon at any point after the *sourcefile*.

For example, any of the following command lines would be legal:

```
MASM test.asm;
MASM /ZI test.asm;
MASM test.asm,,test.lst;
MASM test.asm,,,test.crf
MASM test.asm,test.obj,test.lst,test.crf
MASM test.asm,,,;
```

Getting Assistance or Reporting Problems

If you need help or you feel you have discovered a problem in the software, please provide the following information to help us locate the problem:

- The assembler version number (from the logo that is printed when you invoke the assembler with **MASM**)
- The version of DOS you are running (use the DOS **VER** command)
- Your system configuration (type of machine you are using and its total memory, total free memory at assembler execution time, as well as any other information you think might be useful)
- The assembly command line used (or the link command line if the problem occurred during linking)
- If the problem occurred at link time, any object files or libraries you linked with

If your program is very large, please try to reduce its size to the smallest possible program that still produces the problem.

Use the Software Problem Report form at the back of this manual to send this information to Microsoft.

If you have comments or suggestions regarding any of the manuals accompanying this product, please indicate them on the Documentation Feedback Card at the back of this manual.

Using Assembler Programs

Chapter 1
Getting Started

This chapter tells how to set up Microsoft Macro Assembler files and start writing assembly language programs. It gives an overview of the development process, and shows examples using simple programs. It also refers you to the chapters where you can learn more about each subject.

1.1 Setting Up Your System

After opening the Microsoft Macro Assembler package, you should take the following setup steps before you begin developing assembler programs:

1. Make backup copies of the disks in the assembler package.
2. Decide on a configuration strategy.
3. Copy the assembler files to the appropriate disks and directories.
4. Set environment variables.

1.1.1 Making Backup Copies

You should make backup copies of the assembler disks before attempting to use any of the programs in the package. Put the copies in a safe place and use them only to restore the originals if they are damaged or destroyed.

All the files on the disks are listed in the file **PACKING.LST** on Disk 1.

The files on the disk are not copy protected. You may make as many copies as you like for your own use. You may not distribute any executable, object, or library files on the disk. The sample programs are in the public domain.

No license is required to distribute executable files created with the assembler.

You should also fill out and return the owner registration card if you wish to be informed of updates and other information about the assembler.

1.1.2 Choosing a Configuration Strategy

There are several kinds of files on the distribution disk. You can arrange these files in a variety of ways. The two most important factors in your decision are whether you have a hard disk or floppy disks, and whether you want to use environment variables.

Program development can be affected by the environment variables described below:

Variable	Description
PATH	Determines the order in which DOS will search for executable files. A common setup with language products is to place executable files in the directory **\BIN** and include this directory in the **PATH** environment string.
LIB	Specifies the directory where **LINK** will look for library and object files. A common setup with language products is to put library and object files in directory **\LIB** and include this directory in the **LIB** environment string.
INCLUDE	Specifies the directory where **MASM** will look for include files. A common setup with language products is to put macro files and other include files in directory **\INCLUDE** and include this directory in the **INCLUDE** environment string.
MASM	Specifies default options that **MASM** will use on start-up.
LINK	Specifies default options that **LINK** will use on start-up.

If you have a hard disk, you will probably want to use environment variables to specify locations for library, macro, and executable files. If you have floppy disks, you may prefer to leave all files in the root directory.

If you already have other language products on a hard disk, you should consider how your assembler setup interacts with your other languages. Some users may prefer to have separate directories for library and include files for each language. Others may prefer to have all library and include files in the same directories. If you want all language files in the same directories, make sure you do not have any files with the same names as the ones provided with the Microsoft Macro Assembler.

If you have floppy disks, you will not be able to get all the tools you need for assembly language development on one disk. A typical setup is shown below:

Disk	Files
1	Source, object, library, and macro files on Disk 1 with source and working object files in the root directory, library and standard object files in directory **\LIB**, and macro files in directory **\INCLUDE**.
2	Executable files for developing programs on Disk 2. This could include **MASM**, **LINK**, a text editor, and possibly **MAKE**, **LIB**, or **CREF**. These files may not all fit on a standard 360K disk, so you will have to decide which are most important for you.
3	The CodeView debugger and any additional utilities on Disk 3.

With this setup, you could keep Disk 1 in drive A. Then swap disks 2 and 3 depending on whether you are developing programs or debugging.

1.1.3 Copying Files

A setup batch file called **SETUP.BAT** is provided on Disk 1. You can run it to automatically copy the assembler files to your work disk. The setup program will ask for information about your system and how you want to set it up. Before copying anything to your system, it tells you what it is about to do and gets your confirmation.

If you prefer, you can ignore the setup program and copy the files yourself. See the **PACKING.LST** file for a list of files.

Warning

> If you have previous versions of the assembler or other programs such as **LINK**, **LIB**, or **MAKE**, you may want to make backup copies or rename the old files so that you do not overwrite them with the new versions.

1.1.4 Setting Environment Variables

If you wish to use environment variables to establish default file locations and options, you will probably want to set the environment variables in your **AUTOEXEC.BAT** file. The setup program does not attempt to set any environment variables, so you must modify any batch files yourself.

The following lines could be added for a typical hard-disk setup:

```
PATH C:\BIN
SET LIB=C:\LIB
SET INCLUDE=C:\INCLUDE
SET MASM=/ZI
SET LINK=/CO
```

The following lines might be used for the floppy-disk setup described in Section 1.1.2.

```
PATH B:\;A:\
SET LIB=A:\LIB
SET INCLUDE=A:\INCLUDE
SET MASM=/ZI
SET LINK=/CO
```

1.2 Choosing a Program Type

MASM can be used to create different kinds of program files. The source code format is different for each kind of program. The primary formats are described below:

Type	Description
.EXE	The **.EXE** format is the most common format for programs that will execute under DOS. In future versions of DOS, a similar **.EXE** format will be the only format available for stand-alone programs that take advantage of multitasking. Programs in the **.EXE** format can have multiple segments and can be of any size. Modules can be created and linked using either the assembler or most high-level language compilers, including all the Microsoft compilers. Modules created in different languages can be combined into a single program. This is the format recommended by Microsoft for programs of significant size and purpose. The source format for

creating this kind of program is described and illustrated throughout the rest of the manual.

.COM The **.COM** format is sometimes convenient for small programs. Programs in this format are limited to one segment. They can be no larger than 64K (unless they use overlays). They have no file header and are thus smaller than comparable **.EXE** files. This makes them a good choice for small stand-alone assembler programs of several thousand bytes or less. One disadvantage of the **.COM** format is that executable files cannot contain symbolic and source line information for the CodeView debugger. You can only debug them in assembly mode. The source format for **.COM** programs is illustrated briefly in this chapter and described fully in the *Microsoft MS-DOS Programmer's Reference Guide.*

Binary Files Binary files are used for procedures that will be called by the Microsoft and IBM BASIC interpreters. They are also used by some non-Microsoft compilers. See the manual for the language you are using for details on preparing source files.

Device Drivers Device drivers that set up and control I/O for hardware devices can be developed with the assembler. The source format is described in the *Microsoft MS-DOS Programmer's Reference Guide.*

Code for ROMs The assembler can be used to prepare code that is downloaded to programmable ROM chips. The format is usually a binary format. Methods of preparing and using this code vary depending on hardware.

1.3 The Program-Development Cycle

The program-development cycle for assembly language is illustrated in Figure 2.1. The specific steps for developing a stand-alone assembler program are listed below:

1. Use a text editor to create or modify assembly-language source modules. By convention, source modules are given the extension **.ASM**. For most programs, source modules can be organized in a variety of ways. For example, you can put all the procedures for a program into one large module, or you can split the procedures between several modules. If your program will be linked with high-level language modules, the source code for these modules is also

prepared at this point.

2. Use **MASM** to assemble each of the modules for the program. If assembly errors are encountered in a module, you must go back to Step 1 and correct the errors before continuing. For each source (**.ASM**) file, MASM creates an object file with the default extension **.OBJ**. Optional listing (**.LST**) and cross-reference (**.CRF**) files can also be created during assembly. If your program will be linked with high-level-language modules, the source modules are compiled to object files at this point.

3. You can use **LIB** to gather multiple object files (**.OBJ**) into a single library file having the default extension **.LIB**. This step is optional. It is generally used for object files that will be linked with several different programs. An optional library list file can also be created with **LIB**.

4. Use **LINK** to link all the object files and library modules that will make up a program. **LINK** creates a single executable file with the default extension **.EXE**. An optional map (**.MAP**) file can also be created with **LINK**.

5. If the executable file must be converted to a binary format, use **EXE2BIN** to do the conversion. Skip this step for programs in the **.EXE** format. It is necessary for programs in the **.COM** format and for binary files that will be read into an interpreter or compiler.

6. Debug your program to discover logical errors. Debugging may involve several techniques, including the following:

   - Run the program and study its input and output.
   - Study source and listing files.
   - Use **CREF** to create a cross-reference-listing (.REF) file.
   - Use CodeView (**CV**) to study the program during step-by-step execution.

   If logical errors are discovered, you must return to Step 1 to correct the source code.

All or part of the program-development cycle can be automated by using **MAKE** with make description files. **MAKE** is most useful for developing complex programs involving numerous source modules. Ordinary DOS batch files may be more efficient for developing single-module programs.

1.4 Developing Programs

The next sections take you through the steps in developing programs. Examples of simple programs and command lines are shown for each step. The chapters where you can learn more on each topic are cross-referenced.

1.4.1 Writing and Editing Assembly-Language Source Code

Assembly-language programs are created from one or more source files. Source files are text files that contain statements defining the program's data and instructions.

To create assembly-language source files, you need a text editor capable of producing ASCII (American Standard Code for Information Interchange) files. Each line must be separated by a carriage-return–line-feed combination.

If your text editor has a programming or nondocument mode for producing ASCII files, use that mode. You cannot prepare source files with text editors that insert control codes or use other special formats.

The following examples illustrates source code that produces a stand-alone executable programs. Example 1 creates a program in the **.EXE** format while Example 2 creates the same program in the **.COM** format.

If you are a beginner to assembly language, you can start experimenting by copying these programs. Use the segment shell of the programs, but insert your own data and code.

■ Example 1

```
            TITLE   hello

            DOSSEG                      ; Use Microsoft segment conventions
            .MODEL  SMALL               ;   conventions and small model (1)

            .STACK  100h                ; Allocate 256-byte stack (2)

            .DATA
message     DB      "Hello, world.",13,10  ; Message to be written (3)
lmessage    EQU     $ - message            ; Length of message

            .CODE
start: (4)  mov     ax,DGROUP           ; Load segment location (5)
```

```
        mov     ds,ax                 ;   into DS register

        mov     bx,1                  ; Load 1 - file handle for (6)
                                      ;   standard output
        mov     cx,lmessage           ; Load length of message
        mov     dx,OFFSET message     ; Load address of message
        mov     ah,40h                ; Load number for DOS Write function
        int     21h                   ; Call DOS

        mov     ax,4C00h              ; Load DOS Exit function (4Ch) (7)
                                      ;   in AH and 0 errorlevel in AL
        int     21h                   ; Call DOS

        END     start (4)
```

Note the following points about the source file:

1. The **.MODEL** and **DOSSEG** directives tells **MASM** that you intend to use the Microsoft order and name conventions for segments. These statements automatically define the segments in the correct order and specify **ASSUME** and **GROUP** statements. You can then place segments in your source file in whatever order you find convenient using the **.STACK**, **.DATA**, **.CODE**, and other segment directives. These simplified segment directives are a new feature of Version 5.0. They are optional; you can still define the segments completely using the directives required by earlier versions of **MASM**. The simplified segment directives and the Microsoft naming conventions are explained in Section 5.1.

2. A stack of 256 (100 hexadecimal) bytes is defined using the **.STACK** directive. This is an adequate size for most small programs. Programs with many nested procedures may require a larger stack. See Sections 5.1.4 and 5.2.2 for more information on defining a stack.

3. The **.DATA** directive marks the start of the data segment. A string variable and its length are defined in this segment.

4. The instruction label `start` in the code segment follows the **.CODE** directive and marks the start of the program instructions. The same label is used after the **END** statement to define the starting point where program execution will start. See Sections 4.7.2 and 5.5.1 for more information on using the **END** statement and defining the execution starting point.

5. The first two code instructions load the address of the data segment into the **DS** register. With small-model programs, the data segment is actually a group called **DGROUP**. The **DS** register must always be initialized for source files in the **.EXE** format. No comparable instructions are required for the code and stack segments, since they are initialized automatically. Section 5.5 tells

how each segment is initialized.

6. The string variable defined earlier is displayed using DOS function 40h. File handle 1 (the predefined handle for standard output) is specified to display to the screen. Strings can also be displayed using function 09h. See the *Microsoft MS-DOS Programmer's Reference* or other DOS reference books for more information on DOS calls.

7. DOS function 4C hexadecimal is used to terminate the program. While there are other techniques for returning to DOS, this is the one recommended by Microsoft.

The following example shows source code that can be used to create the same program shown earlier, but in the **.COM** format:

■ Example 2

```
            TITLE    hello

_TEXT       SEGMENT                     ; Define code segment (1)
            ASSUME   cs:_TEXT,ds:_TEXT,ss:_TEXT   (2)
            ORG      100h               ; Set location counter to 256 (3)

start:      jmp      again              ; Jump over data (4)

message     DB       13,"Hello, world.",13,10     ; Message to be written
lmessage    EQU      $ - message                  ; Length of message

begin:      mov      bx,1               ; Load 1 - file handle for
                                        ;   standard output
            mov      cx,lmessage        ; Load length of message
            mov      dx,OFFSET message  ; Load address of message
            mov      ah,40h             ; Load number for DOS Write function
            int      21h                ; Call DOS

            mov      ax,4C00h           ; Load DOS Exit function (4Ch)
                                        ;   in AH and 0 errorlevel in AL
            int      21h                ; Call DOS
                                        ; Data could be placed here (4)
_TEXT       ENDS
            END      start
```

Note the following points in which **.COM** programs differ from **.EXE** programs:

1. The **.MODEL** directive cannot be used to define default segments for **.COM** files. However, segment definition is easy, since only one segment can be used. The align, combine, and class types need not be given, since they make no difference for **.COM** files.

2. All segment registers are initialized to the same segment using the **ASSUME** directive. This tells the assembler which segment to associate with each segment register. See Section 5.4 for more information on the **ASSUME** directive.

3. The **ORG** directive must be used to start assembly at byte 256 (100 hexadecimal). This leaves room for the DOS Program Segment Prefix (PSP), which is automatically loaded into memory at run time. See Section 6.3 for information on how the **ORG** directive changes the location counter.

4. Although data may be included in the segment, it must not be executed. You can use the **JMP** instruction to skip over data (as shown in the example) or you can put the data at the end after the program returns to DOS.

1.4.2 Assembling Source Files

Source modules are assembled witH **MASM**. The **MASM** command-line syntax is shown below:

MASM ⟦*options*⟧ *sourcefile* ⟦,⟦*objectfile*⟧ ⟦,⟦*listingfile*⟧ ⟦,⟦*crossreferencefile*⟧⟧⟧⟧ ⟦;⟧

Assume you had an assembly source file called `hello.asm`. For the fastest possible assembly, you could start **MASM** with the following command line:

```
MASM hello;
```

The output would be an object file called `hello.obj`. To assemble the same source file with the maximum amount of debugging information, use the following command line:

```
MASM /V /Z /ZI hello,,,;
```

The **/V** and **/Z** options instruct **MASM** to send additional statistics and error information to the screen during assembly. The **/ZI** option instructs **MASM** to include debugging information in the object file. The output of this command is three files: the object file `hello.obj`, the assembly listing file `hello.lst`, and the cross-reference file `hello.crf`.

Chapter 2, "Using MASM," describes the **MASM** command line, options, and listing format in more detail.

1.4.3 Converting Cross-Reference Files

Cross-reference files produced by **MASM** are in a binary format and must be converted using **CREF**. The command-line syntax is shown below:

CREF *crossreferencefile* ⟦,*crossreferencelisting*⟧ ⟦;⟧

To convert the cross-reference file `hello.crf` into an ASCII file that cross references symbols used in `hello.asm`, use the following command line:

```
CREF hello;
```

The output file is called `hello.ref`.

The **CREF** command line and listing format are described in Chapter 3, "Using CREF."

1.4.4 Creating Library Files

Object files created with **MASM** or with Microsoft high-level-language compilers can be converted to library files using **LIB**. The command-line syntax is shown below:

LIB *oldlibrary* ⟦**/PAGESIZE:***number*⟧ ⟦*commands*⟧ ⟦,⟦*listfile*⟧ ⟦,⟦*newlibrary*⟧⟧⟧ ⟦;⟧

For example, assume you had used **MASM** to assemble two source files containing graphics procedures and you want to be able to call these procedures from several different programs. The object files containing these procedures are called `dots.obj` and `lines.obj`.

You could combine them into a file called `graphics.lib` using the following command line:

```
LIB graphics +dots +lines;
```

If you later wanted to add another object file called `circles.obj` and at the same time get a listing of the procedures in the library, you could use the following command line:

```
LIB graphics +circles,graphics.lbl
```

The **LIB** command line, commands, and listing format are explained in the Chapter 3, "Using CREF.

1.4.5 Linking Object Files

Object files are linked into executable files using **LINK**. The **LINK** command-line syntax is shown below:

LINK ⟦*options*⟧ *objectfiles* ⟦,⟦*executablefile*⟧ ⟦,⟦*mapfile*⟧ ⟦,⟦*libraryfiles*⟧⟧⟧⟧ ⟦;⟧

Assume you want to create an executable file from the single module `hello.obj`. The source file was written for the **.EXE** format (see Section 1.4.1) and was assembled using the **/ZI** option. You plan to debug the program with the CodeView debugger. Use the following command line:

```
LINK /CO hello;
```

The output file is `hello.exe`. It contains symbolic and line-number information for the debugger. The file can now be run from the DOS command line or from within the CodeView debugger.

After you have debugged the program, you will probably want to create a final version with no symbolic information. Use the following command line:

```
LINK hello;
```

This command line could also be used if the source file was prepared in the **.COM** format. However, in this case the output file `hello.exe` could not be run. Another step is required, as described in Section 1.4.6.

Now assume that you want to create a large program called `picture.exe` that has two object files (`picture` and `picture2`) and calls external procedures from the library file described in Section 1.4.4. Use the following command line:

```
LINK /CO picture picture2,,,graphics;
```

The library file `graphics.lib` would need to be in the current directory or in the directory described by the **LIB** environment variable. The procedure calls would have to be declared external in the source file, as described in Section 8.2.

The **LINK** options, command line, and listing format are described in the Microsoft CodeView and Utilities manual.

1.4.6 Converting to .COM Format

Source files prepared in the **.COM** format require an additional conversion step after linking. The program that does the conversion is called **EXE2BIN**. It is not included in the Macro Assembler package, but it does come with the MS-DOS and PC-DOS operating systems. The syntax is shown below:

EXE2BIN *exefile* ⟦*binaryfile*⟧

To convert a file called `hello.exe` to an executable file called `hello.com`, use the following command line:

```
EXE2BIN hello hello.com
```

Note that you must specify the extension **.COM**, since **BIN** is the default extension. The **.EXE** file must have been prepared from source and object files in the valid **.COM** format.

EXE2BIN can also be used to prepare binary files for use with the Microsoft or IBM BASIC interpreters. See the BASIC interpreter manual.

1.4.7 Debugging

The CodeView debugger is usually the most efficient tool for debugging assembler programs. The command-line syntax is shown below:

CV ⟦*options*⟧ *executablefile* ⟦*arguments*⟧

To debug a program called `hello.exe` using an IBM Personal Computer, use the following command line:

```
CV hello
```

Additional options may be required for other computers. Graphics programs always require the **/S** option. For example, to debug a grapics program called `circles.com` on an IBM-compatible computer, use the following command line:

```
CV /W/I/S circles.com
```

The **/W** and **/I** options tell the debugger to use IBM-compatible features. Note that the **.COM** extension must be specified, since the debugger assumes files without extensions are **.EXE** files.

For information about CodeView command lines, options, and commands, see the *Microsoft CodeView and Utilities Guide.*

Chapter 2
Using MASM

The Microsoft Macro Assembler (**MASM**) assembles 8086, 80186, 80286, and 80386 assembly-language source files and creates relocatable object files. Object files can then be linked to form an executable files.

This chapter tells how to run **MASM**, explains the options and environment variables that control its behavior, and describes the format of the assembly listings it generates.

2.1 Running the Assembler

You can assemble source files with **MASM** using two different methods: by giving a command line at the DOS prompt or by by responding to a series of prompts.

Once you have started **MASM**, it attempts to process the source file you specified. If errors are encountered, they are output to the screen and **MASM** terminates. If no errors are encountered, **MASM** outputs an object file. It can also output listing and cross-reference files if they are specified. You can terminate **MASM** at any time by pressing CONTROL-C or CONTROL-BREAK.

2.1.1 Assembly Using a Command Line

You can assemble a program source file by typing the **MASM** command name and the names of the files you wish to process. The command line has the following form:

MASM ⟦*options*⟧ *sourcefile* ⟦,⟦*objectfile*⟧ ⟦,⟦*listingfile*⟧ ⟦,⟦*crossreferencefile*⟧⟧⟧⟧ ⟦;⟧

The *options* can be any combination of the assembler options described in Section 2.4. The option letter or letters must be preceded by a forward slash (/) or a dash (–). Examples in this manual use a forward slash. The forward slash and dash characters cannot be mixed in the same command line. Although shown at the beginning in the syntax, they may actually be placed anywhere on the command line. An option affects all relevant files in the command line even if the option appears at the end of the line.

The *sourcefile* must be the name of the source file to be assembled. If you do not supply a file-name extension, **MASM** supplies the extension .ASM.

The optional *objectfile* is the name of the file to receive the relocatable object code. If you do not supply a name, **MASM** uses the source-file name, but replaces the extension with **.OBJ**.

The optional *listingfile* is the name of the file to receive the assembly listing. The assembly listing shows the assembled code for each source statement and the names and types of symbols defined in the program. If you do not supply a file-name extension, **MASM** supplies the extension **.LST**.

The optional *crossreferencefile* is the name of the file to receive the cross-reference output. The resulting cross-reference file can be processed with **CREF**, the Microsoft Cross-Reference Utility, to create a cross-reference listing of the symbols in the program. The cross-reference listing can be used for program debugging. If you do not supply a file-name extension, **MASM** supplies **.CRF** by default.

You can use a semicolon (**;**) in the command line to select defaults for the remaining file names. A semicolon after the source-file name selects a default object-file name and suppresses creation of the assembly listing and cross-reference files. A semicolon after the object-file name suppresses just the listing and cross-reference files. A semicolon after the listing-file name suppresses only the cross-reference file.

All files created during the assembly will be written to the current drive and directory unless you specify a different drive for each file. You must separately specify the alternate drive and path for each file that you do not want to go on the current directory.

You can also specify a device name instead of a file name. For example, **NUL** for no file or **PRN** for the printer.

Note

Unless a semicolon (**;**) is used, all the commas in the command line are required. If you want the file name for a given file to be the default (the file name of the source file), place the commas that would otherwise separate the file name from the other names side by side (**,,**).

Spaces in a command line are optional. If you make an error entering any of the file names, **MASM** displays an error message and prompts for new file names, using the method described in Section 2.1.2.

■ **Examples**

```
MASM file.asm, file.obj, file.lst, file.crf
```

The example above is equivalent to the command line below:

```
MASM file,,,;
```

The source file `file.asm` is assembled. The generated relocatable code is copied to the object file `file.obj`. **MASM** also creates an assembly listing and a cross-reference file. These are written to `file.lst` and `file.crf`, respectively.

```
MASM startup,,stest;
```

The example above directs **MASM** to assemble the source file `startup.asm`. The assembler then writes the relocatable object code to the default object file, `startup.obj`. **MASM** creates a listing file named `stest.lst`, but the semicolon keeps the assembler from creating a cross-reference file.

```
MASM startup,,stest,;
```

The example above is exactly the same as the previous example except that the assembler creates a cross-reference file `startup.crf`. This is because the semicolon follows a comma marking the place of the cross-reference file instead of following the file name of the list file.

```
MASM  B:\src\build;
```

The example above directs **MASM** to find and assemble the source file `build.asm` in the directory `\src` on Drive B. The semicolon causes the assembler to create an object file named `build.obj` in the current directory, but prevents **MASM** from creating an assembly listing or cross-reference file. Note that the object file is placed on the current drive, not the drive specified for the source file.

2.1.2 Assembly Using Prompts

You can direct **MASM** to prompt you for the files it needs by starting **MASM** with just the command name. **MASM** prompts you for the input it needs by displaying the following lines, one at a time:

```
Source filename [.ASM]:
Object filename [source.OBJ]:
Source listing  [NUL.LST]:
Cross-reference [NUL.CRF]:
```

The prompts correspond to the fields of **MASM** command lines. **MASM** waits for you to respond to each prompt before printing the next one. You must type a source-file name (though the extension is optional) at the first prompt. For other prompts, you can either type a file name, or press the ENTER key to accept the default displayed in brackets after the prompt.

File names typed at prompts must follow the command-line rules described in Section 2.1.1. You can type options after any of the prompts as long as you separate them from file names with spaces. At any prompt, you can type the rest of the file names in the command-line format. For example, you can choose the default responses for all remaining prompts by typing a semicolon (;) after any prompt (as long as you have supplied a source-file name), or you can type commas (,) to indicate several files.

After you have answered the last prompt and pressed the ENTER key, **MASM** assembles the source file.

2.2 Using Environment Variables

The Macro Assembler recognizes two environment variables: **INCLUDE** and **MASM**. The next two sections describe these environment variables and their use with the assembler.

Environment variables are described in general in the DOS user's guide.

2.2.1 The INCLUDE Environment Variable

The **INCLUDE** environment variable can specify the directory where include files are stored. This makes maintenance of include files easier, particularly on a hard disk. All include files can be kept in the same directory. If you keep source files in different directories, you do not have to keep copies of include files in each directory.

The **INCLUDE** environment variable is used by **MASM** only when you give a file name as an argument to the **INCLUDE** directive (see Section 11.6.1). If you give a complete file specification, including directory or drive, **MASM** only looks for the file in the specified directory.

When a file name is specified, **MASM** looks for the include file first in any directory specified with the **/I** options (see Section 2.4.6). If the **/I** option is not used or if the file is not found, **MASM** next looks in the current directory. If the file is still not found, **MASM** looks in the directories specified with the **INCLUDE** environment variable in the order specified.

■ **Examples**

```
SET INCLUDE=C:\INCLUDE
```

This line defines the **INCLUDE** environment string to be `C:\INCLUDE`. Include files placed in this directory can be found automatically by **MASM**. You can put this line in your **AUTOEXEC.BAT** file to set the environment string each time you turn on your computer.

2.2.2 The MASM Environment Variable

The **MASM** environment variables can be used to specify default assembler options. You can define the options you use most of the time in the environment variable so that you don't need to type them on the command line every time you start **MASM**.

When you first start **MASM**, it reads the options in the environment variable first. Then it reads the options in the command line. If conflicting options are encountered, the last one read takes effect. This means that you can override default options given in the environment variable by giving conflicting options in the command line.

Many assembler options have conflicting or opposite options. Some options define the default action. If given by themselves, they have no effect, since the default action is taken anyway. However, they are useful for overriding a nondefault action specified by an option in the environment variable.

Some assembler directives have the same effect as options. They always override related options.

■ **Examples**

```
SET MASM=/A/ZI/Z
```

The command line above sets the **MASM** environment variable so that the **/A**, **ZI**, and **/Z** options are in effect. The line can be put in an **AUTOEXEC.BAT** file to automatically set these options each time you start your computer.

Assume you have set the **MASM** environment string using the line shown above and you then start **MASM** with the following command line:

```
MASM /S test;
```

The **/S** option, which specifies sequential segment ordering, conflicts with the **/A** option, which specifies alphabetical segment ordering. The command-line option overrides and the source file has sequential ordering. (See Section 5.2.1 for information on the significance of segment order.)

However, if the source file contains the **.ALPHA** directive, it overrides all options and specifies alpabetical segment order.

2.3 Controlling Message Output

During and immediately after assembly, **MASM** sends messages to the standard output device. By default, this device is the screen. However, the display can be redirected so that instead of being displayed on the screen, it goes to a file or to a device such as a printer.

The messages can include a status message for successful assembly and error messages for unsuccessful assembly. The message format and the error and warning messages are described in Appendix B, "Error Messages and Exit Codes."

Some text editing programs can use error information to locate errors in the source file. Typically, **MASM** is run as a shell from the editor and the assembler output is redirected into a file. The editor then opens the file and uses the data in it to locate errors in the source code. The errors may be located by line number, or by searching for the text of the error line.

If your text editor does not support this capability directly, you may still be able to use keystroke macros to set up similar functions. This requires either an editor that supports keystroke macros, or a keyboard enhancer such as Prokey or Superkey.

■ Example

```
MASM file; > errors
```

This command line sends to the file `errors` all messages that would normally be sent to the screen.

2.4 Using MASM Options

The **MASM** options control the operation of the assembler and the format of the output files it generates.

MASM has the following options:

Option	Action
/A	Writes segments in alphabetical order
/B<number>	Sets buffer size
/C	Specifies a cross-reference file
/D	Creates Pass 1 listing
/D*symbol*[[=*value*]]	Defines assembler symbol
/E	Creates code for emulated floating-point instructions
/H	Lists command-line syntax and all assembler options
/I*path*	Sets include-file search path
/L	Specifies an assembly-listing file
/ML	Preserves case sensitivity in names

/MX	Preserves case sensitivity in public and external names
/MU	Converts names to uppercase
/N	Suppresses tables in listing file
/P	Checks for impure code
/R	Creates code for real floating-point instructions
/S	Writes segments in source-code order
/T	Suppresses messages for successful assembly
/V	Displays extra statistics to screen
/W{**1** | **2** | **3**}	Sets error display level
/X	Includes false conditionals in listings
/Z	Displays error lines on screen
/ZI	Puts symbolic and line number information in the object file
/ZD	Puts line number information in the object file

2.4.1 Specifying the Segment Order Method

■ **Syntax**

/A
/S

The **/A** option directs **MASM** to place the assembled segments in alphabetical order before copying them to the object file. The **/S** option directs the assembler to write segments in the order in which they appear in the source code.

Source code order is the default, so if no option is given, **MASM** copies the segments in the order encountered in the source file. The **/S** option is provided for compatibility with XENIX® and for overriding a default option in the **MASM** environment variable.

Note

Some previous versions of the macro assembler ordered segments alphabetically by default. Listings in books and magazines may be written with these early versions in mind. If you have trouble assembling and linking a listing taken from a book or magazine, try using the **/A** option.

The order in which segments are written to the object file is only one factor in the order in which they will appear in the executable file. The significance of segment order and ways to control it are discussed in Sections 5.2.1 and 5.2.2.3.

■ **Example**

```
MASM /A file;
```

The example above creates an object file, `FILE.OBJ`, whose segments are arranged in alphabetical order. If the **/S** option were used instead, or if no option were specified, the segments would be arranged in sequential order.

2.4.2 Setting the File-Buffer Size

■ **Syntax**

/B*number*

The **/B** option directs the assembler to change the size of the file buffer used for the source file. The *number* is the number of 1024-byte (1K) memory blocks allocated for the buffer. You can set the buffer to any size from 1K to 63K (but not 64K). The default size of the buffer is 32K.

A buffer larger than your source file allows you to do the entire assembly in memory, greatly increasing assembly speed. However, you may not be able to use a large buffer if your computer does not have enough memory or if you have too many resident programs using up memory. If you get an error message indicating insufficient memory, you can decrease the buffer size and try again.

■ **Examples**

```
MASM /B16 file;
```

The example above decreases the buffer size to 16K.

```
MASM /B63 file;
```

The example above increases the buffer size to 63K.

2.4.3 Creating a Pass 1 Listing

■ **Syntax**

/D

The **/D** option tells **MASM** to add a Pass 1 listing to the assembly-listing file, making the assembly listing show the results of both assembler passes. A Pass 1 listing is typically used to locate phase errors. Phase errors occur when the assembler makes assumptions about the program in Pass 1 that are not valid in Pass 2.

The **/D** option does not create a Pass 1 listing unless you also direct **MASM** to create an assembly listing. It does direct the assembler to display error messages for both Pass 1 and Pass 2 of the assembly, even if no assembly listing is created. See Section 2.5.7 for more information about Pass 1 listings.

■ **Example**

```
MASM /D file,,;
```

This example directs the assembler to create a Pass 1 listing for the source file `file.asm`. The file `file.lst` will contain both the first and second pass listings.

2.4.4 Defining Assembler Symbols

■ Syntax

/**D***symbol*⟦= *value*⟧

The **/D** option when given with a *symbol* argument directs **MASM** to define a symbol that can be used during the assembly as if it were defined as a text equate in the source file. Multiple symbols can defined in a single command line.

The *value* can be any text string that does not include a space, comma, or semicolon. If no *value* is given, the symbol is assigned a null string.

■ Example

```
MASM /Dwide /Dmode=3 file,,;
```

This example defines the symbol `wide` and gives it a null value. The symbol could then be used in the following conditional-assembly block:

```
IFDEF wide
      PAGE 50,132
ENDIF
```

When the symbol is defined in the command line, the listing file is formatted for a 132-column printer. When the symbol is not defined in the command line, the listing file is given the default width of 80 (see the description of the **PAGE** directive in Section 12.2).

The example also defines the symbol `mode` and gives it the value 3. The symbol could then be used in a variety of contexts as shown below:

```
scrmode     DB      mode              ; Initiate variable
            IF      mode EQ 3         ; Use in expression
            mov     ax,mode           ; Us in instruction
```

2.4.5 Getting Command-Line Help

■ **Syntax**

/H

The **/H** displays the command-line syntax and all the **MASM** options on the screen. You should not give any file names or other options with the **/H** option.

■ **Example**

```
MASM /H
```

2.4.6 Setting a Search Path for Include Files

■ **Syntax**

/I*path*

The **/I** option is used to set search paths for include files. You can set up to 10 search paths by using the option for each path. The order of searching is the order in which the paths are listed in the command line. The **INCLUDE** directive and include files are discussed in Section 11.6.1.

■ **Example**

```
MASM /Ib:\io /I\macro file;
```

This command line might be used if the source file contains the following statement:

```
INCLUDE dos.inc
```

In this case, **MASM** would search for the file `dos.inc` first in directory `\io` on Drive B, then in directory `\macro` on the current drive. If the file was not found in either of these directories, **MASM** would look next in the current directory and finally in any directories specified with the **INCLUDE** environment variable.

You should not specify a path name with the **INCLUDE** directive if you plan to specify search paths from the command line. For example, if the source file contained any of the following statements, **MASM** would only search path `a:\macro` and would ignore any search paths specified in the command line:

```
INCLUDE a:\macro\dos.mac
INCLUDE ..\dos.mac
INCLUDE .\dos.mac
```

2.4.7 Specifying Case Sensitivity

■ **Syntax**

/ML
/MX
/MU

The **/ML** option directs the assembler to preserve case-sensitivity in all label, variable, and symbol names. The **/MX** option directs the assembler to preserve case sensitivity in public and external names only. The **/MU** option directs the assembler to convert all names to uppercase.

By default, **MASM** converts all names to uppercase. The **/MU** option is provided for XENIX compatibility and to override options given in the environment variable.

If case sensitivity is turned on, all names that have the same spelling, but use letters of different cases, are considered different. For example, with the **/ML** option, DATA and `data` are different. They would also be different with the **/MX** option if they were declared external or public. Public and external names include any label, variable, or symbol names defined using the **EXTRN**, **PUBLIC**, or **COMM** directives (see Chapter 8).

The **/ML** and **/MX** options are typically used when object modules created with **MASM** are to be linked with object modules created by a case-sensitive compiler such as the Microsoft C Compiler. If case sensitivity is important, you should also use the linker **/NOI** option.

■ **Example**

```
MASM /MX module.asm;
```

This example shows how to use the /fB/MX option with **MASM** to assemble a file with case sensitive public symbols.

2.4.8 Suppressing Tables in the Listing File

■ **Syntax**

/N

The **/N** option tells the assembler to omit all tables from the end of the listing file. If this option is not chosen, **MASM** will include tables of macros, structures, records, segments and groups, and symbols. The code portion of the listing file is not changed by the **/N** option.

■ **Example**

```
MASM /N file,,;
```

2.4.9 Checking for Impure Code

■ **Syntax**

/P

The **/P** option directs **MASM** to check for impure code in the 80286 or 80386 protected mode. Real and protected modes are explained in Chapter 13, "Understanding 8086-Family Processors." Versions of DOS available at release time do not implement protected mode.

Code that moves data into memory with a **CS:** override is acceptable in real mode. However, such code may cause problems in protected mode. When the **/P** option is in effect, the assembler checks for these situations and generates an error if it encounters them.

This option is provided for **XENIX** compatibility and to warn about programming practices that would be illegal under future multitasking versions of DOS.

■ **Example**

```
              .CODE
              jmp     past          ; Don't execute data
addr          DW      ?             ; Allocate code space for data
past:         .
              .
              .
              mov     cs:addr,si ; Load register address
```

The example shows a **CS** override. If assembled with the **/P** option, an error will be generated.

2.4.10 Creating Code for a Floating-Point Processor

■ **Syntax**

/R

The **/R** option directs the assembler to generate code and data in the format expected by the 8087, 80287, and 80387 math coprocessors.

If the **/R** option is used, real numbers declared as data are assembled in the IEEE (Institute of Electical and Electronic Engineers, Inc.) format. See Section 6.2.1.4 for a description of the real-number formats used by **MASM**. The **/R** option also enables assembly of the 8087 instruction set. See Chapter 19, "Calculating with a Math Coprocessor," for a description of 8087 instructions.

Using the **/R** option on the command line is equivalent to using the **.8087** directive (see Section 4.7.1) in the source code.

■ **Example**

```
MASM /R file;
```

2.4.11 Creating Code for a Floating-Point Emulator

■ **Syntax**

/E

The **/E** option directs the assembler to generate data in the format expected by the 8087, 80287, and 80387 math coprocessors and to generate code that emulates the instruction sets for those coprocessors.

Real numbers declared as data are assembled in the IEEE format. See Section 6.2.1.4 for a description of the real-number formats used by **MASM**.

To the programmer, writing code for the emulator is like writing code for a coprocessor. The instruction sets are the same. However, at run time the coprocessor instructions are used only if there is a coprocessor available on the machine. If there is no coprocessor, code from an emulator library is used instead. The emulator library emulates coprocessor instructions using the 8088/8086 instruction set.

The **/E** option is for routines called from high-level languages that use a math-emulation library. The Microsoft Macro Assembler package does not include a math-emulation library, but the Microsoft C, BASIC, FORTRAN, and Pascal compilers do. This option should not be used with stand-alone assembler programs. You cannot simply link with the emulator library from a high-level language, since the library requires that the compiler start-up code be executed.

The Microsoft high-level-language compilers allow you to use options to specify whether you want to use emulator code. If you link a high-level-language module prepared with emulator options with an assembler module that uses coprocessor instructions, you should use the **/E** option. The emulator is usually used if you want your code to take advantage of a math coprocessor when run on a machine that has one, but to emulate a coprocessor if the machine does not have one.

■ **Example**

```
MASM /E /MX math.asm;
CL /FPi calc.c math
```

In the first command line, the source file `math.asm` is assembled with **MASM** using the **/E** option. Then the **CL** program of the C compiler is used to compile the C source file `calc.c` with the **/FPi** option, and then link the resulting object file (`calc.obj`) with `math.obj`. The compiler generates emulator code for floating-point instructions. There are similar options for the FORTRAN, BASIC, and Pascal compilers.

2.4.12 Controlling Display of Assembly Statistics

■ **Syntax**

/V
/T

The **/V** and **/T** options specify the level of information to display to the screen at the end of assembly. (V is a mnemonic for verbose, while T is a mnemonic for terse.)

If neither option is given, **MASM** outputs a lines telling the symbol space free and the number of warnings and errors.

If the **V** option is given, **MASM** also reports the number of lines and symbols processed.

If the **/T** option is given, **MASM** does not output anything to the screen unless errors are encountered. This option may be useful in batch or make files if you do not want the output cluttered with unnecessary messages.

If errors are encountered, they will be displayed whether these options are given or not. Appendix B, "Error Messages and Exit Codes," describes the messages displayed after assembly.

2.4.13 Setting the Warning Level

- **Syntax**

/W{0 | 1 | 2}

The **/W** option sets the assembler warning level. **MASM** gives warning messages for assembly statements that are ambiguous or questionable, but not necessarily illegal. Some programmers purposely use practices that generate warnings. By setting the appropriate warning level, they can turn off warnings if they are aware the problem and don't wish to take action to remedy it.

MASM has three levels of errors as shown in Table 2.1.

Table 2.1

Warning Levels

Level	Type	Description
0	Severe errors	Illegal statements
1	Serious warnings	Ambiguous statements or questionable programming practices
2	Advisory warnings	Statements that may produce inefficient code

The default warning level is 1. The higher warning levels include the lower levels. Level 2 includes severe errors, serious warnings, and advisory warnings. If severe errors are encountered, no object file is produced by the assembly.

The advisory warnings are listed below:

Number **Message**

104	`Operand size does not match word size`
105	`Address size does not match word size`
106	`Jump shortened. NOP inserted`

The serious warnings are listed below:

Number	Message
1	`Extra characters on line`
16	`Symbol is reserved word`
31	`Operand types must match`
57	`Illegal size for item`
85	`End of file, no END directive`
101	`Missing data; zero assumed`
102	`Segment near (or at) 64k limit`

All other errors are severe.

2.4.14 Listing False Conditionals

■ **Syntax**

/X

The **/X** option directs **MASM** to copy to the assembly listing all statements forming the body of conditional-assembly statements whose condition is false. If you do not give the **/X** option in the command line, **MASM** suppresses all such statements. The **/X** option lets you display conditionals that do not generate code. Conditional-assembly directives are explained in Chapter 12, "Controlling Assembly Output."

The **.LFCOND**, **.SFCOND**, and **.TFCOND** directives can override the effect of the **/X** option, as described in Section 12.3.2. The **/X** option does not affect the assembly listing unless you direct the assembler to create an assembly-listing file.

■ **Example**

```
MASM /X file,,;
```

Listing of false conditionals is turned on when `file.asm` is assembled. Directives in the source file can override the **/X** option to change the status of false conditional listing.

2.4.15 Writing Symbolic Information to the Object File

■ **Syntax**

/ZI
/ZD

The **/ZI** option directs **MASM** to write symbolic information to the object file. There are two types of symbolic information available: line number data and type data.

Line number data relates each instruction to the source line that created it. The CodeView debugger and **SYMDEB** (the debugger provided with some earlier versions of **MASM**) need this information for source-level debugging.

Type data specifies a size for each variable or label used in the program. The CodeView debugger (but not **SYMDEB**) uses this information to specify the correct size for data objects so that they can be used in expressions.

The **/ZI** option writes both line number and type data to the object file. If you plan to debug your programs with the CodeView debugger, use the **/ZI** option when assembling and the **/CO** option when linking. All the necessary debugging information will be available in executable files prepared in the **.EXE** format. Debugging information will be stripped out of programs prepared in **.COM** format.

The **/ZD** option writes line-number information only to the object file. It can be used if you plan to debug with **SYMDEB** or if you want to see line numbers in map files. The **/ZI** option can also be used for these purposes, but it produces larger object files. If you do not have enough memory to debug a program with the CodeView debugger, you can reduce the program size by using **/ZD** instead of **/ZI** for all or some modules.

The option names **/ZI** and **/ZD** are the same as corresponding option names for recent versions of Microsoft compilers.

2.4.16 Displaying Error Lines on the Screen

■ **Syntax**

/Z

The **/Z** option directs **MASM** to display lines containing errors on the screen. Normally when the assembler encounters an error, it displays only an error message describing the problem. When you use the **/Z** option in the command line, the assembler displays the source line that produced the error in addition to the error message. **MASM** assembles faster without the **/Z** option, but you may find the convenience of seeing the incorrect source lines worth the slight cost in processing speed.

■ **Example**

```
MASM /Z file;
```

2.4.17 Specifying Listing and Cross-Reference Files

■ **Syntax**

/L
/C

The **L** option directs **MASM** to create a listing file even if one was not specified in the command line or in response to prompts. The **/C** option has the same effect for cross-reference files. Files specified with these options always have the base name of the source file plus the extension **.LST** for listing files or **.CRF** for cross-reference files. You cannot specify any other file name. Both options are provided for compatibility with XENIX.

■ **Example**

```
MASM /L /C file;
```

This line creates `file.lst` and `file.crf`. It is equivalent to the following command line:

```
MASM file,,,;
```

2.5 Reading Assembly Listings

MASM creates an assembly listing of your source file whenever you give an assembly-listing file name on the **MASM** command line or in response to the **MASM** prompts. The assembly listing contains both the statements in the source file, and the object code (if any) generated for each statement. The listing also shows the names and values of all labels, variables, and symbols in your source file.

The assembler creates tables for macros, structures, records, segments, groups, and other symbols. These tables are placed at the end of the assembly listing (unless you suppress them with the **/N** option). **MASM** lists only the types of symbols encountered in the program. For example, if your program has no macros, there will be no macro section in the symbol table. All symbol names will be shown in uppercase unless you use the **/ML** or **/MX** to specify case sensitivity.

2.5.1 Reading Code in a Listing

The assembler lists the code generated from the statements of a source file. Each line has the syntax shown below:

⟦*linenumber*⟧ *offset* ⟦*code*⟧ *statement*

The *linenumber* is the number of the line starting from the first statement in the assembly listing. Line numbers are produced only if you request a cross-reference file. Line numbers in the listing do not always correspond to the same lines in the source file.

The *offset* is the offset from the beginning of the current segment to the code. If the statement generates code or data, *code* shows the numeric value in hexadecimal if the value is known as assembly time. If the value is calculated at run time, **MASM** indicates what action is necessary to compute the value. The *statement* is the source statement shown exactly as it appears in the source file, or as expanded by a macro.

If any errors occur during assembly, each error message and error number will be directly below the statement where the error occurred. Refer to Appendix B, "Error Messages and Exit Codes," for a list of **MASM** errors and a discussion of the format in which errors are displayed. An example error line and message is shown below:

```
      71 0012  E8 001C R                              call    doit
test.ASM(46): error A2071: Forward needs override or FAR
```

Note that number 46 in the error message is the source line where the error occurred. Number 71 on the code line is the listing line where the error occured. These lines will seldom be the same. Line numbers in the listing file are produced only if you request a cross-reference file.

The assembler uses the symbols and abbreviations in Table 2.2 to indicate addresses that need to be resolved by the linker or values that were generated in a special way.

Table 2.2

Symbols and Abbreviations in Listings

Character	Meaning
R	Relocatable address; linker must resolve
E	External address; linker must resolve
----	Segment/group address; linker must resolve
=	**EQU** or equal-sign (=) directive
nn:	Segment override in statement
nn/	**REP** or **LOCK** prefix instruction
nn[*xx*]	**DUP** expression; *nn*copies of the value *xx*
n	Macro expansion nesting level (+ if more than nine)
C	Line from **INCLUDE** file

■ Example

The sample listing shown in this section is produced using the **/ZI** option. A cross-reference file specified so that line numbers will appear in the listing. The command line is shown in the following command line:

```
MASM /ZI listdemo,,,;
```

The code portion of the resulting listing is shown below. The tables normally seen at the end of the listing are explained later.

```
Microsoft (R) Macro Assembler Version 5.00              1/24/87 09:28:07
LISTDEMO                                                Page     1-1


     1                                                PAGE    65,132
     2                                                TITLE   LISTDEMO
     3                                    C           INCLUDE dos.mac
     4                                    C   @StrAlloc   MACRO   name,text
     5                                    C   name        DB      &text
     6                                    C               DB      13d,10d
     7                                    C   l&name      EQU     $-name
     8                                    C               ENDM
     9
    10
    11 = 0080                                 larg    EQU     80h
    12
    13                                                DOSSEG
    14                                                .MODEL  small
    15
    16 0800                                           .STACK  256
    17
    18                                        color   RECORD  b:1,r:3=1,i:1=1,f:3=7
    19
    20                                        date    STRUC
    21 0000   05                              month   DB      5
    22 0001   07                              day     DB      7
    23 0002   07C3                            year    DW      1987
    24 0004                                   date    ENDS
    25
    26 0000                                           .DATA
    27 0000   1F                              text    color   <>
    28 0001   09                              today   date    <9,22,1987>
    29 0002   16
    30 0003   07C3
    31
    32 0005   0064[                           buffer  DW      100 DUP(?)
    33 0005   0064[????
    34 0005   0064[????          ]
    35 0005   0064[????          ]
    36                                                @StrAlloc ending,"Finished."
    37 00CD   46 69 6E 69 73 68 65   1   ending          DB       "Finished."
    38 00CD   64 2E 6E 69 73 68 65   1
    39 00D6   0D 0A                  1                DB      13d,10d
    40
    41 0000                                           .CODE
    42
    43 0000   B8 ---- R                       start:  mov     ax,DGROUP
```

```
    44 0003   8E D8                                 mov     ds,ax
    45
    46 0005   B8 0063                               mov     ax,'c'
    47 0008   26: 8B 0E 0080                        mov     cx,es:larg
    48 000D   BF 0052                               mov     di,82
    49 0010   F2/ AE                                repne   scasb
    50 0012   57                                    push    di
    51
    52                                              EXTRN   work:NEAR
    53 0013   E8 0000 E                             call    work
    54
    55 0016   B8 170C                               mov     ax,4C00
listdemo.ASM(40): error A2107: Non-digit in number
    56 0019   CD 21                                 int     21h
    57
    58 001B                                         END     start
```

2.5.2 Reading a Macro Table

A macro table at the end of a listing file gives the names and sizes (in lines) of all macros called or defined in the source file. The macros are listed in alphabetical order.

■ Example

```
Macros:

                N a m e                  Lines

@STRALLOC . . . . . . . . . . .            3
```

2.5.3 Reading a Structure and Record Table

Any structures and records declared in the source file are given at the end of the listing file. The names are listed in alphabetical order. Each name is followed by the fields in the order in which they are declared.

■ Example

```
Structures and Records:

                N a m e                  Width   # fields
                                         Shift   Width    Mask    Initial

COLOR . . . . . . . . . . . . . . .      0008    0004
  B . . . . . . . . . . . . . . . .      0007    0001     0080    0000
  R . . . . . . . . . . . . . . . .      0004    0003     0070    0010
  I . . . . . . . . . . . . . . . .      0003    0001     0008    0008
  F . . . . . . . . . . . . . . . .      0000    0003     0007    0007
```

```
DATE . . . . . . . . . . . . . . .          0004    0003
  MONTH . . . . . . . . . . . . . .         0000
  DAY . . . . . . . . . . . . . . .         0001
  YEAR . . . . . . . . . . . . . .          0002
```

The first row of headings only applies to the record or structure itself. For a record, the "Width" column shows the width in bits while the "# Fields" column tells the total number of fields.

The second row of headings applies only to fields of the record or structure. For records, the "Shift" column lists the offset (in bits) from the low-order bit of the record to the low-order bit in the field. The "Width" column lists the number of bits in the field. The "Mask" column lists the maximum value of the field, expressed in hexadecimal. The "Initial" column lists the initial value of the field, if any. For each field, the table shows the mask and initial values as if they were placed in the record and all other fields were set to 0.

For a structure, the "Width" column lists the size of the structure in bytes. The "# fields" column lists the number of fields in the structure. Both values are in hexadecimal.

For structure fields, the `Shift` column lists the offset in bytes from the beginning of the structure to the field. This value is in hexadecimal. The other columns are not used.

2.5.4 Reading a Segment and Group Table

Segments and groups used in the source file are listed at the end of the program with their size, align type, combine type, and class. If you used simplified segment directives in the source file, the actual segment names generated by **MASM** will be listed in the table.

■ Example

```
Segments and Groups:

                N a m e                     Size    Align   Combine Class

DGROUP . . . . . . . . . . . . . .          GROUP
  _DATA  . . . . . . . . . . . . .          00D8    WORD    PUBLIC  'DATA'
  STACK  . . . . . . . . . . . . .          0800    PARA    STACK   'STACK'
$$SYMBOLS  . . . . . . . . . . . .          0044    PARA    NONE    'DEBSYM'
$$TYPES  . . . . . . . . . . . . .          0041    PARA    NONE    'DEBTYP'
_TEXT  . . . . . . . . . . . . . .          0018    BYTE    PUBLIC  'CODE'
```

The "Name" column lists the names of all segments and groups. The names in the list are given in alphabetical order, except that the names of segments belonging to a group are placed under the group name. The $$SYMBOLS and $$TYPES segments shown in the example are the segments where symbolic information is stored.

The "Size" column lists the byte size (in hexadecimal) of each segment. The size of groups is not shown.

The "Align" column lists the align type of the segment.

The "Combine" column lists the combine type of the segment. If no explicit combine type is defined for the segment, the listing shows NONE, representing the private combine type. If the Align column contains AT, the Combine column contains the hexadecimal address of the beginning of the segment.

The Class column lists the class name of the segment. For a complete explanation of the align, combine, and class types, see Section 5.2.2.

2.5.5 Reading a Symbol Table

All symbols (except names for macros, structures, records, and segments) are listed in a symbol table at the end of the listing.

■ Example

```
Symbols:

                N a m e                 Type     Value    Attr

BUFFER . . . . . . . . . . . . .        L WORD   0005     _DATA    Length = 0064

ENDING . . . . . . . . . . . . .        L BYTE   00CD     _DATA

LARG . . . . . . . . . . . . . .        Number   0080
LENDING  . . . . . . . . . . . .        Number   000B

START  . . . . . . . . . . . . .        L NEAR   0000     _TEXT

TEXT . . . . . . . . . . . . . .        L BYTE   0000     _DATA
TODAY  . . . . . . . . . . . . .        L 0004   0001     _DATA

WORK . . . . . . . . . . . . . .        L NEAR   0000     _TEXT    External

@CURSEG  . . . . . . . . . . . .        Text
@FARCODE . . . . . . . . . . . .        Text   0
@FARDATA . . . . . . . . . . . .        Text   0
@FILENAME  . . . . . . . . . . .        Text   listdemo
```

The "Name" column lists the names in alphabetical order. The "Type" column lists each symbol's type. A type is given as one of the following:

Type	Definition
L NEAR	A near label
L FAR	A far label
N PROC	A near procedure label
F PROC	A far procedure label
Number	An absolute label
Alias	An alias for another symbol
Opcode	An instruction opcode
Text	A memory operand, string, or other value

If the symbol is defined by an **EQU** directive or an equal-sign (=) directive, the `Type` column will show either `Number`, `Opcode`, `Alias`, or `Text`. If the symbol represents a variable, label, or procedure, the `Type` column will show the symbol's length if it is known. A length is given as one of the following:

Type	Length
BYTE	One byte
WORD	One word (2 bytes)
DWORD	Doubleword (4 bytes)
QWORD	Quadword (8 bytes)
FWORD	Farword (6 bytes)
TBYTE	Ten-bytes (10 bytes)
number	Length in bytes of a structure variable

If the symbol represents an absolute value defined with an **EQU** or equal-sign (=) directive, the "Value" column shows the symbol's value. The value may be another symbol, a string, or a constant numeric value (in hexadecimal), depending on whether the type is **Alias**, **Text**, or **Number**. If the type is **Opcode**, the "Value" column will be blank. If the symbol represents a variable, label, or procedure, the "Value" column shows the symbol's hexadecimal offset from the beginning of the segment in which it is defined.

The "Attr" column shows the attributes of the symbol. The attributes include the name of the segment (if any) in which the symbol is defined, the scope of the symbol, and the code length. A symbol's scope is given only if the symbol is defined using the **EXTRN** and **PUBLIC** directives. The scope can be `External` or `Global`. The code length (in hexadecimal) is given only for procedures. The `Attr` column is blank if the symbol has no attribute.

The four text equates shown at the end of the sample table are the ones defined automatically when you use simplified segment directives (see Section 5.1.1).

2.5.6 Reading Assembly Statistics

Data on the assembly, including the number of lines and symbols processed and the error or warnings encountered, are shown at the end of the listing. See Appendix B, "Error Messages and Exit Codes," for further information on this data.

■ **Example**

```
    48 Source  Lines
    52 Total   Lines
    53 Symbols

 45988 Bytes symbol space free

     0 Warning Errors
     1 Severe  Errors
```

2.5.7 Reading a Pass 1 Listing

When you specify the **/D** option in the **MASM** command line, the assembler puts a Pass 1 listing in the assembly-listing file. The listing file show the results of both assembler passes. Pass 1 listings are useful in analyzing phase errors.

The following example illustrates a Pass 1 listing for a source file that assembled without error on the second pass.

```
 0017  7E 00                jle     label1
PASS_CMP.ASM(20) : error 9 : Symbol not defined LABEL1
 0019  BB 1000              mov     bx,4096
 001C              label1:
```

During Pass 1, the **JLE** instruction to a forward reference produces an error message and the value 0 is encoded as the operand. **MASM** displays this error because it has not yet encountered the symbol `label1`.

Later in Pass 1, `label1` is defined. Therefore, the assembler knows about `label1` on Pass 2 and can fix the Pass 1 error. The Pass 2 listing is shown below:

```
0017  7E 03                    jle     label1
0019  BB 1000                  mov     bx,4096
001C                label1:
```

The operand for the **JLE** instruction is now coded as 3 instead of 0 to indicate that the distance of the jump to `label1` is 3 bytes.

Since **MASM** generated the same number of bytes for both passes, there was no error. Phase errors occur if the assembler makes an assumption on pass 1 that it cannot change on pass 2. If you get a phase error, you can examine the Pass 1 listing to see what assumptions the assembler made.

Chapter 3
Using CREF

The Microsoft Cross-Reference Utility (**CREF**), creates a cross-reference listing of all symbols in an assembly-language program. A cross-reference listing is an alphabetical list of symbols in which each symbol is followed by a series of line numbers. The line numbers indicate the lines in the source program that contain a reference to the symbol.

CREF is intended for use as a debugging aid to speed up the search for symbols encountered during a debugging session. The cross-reference listing, together with the symbol table created by the assembler, can make debugging and correcting a program easier.

3.1 Using CREF

CREF creates a cross-reference listing for a program by converting a binary cross-reference file, produced by the assembler, into a readable ASCII file. You create the cross-reference file by supplying a cross-reference-file name when you invoke the assembler. See Section 2.1.1 for more information on creating a binary cross-reference file. You create the cross-reference listing by invoking **CREF** and supplying the name of the cross-reference file.

3.1.1 Using a Command Line to Create a Cross-Reference Listing

To convert a binary cross-reference file created by **MASM** into an ASCII cross-reference listing, type **CREF** followed by the names of the files you want to process.

■ **Syntax**

CREF *crossreferencefile* ⟦*,crossreferencelisting*⟧ ⟦;⟧

The *crossreferencefile* is the name of the cross-reference file created by **MASM**, and the *crossreferencelisting* is the name of the readable ASCII file you wish to create.

If you do not supply file-name extensions when you name the files, **CREF** will automatically provide **.CRF** for the cross-reference file and **.REF** for the cross-reference-listing file. If you do not want these extensions, you must supply your own.

You can select a default file name for the listing file by typing a semicolon immediately after *crossreferencefile.*

You can specify a directory or disk drive for either of the files. You can also name output devices such as CON (display console) and PRN (printer).

When **CREF** finishes creating the cross-reference-listing file, it displays the number of symbols processed.

Examples

```
CREF  test.crf,test.ref
```

The example above converts the cross-reference file `test.crf` to the cross-reference-listing file `test.ref`. It is equivalent to

```
CREF  test,test
```

or

```
CREF  test;
```

The following example directs the cross-reference listing to the screen. No file is created.

```
CREF  test,con
```

3.1.2 Using Prompts to Create a Cross-Reference Listing

You can direct **CREF** to prompt you for the files it needs by starting **CREF** with just the command name. **CREF** prompts you for the input it needs by displaying the following lines, one at a time:

```
Cross-Reference [.CRF]:
Listing [filename.REF]:
```

The prompts correspond to the fields of **CREF** command lines. **CREF** waits for you to respond to each prompt before printing the next one. You must type a cross-reference file name (though the extension is optional) at the first prompt. For the second prompt, you can either type a file name, or press the ENTER key to accept the default displayed in brackets after the prompt.

After you have answered the last prompt and pressed the ENTER key, **CREF** reads the cross-reference file and creates the new listing. It also displays the number of symbols in the cross-reference file.

3.2 Reading Cross-Reference Listings

The cross-reference listing contains the name of each symbol defined in your program. Each name is followed by a list of line numbers representing the line or lines in the listing file in which a symbol is defined or used. Line numbers in which a symbol is defined are marked with a pound sign (#).

Each page in the listing begins with the title of the program. The title is the name or string defined by the **TITLE** directive in the source file (see Section 12.2.1).

■ Example 1

Example 1 shows a source program called `hello.asm`:

```
                TITLE   hello
                PAGE    ,128

                DOSSEG
                .MODEL  small

                .STACK  100 DUP(?)

                .DATA
                PUBLIC  message,lmessage
message         DB      13,"Hello, world.",13,10
lmessage        EQU     $ - message

                .CODE

start:          mov     ax,DGROUP
                mov     ds,ax

                EXTRN   display:NEAR
                call    display

                mov     ax,4C00h
                int     21h

                END     start
```

To assemble the program and create a cross-reference file, enter the following command line:

```
MASM hello,,,;
```

■ Example 2

Example 2 shows the listing file `hello.lst` produced by this assembly.

```
Microsoft (R) Macro Assembler Version 5.00              1/23/87 19:50:38
hello                                                   Page     1-1


     1                                        TITLE   hello
     2                                        PAGE    ,128
     3
     4                                        DOSSEG
     5                                        .MODEL small
     6
     7 0800                                   .STACK 100 DUP(?)
     8
     9 0000                                   .DATA
    10                                        PUBLIC message,lmessage
    11 0000  0D 48 65 6C 6C 6F 2C   message   DB      13,"Hello, world.",13,10
    12 0000  20 77 6F 72 6C 64 2E
    13 0000  0D 0A 6F 72 6C 64 2E
    14 = 0010                       lmessage  EQU     $ - message
    15
    16 0000                                   .CODE
    17
    18 0000  B8 ---- R              start:    mov     ax,DGROUP
    19 0003  8E D8                            mov     ds,ax
    20
    21                                        EXTRN   display:NEAR
    22 0005  E8 0000 E                        call    display
    23
    24 0008  B8 4C00                          mov     ax,4C00h
    25 000B  CD 21                            int     21h
    26
    27 000D                                   END     start

Microsoft (R) Macro Assembler Version 5.00              1/23/87 19:50:38
hello                                                   Symbols-1


Segments and Groups:

                N a m e                 Size    Align   Combine Class

DGROUP . . . . . . . . . . . . .        GROUP
  _DATA  . . . . . . . . . . . .        0010    WORD    PUBLIC  'DATA'
  STACK  . . . . . . . . . . . .        0800    PARA    STACK   'STACK'
_TEXT  . . . . . . . . . . . . .        000D    BYTE    PUBLIC  'CODE'

Symbols:

                N a m e                 Type    Value   Attr

DISPLAY  . . . . . . . . . . . .        L NEAR  0000    _TEXT   External

LMESSAGE . . . . . . . . . . . .        Number  0010            Global
```

```
MESSAGE . . . . . . . . . . . .        L BYTE  0000    _DATA   Global

START . . . . . . . . . . . . .        L NEAR  0000    _TEXT

@CURSEG . . . . . . . . . . . .        Text
@FARCODE . . . . . . . . . . .         Text  0
@FARDATA . . . . . . . . . . .         Text  0
@FILENAME . . . . . . . . . . .        Text  public


     25 Source  Lines
     25 Total   Lines
     35 Symbols

  46408 Bytes symbol space free

      0 Warning Errors
      0 Severe  Errors
```

To create a cross-reference listing of the file `hello.crf`, enter the following command line:

```
CREF hello;
```

Example 3 shown resulting cross-reference-listing file (`hello.ref`).

■ Example 3

```
Microsoft Cross-Reference  Version 4.00        Fri Jan 23 19:50:54 1987
hello

  Symbol Cross-Reference       (# is definition)                 Cref-1

@CURSEG. . . . . . . . . . .    7     7#     7     7     7#     9
                                9#    16     16    16#   16     16#
                               27     27     27#

CODE . . . . . . . . . . . .   16

DATA . . . . . . . . . . . .    9
DGROUP . . . . . . . . . . .    5     5     18
DISPLAY. . . . . . . . . . .   21    21#    22

LMESSAGE . . . . . . . . . .   10    14     14#

MESSAGE. . . . . . . . . . .   10    11     11#    14

STACK. . . . . . . . . . . .    7     7#     7     7
START. . . . . . . . . . . .   18    18#    27

_DATA. . . . . . . . . . . .    9     9#    16
_TEXT. . . . . . . . . . . .    5    16     16#    27

 11 Symbols
```

Compare the line numbers in the cross-reference listing to the line numbers in the listing file. Don't try to compare lines in the source file, since source-line numbers may not match line numbers in the listing and cross-reference-listing files.

Using Directives

Chapter 4
Writing Source Code

Assembly-language programs are written as source files, which can then be assembled into object files by **MASM**. Object files can then be processed and combined with **LINK** to form executable files.

Source files are made up of assembly-language statements. Statements are in turn made up of operators, symbols, constants, and mnemonics. This chapter defines these basic building blocks of assembly-language source code. It also tells how to start and end assembly-language source files.

4.1 Assigning Names to Symbols

A symbol is a name that represents a value. Symbols are one of the most important elements of assembly-language programs. Elements that must be represented symbolically in assembly-language source code include variables, address labels, macros, segments, procedures, records, and structures. Constants, expressions, and strings can also be represented symbolically.

Symbol names are combinations of letters, digits, and special characters. **MASM** recognizes the following character set:

```
A-Z a-z 0-9
?  @  _  $ : . [ ] ( ) < > { } + - / *
& % ! ' ~ ¦ \ = # ^ ; , ` "
```

Letters, digits, and most some characters can be used in symbol names, but there are some restrictions on how certain characters can be used or combined:

- A name can have any combination of uppercase and lowercase letters. All lowercase letters are converted to uppercase by the assembler, unless the **/ML** assembly option is used, or the name is declared with a **PUBLIC** or **EXTRN** directive and the **/MX** option is used.
- Digits may be used within a name, but not as the first character.
- A name can have any number of characters, but only the first 31 are used. All other characters are ignored.

- The following special characters may be used at the beginning or within a name: underscore (_), percent sign (%), question mark (?), dollar sign ($), and at sign (@).
- The period (.) is an operator and cannot be used within a name, but it can be used as the first character of a name.
- A name may not be the same as any reserved name. Note that two special characters, question mark (?) and dollar sign ($), are reserved names and may not be used as single character symbols, though they may be contained within or at the start of symbols.

■ **Examples**

```
; Legal
sub3        EQU     3
str$        DB      "Test"
            mov     ax,lines_per_inch

; Illegal
3sub        EQU     3                   ; Digit can't start name
word        =       5                   ; Can't use reserved name
array-name  DB      40 DUP(?)           ; Can't use operator "-"

; Questionable
            mov     ax,.lines.per.inch  ; Operator "." can't be
                                        ;   within name
```

The last example is a legal statement, but the operand `.lines.per.inch` is an expression, not a name. It means `.lines` plus `per` plus `inch`. See Chapter 9, "Using Operands and Expressions," for information on expressions and operators.

4.2 Reserved Names

A reserved name is any name with a special, predefined meaning to the assembler. Reserved names include instruction and directive mnemonics, register names, and operator names. These names can be used only as defined and must not be redefined. All uppercase and lowercase letter combinations of these names are treated as the same name.

Table 4.1 lists names that are always reserved by the assembler. Using any of these names as a symbol results in an error.

Table 4.1

Reserved Names

$	DQ	EXTRN	LENGTH	RECORD
=	DS	FAR	.LFCOND	REPT
?	DT	.FARDATA	.LIST	.SALL
.186	DW	.FARDATA?	LOCAL	SEG
.286	DWORD	FWORD	LOW	SEGMENT
.287	ELSE	GE	LT	SEQ
.386	END	GROUP	MACRO	.SFCOND
.387	ENDIF	GT	MASK	SHL
.8086	ENDM	HIGH	MOD	SHORT
.8087	ENDP	IF	.MODEL	SHR
ALIGN	ENDS	IF1	NAME	SIZE
.ALPHA	EQ	IF2	NE	STACK
AND	EQU	IFB	NEAR	STRUC
ASSUME	ERR	IFDEF	NOT	SUBTTL
BYTE	.ERR1	IFDIF	OFFSET	TBYTE
.CODE	.ERR2	IFE	OR	.TFCOND
COMM	.ERRB	IFIDN	ORG	THIS
COMMENT	.ERRDEF	IFNB	%OUT	TITLE
.CONST	.ERRDIF	IFNDEF	PAGE	TYPE
.CREF	.ERRE	INCLUDE	.PRIV	.TYPE
.DATA	.ERRIDN	INCLUDELIB	PROC	WIDTH
.DATA?	.ERRNB	IRP	PTR	WORD
DB	.ERRNDEF	IRPC	PUBLIC	.XALL
DD	.ERRNZ	LABEL	PURGE	.XCREF
DF	EVEN	.LALL	QWORD	.XLIST
DOSSEG	EXITM	LE	.RADIX	XOR

In addition to these names, instruction mnemonics and register names are considered reserved names. They may vary according to the processor directive specified. Section 4.6.1 describes processor directives. Register names are listed in Section 14.2. Instruction mnemonics for each processor are listed in the *Microsoft Macro Assembler Reference.*

4.3 Constants

Constants can be used in source files to specify numbers or strings that are set or initialized at assembly time. **MASM** recognizes five types of constant values:

- Integers
- Real numbers
- Encoded real numbers
- Packed decimal numbers
- Strings

4.3.1 Integer Constants

Integer constants represent integer values. They can be used in a variety of contexts in assembly-language source code. For example, they can be used in data declarations, equates, and as immediate operands.

Packed decimal integers are a special kind of integer constant that can only be used to initialize binary coded decimal (BCD) variables. They are described in Section 4.3.1.3 and 6.2.1.2.

Integer constants can be specified in binary, octal, decimal, or hexadecimal. Table 4.2 shows the legal digits for each of these radixes.

Table 4.2

Digits Used with Each Radix

Name	Base	Digits
Binary	2	0 1
Octal	8	0 1 2 3 4 5 6 7
Decimal	10	0 1 2 3 4 5 6 7 8 9
Hexadecimal	16	0 1 2 3 4 5 6 7 8 9 A B C D E F

The radix for an integer can be defined for a specific integer using radix specifiers, or a default radix can be defined globally with the **.RADIX** directive.

4.3.1.1 Specifying Integers with Radix Specifiers

The radix for an integer constant can be given by putting one of the following radix specifiers after the last digit of the number:

Radix	Specifier
Binary	**B**
Octal	**Q** or **O**
Decimal	**D**
Hexadecimal	**H**

Radix specifiers can be given in either uppercase or lowercase; sample code in this manual uses lowercase.

Hexadecimal numbers must always start with a decimal digit (0 to 9). If necessary, put a leading 0 at the left of the number to distinguish between symbols and hexadecimal numbers that start with a letter. For example, `0ABCh` is interpreted as a hexadecimal number, but `ABCh` is interpreted as a symbol. The hexadecimal digits A through F can be either uppercase or lowercase. Sample code in this manual uses uppercase.

If no radix is given, the assembler interprets the integer using the current default radix. The initial default radix is decimal, but you can change the default with the **.RADIX** directive.

■ **Examples**

```
n360        EQU   01011010b + 132q + 5Ah + 90d ; 4 * 90
n60         EQU   01111b    +  17o + 0Fh + 15d ; 4 * 15
```

4.3.1.2 Setting the Default Radix

The **.RADIX** directive sets the default radix for integer constants in the source file.

■ Syntax

.RADIX *expression*

The *expression* must evaluate to a number in the range 2 to 16. It defines whether the numbers are binary, octal, decimal, hexadecimal, or numbers of some other base.

Numbers given in the *expression* are always considered decimal, regardless of the current default radix. The initial default radix is decimal.

Note

The **.RADIX** directive does not affect real numbers initialized as variables with the **DD**, **DQ**, or **DT** directives. Initial values for values declared with these directives are always evaluated as decimal unless a radix specifier is appended.

The **.RADIX** directive does not affect the optional radix specifiers, **B** and **D**, used with integer numbers. When **B** or **D** appears at the end of any integer, it is always considered to be a radix specifier even if the current input radix is 16.

For example, if the input radix is 16, the number `0ABCD` will be interpreted as 0ABC decimal, an illegal number, instead of as 0ABCD hexadecimal, as intended. Type `0ABCDh` to specify 0ABCD in hexadecimal. Similarly, the number `11B` will be treated as 11 binary, a legal number, but not 11B hexadecimal, as intended. Type `11Bh` to specify 11B in hexadecimal.

■ Examples

```
.RADIX  16          ; Set default radix to hexadecimal
.RADIX  2           ; Set default radix to binary
```

4.3.1.3 Binary Coded Decimal Constants

When an integer constant is used with the **DT** directive, the number is interpreted by default as a packed binary coded decimal number. You can use the **D** radix specifier to override the default and initialize ten-byte integers as binary-format integers.

The syntax for specifying binary coded decimals is exactly the same as for other integers. However, **MASM** encodes binary coded decimals in a completely different way. See Section 6.2.1.2 for complete information on storage of binary coded decimals.

■ **Examples**

```
positive    DT      1234567890  ; Encoded as 00000000001234567890h
negative    DT      -1234567890 ; Encoded as 80000000001234567890h
```

4.3.2 Real-Number Constants

A real number is a number consisting of an integer part, a fractional part, and an exponent. Real numbers are normally represented in decimal format.

■ **Syntax**

⟦+ | –⟧ *integer.fraction*⟦**E**⟦+ | –⟧*exponent*⟧

The *integer* and *fraction* parts combine to form the value of the number. This value is stored internally as a unit and is called the mantissa. It may be signed. The optional *exponent* follows the exponent indicator (**E**). It represents the magnitude of the value, and is stored internally as a unit. If no *exponent* is given, 1 is assumed. If an exponent is given, it may be signed.

During assembly, **MASM** converts real-number constants given in the decimal format to a binary format. The sign, exponent, and mantissa of the real number are encoded as bit fields within the number. See Section 6.2.1.5 for an explanation of how real numbers are encoded.

You can specify the encoded format directly using hexadecimal digits (0–9 or A–E). The number must begin with a decimal digit (0-9) and must be followed by the real-number designator (**R**). This designator is used exactly like a radix designator except that it specifies that the given hexadecimal number should be interpreted as a real number.

Real numbers can only be used to initialize variables with the **DD**, **DQ**, and **DT** directives. They cannot be used in expressions. The maximum number of digits in the number and the maximum range of exponent values depend on the directive. The number of digits for encoded numbers used with **DD**, **DQ**, and **DT** must be 8, 16, and 20 digits, respectively. (If a leading 0 is supplied, the number must be 9, 17, or 21 digits.) See Section 6.2.1.5 for an explaination of how real numbers are encoded.

Note

Real numbers will be encoded differently depending on assembly options and processor directives. By default, real numbers are encoded in the Microsoft Binary Real format. If the **/R** or **/E** assembly option is specified, or if the **.8087**, **.287**, or **.387** directive is used, real numbers will be encoded in the IEEE format. See Section 6.2.1.5 for a description of these formats.

■ Example

```
shrt        DD      25.23
long        DQ      2.523E1
ten_byte    DT      2523.0E-2

; Assumes no /R, /E, .8087, .287, or .387
eshrt       DD      3F800000r              ; 1.0 as a short real
elong       DQ      3FF0000000000000r      ; 1.0 as a long real
eten_byte   DT      3FFF8000000000000000r  ; 1.0 as a ten-byte real
```

4.3.3 String Constants

A string constant consists of one or more ASCII characters enclosed in single quotation marks or double quotation marks.

■ Syntax

'characters'
"characters"

String constants are case sensitive. A string constant consisting of a single character is sometimes called a character constant.

Single quotation marks must be encoded twice when used literally within string constants that are also enclosed by single quotation marks. Similarly, double quotation marks must be encoded twice when used in string constants that are also enclosed by double quotation marks.

■ Examples

```
char        DB      'a'
twochar     DW      'ab'
char2       DB      "a"
message     DB      "This is a message."
warn        DB      'Can''t find file.'          ; Can't find file.
warn2       DB      "Can't find file."           ; Can't find file.
string      DB      "This ""value"" not found."  ; This "value" not found.
string2     DB      'This "value" not found.'    ; This "value" not found.
```

4.4 Using Type Specifiers

Some statements require type specifiers to give the size or type of an operand. There are two kinds of type specifiers: those that specify the size of a variable or other memory operand, and those that specify the distance of a label.

The type specifiers that give the size of a memory operand are listed below with the number of bytes specified by each:

Specifier	Number of Bytes
BYTE	1
WORD	2
DWORD	4

FWORD	6
QWORD	8
TBYTE	10

In some contexts, **ABS** can also be used as a type specifier that indicates that an operand is a constant rather than a memory operand.

The type specifiers that give the distance of a label are listed below:

Specifier	Description
FAR	The label references both the segment and offset of the label.
NEAR	The label references only the offset of the label.
PROC	The label has the default type (near or far) of the current memory model. The default size is always near if you use full segment definitions. If you use simplified segment definitions (see Section 5.1) the default type is near for small and compact models or far for medium, large, and huge models.

Directives that require or can use type specifiers include **LABEL**, **PROC**, **EXTRN**, and **COMM**. Operators that require a type specifier include **PTR** and **THIS**.

4.5 Writing Assembly-Language Statements

A statement is a combination of symbols, mnemonics, operands, and comments that defines the object code to be created at assembly time. Each line of source code consists of a single statement. Multiline statements are not allowed. Statements must not have more than 128 characters. Statements can have up to four fields, as shown below:

■ Syntax

⟦*name*⟧ ⟦*operation*⟧ ⟦*operands*⟧ ⟦*;comment*⟧

The fields are explained below, starting with the leftmost field:

Field	**Purpose**
name	Labels the statement so that it can be accessed by name in other statements
operation	Defines the action of the statement
operands	Define the data to be operated by the statement
comment	Describes the statement without having any effect on assembly

All fields are optional, although the operand or name fields may be required if certain directives or instructions are given in the operation field. A blank line is simply a statement in which all fields are blank. A comment line is a statement in which all fields except the comment are blank.

Statements can be entered in uppercase or lowercase. Sample code in this manual uses uppercase letters for directives, hexadecimal letter digits, and segment definitions. Your code will be clearer if you choose a case convention and use it consistently.

Each field must be separated from other fields by a space or tab character. That is the only limitation on structure imposed by **MASM**. For example, the following code is legal:

```
@device equ 1;define device macros
include dos.inc;include macros
.stack 100h; allocate 100h-byte stack
.data
message db 13,"hello, world.",13,10;message to be written
lmessage equ $ - message;length of message
.code
start: mov ax,dgroup;load segment location
mov ds,ax;into ds register
@write message,lmessage;write "message"
```

However, the code is much easier to interpret if each field is assigned a specified tab positions and a standard convention is used for capitalization:

```
@device     EQU     1                    ; Define device macros
            INCLUDE DOS.INC              ; Include macros

            .STACK  100h                 ; Allocate 100h-byte stack

            .DATA
message     DB      13,"Hello, world.",13,10    ; Message to be written
lmessage    EQU     $ - message                 ; Length of message
```

```
            .CODE
start:      mov     ax,DGROUP           ; Load segment location
            mov     ds,ax               ;   into DS register

            @Write  message,lmessage    ; Write "message"
```

4.5.1 Using Mnemonics and Operands

Mnemonics are the names assigned to commands that tell either the assembler or the processor what to do. There are two types of mnemonics: directives and instructions.

Directives are instructions to the assembler. They specify the manner in which the assembler is to generate object code at assembly time. Part 2, "Using Directives to Control Assembly-Time Processing," describes the directives recognized by the assembler. Directives are also discussed where necessary in Part 3, "Using Instructions to Control Run-Time Processing."

Instructions are instructions to the processor. At assembly time, they are translated into object code. At run time, the object code controls the behavior of the processor. Instructions are described in Part 3, "Using Instructions to Control Run-Time Processing."

Operands define the data that will be used by directives and instructions. They can be made up of constants, expressions, registers, and symbols. Operands are discussed throughout the manual, but particularly in Chapter 9, "Using Operands and Expressions," and Chapter 14, "Using Addressing Modes."

4.5.2 Writing Comments

Comments are descriptions of the code. They are for documentation only and are ignored by the assembler.

Any text following a semicolon is considered a comment. Comments commonly start in the column assigned for the comment field, or in the first column of the source code. The comment must follow all other fields in the statement.

Multiline comments can either be specified with multiple comment statements, or with the **COMMENT** directive.

■ **Syntax**

COMMENT *delimiter* ⟦*text*⟧
text
delimiter ⟦*text*⟧

All *text* between the first *delimiter* and the line containing a second *delimiter* is ignored by the assembler. The *delimiter* character is the first non-blank character after the **COMMENT** directive. The *text* includes the comments up to and including the line containg the next occurrence of the delimiter.

■ **Example**

```
            COMMENT + The plus
                    sign is the delimiter. The
                    assembler ignores the statement
                    following the last delimiter
+           mov     ax,1   (ignored)
```

4.6 Starting and Ending Source Files

Most assembly-language statement can appear at any point in the source code. However, certain statements that apply to the entire source file rather than one statement must come at the beginning or end.

4.6.1 Starting a Source File

Since the assembler processes sequentially, any directives that define the behavior of the assembler for the entire source file must come at the beginning of the file.

The **.MODEL** directive defines the memory model for the entire assembly. The processor directives define the processor and/or coprocessor for the entire assembly. These directives are optional. If you do not use them, **MASM** makes default assumptions about the memory model and processor. However, if you do use them, you must put them at or near the start of the source file before any segment definitions so that all segments and instructions will be assembled consistently.

The **.MODEL** directive is explained in Section 5.1.3. The processor directives are listed and explained below:

Directive	Description
.8086	Enables assembly of instructions for the 8086 and 8088 microprocessors and disables assembly of the instructions unique to the 80186, 80286, and 808386 processors. This is the default directive and will be used if no instruction set directive is specified. Using the default instruction set ensures that your program will be usable on all 8086-family processors. However, the program will not take advantage of the more powerful instructions available on more advanced processors.
.186	The **.186** directive enables assembly of the 8086 instructions plus the additional instructions for the 80186 processor.
.286	The **.286** directive enables assembly of the 8086 instructions plus the additional instructions for the 80186 and 80286 processors. The **.286** directive can be used with the **.PRIV** directive to enable protected-mode instructions of the 80286. This directive should be used for programs that will be executed only by an 80186, 80286, or 80386 processor. For compatibility with previous versions of **MASM**, the **.286C** directive is also available. It is equivalent to the **.286** directive.
.386	The **.386** directive enables assembly of the 8086, 80186, and 80286 instructions plus the additional instructions for the 80386 processor. The **.386** directive can be used with the **.PRIV** directive to enable protected-mode instructions of the 80386. This directive should be used for programs that will be executed only by an 80386 processor. For compatibility with previous versions of **MASM**, the **.386C** directive is also available. It is equivalent to the **.386** directive.
.PRIV	The **.PRIV** directive (a mnemonic for privileged) enables assembly of protected-mode instructions of the 80286 or 80386 processors. It should be used with the **.286** or **.386** directives. It will be ignored if used with the **.8086** or **.186** directives, since these processors have no protected mode. The **.PRIV** directive can be used with programs that will be executed only by an 80286 or 80386 processor using both protected and nonprotected instructions. This

does not mean that the directive is required if the program will run in protected mode; only that it is required if the program uses the instructions that initiate and manage protected-mode processes. These instructions (see Section 20.4) are normally used only by systems programmers. For compatibility with previous versions of **MASM**, the **.286P** and **.386P** directives are provided. They are equivalent to using **.PRIV** with the **.286** or **.386** directive.

.8087 The **.8087** directive enables assembly of instructions for the 8087 math coprocessor and disables assembly of instructions unique to the 80287 coprocessor. It also specifies that real numbers declared as data in the source code will be assembled in the IEEE format expected by 8087-family coprocessors. This directive should be used for programs that must run with either the 8087, 80287, or 80386 coprocessors. Using the **/R** option in the **MASM** command line is equivalent to using the **.8087** directive in the source code.

.287 The **.287** directive enables assembly of instructions for the 8087 floating-point coprocessor and the additional instructions for the 80287. It also specifies that real numbers declared as data in the source code will be assembled in the IEEE format expected by the 8087-family processors. Coprocessor instructions will be optimized if you use this directive rather than the **.8087** directive. Therefore you should use it if you know your program will never need to run under an 8087 processor.

.387 The **.387** directive enables assembly of instructions for the 8087 and 80287 floating-point coprocessors and the additional instructions and addressing modes of the 80387. It also specifies that real numbers declared as data in the source code will be assembled in the IEEE format expected by the 8087-family coprocessors. The directive must be used if the coprocessor is to operate in 32-bit mode with an 80386 processor.

If you do not specify a processor directive, **MASM** assumes **.8086**. If you do not specify a coprocessor directive, **MASM** assumes there is no coprocessor. With no coprocessor specified, **MASM** assembles real-number data in the Microsoft Binary Real format, as described in Section 6.2.1.5.

4.6.2 Ending a Source File

Source files are always terminated with the **END** directive. This directive has two purposes: it marks the end of the source file, and it can indicate the address where execution will begin when the program is loaded.

■ Syntax

END [[*startaddress*]]

Any statements following the **END** directive will be ignored by the assembler. For example, you can put comments after the **END** directive without using comment specifiers (;) or the **COMMENT** directive.

The *startaddress* is a label or expression identifying the address where you want execution to begin when the program is loaded. Specifying a start address is discussed in detail in Section 5.5.1.

Chapter 5
Defining Segment Structure

5.6 Nesting Segments 111

Segments are a fundamental part of assembly-language programming for the 8086-family of processors. They are related to the segmented architecture used by Intel for its 16-bit and 32-bit microprocessors. This architecture is explained in more detail in Chapter 13, "Understanding 8086-Family Processors."

A segment is a collection of instructions or data whose addresses are all relative to the same segment register. Segments can be defined using simplified segment directives or with full segment definitions.

In most cases, simplified segment definitions are a better choice. They are easier to use and more consistent, yet you seldom sacrifice any functionality by using them.

Although more difficult to use, full segment definitions give more complete control over segments. A few complex programs may require complete segment definitions in order to get unusual segment orders and types. This was the only way to define segments in previous versions of **MASM**, so you may need to use it to maintain existing source code.

This chapter describes both methods. In most cases, you can read about one method and ignore the other.

5.1 Simplified Segment Definitions

Version 4.5 of **MASM** implements a new simplified system for declaring segments. By default, the simplified segment directives use the segment names and conventions used by Microsoft high-level languages. If you are willing to accept these conventions, the more difficult aspects of segment definition are handled automatically.

If you are writing stand-alone assembler programs where segment names, order, and other definition factors are not crucial, the simplified segment directives make programming easier. The Microsoft conventions are flexible enough to work for most kinds of programs. If you are new to assembly-language programming, you should use the simplified segment directives for your first programs.

If you are writing assembler routines to be linked with Microsoft high-level languages, the simplified segment directives ensure against mistakes that would make your modules incompatible. The names are automatically defined consistently and correctly.

When you use simplified segment directives, **ASSUME** and **GROUP** statements consistent with Microsoft conventions are generated automatically. You can learn more about the **ASSUME** and **GROUP** directives in Sections 5.3 and 5.4. However, for most programs you do not need to understand these statements. You can simply use the statements in the format shown in the examples.

Note

The simplified segment directives cannot be used for programs written in the **.COM** format. You must specifically define the single segment required for this format. See Section 1.4.1 for information on the **.COM** format.

5.1.1 Understanding Memory Models

To use simplified segment directives, you must declare a memory model for your program. The memory model specifes the default size of data and code used in a program.

Microsoft high-level languages require that each program have a default size (or memory model). Any assembly-language routine called from a high-level-language program should have the same memory model as the calling program. See the documentation for your language to see what memory models it can use.

Stand-alone assembler programs can have any model. Small model is adequate for most programs written entirely in assembly language.

The most commonly used memory models are described below:

Model	**Description**
Tiny	All data and code fits in a single segment. Tiny model programs must be written in the **.COM** format. Microsoft languages do not support this model. Some compilers from other companies support tiny model either as an option or as a requirement. You cannot use simplified segment directives for tiny-model programs.

Small	All data fits within a single 64K segment and all code fits within a 64K segment. Therefore, all code and data can be accessed as near. This is the most common model for stand-alone assembler programs. C is the only Microsoft language that supports this model.
Medium	All data fits within a single 64K segment, but code may be greater than 64K. Therefore, data is near, but code is far. Most recent versions of Microsoft languages support this model.
Compact	All code fits within a single 64K segment, but the total amount of data may be greater than 64K (though no array can be larger than 64K). Therefore, code is near, but data is far. C is the only Microsoft language that supports this model.
Large	Both code and data may be greater than 64K (though no array can be larger than 64K). Therefore, both code and data are far. All Microsoft languages support this model.
Huge	Both code and data may be greater than 64K. In addition, data arrays may be larger than 64k. Both code and data must are far, and pointers to elements within an array must also be far. Most recent versions of Microsoft languages support this model. Segments are the same for large and huge models.

Since near data or code can be accessed more quickly, the smallest memory model that can accommodate your code and data is usually the most efficient.

Mixed model programs use the default size for most code and data, but override the default for particular data items. Stand-alone assembler programs can be written as mixed model by making specific procedures or variables near or far. Some Microsoft high-level languages have **NEAR**, **FAR**, and **HUGE** keywords that enable you to override the default size of individual data or code items.

5.1.2 Specifying DOS Segment Order

The **DOSSEG** directive specifies that segments be ordered according the to DOS segment-order convention. This is the convention used by Microsoft high-level language compilers.

■ Syntax

DOSSEG

You should use the **DOSSEG** argument in the main module of stand-alone assembler programs. Modules called from the main module need not use the **DOSSEG** argument. You do not need to use the **DOSSEG** directive for modules called from high-level languages, since the compiler already defines DOS segment order.

The DOS segment-order convention has the following rules:

1. All segment names having the class name `'CODE'` are placed at the beginning for the executable file.
2. Any segments that do not have class name `'CODE'` and are not part of the group DGROUP are placed after the code segments.
3. Segments that are part of DGROUP come at the end of the executable file.

Using the **DOSSEG** directive has the same effect as using the **/DOSSEG** linker option.

The directive works by writing to the comment record of the object file. The Intel title for this record is **COMENT**. If the linker detects a certain sequence of bytes in this record, it automatically puts segments in the DOS order.

5.1.3 Defining the Memory Model

The **.MODEL** directive is used to initialize the memory model. This directive should be used early in the source code before any other segment directive.

■ Syntax

.MODEL *memorymodel*

The *memorymodel* can be **SMALL**, **MEDIUM**, **COMPACT**, **LARGE**, or **HUGE**. Segments are defined exactly the same for large and huge models, but the `@fardata` equate (explained in Section 5.1.5) is different.

If you are writing an assembler routine for a high-level language, the *memorymodel* should match the memory model used by the compiler or interpreter.

If you are writing a stand-alone assembler program, you can use any model. Section 5.1.1 describes each memory model. Small model is the best choice for most stand-alone assembler programs.

Note

You must use the **.MODEL** directive before defining any segment. If one of the other simplified segment directives (such as **.CODE** or **.DATA**) is given before the **.MODEL** directive, an error will be generated.

■ Example 1

```
DOSSEG
.MODEL  small
```

This statement defines default segments for small model and creates the **ASSUME** and **GROUP** statements used by small-model programs. The segments are automatically ordered according to the Microsoft convention. The statement might be used at the start of the main (or only) module of a stand-alone assembler program.

■ Example 2

```
.MODEL  LARGE
```

This statement defines default segments for large model and creates the **ASSUME** and **GROUP** statements used by large-model programs. It does not automatically order segments according to the Microsoft convention. The statement might be used at the start of an assembly module that would be called from a large model C, BASIC, FORTRAN, or Pascal program.

■ **80386 Processor Only**

If you use the **.386** directive before the **.MODEL** directive, the segments definitions will define 32-bit segments. If you want to enable the 80386 processor with 16-bit segments, you should give the **.386** directive after the **.MODEL** directive.

5.1.4 Defining Segments

The **.CODE**, **.DATA**, **.DATA?**, **.FARDATA**, **.FARDATA?**, **.CONST**, and **.STACK** directives indicates the start of a segment. They also end any open segment definition used earlier in the source code.

■ **Syntax**

.STACK [[*size*]]	Stack segment
.CODE [[*name*]]	Code segment
.DATA	Initialized near data segment
.DATA?	Uninitialized near data segment
.FARDATA [[*name*]]	Initialized far data segment
.FARDATA? [[*name*]]	Uninitialized far data segment
.CONST	Constant data segment

For segments that take an optional *name*, a default name is used if none is specified. See Section 5.1.7 for information on default segment names.

Each new segment directive ends the previous segment. The **END** directive closes the last open segment in the source file.

The *size* argument of the **.STACK** directive is the number of bytes to be declared in the stack. If no *size* is given, the segment is defined, but no memory is allocated for it.

Stand-alone assembler programs in the **.EXE** format should define a stack for the main (or only) module. Modules that will be linked with a main module from a high-level language need not define a stack, since one is already defined by the compiler or interpreter.

Code should be placed in a segment initialized with the **.CODE** directive, regardless of the memory model. Normally, only one code segment is defined in a source module. If you put multiple code segments in one source file, you must specify *name* to distinguish the segments. The *name* can only be specified for models that allow multiple code segments (medium and large). If you give *name* with small or compact model, it will be ignored.

Uninitialized data is any variable declared using the indeterminate symbol (**?**) and the **DUP** operator. When declaring data for modules that will be used with a Microsoft high-level language, you should follow the convention of using **.DATA** or **.FARDATA** for initialized data and **.DATA?** or **.FARDATA?** for uninitialized data. For stand-alone assembler programs, using the **.DATA?** and **.FARDATA?** directives is optional. You can put unitialized data in any data segment.

Constant data is data that must be declared in a data segment, but that is not subject to change at run time. Use of this segment is optional for stand-alone assembler programs. However, the Microsoft FORTRAN and Pascal compilers protect constant data against change. If you are writing assembler routines to be called from these compilers, you should use the **.CONST** directive to declare strings, real numbers, and other constant data that must be allocated in data segments.

Data in segments defined with the **.STACK**, **.CONST**, **.DATA** or **.DATA?** directives will be placed in a group called **DGROUP**. Data in segments defined with the **.FARDATA** or **.FARDATA?** directives will not be in any group. See Section ?.? for more information on groups. When initializing the **DS** register to access data in a group-associated segment, the value of **DGROUP** should be loaded into **DS**. See Section 5.5.2 for information on initializing data segments.

■ Example 1

```
            DOSSEG
            .MODEL  SMALL
            .STACK  100h
            .DATA
ivariable   DB      5
iarray      DW      50 DUP (5)
            EXTRN   xvariable:WORD
            .CONST
string      DB      "This is a string"
            .DATA?
uvariable   DB      1 DUP (?)
uarray      DW      50 DUP (?)
            .CODE
start:      mov     ax,DGROUP
            mov     ds,ax
            EXTRN   xprocedure:NEAR
            .
```

```
                .
                .
                END     start
```

This code using simplified segment directives. See Section 5.1.7 for an equivalent version that uses full segment definitions.

■ **Example 2**

```
                .MODEL  LARGE
                .FARDATA? STUFF
array           DW      10 DUP (?)
                .CONST
string1         DB      "This is a string."
                .CODE   ACTION
task            PROC
                .
                .
                .
                ret
task            ENDP
                END
```

This example uses simplified segment directives to create a module that might be called from an assembly-language program that does not follow the Microsoft naming conventions. See Section 5.1.7 for an equivalent version using full segment definitions.

5.1.5 Using Predefined Equates

When you use simplified segment directives, several equates are predefined for you. You can use the equate names at any point in your code to represent the equate values. You should not assign equates having these names.

The predefined equates are listed below:

Name	**Value**
@segcur	This name has the segment name of the current segment. This value may be convenient for **ASSUME** statements or segment overrides. It can also be used to terminate a segment if you want to mix simplified segments and full segments.

The example below illustrates the technique:

```
             .DATA
             .
             .
             .
@segcur      ENDS                ; End current (data) segm
MYSEG        SEGMENT AT 0        ; Start full segment defi
             .
             .
             .
MYSEG        ENDS                ; End segment
             .CODE               ; Start new simplified se
```

`@filename`

This value represents the base name of the current source file. For example, if the current source file is `task.asm`, the value of `@filename` is `task`. This value can be used in any name you would like to change if the file name changes. For example, it can be used as a procedure name:

```
@filename    PROC
             .
             .
             .
@filename    ENDP
```

`@farcode` and `@fardata`

If the **.MODEL** directive has been used, the `@farcode` value is 0 for small and compact models or 1 for medium and large models. The `@fardata` value is 0 for small and medium models, 1 for compact and large models, and 2 for huge model. These values can be used in conditional-assembly statements:

```
IF       @farcode
EXTRN    task:FAR
ELSE
EXTRN    task:NEAR
ENDIF
```

5.1.6 Simplified Segment Defaults

When you use the simplified segment directives, defaults are different in certain situations than they would be if you gave full segment definitions. Defaults that change are listed below:

- If you give full segment definitions, the default size for the **PROC** directive is always **NEAR**. If you use the **.MODEL** directive, the **PROC** directive is associated with the specified memory model: **NEAR** for small and compact models or **FAR** for medium and large models. See Section 6.1.2 for further discussion of the **PROC** directive.

- If you give full segment definitions, the segment address used as the base when calculating an offset with the **OFFSET** operator is the data segment. (the segment associated with the **DS** register). With the simplified segment directives, the base address is the **DGROUP** segment for segments that are associated with a group. This includes segments declared with the **.DATA**, **.DATA?**, and **.STACK** directives, but not segments declared with the **.CODE**, **.FARDATA**, and **.FARDATA?** directives.

  For example, assume the variable `test1` was declared in a segment defined with the **.DATA** directive and `test2` was declared in a segment defined with the **.FARDATA** directive. The statement

  ```
          mov     ax,OFFSET test1
  ```

  loads the address of `test1` relative to **DGROUP**. The statement

  ```
          mov     ax,OFFSET test2
  ```

  loads the address of `test2` relative to the segment defined by the **.FARDATA** directive. See Section 5.3 for more information on groups.

- With full segment definitions, assumptions are made for external declarations depending on where the declarations are given. See Section 8.2 for details. With simplified segment directives, external declarations always have the size specified in the declaration. Therefore, they can be declared anywhere. For example, external data can be declared from within a code segment or external code can be declared from within a data segment.

5.1.7 Default Segment Names

If you use the simplified segment directives by themselves, you do not need to know the names assigned for each segment. However, it is possible to mix full segment definitions with simplified segment definitions. Therefore, advanced programmers may wish to know the actual names assigned to all segments.

Table 5.1 shows the default segment names created by each directive.

Table 5.1

Default Segments and Types for Standard Memory Models

Model	Directive	Name	Align	Combine	Class	Group
Small	.CODE	_TEXT	BYTE	PUBLIC	'CODE'	
	.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
	.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
	.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
	.STACK	STACK	PARA	STACK	'STACK'	DGROUP
Medium	.CODE	*name*_TEXT	BYTE	PUBLIC	'CODE'	
	.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
	.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
	.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
	.STACK	STACK	PARA	STACK	'STACK'	DGROUP
Compact	.CODE	_TEXT	BYTE	PUBLIC	'CODE'	
	.FARDATA	FAR_DATA	PARA	private	'FAR_DATA'	
	.FARDATA?	FAR_BSS	PARA	private	'FAR_BSS'	
	.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
	.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
	.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
	.STACK	STACK	PARA	STACK	'STACK'	DGROUP
Large	.CODE	*name*_TEXT	BYTE	PUBLIC	'CODE'	
	.FARDATA	FAR_DATA	PARA	private	'FAR_DATA'	
	.FARDATA?	FAR_BSS	PARA	private	'FAR_BSS'	
	.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP

.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
.STACK	STACK	PARA	STACK	'STACK'	DGROUP

The **name** used as part of far code segment names is the file name of the module. The default name associated with the **.CODE** directive can be overridden in medium and large model. The default names for the **.FARDATA** and **.FARDATA?** directives can always be overridden.

The segment and group table at the end of listings always shows the actual segment names.

80386 Only

For 32-bit data and code segments, the alignment is **DWORD** for all segments.

■ Example 1

```
            EXTRN   xvariable:WORD
            EXTRN   xprocedure:NEAR
DGROUP      GROUP   _DATA,_BSS
            ASSUME  cs:_TEXT,ds:DGROUP,ss:DGROUP,es:NOTHING
_TEXT       SEGMENT BYTE PUBLIC 'CODE'
start:      mov     ax,DGROUP
            mov     ds,ax
            .
            .
            .
_TEXT       ENDS
_DATA       SEGMENT WORD PUBLIC 'DATA'
ivariable   DB      5
iarray      DW      50 DUP (5)
_DATA       ENDS
CONST       SEGMENT WORD PUBLIC 'CONST'
string      DB      "This is a string"
CONST       ENDS
_BSS        SEGMENT WORD PUBLIC 'BSS'
uvariable   DB      1 DUP (?)
uarray      DW      50 DUP (?)
_BSS        ENDS
STACK       SEGMENT PARA STACK 'STACK'
            DB      100h DUP (?)
STACK       ENDS
            END     start
```

This example is equivalent to Example 1 in Section 5.1.4. Note that the segment order is different in this version. The **DOSSEG** directive automatically specifies the order shown here. The external variables are declared at the start of the source code. With simplified segment directives, they should be declared in the appropriate segment.

■ **Example 2**

```
DGROUP      GROUP   STRINGS
STUFF       SEGMENT WORD 'BSS'
array       DW      10 DUP (?)
STUFF       ENDS
CONST       SEGMENT WORD PUBLIC 'CONST'
string1     DB      "This is a string."
CONST       ENDS
ACTION      SEGMENT WORD PUBLIC 'CODE'
task        PROC    FAR
            .
            .
            .
            ret
task        ENDP
ACTION      ENDS
            END
```

This example is equivalent to Example 2 in Section 5.1.4. Notice that the segment order is the same in both versions (though a **DOSSEG** directive in the main module could change it). The constant data segment is placed in **DGROUP**, but the far data segment is not.

5.2 Full Segment Definitions

If you need complete control over segments, you may want to give complete segment definitions. This section explains all aspects of segment definitions, including how to order segments and how to define all the segment types.

5.2.1 Setting the Segment-Order Method

The order in which **MASM** writes segments to the object file can be either sequential or alphabetical. If the sequential method is specified, segments will be written in the order in which they appear in the source code. If the alphabetical method is specified, segments will be written in the alphabetical order of their segment names.

The default is sequential. If no segment-order directive or option is given, segments will be ordered sequentially. The segment-order method is only one factor in determining the final order of segments in memory. Class types can also affect segment order, as described in Section 5.2.2.4.

The ordering method can be set using the **.ALPHA** or **.SEQ** directive in the source code. The method can also be set using the **/S** (sequential) or **/A** (alphabetical) assembler options (see Section 2.4.1). The directives have precedence over the options. For example, if the source code contains the **.ALPHA** directive, but the **/S** option is given on the command line, the segments will be ordered alphabetically.

Changing the segment order is an advanced technique. In most cases you can simply leave the default sequential order in effect. If you are linking with high-level language code, the compiler automatically sets the segment order. The **DOSSEG** directive also overrides any segment-order directives or options.

Note

Some previous versions of the IBM Macro Assembler ordered segments alphabetically by default. If you have trouble assembling and linking source-code listings from books or magazines, try using the **/A** option. Listings written for previous IBM versions of the assembler may not work without this option.

■ **Example 1**

```
            .SEQ
DATA        SEGMENT WORD PUBLIC 'DATA'
DATA        ENDS
CODE        SEGMENT BYTE PUBLIC 'CODE'
CODE        ENDS
```

■ **Example 2**

```
              .ALPHA
DATA          SEGMENT WORD PUBLIC 'DATA'
DATA          ENDS
CODE          SEGMENT BYTE PUBLIC 'CODE'
CODE          ENDS
```

In Example 1, the DATA segment is written to the object file first because it appears first in the source code. In example 2, the CODE segment is written to the object file first because its name comes first alphabetically.

5.2.2 Defining Segments

The beginning of a program segment is defined with the **SEGMENT** directive, and the end of the segment is defined with the **ENDS** directive.

■ **Syntax**

name **SEGMENT** ⟦*align*⟧ ⟦*use*⟧ ⟦*combine*⟧ ⟦'*class*'⟧

.

.

.

name **ENDS**

The *name* defines the name of the segment. This name can be unique or it can be the same name given to other segments in the program. Segments with identical names are treated as the same segment. For example, if it is convenient to put different portions of a single segment in different source modules, the segment is given the same name in both modules.

The optional *align*, *use*, *combine*, and *class* types give the linker and the assembler instructions on how to set up and combine segments. Types should be specified in order, but it is not necessary to enter all types, or any type, for a given segment.

Defining segment types is an advanced technique. Beginning assembly-language programmers are better off using the simplified segment directives discussed in Section 5.1.

Note

Don't confuse the **BYTE**, **WORD**, and **DWORD** align types with the **BYTE**, **WORD**, and **DWORD** reserved names used to specify data type with the **LABEL** directive and the **PTR** and **THIS** operators. Also, the **PAGE** align type and the **PUBLIC** combine type should not be confused with the **PAGE** and **PUBLIC** directives. The distinction should be clear from context since the align and combine types are only used on the same line as the **SEGMENT** directive.

Segment types have no effect on programs prepared in the **.COM** format. Since there is only one segment, there is no need to specify how segments are combined or ordered.

5.2.2.1 Controlling Alignment with Align Type

The optional *align* type defines the range of memory addresses from which a starting address for the segment can be selected. The align type can be any one of the following:

Align Type	Meaning
BYTE	Use the next available byte address
WORD	Use the next available word address (2 bytes per word)
DWORD	Use the next available double-word address (4 bytes per double word); the **DWORD** align type is normally used 32-bit segments with the 80386
PARA	Use the next available paragraph address (16 bytes per paragraph)
PAGE	Use the next available page address (256 bytes per page)

If no *align* type is given, **PARA** is used by default.

The linker uses the alignment information to determine the relative start address for each segment. DOS uses the information to calculate the actual start address when the program is loaded.

Align types are illustrated in Figure 5.1.

5.2.2.2 Setting Segment Size with Use Type

■ **80386 Processor Only**

The *use* type specifies the segment size on the 80386 processor. It is only relevant if you have enabled 80386 instructions and addressing modes with the **.386** directive. The assembler will ignore the *use* type if the 80386 processor is not enabled.

With the 80286 and other 16-bit processors, the segment size is always 16 bits. A 16-bit segment can contain up to 65,536 (64K) bytes. However, the 80386 is capable of using either 16-bit or 32-bit segments. A 32-bit segment can contain up to 4,294,967,296 bytes (4 gigabytes).

If you do not specify a *use* type, segments will be 32 bits wide by default when the **.386** directive is used.

The segment size you specify for the code segment changes the effect of addressing modes. See Section 14.3.3 for more information on 80386 addressing modes.

Note

Although the assembler allows you to use 16-bit and 32-bit segments in the same program, you should normally make all segments the same size. Mixing segment sizes is an advanced technique that can have unexpected side effects. It is normally used only by systems programmers.

■ **Example 1**

```
; 16-bit segment
            .386
_DATA       SEGMENT WORD USE16 PUBLIC 'DATA'
            .
            .
            .
_DATA       ENDS
```

■ Example 2

```
; 32-bit segment
_TEXT       SEGMENT DWORD USE32 PUBLIC 'CODE'
            .
            .
            .
_TEXT       ENDS
```

5.2.2.3 Defining Segment Combinations with Combine Type

The optional *combine* type defines how to combine segments having the same name. The combine type can be any one of the following:

Combine Type	Meaning
PUBLIC	Concatenates all segments having the same name to form a single, contiguous segment. All instruction and data addresses in the new segment are relative to a single segment register, and all offsets are adjusted to represent the distance from the beginning of the new segment.
STACK	Concatenates all segments having the same name to form a single, contiguous segment. This combine type is the same as the **PUBLIC** combine type, except that all addresses in the new segment are relative to the **SS** segment register. The stack pointer (**SP**) register is initialized to the ending address of the segment. Stack segments should normally use the **STACK** type, since this automatically initializes the **SS** register. If you create a stack segment and do not use the **STACK** type, you must give instructions to load the segment address into the **SS** register.
COMMON	Creates overlapping segments by placing the start of all segments having the same name at the same address. The length of the resulting area is the length of the longest segment. All addresses in the segments are relative to the same base address. If variables are initialized in more than one segment having the same name and **COMMON** type, the most recently initialized data replace any previously initialized data.

MEMORY Concatenates all segments having the same name to form a single, contiguous segment. The Microsoft 8086 Overlay Linker treats **MEMORY** segments exactly the same as **PUBLIC** segments. **MASM** allows you to use **MEMORY** type even though **LINK** does not recognize a separate **MEMORY** type. This feature is provided for compatibility with other linkers that may support a combine type conforming to the Intel definition of **MEMORY** type.

AT *address* Causes all label and variable addresses defined in the segment to be relative to *address*. The *address* can be any valid expression, but must not contain a forward reference—that is, a reference to a symbol defined later in the source file. An **AT** segment typically contains no code or initialized data. Instead, it represents an address template that can be placed over code or data already in memory, such as a screen buffer or other absolute memory location defined by hardware. The labels or variables in the **AT** segments can then be used to access fixed instructions or data.

The **AT** combine type has no meaning in protected-mode programs, since the segment represents a movable selector rather than a physical address. Real-mode programs that use **AT** segments must be modified before they can be used in protected mode. Future multitasking versions of DOS will provide DOS calls for doing tasks that are often done by manipulating memory directly under current versions of DOS.

If no *combine* type is given, the segment has **PRIVATE** type. Segments having the same name are not combined. Instead, each segment receives its own physical segment when loaded into memory.

Notes

Although a given segment name can be used more than once in a source file, each segment definition using that name must have either exactly the same attributes, or attributes that do not conflict. If a segment is defined once with all types defined, then subsequent definitions for that segment need not specify any types.

Normally you should provide at least one stack segment (having **STACK** combine type) in a program. If no stack segment is declared, **LINK** will display a warning message. You can ignore this message if you have a specific reason for not declaring a stack segment. For example, you would not have a separate stack segment in a program in the **.COM** format.

■ Example

The following source code shell illustrates one way in which the *combine* and *align* types can be used. Figure 3.1 shows the way **LINK** would load the example program into memory.

```
            NAME module_1

SEG_A       SEGMENT BYTE PUBLIC 'CODE'
start:      .
            .
            .
SEG_A       ENDS

SEG_B       SEGMENT WORD COMMON 'DATA'
            .
            .
            .
SEG_B       ENDS

SEG_C       SEGMENT PARA STACK 'STACK'
            .
            .
            .
SEG_C       ENDS

SEG_D       SEGMENT AT 0B800H
            .
            .
            .
SEG_D       ENDS
            END start

            NAME module_2

SEG_A       SEGMENT BYTE PUBLIC 'CODE'
            .
            .
            .
```

```
SEG_A       ENDS

SEG_B       SEGMENT WORD COMMON 'DATA'
            .
            .
            .
SEG_B       ENDS
```

```
High memory
                              .
                              .
                              .
                              +-----------+
                              |           |
                              |           | --- SEG_D   SEGMENT AT 0B800h
                              |           |
0B800h                     ---+-----------+
                              |
                              .
                              .
                              .
                              |
                              +-----------+
                              |           |
                              |           | --- SEG_C   SEGMENT PARA STACK 'STACK'
First available               |           |
paragraph address           --+-----------+     SS initialized to this address
after seg_b                   |
                              +-----------+
                              |           |
                              |           | --- SEG_B   SEGMENT WORD COMMON 'DATA'
                              |           |     in module 2
                              +-------------+
                              |           | |
                              |           | | - SEG_B   SEGMENT WORD COMMON 'DATA'
First available               |           | |   in module 1
word address               ---+-----------+-+
after seg_a                   |
                              +-----------+
                              |           |
                              |           | --- SEG_A   SEGMENT BYTE PUBLIC 'CODE'
                              |           |     in module 2
                              +-----------+
                              |           |
                              |           | --- SEG_A   SEGMENT BYTE PUBLIC 'CODE'
First available               |           |     in module 1
byte address               ---+-----------+
                              |
                              .
                              .             --- DOS
                              .
Low memory
```

5.2.2.4 Controlling Segment Structure with Class Type

Class type can be used to control segment order and to identify the code segment.

The *class* name must be enclosed in single quotation marks ('). Class names are not case sensitive unless the **/ML** or **/MX** option is used during assembly.

All segments belong to a class. Segments for which no class name is explicitly stated have the null class name. **LINK** imposes no restriction on the number or size of segments in a class. The total size of all segments in a class can exceed 64K.

Note

The names assigned for class types of segments should not be used for other symbol definitions in the source file. For example, if you give a segment the class name `'CONSTANT'`, you should not give the name `constant` to variables or labels in the source file. If you do, the error `Symbol already different kind` will be generated.

The linker expects segments having the class name `'CODE'` or a name with the suffix `'CODE'` to contain program code. You should always give this class name to segments containing code.

The CodeView debugger also expects code segments to have the class name `'CODE'`. If you fail to assign a class type to a code segment, or if you give it a class type other than `'CODE'`, then labels may not be properly aligned for symbolic debugging.

Class type is one of the factors that control the final order of segments in an executable file. The other factor is either the order of the segments in the source file (with the **/S** option or the **.SEQ** directive) or the alphabetical order of segments (with the **/A** option or the **.ALPHA** directive).

These two factors control different internal behavior, but both affect final order of segments in the executable file. The sequential or alphabetical order of segments in the source file determines the order in which the assembler writes segments to the object file. The class type can affect the order in which the linker writes segments from object files to the executable file.

Segments having the same class type are loaded into memory together, regardless of their sequential or alphabetical order in the source file.

■ Example

```
A_SEG   SEGMENT 'SEG_1'
A_SEG   ENDS

B_SEG   SEGMENT 'SEG_2'
B_SEG   ENDS

C_SEG   SEGMENT 'SEG_1'
C_SEG   ENDS
```

When **MASM** assembles the preceding program fragment, it writes the segments to the object file in sequential or alphabetical order, depending on whether the **/A** option or **.ALPHA** directive was used. In this case, the sequential and alphabetical order are the same, so the order will be A_SEG, B_SEG, C_SEG in either case.

When the linker writes the segments to the executable file, it first checks to see if any segments have the same class type. If they do, it writes them to the executable file together. Thus A_SEG and C_SEG will be placed together because they both have class type 'SEG_1'. The final order in memory will be A_SEG, C_SEG, B_SEG.

Since **LINK** processes modules in the order in which it receives them on the command line, you may not always be able to easily specify the order in which you want segments to be loaded. For example, assume your program has four segments that you want loaded in the following order: _TEXT, _DATA, CONST, and STACK.

The _TEXT, CONST, and STACK segments are defined in the first module of your program, but the _DATA segment is defined in the second module. **LINK** will not put the segments in the proper order because it will first load the segments encountered in the first module.

You can avoid this problem by starting your program with dummy segment definitions in the order in which you wish to load your real segments. The dummy segments can either go at the start of the first module, or they can be placed in a separate include file that is called at the start of the first module. You can then put the actual segment definitions in any order or any module you find convenient.

For example, you might call the following include file at the start of the first module of your program:

```
_TEXT       SEGMENT BYTE PUBLIC 'CODE'
_TEXT       ENDS
```

```
_DATA       SEGMENT WORD PUBLIC 'DATA'
_DATA       ENDS
CONST       SEGMENT WORD PUBLIC 'CONST'
CONST       ENDS
STACK       SEGMENT PARA STACK 'STACK'
STACK       ENDS
```

If the initial dummy segments do not define all classes to be used in your program, **LINK** will choose a default loading order that may not correspond to the order you desire.

5.3 Defining Segment Groups

A group is a collection of segments that are all associated with the same starting address. You may wish to use a group if you want several types of data to be organized in separate segments in your source code, but you want them all to be accessible from a single, common segment register at run time.

■ Syntax

name **GROUP** *segmentname* ⟦*,segmentname...*⟧

The *name* is the symbol assigned to the starting address of the group. All labels and variables defined within the segments of the group are relative to the start of the group, rather than to the start of the segments in which they are defined.

The *segmentname* can be any previously defined segment or a **SEG** expression (see Section 9.2.4.5).

Segments can be added to a group one at a time. For example, you can define a segment, add it to a group, define another segment, add it to the same group, and so on. This is a new feature of Version 4.5. Previous versions required that all segments in a group be defined at one time.

The **GROUP** directive does not affect the order in which segments of a group are loaded. Loading order depends on each segment's class, or on the order in which object modules are given to the linker.

Segments in a group need not be contiguous. Segments that do not belong to the group can be loaded between segments that do. The only restriction is that the distance (in bytes) between the first byte in the first segment of the group and the last byte in the last segment must not exceed 65,535. Therefore, if the segments of a group are contiguous, the group can occupy up to 64K of memory.

Group names can be used with the **ASSUME** directive (discussed in Section 3.7) and as an operand prefix with the segment override operator (discussed in Section 5.3.7).

■ Example

```
DGROUP      GROUP   ASEG,CSEG
            ASSUME  ds:DGROUP

ASEG        SEGMENT WORD PUBLIC 'DATA'
            .
sym_a       .
            .
ASEG        ENDS

BSEG        SEGMENT WORD PUBLIC 'DATA'
            .
sym_b       .
            .
BSEG        ENDS

CSEG        SEGMENT WORD PUBLIC 'DATA'
            .
sym_c       .
            .
CSEG        ENDS
            END
```

Figure 5.1 shows the order of the example segments in memory. They are loaded in the order in which they appear in the source code (or in alphabetical order if the **.ALPHA** directive or **/A** option is specified).

Since ASEG and CSEG are declared part of the same group, they have the same base despite their separation in memory. This means that the symbols sym_a and sym_c have offsets from the beginning of the group, which is also the beginning of ASEG. The offset of sym_b is from the beginning of BSEG, since it is not part of the group. This sample illustrates the way **LINK** organizes segments in a group. It is not intended as a typical use of a group.

```
High memory
                     .
                     .
                     .
                     +----------+
                     |          |
            ---------| - sym_c  |
              ^      |          | --- CSEG    SEGMENT WORD PUBLIC 'DATA'
              |      |          |             (Part of DGROUP)
offset        |      |          |
sym_c         |------+----------+
              |      +----------+
              |      |          |
       -------+------| - sym_b  |
         ^    |      |          | --- BSEG    SEGMENT WORD PUBLIC 'DATA'
offset   |    |      |          |             (Not part of DGROUP)
sym_b    v    |      |          |
       -------+------+----------+
              |      +----------+
              |      |          |
       -------+------| - sym_a  |
         ^    |      |          | --- ASEG    SEGMENT WORD PUBLIC 'DATA'
offset   |    |      |          |             (Part of DGROUP)
sym_a    v    v      |          |
       --------------+----------+     Base of DGROUP
                     .
                     .
                     .
Low memory
```

5.4 Associating Segments with Registers

Many instructions assume a default segment. For example, **JMP** instructions assume the segment associated with the **CS** register, **PUSH** and **POP** instructions assume the segment associated with the **SS** register, and **MOV** instructions assume the segment associated with the **DS** register.

When the assembler needs to reference an address, it must know what segment the address is in. It does this by using default segment or group addresses assigned with the **ASSUME** directive.

Note

Using the **ASSUME** directive to tell the assembler which segment to associate with a segment is not the same as telling the processor. The **ASSUME** directive only affects assembly-time assumptions. You may need to use instructions to change run-time assumptions. Initializing segment registers at run time is discussed in Section 5.5.

■ Syntax

ASSUME *segmentregister***:***name* [[,*segmentregister***:***name*...]]
ASSUME NOTHING

The *name* must be the name of the segment or group that is to be associated with the *segmentregister*. Subsequent instructions that assume a default register for referencing labels or variables will automatically assume that if the default segment is *segmentregister*, then the label or variable will be in the *name* segment or group.

The **ASSUME** directive can define a segment for each of the segment registers. The *segmentregister* can be **CS**, **DS**, **ES**, or **SS** (**FS** and **GS** are also available on the 80386). The *name* must be one of the following:

- The name of a segment defined in the source file with the **SEGMENT** directive
- The name of a group defined in the source file with the **GROUP** directive
- The keyword **NOTHING**

The keyword **NOTHING** cancels the current segment selection. The statement `ASSUME NOTHING` cancels all register selections made by a previous **ASSUME** statement.

Usually a single **ASSUME** statement defines all four segment registers at the start of the source file. However, you can use the **ASSUME** directive at any point to change segment assumptions.

Using the **ASSUME** directive to change segment assumptions is often equivalent to changing assumptions with the segment-override operator (see Section 9.2.3). The segment-override operator is more convenient for one-time overrides, while the **ASSUME** directive may be more convenient if previous assumptions must be overridden for a sequence of instructions.

■ Example

```
DATA1       SEGMENT WORD PUBLIC 'DATA'
d1          DW      ?
DATA1       ENDS

DATA2       SEGMENT WORD PUBLIC 'DATA'
d2          DW      ?
DATA2       ENDS

            ASSUME  cs:CODE, ds:DATA1, es:DATA2
CODE        SEGMENT BYTE PUBLIC 'CODE'
            .
            .
            .

; Method 1 for series of instructions that need override
; Use segment override for each instruction
            mov     es:d2,ax
            .
            .
            .
            mov     bx,es:d2

; Method 2 for series of instructions that need override
; Use ASSUME at beginning and end of series of instructions
            ASSUME  ds:DATA2
            mov     d2,ax
            .
            .
            .
            mov     bx,d2
            ASSUME  ds:DATA1
```

5.5 Initializing Segment Registers

Assembly-language programs must initialize segment values for each segment register before instructions that reference the segment register can be used in the source program.

Initializing segment registers is different than assigning default values for segment registers with the **ASSUME** statement. The **ASSUME** directive tells the assembler what segments to use at assembly time. Initializing segments gives them an initial value that will be used at run time.

Each of the segment registers is initialized in a different way.

5.5.1 Initializing the CS and IP Registers

The **CS** and **IP** registers are initialized by specifying a starting address with the **END** directive.

■ **Syntax**

END [[*startaddress*]]

The *startaddress* is a label or expression identifying the address where you want execution to begin when the program is loaded. Normally a label for the *startaddress* should be placed at the address of the first instruction in the code segment.

The **CS** segment is initialized to the value of *startaddress*. The **IP** segment is normally initialized to 0. You can change the initial value of the **IP** register by specifying using the **ORG** directive (see Section 6.3) just before the *startaddress* label. For example, programs in the **.COM** format use **ORG 100h** to initialize the **IP** register to 256 (100 hexadecimal).

If a program consists of a single source module, then the *startaddress* is required for that module. If a program has several modules, all modules must terminate with an **END** directive, but only one of them can define a *startaddress*.

Warning

One, and only one, module must define a *startaddress*. If you do not specify a *startaddress*, none will be assumed. Neither **MASM** nor **LINK** will generate an error message, but your program will attempt to start execution at the wrong address.

■ **Example**

```
; Module 1
            EXTRN   task:NEAR
            .TEXT
start:      .               ; First executable instruction
            .
            .
            call    task
            .
```

```
            .
            .
            END     start

; Module 2
            PUBLIC  task
            .TEXT
task        PROC
            .
            .
            .
task        ENDP
            END
```

If Module 1 and Module 2 are linked into a single program, it is essential that only the calling module define a starting address.

5.5.2 Initializing the DS Register

The **DS** register must be initialized to the address of the segment that will be used for data.

The address of the segment or group that will be the initial data segment must be loaded into the **DS** register. This is done in two statements because a memory value cannot be loaded directly into a segment register. The segment setup lines typically appear at the start or very near the start of the code segment.

■ Example 1

```
_DATA       SEGMENT WORD PUBLIC 'DATA'
            .
            .
            .
_DATA       ENDS
_TEXT       SEGMENT BYTE PUBLIC 'CODE'
            ASSUME  cs:_TEXT,ds_DATA
start:      mov     ax,_DATA            ; Load start of data segment
            mov     ds,ax               ; Transfer to DS register
            .
            .
            .
_TEXT       ENDS
            END     start
```

If you are using the Microsoft naming conventions and segment order (for example, with simplified segment directives), the address loaded into the **DS** register for small and medium memory models will not be a segment address, but the address of **DGROUP**, as shown in Example 2.

■ **Example 2**

```
            .MODEL  SMALL,DOSSEG
            .DATA
            .
            .
            .
            .CODE
start:      mov     ax,DGROUP           ; Load start of DGROUP
            mov     ds,ax               ; Transfer to DS register
            .
            .
            .
            END     start
```

5.5.3 Initializing the SS and SP Registers

The **SS** register is automatically initialized to the value of the last segment in the source code having combine type **STACK**. The **SP** register is automatically initialized to the end of the same segment. This is the normal method of setting up a stack; it is used automatically with the simplified segment directives.

However, you can initialize or reinitialize the stack segment directly by changing the values of **SS** and **SP**.

■ **Example**

```
STACK1      SEGMENT PARA STACK 'STACK'
            DB      100h
stacktop1   EQU     $
STACK1      ENDS

STACK2      SEGMENT PARA PUBLIC 'STACK'
            DB      200h
stacktop2   EQU     $
STACK2      ENDS

_TEXT       SEGMENT BYTE PUBLIC 'CODE'
            ASSUME  cs:_TEXT,ss:_STACK1
; SS and SP automatically initialized to start and end of STACK1
            .
            .
            .
; SS and SP reinitialized to start and end of STACK2
            ASSUME  ss:STACK2           ; Tell the assembler
            mov     ax,STACK2           ; Tell the processor
            mov     ss,ax
            mov     sp,stacktop2
            .
            .
            .
```

5.5.4 Initializing the ES Register

The **ES** register is not automatically initialized. If your program uses the **ES** register, you must initialize it by moving the appropriate segment value into the register.

```
EXTRA       SEGMENT WORD PUBLIC 'DATA'
            .
            .
            .
EXTRA       ENDS
_DATA       SEGMENT WORD PUBLIC 'DATA'
            .
            .
            .
_DATA       ENDS
_TEXT       SEGMENT BYTE PUBLIC 'CODE'
            ASSUME  cs:_TEXT,ds:_DATA,es:EXTRA
            mov     ax,_DATA            ; Iniitial
            mov     ds,ax
            mov     ax,EXTRA
            mov     es,ax
            .
            .
            .
```

5.6 Nesting Segments

Segments can be nested. When **MASM** encounters a nested segment, it temporarily suspends assembly of the enclosing segment and begins assembly of the nested segment. When the nested segment has been assembled, **MASM** continues assembly of the enclosing segment.

Nested segments, where **MASM** remembers and reopens the outer segment, are only possible with full segment definitions. However, similar segment structures can be created using simplified segment directives.

■ Example 1

```
; Macro to print message on the screen
; Uses full segment definitions - segments nested

message     MACRO   text
            LOCAL   symbol
_DATA       SEGMENT WORD PUBLIC 'DATA'
symbol      DB      &text
            DB      13,10,"$"
_DATA       ENDS
```

```
            mov     ah,09h
            mov     dx,OFFSET symbol
            int     21h
            ENDM

_TEXT       SEGMENT BYTE PUBLIC 'CODE'
            .
            .
            .
            message "Please insert disk"
```

In the example above a macro called from inside the code segment (_TEXT) allocates a variable within a nested data segment (_DATA). This has the effect of allocating more variable space onto the end of the data segment each time the macro is called. The macro can be used for messages that will appear only once in the source code.

■ Example 2

```
; Macro to print message on the screen
; Uses simplified segment directives - segments not nested

message     MACRO   text
            LOCAL   symbol
            IF      @fardata
            .FARDATA
            ELSE
            .DATA
            ENDIF
symbol      DB      &text
            DB      13,10,"$"
            .CODE
            mov     ah,09h
            mov     dx,OFFSET symbol
            int     21h
            ENDM

            .CODE
            .
            .
            .
            message "Please insert disk"
```

Although Example 2 has the same effect as Example 1, **MASM** handles the two macros differently. In Example 1, assembly of the outer (code) segment is suspended rather than terminated. In Example 2, assembly of the code segment terminates, assembly of the data segment starts and terminates, then assembly of the code segment is restarted.

Chapter 6
Defining Labels and Variables

This chapter explains how to define labels, variables, and other symbols that refer to instruction and data locations within segments.

The label and variable definition directives described in this chapter are closely related to the segment definition directives described in Chapter 5, "Defining Segment Structure." Segment directives assign the addresses for segments. The variable and label definition directives assign offset addresses within segments.

The assembler assigns offset addresses for each segment by keeping track of a value called the location counter. The location counter is incremented as each source statement is processed, so that it always contains the address of the location being assembled. When a label or a variable name is encountered, the current value of the location counter is assigned to the symbol.

This chapter tells how to assign labels and most kinds of variables. (Multifield variables such as structures and records are discussed in Chapter 7, "Using Structures and Records"). The chapter also discusses related directives, including those that control the location counter directly.

6.1 Defining Code Labels

Code labels give symbolic names to the addresses of instructions in the code segment. These labels can be used as the operands to jump, call, and loop instructions to transfer program control to a new instruction. There are three types of code labels: near labels, procedure labels, and labels created with the **LABEL** directive.

6.1.1 Defining Near Code Labels

Near-label definitions create instruction labels that have **NEAR** type. Such labels can be used to access the address of the label from other statements.

■ **Syntax**

name:

The *name* must not be previously defined in the module and it must be followed by a colon (:). Furthermore, the segment containing the definition must be the one the assembler currently associates with the **CS** register. The **ASSUME** directive is used to associate a segment with a segment register, as described in Section 5.4.

The assembler assigns the label an address by setting it equal to the value of the location counter at the statement where the label is encounters. A near label can appear on a line by itself or on a line with an instruction.

The same label name can be used in different modules as long as each label is only referenced by instructions in its own module. If a label must be referenced by instructions in another module, it must be given a unique name and declared with the **PUBLIC** and **EXTRN** directives, as described in Chapter 8, "Creating Programs from Multiple Modules."

■ **Examples**

```
            cmp     ax,5          ; Compare with 5
            ja      bigger
            jb      smaller
            .                     ; Instructions if AX=5
            .
            .
            jmp     done
bigger:     .                    ; Instructions if AX > 5
            .
            .
            jmp     done
smaller:    .                    ; Instructions if AX < 5
            .
            .
done:
```

6.1.2 Defining Procedure Labels

The start of an assembly-language procedure can be defined with the **PROC** directive, and the end of the procedure can be defined with the **ENDP** directive.

■ Syntax

name **PROC** ⟦*distance*⟧

.

.

.

name **ENDP**

The *name* assigns a symbol to the procedure. The *distance* can be **NEAR** or **FAR**. Any **RET** instructions within the procedure will automatically have the same *distance* (**NEAR** or **FAR**) as the procedure.

The **ENDP** directive labels the address where the procedure ends. Every procedure label must have a matching **ENDP** label to mark the end of the procedure. **MASM** generates an error message if it does not find an **ENDP** directive to match each **PROC** directive.

When the **PROC** label definition is encountered, the assembler sets the label's value to the current value of the location counter and sets its type to **NEAR** or **FAR**. If the label has **FAR** type, the assembler also sets its segment value to that of the enclosing segment. If you have specified full segment definitions, the default *distance* is **NEAR**. If you are using simplified segment definitions, the default *distance* is the distance associated with the declared memory model: **NEAR** for small and compact models or **FAR** for medium and large models.

The procedure label can be used in a **CALL** instruction to direct execution control to the first instruction of the procedure. Control can be transferred to a **NEAR** procedure label from any address in the same segment as the label. Control can be transferred to a **FAR** procedure label from any segment.

Procedure labels must be declared with the **PUBLIC** and **EXTRN** directives if they are located in one module but called from another module, as described in Chapter 8, "Creating Programs from Multiple Modules."

Procedures are discussed in more detail in Section 17.4.

■ Examples

```
            call    task        ; Call procedure
            .
            .
            .
task        PROC    NEAR        ; Start of procedure
```

```
            .
            .
            .
            ret
task        ENDP                   ; End of procedure
```

6.1.3 Defining Code Labels with the LABEL Directive

The **LABEL** directive provides an alternative method of defining code labels. Unlike labels defined with the **PROC** and **ENDP** directives, these labels do not require a closing statement.

■ **Syntax**

name **LABEL** *distance*

The *name* is the symbol name assigned to the label. The *distance* can be a type specifier such as **NEAR**, **FAR**, or **PROC**. You can use the **LABEL** directive to define a **FAR** code label or to define a second entry point into a procedure.

■ **Example**

```
distant     LABEL  FAR         ; Example 1
                               ; Can be accessed by jump or loop
                               ;   from another segment

task        PROC   FAR         ; Example 2 -- Main entry point
            .
            .
            .
task1       LABEL  FAR         ; Secondary entry point
            .
            .
            .
            ret
task        ENDP               ; End of procedure
```

6.2 Defining and Initializing Data

The data-definition directives enable you to allocate memory for data. At the same time, you can specify the initial values for the allocated data. Data can be specified as numbers, strings, or expressions that evaluate to constants. The assembler translates these constant values into binary bytes, words, or other units of data. The encoded data are written to the object file at assembly time.

6.2.1 Defining Variables

Variables consist of one or more named data objects of a specified size.

■ Syntax

⟦*name*⟧ ***directive initializer*** ⟦***,initializer...***⟧

The *name* is the symbol name assigned to the variable. If no *name* is assigned, the data is allocated, but the starting address of the variable has no symbolic name.

The size of the variable is determined by *directive.* The directives that can be used to define single-item data objects are listed below:

Directive	Meaning
DB	Defines byte
DW	Defines word (2 bytes)
DD	Defines doubleword (4 bytes)
DF	Defines farword (6 bytes); normally used only with 80386 processor
DQ	Defines quadword (8 bytes)
DT	Defines 10-byte variable

The optional *initializer* can be a constant, an expression that evaluates to a constant, or a question mark (**?**). The question mark is the indeterminate symbol, which indicates that the value of the variable is undefined. You can define multiple values using multiple initializers separated by commas, or you can use the **DUP** operator as explained in Section 6.2.2.

Simple data types can allocate memory for integers, strings, addresses, or real numbers.

6.2.1.1 Defining Integer Variables

When defining an integer variable, you can specify an initial value as an integer constant or as a constant expression. **MASM** will generate an error if you specify an initial value that is too large for the specified variable.

Integer values for all sizes except 10-byte variables are stored in the binary two's complement format. They can be interpreted as either signed or unsigned numbers. The same value represents two different values. For example, the hexadecimal value 0FFCD can be interpreted either as the signed number –51 or the unsigned number 65,485.

The processor cannot tell the difference between signed and unsigned numbers. Some instructions are designed specifically for signed numbers. It is the programmer's responsibility to decide whether a value is to be interpreted as signed or unsigned, and to use the appropriate instructions to handle the value correctly.

The directives for defining integer variables are listed below with the sizes of integer they can define:

Directive	**Size**
DB (bytes)	Allocates unsigned numbers from 0 to 255 or signed numbers from –128 to 127. These values can be used directly in 8086-family instructions.
DW (words)	Allocates unsigned numbers from 0 to 65,535 or signed numbers from –32,768 to 32,787. These values can be used directly in 8086-family instructions. They can also be loaded, used in calculations, and stored with 8087-family instructions.
DD (doublewords)	Allocates unsigned numbers from 0 to 4,294,967,295 or signed numbers from -2,147,483,648 to 2,147,483,647. These 32-bit values (called short integers) can be loaded, used in calculations, and stored with 8087-family instructions. Some calculations can be done on these numbers directly with 16-bit 8086-family

processors, while others involve an indirect method called "bit splicing" (see Section 16.1). They can be used directly without bit splicing in calculations with the 80386 processor.

DF (farwords)
Allocates six-byte (48-bit) integers. These values are normally only used as pointer variables on the 80386 processor (see Section 6.2.1.4).

DQ (quadwords)
Allocates long (64-bit) integers. These values can be loaded, used in calculations, and stored with 8087-family instructions. You must write your own routines to use them with 16-bit 8086-family processors. Some calculations can be done on these numbers directly with the 80386 processor, while others involve bit splicing.

DT
Allocates 10-byte (80-bit) integers if the **D** radix specifier is used. By default, **DT** allocates packed BCD numbers, as described in Section 6.2.1.2. If you define binary 10-byte integers, you must write your own routines to use them in calculations.

■ Example

```
integer     DB     16            ; Initialize byte to 16
expression  DW     4*3           ; Initialize word to 12
empty       DQ     ?             ; Allocate uninitialized long integer
            DB     1,2,3,4,5,6   ; Initialize six unnamed bytes
high_byte   DD     4294967295    ; Initialize double word to 4,294,967,295
tb          DT     2345d         ; Initialize 10-byte binary integer
```

6.2.1.2 Defining Binary Coded Decimal Variables

Binary coded decimals provide a method of doing calculations on large numbers without rounding errors. They are sometimes used in financial applications. There are two kinds: packed and unpacked.

Unpacked BCD numbers are stored one digit to a byte, with the value in the lower four bits. This means that memory is not used very efficiently when storing them. For example, a byte used to store unsigned integers can have any value between 0 and 255, while a byte used to store BCD values can have any value between 0 and 9.

Unpacked BCD variables can be defined with the **DB** directive. For example, an unpacked BCD number could be defined and initialized as shown below:

```
unpacked    DB      1,5,8,2,5,2,9       ; Initialized to 9,252,851
```

Notice that the least significant digits come first. This is a convention that makes calculations with the numbers easier. Calculations with unpacked BCD numbers are discussed in Section 6.2.1.2.

Packed BCD numbers are stored two digits to a byte, with one digit in the lower four bits and one in the upper four bits. The leftmost bit holds the sign (0 for positive or 1 for negative).

Packed BCD variables can be defined with the **DT** directive as shown below:

```
packed      DT      9252851     ; Allocate BCD 9,252,851
```

The 8087-family processors can do fast calculations with packed BCD numbers, as described in Chapter 19, "Calculating with a Math Coprocessor." The 8086-family processors can also do some calculations with packed BCD numbers, but the process is slower and more complicated. See Section 16.5 for details.

6.2.1.3 Defining String Variables

Strings are normally initialized with the **DB** directive. The initializing value is specified as a string constant. Strings can also be initialized by specifying each value in the string. For example, the following definations are equivalent:

```
version1    DB      97,98,99      ; As ASCII values
version2    DB      'a','b','c'   ; As characters
version3    DB      "abc"         ; As a string
```

One- and two-character strings can also be initialized with any of the other data definition directives. The last (or only) character in the string is placed in the byte with the lowest address. Either 0 or the first character is placed in the next byte. The unused portion of such variables will be filled with zeros.

■ Examples

```
function9   DB    'Hello',13,10,'$'    ; Use with DOS INT 21h
                                       ;   function 9

asciiz      DB    "\ASM\TEST.ASM",0    ; Use as ASCIIZ string

message     DB    "Enter file name: "  ; Use with DOS INT 21h
l_message   EQU   $-message            ;   function 40h
a_message   EQU   OFFSET message

str1        DD    "ab"                 ; Stored as 62 61 00 00

str2        DD    "a"                  ; Stored as 61 00 00 00
```

6.2.1.4 Defining Pointer Variables

Pointer variables (or pointers) are variables that contain the address of a data or code object rather than the object itself. The address in the variable "points" to another address. Pointers can be either near addresses or far addresses.

Near pointers consist of the offset portion of the address. They can be initialized in word variables using the **DW** directive. Values in near address variables can be used in situations where the segment portion of the address is known to be the current segment.

Far pointers consist of both the segment and offset portions of the address. They can be initialized in doubleword variables using the **DD** directive. Values in far address variables must be used in situations where the segment portion of the address may be outside the current segment.

■ Examples

```
string      DB    "Text",0      ; Null-terminated string
npstring    DW    string        ; Near pointer to "string"
fpstring    DD    string        ; Far pointer to "string"
```

■ 80386 Processor Only

Pointers are different on the 80386 processor if the **USE32** use type has been specified. In this case the offset portion of an address consists of 32 bits, while the segment portion consists of 16 bits. Therefore a near pointer is 32-bits (a doubleword), while a far pointer is 48-bits (a farword).

■ Example

```
_DATA       SEGMENT WORD USE32 PUBLIC 'DATA'
string      DB     "Text",0     ; Null-terminated string
npstring    DD     string       ; Near (32-bit) pointer to "string"
fpstring    DF     string       ; Far (48-bit) pointer to "string"
_DATA       ENDS
```

6.2.1.5 Defining Real-Number Variables

Real numbers must be stored in binary format. However, when initializing variables, you can specify constants in the real number formats recognized by **MASM**. The assembler automatically encodes the decimal constants into their binary equivalents. This section tells how to initialize real-number variables and explains real number encoding.

Initializing and Allocating Real-Number Variables

Real numbers can be defined by initializing them either with real-number constants or with encoded hexadecimal constants. The real-number designator (**R**) must follow numbers specified in encoded format.

The directives for defining real numbers are listed below with the sizes of the numbers they can allocate:

Directive	Size
DD	Allocates short (32-bit) reals in either the Microsoft Binary Real or IEEE format.
DQ	Allocates long (64-bit) reals in either the Microsoft Binary Real or IEEE format.
DT	Allocates temporary or 10-byte (80-bit) reals. The format of these numbers is similar to the IEEE format. They are always encoded the same regardless of coprocessor directives or the **/R** or **/E** options. Their size is nonstandard and will be incompatible with Microsoft high-level languages. Temporary real format is provided for those who want to intitialize real numbers in the format used internally by 8087-family processors.

The 8086-family microprocessors do not have any instructions for handling real numbers. You must write your own routines, use a library that includes real-number calculation routines, or use a coprocessor. The 8087-family coprocessors can load real numbers in the IEEE format, use the values in calculations, and store the results back to memory, as explained in Chapter 19, "Calculating with a Math Coprocessor."

■ Examples

```
shrt        DD      98.6                    ; MASM automatically encodes
long        DQ      5.391E-4                ;   in current format
ten_byte    DT      -7.31E7

eshrt       DD      87453333r               ; 98.6 encoded in Microsoft
                                            ;   Binary Real format
elong       DQ      3F41AA4C6F445B7Ar       ; 5.391E-4 encoded in
                                            ;   IEEE format
```

Selecting a Real-Number Format

MASM can encode 4- and 8-byte real numbers in two different formats: Microsoft Binary Real and IEEE. Your choice depends on the type of program you are writing. The primary alternatives are listed below:

1. If your program requires a coprocessor for calculations, you must use the IEEE format.

2. Most high-level languages use the IEEE format. If you are writing a procedures that will be called from such a language, your program must use the IEEE format. All versions of the C, FORTRAN, and Pascal compilers sold by Microsoft and IBM use the IEEE format.

3. If you are writing a procedures that will be called from most previous versions of Microsoft or IBM BASIC, your program must use the Microsoft Binary Real format. Versions that support only the Microsoft Binary Real format include:

   - Microsoft QuickBASIC through Version 2.1
   - Microsoft BASIC Compiler through Version 5.3
   - IBM BASIC Compiler through Version 2.0
   - Microsoft GW-BASIC interpreter (all versions)

- IBM BASICA interpreter (all versions)

Microsoft QuickBASIC Version 3.0 supports both the Microsoft Binary Real and IEEE formats as options.

Current or soon-to-be-released BASIC versions support only the IEEE format. These include:

- Microsoft QuickBASIC Version 4.0
- Microsoft BASIC Compiler Version 6.0
- IBM BASIC Compiler Version 3.0

4. If you are creating a stand-alone program that does not use a coprocessor, you can choose either format. The IEEE format is better for overall compatibility with high-level languages. Also, the CodeView debugger can only display real numbers in the IEEE format. The Microsoft Binary Real format may be necessary for compatibility with existing source code.

By default, **MASM** assembles real-number data in the Microsoft Binary Real format. To assemble data in the IEEE format, you must specify the **.8087**, **.287**, or **.387** directive, or the **/R** or **/E** options.

Real-Number Encoding

The IEEE format for encoding 4- and 8-byte real numbers is illustrated in Figure 6.1.

```
Short real

 3 3          2 2 2          1 1          8 7           0
 1 0          4 3 2          6 5
+              +              +              +              +
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| |              |                                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 | \            /  \                                          /
 |  \        /       \                                      /
Sign  Exponent                 Mantissa
1      2                        3

Long real

 6 6          5 5            4 4          8 7           0
 3 2          6 5            8 7
+              +              +              +              +
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-  . . . -+-+-+-+-+-+-+-+-+-+
| |                  |                                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-  . . . -+-+-+-+-+-+-+-+-+-+
 | \                    /  \                                      /
```

```
  |    \                 /       \                           /
 Sign     Exponent                           Mantissa
 1        2                                  3
```

The parts of the real numbers are described below:

1. Sign bit (0 for positive or 1 for negative) in the upper bit of the first byte.
2. Exponent in the next bits in sequence (8 bits for short real or 11 bits for long real).
3. All except the first set bit of mantissa in the remaining bits of the variable. Since the first significant bit is known to be set, it need not be actually stored. The length is 23 bits for short reals or 52 bits for long reals.

The Microsoft Binary Real format for encoding real numbers is illustrated in Figure 6.2.

```
Short real

  3              2 2 2           1 1             8 7             0
  1              4 3 2           6 5
 +               +               +               +               +
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |               | |             |               |               |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   \           /   |  \                                        /
     \       /     |    \                                    /
   Exponent     Sign              Mantissa
   1            2                 3

Long real

  6              5 5 5                           8 7             0
  3              6 5 4
 +               +                               +               +
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+- . . . -+-+-+-+-+-+-+-+-+-+-+-+-+
 |               | |                             |               |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+- . . . -+-+-+-+-+-+-+-+-+-+-+-+-+
   \           /   |  \                                        /
     \       /     |    \                                    /
   Exponent     Sign             Mantissa
   1            2                3
```

The parts of real numbers are described below:

1. Biased exponent (8 bits) in the high-address byte; the bias is 81h for short reals or 401h for long reals.
2. Sign bit (0 for positive or 1 for negative) in the upper bit of the second highest byte.

3. All except the first set bit of mantissa in the remaining 7 bits of the second highest byte and in the remaining bytes of the variable. Since the first significant bit is known to be set, it need not be actually stored. The length is 23 bits for short reals or 55 bits for long reals.

MASM also supports the 10-byte temporary real format used internally by 8087-family coprocessors. This format is similar to IEEE format. The size is nonstandard and is not used by Microsoft compilers or interpreters. Since the coprocessors can load and automatically convert numbers in the more standard 4- and 8-byte formats, the 10-byte format is seldom used in assembly-language programming.

The format for 10-byte temporary real numbers is shown in Figure 6.3.

```
Ten-byte real

 7 7             7 7             6 6 6           8 7             0
 9 8             2 1             4 3 2
+               +               +               +               +
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+- . . . -+-+-+-+-+-+-+-+-+-+
| |                             | |               |               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+- . . . -+-+-+-+-+-+-+-+-+-+
 | \                           / | \                           /
 |  \                         /  |  \                         /
Sign            Exponent         Integer     Mantissa
1               2                3           4
```

The parts of the real numbers are described below:

1. Sign bit (0 for positive or 1 for negative) in the upper bit of the first byte.
2. Exponent in the next bits in sequence (15 bits for 10-byte real).
3. First set bit of mantissa in the next bit in sequence (bit 63).
4. Remaining bits of mantissa in the remaining bits of the variable. The length is 63 bits.

Notice that the 10-byte temporary real format stores the first set bit of the mantissa. This differs from the 4- and 8-byte formats, where the first set bit is implicit.

6.2.2 Defining Arrays and Buffers

Arrays, buffers, and other data structures consisting of multiple data objects of the same size can be defined with the **DUP** operator. This operator can be used with any of the data definition directives described in this chapter.

■ Syntax

count **DUP** (*initialvalue*⟦,*initialvalue*⟧...)

The *count* sets the number of times to define *initialvalue*. The initial value can be any expression that evaluates to an integer value, a character constant, or another **DUP** operator. It can also be the indeterminate symbol (**?**) if the initial value is to be undefined. Multiple initial values must be separated by commas.

DUP operators can be nested up to 17 levels. The initial value (or values) must always be placed within parentheses.

■ Examples

```
array       DD      10 DUP (1)                   ; 10 doublewords
                                                 ;   initialized to 1

buffer      DB      256 DUP (?)                  ; 256 byte buffer

masks       DB      20 DUP (040h,020h,04h,02h)   ; 80 byte buffer
                                                 ;   with bit masks
            DB      32 DUP ("stuffer ")          ; 256 byte buffer
                                                 ;   with signature
three_d     DD      5 DUP (5 DUP (5 DUP (0)))    ; 125 doublewords
                                                 ;  initialized to 0
```

Note

MASM sometimes generates different object code when the **DUP** operator is used than when multiple values are given. For example, the statement

```
test1       DB      ?,?,?,?,?  ; Indeterminate
```

is "undefined." It causes **MASM** to write five zero-value bytes to the object file. The statement

```
test2       DB      5 DUP (?)   ; Undefined
```

is "indeterminate." It causes **MASM** to increase the offset of the next record in the object file by five bytes. Therefore an object file created with the first statement will be larger than one created with the second statement.

Microsoft high-level languages take advantage of this difference by keeping uninitialized data that is defined with the **DUP** operator in separate segments called **_BSS** or **FAR_BSS**. The result is shorter object files.

In most cases, the distinction between indeterminate and undefined definitions is trivial. The linker adjust the offsets so that the same executable file is generated in either case. However, the difference is significant in segments with the **COMMON** combine type. If **COMMON** segments in two modules contain definitions for the same variable, one with an indeterminate value and one with an explicit value, the actual value in the executable file will vary depending on link order. If the module with the indeterminate value is linked last, the 0 initialized for it will override the explicit value. You can prevent this by always using undefined rather than indeterminate values in **COMMON** segments. For example, use the first of the following statements:

```
test3       DB      1 DUP (?)   ; Undefined - doesn't initialize
test4       DB      ?           ; Indeterminate - initializes 0
```

If you use the undefined, the explicit value will always be used in the executable file regardless of link order.

6.2.3 Labeling Variables

The **LABEL** directive can be used to define a variable of a given size at a specified location. It is useful if you want to refer to the same data as variables of different sizes.

■ Syntax

name **LABEL** *type*

The *name* is the symbol assigned to the variable, and *type* is the variable size. The type can be any one of the following type specifiers: **BYTE**,

WORD, **DWORD**, **FWORD**, **QWORD**, or **TBYTE**. It can also be the name of a previously defined structure.

■ Examples

```
warray      LABEL   WORD         ; Access array as 50 words
darray      LABEL   DWORD        ; Access same array as 25 doublewords
barray      DB      100 DUP(?)   ; Access same array as 100 bytes

crlf        LABEL   BYTE         ; Access carriage return-line feed
cr          DB      13           ; Access carriage return
lf          DB      10           ; Access line feed
```

6.3 Setting the Location Counter

The location counter is the value **MASM** maintains to keep track of its current location in the source file. The location counter is incremented automatically as each source statement is processed. However, the location counter can be set specifically using the **ORG** directive.

■ Syntax

ORG *expression*

Subsequent code and data offsets begin at the new value set by *expression*. The *expression* must resolve to a constant number. In other words, all symbols used in the expression must be known on the first pass of the assembler.

Note

The value of the location counter, represented by the dollar sign ($), can be used in the *expression*, as described in Section 9.3.

■ **Example 1**

```
; Labeling absolute addresses

STUFF        SEGMENT AT 0      ; Segment has constant value 0
             ORG    410h       ; Offset has constant value 410h
equipment    LABEL  WORD       ; Word at 0000:0410 is "equipment"
             ORG    417h       ; Offset has constant value 417h
keyboard     LABEL WORD        ; Word at 0000:0417 is "keyboard"
STUFF        ENDS
```

Example 1 illustrates one way of assigning symbolic names to absolute addresses. This technique is not possible under protected-mode operating systems.

■ **Example 2**

```
; Format for .COM files

_TEXT        SEGMENT
             ASSUME cs:_TEXT,ds:_TEXT,ss:_TEXT,es:_TEXT
             ORG    100h       ; Skip 100h bytes of DOS header

entry:       jmp    begin      ; Jump over data
variable     DW     ?          ; Put more data here
             .
begin:       .                 ; First line of code
             .                 ; Put more code here
_TEXT        ENDS
             END    entry
```

Example 2 illustrates how the **ORG** directive is used to initialize the starting execution point in .COM files.

6.4 Aligning Data

Some operations are more efficient when the variable used in the operation is lined up on a boundary of a particular size. The **ALIGN** and **EVEN** directives can be used to pad the object file so that the next variable is aligned on a specified boundary.

■ **Syntax 1**

EVEN

■ **Syntax 2**

ALIGN *number*

The **EVEN** directive always aligns on the next even byte. The **ALIGN** direct aligns on the next byte that is a multiple of *number*. The *number* should be a power of 2. For example, use `ALIGN 2` or `EVEN` to align on word boundaries, or use `ALIGN 4` to align on doubleword boundaries.

If the value of the location counter is not on the specified boundary when an **ALIGN** directive is encountered, the location counter is incremented to a value on the boundary. **NOP** (no operation) instructions are generated to pad the object file. If the location counter is already on the boundary, the directive has no effect.

The **ALIGN** and **EVEN** directives gives no efficiency improvements on processors that have an 8-bit data bus (such as the 8088 or 80188). These processors always fetch data one byte at a time regardless of the alignment. However, using **EVEN** can speed certain operation on processors that have a 16-bit data bus (such as the 8086, 80186, or 80286), since the processor can fetch a word if the data is word aligned, but must fetch 2 bytes if the data is not word aligned. Similarly, using **ALIGN 4** can speed some operations with a 80386 processor, since the processor can fetch 4 bytes at a time if the data is doubleword aligned.

Note

The **ALIGN** directive is new starting with **MASM** 5.0. In previous versions, data could be word aligned using the **EVEN** directive, but other alignments could not be specified.

■ Example

```
            .DATA
            .
            .
            .
            EVEN
stuff       DW      OFFFFh, OFFFEh, OFFFDh, OFFFCh
            .
            .
            .
            EVEN
copy        DW      ?,?,?,?
            .CODE
            .
            .
            .
            mov     si,stuff            ; Set source
            mov     di,copy             ; Set destination
            mov     cx,4                ; Set count
            rep     movsw               ; Move the words
```

In this example, the words at `stuff` and `copy` are forced to even boundaries. This makes the string operation faster with processors that have a 16-bit data bus. If the boundaries were odd, the processor would have to fetch half of each word (a byte) at a time. This coding wouldn't make any difference with an 8-bit data bus since data must always be fetched a byte at a time.

Chapter 7
Using Structures and Records

The Macro Assembler can define and use two kinds of multifield variables: structures and records.

Structures are named variables made up of multiple smaller variables. The variables within a structure are called fields. They can be different sizes, and each of can be accessed individually.

Records are single variables whose bits are broken into groups called fields. Each bit field can be used separately in constant operands or expressions. The processor cannot access bit fields individually at run-time, but they can be used with logical bit instructions to change bits indirectly, as shown in Section 7.2.4.

This chapter describes structures and records, and tells how to use them.

7.1 Using Structures

A structure variable is a collection of data objects that can be accessed symbolically as a single data object. Objects within the structure can have different sizes, and can be accessed symbolically.

There are three steps in using structure variables:

1. Declare a structure type. A structure type is a template for data. It declares the sizes and, optionally, the initial values for objects in the structure. By itself the structure type does not define any data. The structure type is used by **MASM** during assembly, but is not saved as part of the object file.
2. Define one or more variables having the structure type. For each variable defined, memory is allocated to the object file in the format declared by the structure type.
3. The structure variable can then be used as an operand in assembler statements. The structure variable can be accessed as a whole by using the structure name, or individual fields can be accessed by using the structure name and a field name combined with the field-name operator.

7.1.1 Declaring Structure Types

The **STRUC** and **ENDS** directives mark the beginning and end of a type declaration for a structure.

■ **Syntax**

name **STRUC**
fielddeclarations
name **ENDS**

The *name* declares the name of the structure type. It must be unique. The *fielddeclarations* declare the fields of the structure. Any number of field declarations may be given. They must follow the form of data definitions described in Section 6.2. Values may be initialized and the **DUP** operator may be used to declare multiple values.

The names given to fields must be unique. When variables are defined, the field names will represent the offset from the beginning of the structure to the corresponding field.

When declaring strings in a structure type, make sure the initial values are long enough to accommodate the largest possible string. Strings smaller than the field size can be placed in the structure variable, but larger strings will be truncated.

A structure type declaration can contain field declarations and comments. Starting with **MASM** 5.0, conditional-assembly statements are allowed in structure declarations. They were not permitted in previous versions. No other kinds of statements are allowed. Since the **STRUC** directive is not allowed inside structure declarations, structures cannot be nested.

Note

The **ENDS** directive that marks the end of a structure has the same mnemonic as the **ENDS** directive that marks the end of a segment. The assembler recognizes the meaning of the directive from context. Make sure each **SEGMENT** directive and each **STRUC** directive has its own **ENDS** directive.

■ Example

```
student     STRUC                   ; Structure for student records
id          DW      ?               ; Field for identification #
sname       DB      "Last, First Middle    "
scores      DB      10 DUP (100)    ; Field for 10 scores
student     ENDS
```

In this example, the fields `id`, `sname`, and `scores` have the offset values 0, 2, and 24, respectively within the structure `student`.

7.1.2 Defining Structure Variables

A structure variable is a variable with one or more fields of different sizes. The sizes and initial values of the fields are determined by the structure type with which the variable is defined.

■ Syntax

⟦*name*⟧ ***structurename*** <⟦*initialvalue* ⟦*,initialvalue...*⟧⟧>

The *name* is the name assigned to the variable. If no *name* is given, the assembler allocates space for the variable, but does not give it a symbolic name. The *structurename* is the name of a structure type previously declared using the **STRUC** and **ENDS** directives.

An *initialvalue* can be given for each field in the structure. It's type must not be incompatible with the type of the corresponding field. The angle brackets (< >) are required even if no initial value is given.

If *initialvalues* are given for more than one field, the values must be separated by commas. If the **DUP** operator (see Section 6.2.3) is used to initialize several structure variables of the same type, only the values within the parentheses need to be enclosed in angle brackets. In other words, the structure variable is duplicated, not the fields.

You need not initialize all fields in a structure. If an initial value is left blank, the assembler automatically uses the default initial value of the field, which was originally determined by the structure type. If there is no default value, the field is undefined.

■ Examples

The following examples use the student type declared in the example in Section 7.1.2.

```
s1          student <>          ; Uses initial values of type


s2          student <1467,"White, Robert D.",>
                                ; Override initial values of first two
                                ;   fields--use initial value of third

sarray      student DUP 100 (<>); Declare 100 student variables
                                ;   with default initial values
```

Note

You cannot initialize any structure field that has multiple values if this field was given a default initial value when the structure was declared. For example, assume the following structure declaration:

```
stuff       STRUC
buffer      DB      100 DUP (?)      ; Can't override
crlf        DB      13,10            ; Can't override
query       DB      'Filename: '     ; String <= can override
endmark     DB      36               ; Can override
stuff       ENDS
```

The buffer and crlf fields cannot be overridden by initial values in the structure definition because they have multiple values. The query field can be overridden as long as the overriding string is no longer than query (10 bytes). A longer string would be truncated. The endmark field can be overridden by any byte value.

7.1.3 Using Structure Operands

The starting address of a structure variable can be accessed by name the same as other variables. Fields within a structure variable can also be accessed using the syntax shown below:

■ Syntax

variable.field

The *variable* must be the name of a structure (or an operand that resolves to the address of a structure). The *field* must be the name of a field within that structure. The *variable* is separated from *field* by a period. The period is discussed as a structure field-name operator in Section 9.2.1.2.

The address of a structure operand is the sum of the offsets of *variable* and *field*. The address is relative to the segment or group in which the variable is declared.

■ Examples

```
date        STRUC                          ; Declare structure
month       DB      ?
day         DB      ?
year        DW      ?
date        ENDS

            .DATA
yesterday   date    <9,30,1987>             ; Declare structure
today       date    <10,1,1987>             ;   variables
tomorrow    date    <10,2,1987>

            .CODE
            .
            .
            .
            mov     al,yesterday.day        ; Use structure variables
            mov     ah,today.month          ;   as operands
            mov     tomorrow.year,dx
            mov     bx,OFFSET yesterday     ; Load structure address
            mov     ax,[bx].day             ; Use as indirect operand
            .
            .
            .
```

7.2 Using Records

A record variable is a byte or word variable in which specific bit fields can be accessed symbolically. Records can be doubleword variables with the 80386 processor. Bit fields within the record can have different sizes.

There are three steps in declaring record variables:

1. Declare a record type. A record type is a template for data. It declares the sizes and, optionally, the initial values for bit fields in the record. By itself the record type does not define any data. The record type is used by **MASM** during assembly, but is not saved as part of the object file.
2. Define one or more variables having the record type. For each variable defined, memory is allocated to the object file in the format declared by the type.
3. The record variable can then be used as an operand in assembler statements. The record variable can be accessed as a whole by using the record name, or individual fields can be specified by using the record name and a field name combined with the field-name operator. A record type can also be used as a constant (immediate data).

7.2.1 Declaring Record Types

The **RECORD** directive declares a record type for an 8- or 16-bit record that contains one or more bit fields. With the 80386, 32-bit records can also be declared.

■ **Syntax**

recordname **RECORD** *fielddeclaration* ⟦,*fielddeclaration...*⟧

The *recordname* is the name of the record type to be used when creating the record. The *fielddeclaration* declares the name, width, and initial value for the field.

The syntax for each *fielddeclaration* is shown below:

■ **Syntax**

fieldname:*width*⟦= *expression*⟧

The *name* is the name of a field in the record, *width* is the number of bits in the field, and *expression* is the initial (or default) value for the field.

Any number of *fielddeclarations* combinations can be given for a record, as long as each is separated from its predecessor by a comma. The sum of the widths for all fields must not exceed 16 bits.

The width must be a constant. If the total width of all declared fields is larger than 8 bits, then the assembler uses 2 bytes. Otherwise, only 1 byte is used.

80386 Only

Records can be up to 32 bits in width when the 80386 processor is enabled. If the total width is 8 bits or less, the assembler uses 1 byte; if the width is 9 to 16 bytes, the assembler uses 2 bytes; and if the width is larger than 16 bits, the assembler uses 4 bytes.

If *expression* is given, it declares the initial value for the field. If the field is at least 7 bits wide, you can use an ASCII character for *expression*. The expression must not contain a forward reference to any symbol.

In all cases, the first field you declare goes into the most significant bits of the record. Successively declared fields are placed in the succeeding bits to the right. If the fields you declare do not total exactly 8 bits or exactly 16 bits, the entire record is shifted right so that the last bit of the last field is the lowest bit of the record. Unused bits in the high end of the record will be initialized to 0.

■ Example 1

```
color       RECORD  blink:1,back:3,intense:1,fore:3
```

The example above creates a byte record type `color` having four fields: `blink`, `back`, `intense`, and `fore`. The contents of the record type are shown below:

```
blink    back  intense   fore
  |  /       \  |  /       \
+---+---+---+---+---+---+---+---+
| 0   0   0   0   0   0   0   0 |
+---+---+---+---+---+---+---+---+
  7   6   5   4   3   2   1   0
```

Since no initial values are given, all bits are set to 0. Note that this is only a template maintained by the assembler. No data are created.

■ Example 2

```
openmode     RECORD  D:1=0,W:1=0,F:1=1,R:5=0,I:1=1,S:3=4,R:1=0,A:3=2
```

Example 2 creates a record type openmode having eight fields. Each record declared using this type will occupy 16 bits of memory. The bit diagram below shows the contents of the record type:

```
 D=0 W=0 F=1        R=0         I=1     S=4     R=0     A=2
  |   |   |   /             \   |   /       \   |   /       \
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 0   0   1   0   0   0   0   0 | 1   1   0   0   0   0   1   0 | 20C2h
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
  7   6   5   4   3   2   1   0   7   6   5   4   3   2   1   0
```

Default values are given for each field. They can be used when data is declared for the record.

7.2.2 Defining Record Variables

A record variable is an 8-bit or 16-bit variable whose bits are divided into one or more fields. With the 80386, 32-bit variables are also allowed.

■ Syntax

⟦*name*⟧ ***recordname*** **<**⟦*initialvalue* ⟦,*initialvalue*...⟧⟧**>**

The *name* is the symbolic name of the variable. If no *name* is given, the assembler allocates space for the variable, but does not give it a symbolic name. The *recordname* is the name of a record type that was previously declared using the **RECORD** directive.

An *initialvalue* for each field in the record can be given as an integer, character constant, or an expression that resolves to a value compatible with the size of the field. Angle brackets (**< >**) are required even if no initial value is given.

If initial values for more than one field are given, the values must be separated by commas. If the **DUP** operator (see Section 6.2.2) is used to initialize several record variables of the same type, only the values within the parentheses need to be enclosed in angle brackets.

You do not have to initialize all fields in a record. If an initial value is left blank, the assembler automatically uses the default initial value of the field. This is declared by the record type. If there is no default value, each bit in the field is cleared.

Sections 7.2.3 and 7.2.4 illustrate ways to use record data after it has been declared.

■ Examples

```
color      RECORD  blink:1,back:3,intense:1,fore:3 ; Record declaration
warning    color   <1,0,1,4>                       ; Record definition
```

The definition above creates a variable named `warning` whose type is given by the record type `color`. The initial values of the fields in the variable are set to the values given in the record definition. They would override the default record values, had any been given in the declaration. The contents of the record variable are shown below:

```
blink    back  intense    fore
  1        0       1       4
  |   /      \   |   /       \
+---+---+---+---+---+---+---+---+
| 1   0   0   0   1   1   0   0 | 8Ch
+---+---+---+---+---+---+---+---+
  7   6   5   4   3   2   1   0
```

■ Example 2

```
color      RECORD  blink:1,back:3,intense:1,fore:3 ; Record declaration
colors     color   16 DUP (<>)                     ; Record declaration
```

Example 2 creates an array named `colors` containing 16 variables of type `color`. Since no initial values are given in either the declaration or the definition, the variables have undefined (0) values.

■ Example 3

```
openmode   RECORD   D:1=0,W:1=0,F:1=1,R:5=0,I:1=1,S:3=4,R:1=0,A:3=2
om1        openmode <,,,,0,2,,0>
```

Example 3 creates a variable named `om1` with type `openmode`. The default values set in the type declaration are used for all fields except the `I`, `S`, and `A` fields. These are set to 0, 2, and 0 respectively. The contents of the variable are shown below:

```
 D=0 W=0 F=1         R=0          I=1     S=4     R=0     A=2
  0   0   1           0            0       2       0       0
  |   |   |   /              \     |   /      \    |   /       \
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 0   0   1   0   0   0   0   0 | 0   0   1   0   0   0   0   0 | 2020h
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
  7   6   5   4   3   2   1   0   7   6   5   4   3   2   1   0
```

7.2.3 Using Record Operands and Record Variables

A record operand refers to the value of a record type. It should not be confused with a record variable. A record operand is a constant, while a record variable is a value stored in memory. A record operand can be used with the following syntax:

■ Syntax

recordname <[[[[*value*]]]][[,*value...*]]]]>

The *recordname* must be the name of a record type declared in the source file. The optional *value* is the value of a field in the record. If more than one *value* is given, the values must be separated by commas. Values can include expressions or symbols that evaluate to constants. The enclosing angle brackets (< >) are required, even if no value is given. If no value for a field is given, the default value for that field is used.

■ Example

```
             .DATA
color        RECORD  blink:1,back:3,intense:1,fore:3 ; Record declaration
window       color   <0,6,1,6>                       ; Record definition

             .CODE
             mov     ah,color <0,3,0,2> ; Load record operand
                                        ;  (constant value 32h)
             mov     bh,window          ; Load record variable
                                        ;  (memory value 6Eh)
```

In this example, the record operand `color <0,3,0,2>` and the record variable `warning` are loaded into registers. The contents of the values are shown below:

```
        color <0,3,0,2>
```

```
blink    back  intense   fore
  0       3       0       2
  |   /       \   |   /       \
+---+---+---+---+---+---+---+---+
| 0   0   1   1   0   0   1   0 | 32h
+---+---+---+---+---+---+---+---+
  7   6   5   4   3   2   1   0

   window    color <0,6,1,6>

blink    back  intense   fore
  0       6       1       6
  |   /       \   |   /       \
+---+---+---+---+---+---+---+---+
| 0   1   1   0   1   1   1   0 | 6Eh
+---+---+---+---+---+---+---+---+
  7   6   5   4   3   2   1   0
```

7.2.4 Using Record-Field Operands

Record-field operands represent the location of a field in its corresponding record. The operand evaluates to the bit position of the low-order bit in the field and can be used as a constant operand. The field name must be from a previously declared record.

Record-field operands are often used with the **WIDTH** and **MASK** operators, as described in Sections 7.3.

■ Example

```
       .DATA
color  RECORD  blink:1,back:3,intense:1,fore:3 ; Record declaration
cursor color   <1,5,1,1>                       ; Record definition
       .CODE
       .
       .
       .
; Rotate "back" of "cursor" without changing other values

       mov     al,cursor         ; Load value from memory
       mov     ah,al             ; Save a copy for work      1101 1001=ah/al
       and     al,NOT MASK back  ; Mask out old bits     and 1000 1111=mask
                                 ;   to save old cursor      ---------
                                                             1000 1001=al
       mov     cl,back           ; Load bit position
       shr     ah,cl             ; Shift to right            0000 1101=ah
       inc     ah                ; Increment                 0000 1110=ah

       shl     ah,cl             ; Shift left again          1110 0000=ah
       and     ah,MASK back      ; Mask off extra bits   and 0111 0000=mask
                                 ;   to get new cursor       ---------
                                                             0110 0000 ah
       or      ah,al             ; Combine old and new    or 1000 1001 al
                                                             ---------
       mov     cursor,ah         ; Write back to memory      1110 1001 ah
```

This example illustrates several ways record fields can be used as operands and in expressions.

7.2.5 Using Record Operators

The **WIDTH** and **MASK** operators are used exclusively with records to return constant values representing different aspects of previously declared records.

7.2.5.1 Using the MASK Operator

The **MASK** operator returns a bit mask for the bit positions in a record occupied by the given record field. A bit in the mask contains a 1 if that bit corresponds to a field bit. All other bits contain 0.

■ **Syntax**

MASK {*recordfieldname* | *record*}

The *recordfieldname* may be the name of any field in a previously defined record. The *record* may be the name of any previously defined record. The **NOT** operator is sometimes used with the **MASK** operator to reverse the bits of a mask.

■ **Example**

```
             .DATA
color        RECORD  blink:1,back:3,intense:1,fore:3
message      color   <0,5,1,1>
             .CODE
             mov     ah,message          ; Load initial   0101 1001
             and     ah,NOT MASK back    ; Turn off   AND 1000 1111
                                           "back"         ---------
                                                          0000 1001
             or      ah,MASK blink       ; Turn on     OR 1000 0000
                                           "blink"        ---------
                                                          1000 1001
             xor     ah,MASK fore        ; Toggle     XOR 0000 1000
                                           "intense"      ---------
                                                          1000 0001
```

7.2.5.2 Using the WIDTH Operator

The **WIDTH** operator returns the width (in bits) of a record or record field.

■ Syntax

WIDTH {*recordfieldname* | *record*}

The *recordfieldname* may be the name of any field defined in any record. The *record* may be the name of any defined record.

Note that the width of a field is the number or bits assigned for that field, while the value of the field is the starting position (from the right) of the field.

■ Examples

```
            .DATA
color       RECORD  blink:1,back:3,intense:1,fore:3

wblink      EQU     WIDTH blink   ; "wblink"   = 1   "blink"   = 7
wback       EQU     WIDTH back    ; "wback"    = 3   "back"    = 4
wintense    EQU     WIDTH intense ; "wintense" = 1   "intense" = 1
wfore       EQU     WIDTH wfore   ; "wfore"    = 3   "fore"    = 0
color       EQU     WIDTH color   ; "wcolor"   = 8

prompt      color   <>

            .CODE
            .
            .
            .
            IF      (WIDTH color) GE 8 ; If color is 16 bit, load
            mov     ax,color           ;   into 16-bit register
            ELSE                       ; else
            mov     al,color           ;   load into low 8-bit register
            xor     ah,ah              ;   and clear high 8-bit register
            ENDIF
```

Chapter 8
Creating Programs from Multiple Modules

Most assembly-language programs of significant size are created from several source files or modules. When several modules are used, the scope of symbols becomes important. This chapter discusses the scope of symbols and explains how to declare global symbols that can be accessed from any module. It also tells how to specify a module that will be accessed from a library.

Symbols such as labels and variable names can be either local or global in scope. By default, all symbols are local: they are specific to the source file in which they are defined. Local symbols can be convenient if you want to use the same names for symbols in different source files. In particular, you may run out of unique mnemonic names for near code labels if you can't use duplicate names in different source files.

Any symbols that are declared in one source module, yet must be accessed from another module, should be declared global. This includes most procedure names, many variable names, and key near-code labels.

To declare symbols global, they must be declared public in the source module in which they are defined. They must also be declared external in any module that must access the symbol. If the symbol represents uninitialized data, it can be declared communal. Communal means that the symbol is both public and external. The **PUBLIC**, **EXTRN**, and **COMM** directives are used to declare symbols public, external, and communal respectively.

Notes

The term local symbol has a different meaning in assembly language than in many high-level languages. Often, local symbols in compiled languages are symbols that are known only within a procedure (called a function, routine, subprogram, or subroutine in some languages). Local symbols of this type cannot be declared by **MASM**, although procedures can be written to allocate local symbols dynamically at run time, as described in Section 17.4.4.

By default, the assembler converts all lowercase letters in names declared with the **PUBLIC**, **EXTRN**, and **COMM** directives to uppercase before copying the name to the object file. The **/ML** and **/MX** options can be used in the **MASM** command line to direct the assembler to preserve lowercase letters when copying public and external symbols to the object file.

8.1 Declaring Symbols Public

The **PUBLIC** directive is used to declare symbols public so that they can be accessed from other modules. If a symbol is not declared public, the symbol name is not written to the object file. The symbol has the value of its offset address during assembly, but the name and address are not available to the linker.

If the symbol is declared public, its name is associated with its offset address in the object file. During linking, symbols in different modules, but with the same name, will be resolved to a single address.

Public symbol names are also used by some symbolic debuggers (such as **SYMDEB**) to associate addresses with symbols. However, variables and labels do not need to be declared public in order to be visible in the CodeView debugger.

■ Syntax

PUBLIC *name* ⟦,*name*...⟧

The *name* must be the name of a variable, label, or equate symbol defined within the current source file. Equate symbols, if given, can only represent 1- or 2-byte integer or string values. Text macros (or text equates) cannot be public.

■ Example

```
            PUBLIC  true,status,start,clear
true        =       0FFFFh
status      DB      1
start       LABEL   FAR
clear       PROC    NEAR
```

8.2 Declaring Symbols External

If a symbol is not declared in a module, but must be accessed by instructions in that module, it must be declared with the **EXTRN** directive.

This directive tells the assembler not to generate an error message even though it cannot find the symbol in the current module. The assembler assumes that the symbol occurs in another module, and so does not generate an error. However, the symbol must actually exist and must be declared public in some module. Otherwise, the linker will generate an error.

■ Syntax

EXTRN *name:type* ⟦*,name:type...*⟧

The **EXTRN** directive defines an external variable, label, or symbol of the specified *name* and *type*. The *type* must match the type given to the item in its actual definition in some other module. It can be any one of the following:

Description	Types
Distance specifier	**NEAR**, **FAR**, or **PROC**
Size specifier	**BYTE**, **WORD**, **DWORD**, **FWORD**, **QWORD**, or **TBYTE**
Absolute	**ABS**

The **ABS** type is for symbols that represent constant numbers, such as equates declared with the **EQU** and = directives (see Section 11.1).

The **PROC** type represents the default type for a procedure. It is useful if you have defined segments with the simplified segment directives. The type of an external symbol declared with **PROC** type will be near for programs declared small or compact, or far for programs declared medium or large. Section 5.1.3 tells how to declare the memory model using the **.MODEL** directive. If full segment definitions are used, the default type represented by **PROC** is always near.

Although the actual address of an external symbol is not determined until link time, the assembler may assume a default segment for the item, based on where the **EXTRN** directive is placed in the module. This only happens with full segment definitions. If simplified segment directives are used, no assumptions are made about the segment location of the symbol.

If you are using full segment definitions, the safest method is to declare all external symbols outside segments. External symbols can also be declared inside the segment in which the symbol actually occurs. However, sometimes the segment of an external symbol does not occur in a module where it is used. In this case, the external declaration should be outside all segments.

■ **Example 1**

```
EXTRN  task:PROC            ; Procedure label (use default)
EXTRN  act:FAR              ; Other code label (NEAR or FAR)
EXTRN  val:BYTE, va2:WORD   ; Variables (any data size)

call   task
jmp    act
mov    ah,var1
mov    bx,var2
```

In Example 1, the procedure name `task`, the code label `act`, and the variables `var1` and `var2` are expected to be in a different modules. At link-time, the external symbols must actually exist in another module and must have the specified size.

■ **Example 2**

```
            EXTRN    proc1:FAR

_CODE       SEGMENT BYTE PUBLIC 'CODE'

            EXTRN    proc2:FAR
            call     proc1
            call     proc2
            .
            .
            .
_CODE       ENDS
            END
```

In Example 2, `proc1` is a **FAR** procedure as declared because it is declared external outside any segment. However, `proc2` is declared external in the current near segment. Therefore it is a **NEAR** procedure despite being specified **FAR** in the external declaration. This discrepancy would not occur if simplified segment directives were used.

8.3 Declaring Symbols Communal

Uninitialized data can be declared communal. Communal variables are both public and external. They can be called from any module and can be declared in any module.

Communal variables were implemented with Version 4.5 in order to be compatible with the Microsoft C Compiler. In the C language, include files (usually having the extension **.H**) are sometimes used to declare variables. Often the same include file will be used in several different source modules, yet variables declared in these files should exist at only one address. The C compiler makes the variables communal so that a separate copy won't be created for each source module that uses the include file.

Communal variables can be used in assembly language include files in the same way. Data that will be used by several assembly routines can be declared external in an include file. Then variables can be used in each module that includes the file without specifically declaring it external in each module. The only limitation is that communal variables cannot be initialized. This makes them useful for file buffers, arrays, and pointer variables that are not given values until run time.

■ Syntax

name **COMM** ⟦*distance*⟧ *size count* **DUP(?)**

The *name* is the symbolic name of the variable. The *distance* can be **NEAR** or **FAR**. **NEAR** is the default if *distance* is not given and full segment definitions are used. If simplified segment directives are used, the default is the *distance* used by the specified memory model—**NEAR** in small and medium models or **FAR** in compact, large, and huge models. The size can by **BYTE**, **WORD**, **DWORD**, **FWORD**, **TBYTE**, a structure name, or a constant specifying a number of bytes. The *count* specifies the number of elements in arrays or other multiobject variables. A *count* must be specified, even if it is 1.

Since communal variables are implement primarily for compatibility with C, they follow strict segment guidelines that are part of the Microsoft segment naming conventions. Near communal variables are placed in a segment called **c_common**. Far communal variables placed in a segment called **FAR_BSS**. The linker rather than the assembler handles this, so the you cannot override the default to place communal variables in other segments.

The linker places the **c_common** segment in **DGROUP**. To access near communal variables, your program must have a group called **DGROUP**. This group is created and initialized automatically if you use simplified segment directives. If you use full segment directives, you must create a **DGROUP** and use the **ASSUME** directive to associate it with the **DS** register.

The **FAR_BSS** segment where far communal data is stored has combine type private and class type **'FAR_BSS'**. This means that multiple segments with the same name can be created. Such segments cannot be accessed by name. They must be initialized indirectly using the **SEG** operator. For example, if a far communal variable is called `fcomvar`, its segment can be initialized with the following lines:

```
mov     ax,SEG comvar
mov     ds,ax
```

■ Example 1

```
; Include file - COMMUNAL.INC
            IF      @fardata
            .FARDATA
            ELSE
            .DATA
            ENDIF
@max        COMM    DB 1 DUP(?)       ; Default distance of calling
@actual     COMM    DB 1 DUP(?)       ;   source file used
@tempstr    COMM    DB 128 DUP(?)

@GetTempStr MACRO   destination       ; Destination must be address
                                      ;   for pointer to string
            mov     dx,OFFSET @max    ; Address of string buffer
            mov     @max,128          ; Insert maximum length
            mov     ah,0Ah            ; Get string
            int     21h               ;
            mov     dl,@actual        ; Get length of string
            xor     dh,dh
            mov     si,dx
            mov     @tempstr[si],0    ; Overwrite CR with 0
destination EQU     OFFSET @tempstr
            ENDM
```

Example 1 shows an include file that declares temporary variables. The variables are then used in a macro in the same include file. When the file is included in a source file that uses simplified segment directives, the proper data segment will be initialized for the memory model of the source file. Example 2 shows how the macro is used in a source file. Note that once the macro is written, the user does not need to know the names of the temporary communal variables or how they are used used in the macro.

■ **Example 2**

```
            DOSSEG
            .MODEL   small
            INCLUDE  communal.inc
            .DATA
message     DB       "Enter file name: $"
            .CODE
            .
            .
            .
            mov      dx,OFFSET message   ; Load offset of file prompt
            mov      ah,09h              ; Display prompt
            int      21h

            @GetTempStr   place          ; Get file name and
                                         ;   return address as "place"

            mov      al,00000010b        ; Load access code
            mov      dx,place            ; Load address of ASCIIZ
            mov      ah,3Dh              ; Open the file
            int      21h
```

8.4 Using Multiple Modules

The following source files illustrate a program that uses public and external declarations to access instruction labels. The program consists of two modules called `hello` and `display`. The `hello` module is the program's initializing module. Execution starts at the instruction labeled `start` in the `hello` module. After initializing the data segment, the program calls the procedure `display` in the `display` module, where a DOS call is used to display a message on the screen. Execution then returns to the address after the call in the `hello` module.

■ Hello Module

```
            TITLE   hello

            .MODEL  small,dosseg

            .STACK  256

            .DATA
            PUBLIC  message, lmessage
message     DB      "Hello, world.",13,10
lmessage    EQU     $ - message

            .CODE
            EXTRN   display:PROC
start:      mov     ax,DGROUP              ; Load segment location
            mov     ds,ax                  ;   into DS register

            call    display                ; Call other module

            xor     al,al                  ; Set return code to 0
            mov     ah,04Ch                ; Terminate function
            int     21h                    ; Call DOS

            END     start
```

■ Display Module

```
            TITLE   display
            .MODEL  small

            EXTRN   message:BYTE,lmessage:ABS

            .CODE
            PUBLIC  display

display     PROC
            mov     bx,1                   ; File handle for
Standard output
            mov     cx,lmessage            ; Message length
            mov     dx,OFFSET message      ; Message address
            mov     ah,40h                 ; Write Handle Function
            int     21h                    ; Call DOS
            ret display     ENDP
            END
```

This example is a variation of the `hello.asm` program used in examples in Chapter 1 except that it uses a procedure rather than a macro to display to the screen. Notice that all symbols that are defined in one module but used in another are declared **PUBLIC** in the defining module and declared **EXTRN** in the using module.

For example, `message` and `lmessage` are declared **PUBLIC** in `hello` and declared **EXTRN** in `display`. The procedure label `display` is declared **EXTRN** in `hello` and **PUBLIC** in `display`.

To create an executable file for these modules, assemble each module separately. For example:

```
MASM hello;
MASM display;
```

Then link the two modules:

```
LINK hello display;
```

The result is the executable file `hello.exe`.

8.5 Specifying Library Files

The **INCLUDELIB** directive instructs the linker to link the object file with a specified library file. You can use this directive if you find it more convenient to specifying a library file in the assembly source file than on the **LINK** command line.

■ Syntax

INCLUDELIB *libraryname*

The *libraryname* is written to the comment record of the object file. The Intel title for this record is **COMENT**. At link time, the linker reads this record and links with the specified library file.

The *libraryname* must be a file name rather than a complete file specification. **LINK** will search for the library file first in the current directory, and then in any directories listed in the **LIB** environment variable.

■ Example

```
INCLUDELIB grafics.lib
```

This statement passes a message from **MASM** telling **LINK** to include object modules from the file `graphics.lib`.

Chapter 9
Using Operands and Expressions

Operands are the arguments that define values to be acted on by instructions or directives. Operands can be constants, variables, expressions, or key words, depending on the instruction or directive, and the context of the statement.

A common type of operand is an expression. An expression consists of several operands that are combined to describe a value or memory location. Operators indicate the operations to be performed when combining the operands of an expression.

Expressions are evaluated at assembly time. By using expressions, you can instruct the assembler to calculate values that would be difficult or inconvenient to calculate when writing source code.

This chapter discusses operands, expressions, and operators as they are evaluated at assembly time. See Chapter 14, "Using Addressing Modes," for a discussion of the addressing modes that can be used calculate operand values at run time. This chapter also discusses the location counter operand, forward references, and strong typing of operands.

9.1 Using Operands with Directives

Each directive requires a specific type of operand. Most directives take string or numeric constants, or symbols or expressions that evaluate to such constants.

The type of operand varies for each directive, but the operand must always evaluate to a value that is known at assembly time. This differs from instructions, whose operands may not be known at assembly time and may vary at run time. Operands used with instructions are discussed in Chapter 14, "Using Addressing Modes."

Some directives, such as those used in data declarations, accept labels or variables as operands. When a symbol that refers to a memory location is used as an operand to a directive, the symbol represents the address of the symbol rather than its contents, since the contents are never known at assembly time.

■ Example 1

```
                ORG     100h          ; Set address to 100h
var             DB      10h           ; Address of "var" is 100h
                                      ; Value of "var" is 10h
pvar            DW      var           ; Address of "pvar" is 101h
                                      ; Value of "pvar" is
                                      ;   address of "var" (100h)
```

In Example 1, the operand of the **DW** directive in the third statement represents the address of `var` (100h) rather than its contents (10h). The address is relative to the start of the segment in which `var` is defined.

■ Example 2

```
                TITLE   doit                   ; String
_TEXT           SEGMENT BYTE PUBLIC 'CODE'     ; Key words
                INCLUDE \include\bios.inc      ; Pathname
                .RADIX  16                     ; Numeric constant
tst             DW      a / b                  ; Numeric expression
                PAGE    +                      ; Special character
sum             EQU     x * y                  ; Numeric expression
here            LABEL   WORD                   ; Type specifier
```

Example 2 illustrates the different kinds of values that can be used as directive operands.

9.2 Using Operators

The assembler provides a variety of operators for combining, comparing, changing, or analyzing operands. Some operators work with integer constants, some with memory values, and some with both. Operators cannot be used with floating point constants, since **MASM** does not recognize real numbers in expressions.

This section describes the different kinds of operators used in assembly-language statements and gives examples of expressions formed with them. In addition to the operators described in this chapter, you can use the **DUP** operator (Section 6.2.2), the record operators (Section 7.2.5), and the macro operators (Section 11.4).

9.2.1 Using Calculation Operators

MASM provides the common arithmetic operators as well as several other operators for adding, shifting, or doing bit manipulations. The next sections describe operators that can be used for doing numeric calculations.

9.2.1.1 Using Arithmetic Operators

MASM recognizes a variety of arithmetic operators for common mathematical operations. Table 9.1 lists the arithmetic operators.

Table 9.1

Arithmetic Operators

Operator	Meaning	Syntax
+	Positive (unary)	+*expression*
–	Negative (unary)	-*expression*
*	Multiplication	*expression1***expression2*
/	Integer division	*expression1*/*expression2*
MOD	Remainder (modulus)	*expression1*MOD*expression2*
+	Addition	*expression1*+*expression2*
–	Subtraction	*expression1*–*expression2*

For all arithmetic operators except + and –, the expressions operated on must be integer constants.

The + and – operators can be used to add or subtract an integer constant and a memory operand. The result can be used as a memory operand.

The – operator can also be used to subtract one memory operand from another, but only if the operands refer to locations within the same segment. The result is a constant, not a memory operand.

Note

The unary plus and minus (used to designate positive or negative numbers) are not the same as the binary plus and minus (used to designate addition or subtraction). The unary plus and minus have a higher level of precedence, as described in Section 9.2.5.

■ Example 1

```
int         =       14  *  3           ; = 42
int         =       int /  4           ; 42 / 4 = 10
int         =       int MOD  4         ; 10 mod 4 = 2
int         =       int +  4           ; 2 + 4 = 6
int         =       int -  3           ; 6 - 3 = 3
int         =       -int - 8           ; -3 - 8 = -11
int         =       -int - +int        ; -11 - +11 = -22
```

Example 1 illustrates arithmetic operators used in integer expressions.

■ Example 2

```
            ORG     100h
a           DB      ?                  ; Address is 100h
b           DB      ?                  ; Address is 101h
mem1        EQU     a + 5              ; mem1 = 100h + 5 = 105h
mem2        EQU     a - 5              ; mem2 = 100h - 5 = 0FAh
const       EQU     b - a              ; const = 101h - 100h = 1
```

Example 2 illustrates arithmetic operators used in memory expressions.

9.2.1.2 Using the Structure Field-Name Operator

The structure field-name operator (.) indicates addition. It is used to designate a field within a structure.

■ Syntax

variable.field

The *variable* is a memory operand (usually a previously declared structure variable) and *field* is the name of a field within the structure. See Section 7.1 for more information on using structures.

■ Example

```
               .DATA
date           STRUC                          ; Declare structure
month          DB      ?
day            DB      ?
year           DW      ?
date           ENDS
yesterday      date    <12,31,1987>           ; Define structure variables
today          date    <1,1,1988>

               .CODE
               mov     bh,yesterday.day       ; Load structure variable

               mov     bx,OFFSET date         ; Load structure address and
               inc     [bx].year              ;   use in indirect memory operand
```

9.2.1.3 Using the Index Operator

The index operator ([ ]) indicates addition. It is similar to the addition (+) operator.

■ Syntax

⟦*expression1*⟧[*expression2*]

In most cases *expression1* is simply added to *expression2*. The limitations of the addition operator for adding memory operands also apply to the index operator. For example, two direct-memory operands cannot be added. The expression `label1[label2]` is illegal if both are memory operands.

The index operator has an extended function in specifying indirect-memory operands. Section 14.3.2 explains the used of indirect-memory operands. The index brackets must be outside the register or registers that specify the indirect displacement. However, any of the three operators that indicate addition (the addition operator, the index operator, or the structure-field- name operator) may be used for multiple additions within the expression.

For example, the following statements are equivalent:

```
mov     ax,table[bx][di]
mov     ax,table[bx+di]
mov     ax,[table+bx+di]
mov     ax,[table][bx][di]
```

The following statements are illegal because the index operator does not enclose the registers that specify indirect displacement:

```
mov     ax,table+bx+di      ; Illegal - no index operator
mov     ax,[table]+bx+di    ; Illegal - registers not
                            ;  inside index operator
```

The index operator is typically used to index elements of a data object, such as variables in an array or characters in a string.

■ **Example 1**

```
mov     al,string[3]        ; Get 4th element of string
add     ax,array[4]         ; Add 5th element of array
mov     string[7],al        ; Put into 8th element of string
cmp     cx,DGROUP:[1]       ; Compare to 2nd byte of DGROUP
```

Example 1 illustrates the index operator used with direct-memory operands.

■ **Example 2**

```
mov     ax,[bx]             ; Get element BX points to
add     ax,array[si]        ; Add element SI points to
mov     string[di],al       ; Load element DI points to
cmp     cx,table[bx][di]    ; Compare to element BX and DI
                            ;  point to
```

Example 2 illustrates the index operator used with indirect-memory operands.

9.2.1.4 Using Shift Operators

The **SHR** and **SHL** operators can be used to shift bits in constant values. Both perform logical shifts. Bits on the right for **SHL** and on the left for **SHR** are zero-filled as their contents are shifted out of position.

■ **Syntax**

expression SHR *count*
expression SHL *count*

The *expression* is shifted right or left by *count* number of bits. Bits shifted off either end of the expression are lost. If *count* is greater than or equal to

16, the result is 0.

Note

Do not confuse the **SHR** and **SHL** operators with the processor instructions having the same names. The operators work on integer constants only at assembly time. The processor instructions work on register or memory values at run time.

■ **Examples**

```
mov     ax,01110111b SHL 3 ; Move 01110111000b
mov     ah,01110111b SHR 3 ; Move 01110b
```

9.2.1.5 Using Bitwise Logical Operators

The bitwise operators perform logical operations on each bit of an expressions. The expressions must resolve to constant values. Table 9.2 lists the logical operators and their meanings.

Table 9.2

Logical Operators

Operator	Syntax	Meaning
NOT	**NOT** *expression*	Bitwise complement
AND	*expression1* **AND** *expression2*	Bitwise AND
OR	*expression1* **OR** *expression2*	Bitwise inclusive OR
XOR	*expression1* **XOR** *expression2*	Bitwise exclusive OR

Note

Do not confuse the **NOT**, **AND**, **OR**, and **XOR** operators with the processor instructions having the same names. The operators work on integer constants only at assembly time. The processor instructions work on register or memory values at run time.

■ **Examples**

```
mov     ax,NOT 11110000b            ; AX contains 1111111100001111b
mov     ah,NOT 11110000b            ; AH contains 00001111b
mov     ah,01010101b AND 11110000b  ; AH contains 01010000b
mov     ah,01010101b OR  11110000b  ; AH contains 11110101b
mov     ah,01010101b XOR 11110000b  ; AH contains 10100101b
```

9.2.2 Using Relational Operators

The relational operators compare two expressions and and return true (–1) if the condition specified by the operator is satisfied, or false (0) if it is not. The expressions must resolve to constant values. Table 9.3 lists the operators and the values they return if the specified condition is satisfied.

Table 9.3

Relational Operators

Operator	Syntax	Returned Value
EQ	*expression1* **EQ** *expression2*	True if expressions are equal
NE	*expression1* **NE** *expression2*	True if expressions are not equal
LT	*expression1* **LT** *expression2*	True if left expression is less than right
LE	*expression1* **LE** *expression2*	True if left expression is less than or equal to right
GT	*expression1* **GT** *expression2*	True if left expression is greater than right
GE	*expression1* **GE** *expression2*	True if left expression is greater than or equal to right

Relational operators are typically used with conditional directives.

Note

The **EQ** and **NE** operators treat their arguments as 16-bit numbers. Numbers specified with the 16th bit on are considered negative (FFFF hexadecimal is –1 decimal). Therefore, the expression `-1 EQ 0FFFFh` is true, while the expression `-1 NE 0FFFFh` is false.

The **LT**, **LE**, **GT**, and **GE** operators treat their arguments as 17-bit numbers, where the 17th bit specifies the sign. Therefore, FFFF hexadecimal is 65,535 decimal, not –1. Therefore, the expression `1 GT -1` is true, while the expression `1 GT 0FFFFh` is false.

■ Examples

```
mov     ax,4 EQ 3   ; AX contains false ( 0)
mov     ax,4 NE 3   ; AX contains true  (-1)
mov     ax,4 LT 3   ; AX contains false ( 0)
mov     ax,4 LE 3   ; AX contains false ( 0)
mov     ax,4 GT 3   ; AX contains true  (-1)
mov     ax,4 GE 3   ; AX contains true  (-1)
```

9.2.3 Using the Segment-Override Operator

The segment-override operator (**:**) forces the address of a variable or label to be computed relative to a specific segment.

■ Syntax

*segmentregister***:***expression*
*segmentname***:***expression*
*groupname***:***expression*

If the segment is specified as *segmentregister*, the register must be **CS**, **DS**, **SS**, or **ES** (or **FS** or **GS** on the 80386). If the segment is specified as *segmentname* or *groupname*, the name must have been previously assigned to a segment register with an **ASSUME** directive and defined using a **SEGMENT** or **GROUP** directive. The *expression* can be a constant symbol or memory operand.

By default, the effective address of a memory operand is computed relative to the **DS**, **SS**, or **ES** register, depending on the instruction and operand type. Similarly, all labels are assumed to be **NEAR**. These default types can be overridden using the segment-override operator.

Note

When a segment override is given with an indexed operand, the segment must be specified outside the index operators. For example, `es:[di]` is correct, but `[es:di]` will generate an error.

■ **Examples**

```
mov     ax,es:[bx][si]
mov     _TEXT:far_label,ax
mov     ax,DGROUP:variable
mov     al,cs:0001h
```

9.2.4 Using Type Operators

This section describes the assembler operators that specify or analyze the types of memory operands and other expressions.

9.2.4.1 Using the PTR Operator

The **PTR** operator specifies the type for a variable or label.

■ **Syntax**

type **PTR** *expression*

The operator forces *expression* to be treated as having *type*. The *expression* can be any operand. The *type* can be **BYTE**, **WORD**, **DWORD**, **FWORD**, BWORD, or **TBYTE** for memory operands. It can be **NEAR**, **FAR**, or **PROC** for labels.

The **PTR** operator is typically used with forward references to explicitly define what size or distance a reference has. If it is not used, the assembler assumes a default size or distance for the reference. See Section 9.4 for more information on forward references.

The **PTR** operator is also used to enable instructions to access variables in ways that would otherwise generate errors. For example, you could use the **PTR** operator to access the high-order byte of a **WORD** size variable.

■ Example 1

```
            .DATA
stuff       DD      ?
buffer      DB      20 DUP (?)

            .CODE
            call    FAR PTR task             ; Call a near procedure
                                             ;   as far
            mov     bx,WORD PTR stuff[0]     ; Load a word from a
                                             ;   doubleword variable
            add     ax,WORD PTR buffer[bx]   ; Add a word from a
                                                 byte variable

task        PROC    NEAR
            .
            .
            .
task        ENDP
```

Example 1 shows how the **PTR** operator can override a previous definitions.

■ Example 2

```
; 80386 Only
_TEXT       SEGMENT DWORD PUBLIC USE32 'CODE'
            call    NEAR PTR [ebx]      ; Call 16-bit near procedure
            call    NEAR WORD [ebx]     ; Call 16-bit near procedure
            call    FAR PTR [ebx]       ; Call 16-bit far procedure
            call    DWORD PTR [ebx]     ; Call 16-bit far procedure
            call    DWORD PTR [ebx]     ; Call 32-bit near procedure
            call    FWORD PTR [ebx]     ; Call 32-bit far procedure
```

9.2.4.2 Using the SHORT Operator

The **SHORT** operator sets the type of a specified label to **SHORT**. Short labels can be used in **JMP** instructions whenever the distance from the label to the instruction is less than 128 bytes.

■ **Syntax**

SHORT *label*

Instructions using short labels are a byte smaller than identical instructions using the default near labels. See Section 9.4.1 for information on using the **SHORT** operator with the **JMP** instruction.

■ **Example**

```
            jmp     again              ; Jump 128 bytes or more
            .
            .
            .
            jmp     SHORT again        ; Jump less than 128 bytes
            .
            .
            .
again:
```

9.2.4.3 Using the THIS Operator

The **THIS** operator creates an operand whose offset and segment values are equal to the current location-counter value and whose type is specified by the operator.

■ **Syntax**

THIS *type*

The *type* can be **BYTE**, **WORD**, **DWORD**, **FWORD**, BWORD, or **TBYTE** for memory operands. It can be **NEAR**, **FAR**, or **PROC** for labels.

The **THIS** operator is typically used with the **EQU** or equal-sign (=) directive to create labels and variables. The result is similar to using the **LABEL** directive.

■ **Examples**

```
tag1        EQU     THIS BYTE  ; Both represent the same variable
tag2        LABEL   BYTE

check1      EQU     THIS NEAR  ; All represent the same address
check2      LABEL   NEAR
check3      PROC    NEAR
check4:
```

9.2.4.4 Using the HIGH and LOW Operators

The **HIGH** and **LOW** operators return the high and low bytes, respectively, of an expression.

■ **Syntax**

HIGH *expression*
LOW *expression*

The **HIGH** operator returns the high-order 8 bits of *expression*; the **LOW** operator returns the low-order 8 bits. The *expression* must evaluate to a constant or memory operand.

■ **Examples**

```
            .DATA
stuff       DW      ?
            .CODE
            mov     ah,HIGH stuff      ; Load high byte of word
            mov     al,LOW 0ABCDh      ; Load 0CDh
```

9.2.4.5 Using the SEG Operator

The **SEG** operator returns the segment address of an expression.

■ Syntax

SEG *expression*

The *expression* can be any label, variable, segment name, group name, or other memory operand. The **SEG** operator cannot be used with constant expressions. The returned value can be used as a memory operand.

■ Examples

```
        .DATA
var     DB      ?
        .CODE
        mov     ax,SEG var        ; Get address of segment
                                  ;  where variable is declared
here:
        mov     ax,SEG here       ; Get address of segment
                                  ;  where label is declared
```

9.2.4.6 Using the OFFSET Operator

The **OFFSET** operator returns the offset address of an expression.

■ Syntax

OFFSET *expression*

The *expression* can be any label, variable, or other memory operand. Constant expressions return meaningless values.

If simplified segment directives are given, the returned value is the number of bytes between the item and the beginning of **DGROUP** if the item is declared in a near data segment, or the number of bytes between the item and the beginning of the segment if the item is declared in a far segment.

If full segment definitions are given, the returned value is a memory operand equal to the number of bytes between the item and the beginning of the segment in which it is defined.

The segment-override operator (**:**) can be used to force **OFFSET** to return the number of bytes between the item in *expression* and the beginning of a named segment or group. This is the method used to generate valid offsets for items in a group when full segment definitions are used. For example, the statement

```
        mov     bx,OFFSET DGROUP:array
```

is not the same as

```
        mov     bx,OFFSET array
```

if `array` is not the first segment in DGROUP.

The expression used with the **OFFSET** operator must be a direct memory operand. The value it returns is an immediate (constant) operand. You can use the **LEA** instruction, as described in Section 15.3.1, to load the offset of an indirect memory operand.

■ **Examples**

```
        .DATA
var     DB      ?
        .CODE
        jmp     OFFSET place        ; Jump to offset of label
        mov     dx,OFFSET var       ; Load offset of variable
place   LABEL   FAR
```

9.2.4.7 Using the .TYPE Operator

The **.TYPE** operator returns a byte that defines the mode and scope of an expression.

■ **Syntax**

.TYPE *expression*

If the *expression* is not valid, **.TYPE** returns 0. Otherwise **.TYPE** returns a byte having the bit setting shown in Table 9.4. Only bits 0, 1, 5, and 7 are affected. Other bits are always zero.

Table 9.4

.TYPE Operator and Variable Attributes

Bit Position	If Bit = 0	If Bit = 1
0	Not program related	Program related
1	Not data related	Data related

5	Not defined	Defined
7	Local or public scope	External scope

The **.TYPE** operator is typically used in macros where different kinds of arguments may need to be handled differently.

■ Example

```
display     MACRO    string
            IF       ((.TYPE string) SHL 14) NE 8000h
            .ERR2
            IF2
            %OUT     Argument must be a variable
            ENDIF
            ENDIF
            mov      dx,OFFSET string
            mov      ah,09h
            int      21h
            ENDM
```

This macro checks to see if the argument passed to it is data related (a variable). It does this by shifting all bits except the relevant bits (1 and 0) left so that they can be checked. If the data bit is not set, an error is generated and a message sent to the screen.

9.2.4.8 Using the TYPE Operator

The **TYPE** operator returns a number representing the type of an expression.

■ Syntax

TYPE *expression*

If *expression* evaluates to a variable, the operator returns the number of bytes in each data object in the variable. Each byte in a string is considered a separate data object, so the **TYPE** operator returns 1 for strings.

If *expression* evaluates to a structure or structure variable, the operator returns the number of bytes in the structure. If *expression* is a label, the operator returns 0FFFFh if the label is **NEAR**, and 0FFFEh if the label is **FAR**. If *expression* is a constant, the operator returns 0.

The returned value can be used to specify the type for a **PTR** operator.

■ **Examples**

```
             .DATA
var          DW      ?
array        DD      10 DUP (?)
str          DB      "This is a test"
             .CODE
room:        .
             .
             .
             mov     ax,TYPE var            ; Puts 2 in AX
             mov     bx,TYPE array          ; Puts 4 in BX
             mov     cx,TYPE str            ; Puts 1 in CX
             jmp     (TYPE room) PTR room+2 ; Jumps to near label
                                            ;   2 bytes past "room"
```

9.2.4.9 Using the LENGTH Operator

The **LENGTH** operator returns the number of data elements in an array or other variable defined with the **DUP** operator.

■ **Syntax**

LENGTH *variable*

The returned value will be the number of elements of the declared size in the variable. If the variable was declared with nested **DUP** operators, only the value given for the outer **DUP** operator will be returned. If the variable was not declared with the **DUP** operator, the value returned is always 1.

■ **Examples**

```
array        DD      100 DUP(FFFFFFh)
table        DW      100 DUP(1,10 DUP(?))
string       DB      'This is a string'
var          DT      ?
larray       EQU     LENGTH array        ; 100 - numer of elements
ltable       EQU     LENGTH table        ; 100 - inner DUP not counted
lstring      EQU     LENGTH string       ; 1 - string is one element
lvar         EQU     LENGTH var          ; 1
             .
             .
             .
             mov     cx,LENGTH array     ; Load number of elements
again:       .                           ; Perform some operation on
```

```
        .                       ;    each element
        .
        loop    again
```

9.2.4.10 Using the SIZE Operator

The **SIZE** operator returns the total number of bytes allocated for an array or other variable defined with the **DUP** operator.

■ Syntax

SIZE *variable*

The returned value is equal to the value of **LENGTH** *variable* times the value of **TYPE** *variable*. If the variable was declared with nested **DUP** operators, only the value given for the outside **DUP** operator will be considered. If the variable was not declared with the **DUP** operator, the value returned is always **TYPE** *variable*.

■ Example

```
array       DD      100 DUP(1)
table       DW      100 DUP(1,10 DUP(?))
string      DB      'This is a string'
var         DT      ?
sarray      EQU     SIZE array          ; 400 - elements times size
stable      EQU     SIZE table          ; 200 - inner DUP ignored
sstring     EQU     SIZE string         ; 1 - string is one element
svar        EQU     SIZE var            ; 10 - variable equals size
            .
            .
            .
            mov     cx,SIZE array       ; Load number of bytes
again:      .                           ; Perform some operation on
            .                           ;   each byte
            .
            loop    again
```

9.2.5 Operator Precedence

Expressions are evaluated according to the following rules:

- Operations of highest precedence are performed first.
- Operations of equal precedence are performed from left to right.

- The order of evaluation can be overridden by using parentheses. Operations in parentheses are always performed before any adjacent operations.

The order of precedence for all operators is listed in Table 9.5. Operators on the same line have equal precedence.

Table 9.5

Operator Precedence

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

■ Examples

```
a           EQU     8 / 4 * 2           ; Equals 4
b           EQU     8 / (4 * 2)         ; Equals 1
c           EQU     8 + 4 * 2           ; Equals 16
d           EQU     (8 + 4) * 2         ; Equals 24
e           EQU     8 OR 4 AND 2        ; Equals 8
f           EQU     (8 OR 4) AND 3      ; Equals 0
```

9.3 Using the Location Counter

The location counter is a special operand that, during assembly, represents the address of the statement currently being assembled. At assembly-time, the location counter keeps changing, but when used in source code it resolves to a constant representing an address.

The location counter has the same attributes as a near label. It represents an offset that is relative to the current segment and is equal to the number of bytes generated for the segment to that point.

■ Example 1

```
string      DB      "Who wants to count every byte in a string, "
            DB      "especially if you might modify the source "
            DB      "code later."
lstring     EQU     $-string    ; Let the assembler do it
```

Example 1 shows one way of using the location counter operand in expressions relating to data.

■ Example 2

```
            cmp     ax,bx
            jl      shortjump   ; If ax < bx, go to "shortjump"
            .                   ;   else if ax >= bx, continue
            .
shortjump:  .

            cmp     ax,bx
            jge     $+3         ; If ax >= bx, continue
            jmp     longjump    ;   else if ax < bx, go to "longjump"
            .                   ; This is "$+3"
            .
longjump:   .
```

Example 2 illustrates how you can use the location counter to do conditional jumps of more than 128 bytes. The first part shows the normal way of coding jumps of less than 128 bytes, while the second part shows how to code the same jump when the label is more than 128 bytes away.

9.4 Using Forward References

The assembler permits you to refer to labels, variable names, segment names, and other symbols before they are declared in the source code. Such references are called forward references.

The assembler handles forward references by making assumptions about them on the first pass, then attempting to correct the assumptions, if necessary, on the second pass. Checking and correcting assumptions on the second pass takes processing time, so source code with forward references assembles slower than source code with no forward references.

In addition, the assembler may make incorrect assumptions that it cannot correct, or that it corrects at a cost in program efficiency.

9.4.1 Adjusting Forward References to Labels

Forward references to labels may result in incorrect or inefficient code.

In the statement below, the label `target` is a forward reference:

```
            jmp     target              ; Generates 3 bytes
            .                           ;   in 16-bit segment
            .
            .
target:
```

Since the assembler processes source files sequentially, `target` is unknown when it is first encountered. Assuming 16-bit segments, it could be one of three types: short (–128 to 127 bytes from the jump), near (–32,768 to 32,767 bytes from the jump), or far (in a different segment than the jump). **MASM** assumes that `target` is a near label, and generates the number of bytes necessary to specify a near label: 1 byte for the instruction and 2 bytes for the operand.

If on the second pass the assembler learns that `target` is a short label, it will need only 2 bytes: one for the instruction and one for the operand. It will pad the extra byte with a **NOP** instruction. If the assembler learns that `target` is a far label, it will need 5 bytes. Since it can't make this adjustment, it will generate a phase error.

You can override the assembler's assumptions by specifying the exact size of the jump. For example, if you know that a **JMP** instruction refers to a label less than 128 bytes from the jump, you can use the **SHORT** operator, as shown below:

```
            jmp     SHORT target        ; Generates 2 bytes
            .                           ;   in 16-bit segment
            .
            .
target:
```

Using the **SHORT** operator eliminates unnecessary **NOP** instructions that can decrease program speed and increase executable size. If the assembler has to pad with unnecessary **NOP** instructions because of forward references, it will generate a warning message if the warning level is 2. (The warning level can be set with the **/W** option as described in Section 2.4.13.) You can ignore the warning, or you can go back to the source code and change the code to eliminate the forward references.

Note

The **SHORT** operator in the example above would not be needed if `target` were located before the jump. The assembler would have already processed `target` and would be able to make adjustments based on its distance.

If you use the **SHORT** operator when the label being jumped to is more than 128 bytes away, **MASM** will generate an error message. You can either remove the **SHORT** operator, or try to reorganize your program to reduce the distance.

If a far jump is required, you can override the assembler's assumptions with the **FAR** and **PTR** operators, as shown below:

```
            jmp     FAR PTR target      ; Generates 5 bytes
            .                           ;   in 16-bit segment
            .
            .
target:                                 ; Different segment
```

If you use the **PROC** and **ENDP** directives to define procedures, the assembler automatically makes the correct assumption about whether the procedure label is near or far. However, you can override the assembler's assumptions, as shown below:

```
call    FAR PTR nproc     ; Call a near procedure as far
call    NEAR PTR fproc    ; Call a far procedure as near
```

■ **80386 Only**

If the 80386 processor is enabled for 32-bit segments, the limitations on jumps to forward references are different. A short jump (–32,768 to 32,767 bytes) generates 3 bytes. A near jump (–2,147,483,648 to 2,147,483,647) generates 5 bytes. A far jump (in a different segment) generates 7 bytes.

9.4.2 Adjusting Forward References to Variables

When **MASM** encounters code that references variables that have not yet been defined, it makes assumptions about the segment where the variable will be defined. If the assumptions turn out to be wrong, the wrong data may be processed.

These problems usually occur with complex segment structures that do not follow the Microsoft naming conventions. The problems never appear if simplified segment directives are used. Two examples are discussed in this section.

By default, **MASM** assumes that variables are referenced to the **DS** register. If a statement must access a variable in a segment not associated with the **DS** register, and if the variable has not yet been defined, you must use the segment override operator to specify the segment, as shown below:

```
CODE        SEGMENT BYTE PUBLIC 'CODE'
CODE        ENDS
DATA1       SEGMENT WORD PUBLIC 'DATA'
DATA1       ENDS
DATA2       SEGMENT WORD PUBLIC 'DATA'
DATA2       ENDS

CODE        SEGMENT BYTE PUBLIC 'CODE'
            ASSUME  cs:CODE,ds:DATA1,es:DATA2
            mov     ax,DATA2:var2
            .
            .
            .
CODE        ENDS

DATA1       SEGMENT WORD PUBLIC 'DATA'
```

```
var1         DW       10
DATA1        ENDS

DATA2        SEGMENT WORD PUBLIC 'DATA'
var2         DW       20
DATA2        ENDS
```

In this example, the segment-override operator is used to specify that var2 is in the DATA2 segment. When **MASM** first encounters var2, it knows the segment name because it was defined in a dummy segment at the start of the source code. However, the variable var2 is not yet known.

If the segment operator were not used, **MASM** would automatically assume that var2 is relative to **DS**. As a result, **MASM** would incorrectly use var1 instead of var2.

The situation is different if neither the variable nor the segment in which it is defined is known, as shown below:

```
GROUP2       GROUP    DATA2
CODE         SEGMENT BYTE PUBLIC 'CODE'
             ASSUME   cs:CODE,ds:DATA1,es:GROUP2
             mov      ax,DATA2:var2
             .
             .
             .
CODE         ENDS

DATA1        SEGMENT WORD PUBLIC 'DATA'
var1         DW       10
DATA1        ENDS

DATA2        SEGMENT WORD PUBLIC 'DATA'
var2         DW       20
DATA2        ENDS
```

In this case, the DATA2 segment has not yet been defined, but **MASM** can still use it in a segment override because the segment was named in the **GROUP** statement and assigned a segment register in the **ASSUME** statement. Use the **GROUP** statement for forward references where the segment has not yet been defined.

9.5 Strong Typing for Memory Operands

The assembler carries out strict syntax checks for all instruction statements, including strong typing for operands that refer to memory locations. This means that when an instruction uses two operands with implied data types, the operand types must match. Warning messages are generated for nonmatching types.

For example, in the following fragment, the variable `string` is incorrectly used in a move instruction:

```
            .DATA
string      DB       "A message."
            .CODE
            mov      ax,string[1]
```

The `AX` register has **WORD** type, but `string` has **BYTE** type. Therefore, the statement generates the warning message 37:

```
Operand types must match
```

To avoid all ambiguity and prevent the warning error, use the **PTR** operator to override the variable's type, as shown below:

```
            mov      ax,WORD PTR string[1]
```

You can ignore the warnings if you are willing to trust the assembler's assumptions. When a register and memory operand are mixed, the assembler assumes that the register operand is always the correct size. For example, in the statement

```
            mov      ax,string[1]
```

the assembler assumes that the programmer wishes the word size of the register to override the byte size of the variable. A word starting at `string[1]` will be moved into **AX**. In the statement

```
            mov      string[1],ax
```

the assembler assumes the programmer wishes to move the word value in **AX** into the word starting at `string[1]`. However, the assembler's assumptions are not always as clear as these examples. You ignore warnings about type mismatches at your own risk.

Note

Some assemblers (including early versions of the IBM Macro Assembler) do not do strict type checking. For compatibility with these assemblers, type errors are warnings rather than severe errors. Many assembly-language program listings in books and magazines are written for assemblers with weak typing checking. Such programs may produce warning messages, but assemble correctly. You can use the **/W** option to turn off type warnings if you are sure the code is correct.

Chapter 10
Assembling Conditionally

The Macro Assembler provides two types of conditional directives. Conditional-assembly directives test for a specified condition and assemble a block of statements if the condition is true. Conditional-error directives test for a specified condition and generate an error if the condition is true.

Both kinds of conditional directives test assembly-time conditions. They cannot test run-time conditions. Only expressions that evaluate to constants during assembly can be compared or tested.

Since macros and conditional-assembly directives are often used together, you may need to refer to Chapter 11, "Using Equates, Macros, and Repeat Blocks," to understand some of the examples in this chapter. In particular, conditional directives are frequently used with the special macro operators described in Section 11.4.

10.1 Using Conditional-Assembly Directives

The conditional-assembly directives include the following:

IF

IFE

IF1

IF2

IFDEF

IFNDEF

IFB

IFNB

IFIDN

IFDIF

ELSE

ENDIF

The **IF** directives and the **ENDIF** and **ELSE** directives can be used to enclose the statements to be considered for conditional assembly.

■ **Syntax**

IF *condition*

.

.

.

⟦**ELSE**

.

.

.

ENDIF

The statements following the **IF** directive can be any valid statements, including other conditional blocks. The **ELSE** directive and its statements are optional. **ENDIF** ends the block.

The statements in the conditional block are assembled only if the condition specified by the corresponding **IF** statement is satisfied. If the conditional block contains an **ELSE** directive, only the statements up to the **ELSE** directive will be assembled. The statements following the **ELSE** directive are assembled only if the **IF** statment is not met. An **ENDIF** directive must mark the end of any conditional-assembly block. No more than one **ELSE** directive is allowed for each **IF** statement.

IF statements can be nested up to 255 levels. To avoid ambiguity, a nested **ELSE** directive always belongs to the nearest preceding **IF** statement that does not have its own **ELSE**.

10.1.1 Testing Expressions with IF and IFE Directives

The **IF** and **IFE** directives test the value of an expression and grant assembly depending on whether it is true or not.

■ **Syntax**

IF *expression*
IFE *expression*

The **IF** directive grants assembly if the value of *expression* is true (nonzero). The **IFE** directive grants assembly if the value of *expression* is false (0). The *expression* must resolve to a constant value and must not contain forward references.

■ **Example**

```
IF      maclevel GT 4
PURGE   get_list, redir, cancel_redir, get_psp
ENDIF
```

In this example, the macros within the block will only be purged if the symbol `maclevel` is greater than 4.

10.1.2 Testing the Pass with IF1 and IF2 Directives

The **IF1** and **IF2** directives test the current assembly pass and grant assembly only on the specified pass. Multiple passes of the assembler are discussed in Section 2.5.7.

■ **Syntax**

IF1
IF2

The **IF1** directive grants assembly only on Pass 1. **IF2** grants assembly only on Pass 2. The directives take no arguments.

Since many statements are assembled once on each pass, these directives can prevent an action from being taken twice unnecessarily. For example, macros only need to be processed once. You can enclose blocks of macros in **IF1** blocks to prevent them from being reprocessed on the second pass.

■ **Example**

```
IF1
%OUT Beginning Pass 1
ELSE
%OUT Beginning Pass 2
ENDIF
```

10.1.3 Testing Symbol Definition with IFDEF and IFNDEF Directives

The **IFDEF** and **IFNDEF** directives test whether or not a symbol has been defined, and grant assembly based on the result.

■ Syntax

IFDEF *name*
IFNDEF *name*

The **IFDEF** directive grants assembly only if *name* is a defined label, variable, or symbol. The **IFNDEF** directive grants assembly if *name* has not yet been defined.

The name can be any valid name. Note that if *name* is a forward reference, it is considered undefined on Pass 1, but defined on Pass 2.

■ Example

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

In this example, `buff` is allocated only if `buffer` has been previously defined.

One way to use this conditional block would be to leave `buffer` undefined in the source file and define it if needed by using the **/D***symbol* option (see Section 2.4.4) when you start **MASM**. For example, if the conditional block is in `test.asm`, you could start the assembler with the following command line:

```
MASM /Dbuffer=1024 test;
```

The symbol `buffer` would be defined, and as a result the conditional-assembly block would allocate `buff`. However, if you didn't need `buff`, you could use the command line:

```
MASM test;
```

10.1.4 Verifying Macro Parameters with IFB and IFNB Directives

The **IFB** and **IFNB** directives test to see if a specified argument was passed to a macro, and grant assembly based on the result.

■ Syntax

IFB <*argument*>
IFNB <*argument*>

These directives are always used inside macros, and they always test to see if a real argument was passed for a specified dummy argument. The **IFB** directive grants assembly if *argument* is blank. The **IFNB** directive grants assembly if *argument* is not blank. The arguments can be any name, number, or expression. The angle brackets (< >) are required.

■ Example

```
@Write      MACRO    buffer,bytes,handle
            IFNB     <handle>
            mov      bx,handle         ; (1=stdout,2=stderr,3=aux,4=printer)
            ELSE
            mov      bx,1              ; Default standard out
            ENDIF
            mov      dx,OFFSET buffer; Address of buffer to write to
            mov      cx,bytes          ; Number of bytes to write
            mov      ah,40h
            int      21h
            ENDM
```

In this example, a default value is used if no value is specified for the third macro argument.

10.1.5 Comparing Macro Arguments with IFIDN and IFDIF Directives

The **IFIDN** and **IFDIF** directives compare two macro arguments, and grant assembly depending on whether they are the same.

■ Syntax

IFIDN <*argument1*>,<*argument2*>
IFDIF <*argument1*>,<*argument2*>

These directives are always used inside macros, and they always test to see if a real arguments passed for two specified dummy arguments are the same. The **IFIDN** directive grants assembly if *argument1* and *argument2*. are identical. The **IFDIF** directive grants assembly if *argument1* and *argument2* are different. The arguments can be any names, numbers, or expressions. To be identical, each character in *argument1* must match the corresponding character in *argument2*.

Warning

Case is significant. The angle brackets (< >) are required. The arguments must be separated by a comma.

■ Example

```
divide      MACRO   numerator, denominator
            IFDIF   <denominator>,<0>   ;; If not dividing by zero
            mov     ax, numerator       ;;   divide AX by BX
            mov     bx, denominator
            div     bx                  ;; Result in accumulator
            ELSE
            %OUT    Can't divide by zero!
            ENDIF
            ENDM

            divide  6,%tst
```

In this example, a macro uses the **IFDIF** directive to check against dividing by a constant that evaluates to 0. The macro is then called, using a percent sign (%) on the second parameter so that the value of the parameter, rather than its name, will be evaluated. See Section 11.4.4 for a discussion of the expression (%) operator.

If the parameter `tst` was previously defined with the statement

```
tst         EQU     0
```

then the condition fails and the code in the block will not be assembled. However, if the parameter `tst` was defined with the statement

```
tst         DW      0
```

error 42, `Constant was expected`, will be generated. This is because the assembler cannot evaluate the run-time value of `tst`.

10.2 Using Conditional-Error Directives

Conditional-error directives can be used 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.

The conditional-error directives, and the errors they produce, are listed in Table 7.1.

Table 10.1

Conditional Error Directives

Directive	Number	Message
.ERR1	87	`Forced error - pass1`
.ERR2	88	`Forced error - pass2`
.ERR	89	`Forced error`
.ERRE	90	`Forced error - expression equals 0`
.ERRNZ	91	`Forced error - expression not equal 0`
.ERRNDEF	92	`Forced error - symbol not defined`
.ERRDEF	93	`Forced error - symbol defined`
.ERRB	94	`Forced error - string blank`
.ERRNB	95	`Forced error - string not blank`
.ERRIDN	96	`Forced error - strings identical`
.ERRDIF	97	`Forced error - strings different`

Like other severe errors, those generated by conditional-error directives cause the assembler to return exit code 7. If a severe error is encountered during assembly, **MASM** will delete the object module. All conditional error directives except **ERR1** generate severe errors.

10.2.1 Generating Unconditional Errors with .ERR, .ERR1, and .ERR2 Directives

The **.ERR**, **.ERR1**, and **.ERR2** directives force an error at the points at which they occur in the source file. The error is generated unconditionally when the directive is encountered, but they can be placed within conditional assembly blocks to limit the errors to certain situations.

■ **Syntax**

.ERR
.ERR1
.ERR2

The **.ERR** directive forces an error regardless of the pass, while the **.ERR1** and **.ERR2** directives force the error only on their respective passes. The **.ERR1** directive only appears on the screen or in the listing file if you use the **/D** option to request a Pass 1 listing.

You can place these directives within conditional-assembly blocks or macros to see which blocks are being expanded.

■ **Example**

```
IFDEF       dos
            .
            .
            .
ELSE
IFDEF       xenix
            .
            .
            .
            ELSE
            .ERR
            ENDIF
            ENDIF
```

This example makes sure that either the symbol `dos` or the symbol `xenix` is defined. If neither is defined, the nested **ELSE** condition is assembled and an error message is generated. Since the **.ERR** directive is used, an error would be generated on each pass. If you wanted a single severe error, you could use the **.ERR2** directive.

10.2.2 Testing Expressions with .ERRE or .ERRNZ Directives

The **.ERRE** and **.ERRNZ** directives test the value of an expression and conditionally generate an error depending on whether the expression is true (nonzero) or false (0).

■ Syntax

.ERRE *expression*
.ERRNZ *expression*

The **.ERRE** directive generates an error if the *expression* is false (0). The **.ERRNZ** directive generates an error if the *expression* is true (nonzero). The *expression* must resolve to a constant value and must not contain forward references.

■ Example

```
buffer      MACRO    count,bname
            .ERRE    count LE 128        ;; Allocate memory, but
bname       DB       count DUP(0)        ;;   no more than 128 bytes
            ENDM
            .
            .
            .
            buffer   128,buf1            ; Data allocated - no error
            buffer   129,buf2            ; Error generated
```

In this example, the **.ERRE** directive is used to check the boundaries of a parameter passed to the macro `buffer`. If `count` is less than or equal to 128, the expression being tested by the error directive will be true (nonzero) and no error will be generated. If `count` is greater than 128, the expression will be false (0) and the error will be generated.

10.2.3 Verifying Symbol Definition with .ERRDEF and .ERRNDEF Directives

The **.ERRDEF** and **.ERRNDEF** directives test whether or not a symbol is defined, and conditionally generate an error depending on the result.

■ **Syntax**

.ERRDEF *name*
.ERRNDEF *name*

The **.ERRDEF** directive produces an error if *name* is defined as a label, variable, or symbol. The **.ERRNDEF** directive produces an error if *name* has not yet been defined. If *name* is a forward reference, it is considered undefined on Pass 1, but defined on Pass 2.

■ **Example**

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

In this example, the **.ERRNDEF** directive at the beginning of a conditional block makes sure that a symbol being tested in the block actually exists.

10.2.4 Testing for Macro Parameters with .ERRB and .ERRNB Directives

The **.ERRB** and **.ERRNB** directives test to see if a specified argument was passed to a macro, and conditionally generate an error based on the result.

■ **Syntax**

.ERRB <*argument*>
.ERRNB <*argument*>

These directives are always used inside macros, and they always test to see if a real argument was passed for a specified dummy argument. The **.ERRB** directive generates an error if *argument* is blank. The **.ERRNB** directive generates an error if *argument* is not blank. The *argument* can be any name, number, or expression. The angle brackets (< >) are required.

■ Example

```
work        MACRO   realarg,testarg
            .ERRB   <realarg>   ;; Error if no parameters
            .ERRNB  <testarg>   ;; Error if more than one parameter
            .
            .
            .
            ENDM
```

In this example, error directives are used to make sure that one, and only one, argument is passed to the macro. The **.ERRB** directive generates an error if no argument is passed to the macro. The **.ERRNB** directive generates an error if more than one argument is passed to the macro.

10.2.5 Comparing Macro Arguments with .ERRIDN and .ERRDIF Directives

The **.ERRIDN** and **.ERRDIF** directives compare two macro arguments, and conditionally generate an error depending on whether they are identical.

■ Syntax

.ERRIDN <*argument1*>,<*argument2*>
.ERRDIF <*argument1*>,<*argument2*>

These directives are always used inside macros, and they always compare the real arguments specified for two parameters. The **.ERRIDN** directive generates an error if the arguments are identical. The **.ERRDIF** generates an error if the arguments are different. The arguments can be names, numbers, or expressions. To be identical, each character in *argument1* must match the corresponding character in *argument2*.

Warning

Case is significant. The angle brackets (< >) are required. The arguments must be separated by a comma.

■ Example

```
addem       MACRO   ad1,ad2,sum
            .ERRIDN <ax>,<ad2> ;; Error if ad2 is "ax"
            .ERRIDN <AX>,<ad2> ;; Error if ad2 is "AX"
            mov     ax,ad1     ;; Would overwrite if ad2 were AX
            add     ax,ad2
            mov     sum,ax     ;; Sum must be register or memory
            ENDM
```

In this example, the **.ERRIDN** directive is used to protect against passing the **AX** register as the second parameter, since this would cause the macro to fail. Note that the directive is used twice to protect against the two most likely spellings.

Chapter 11
Using Equates, Macros, and Repeat Blocks

This chapter explains how to use equates, macros, and repeat blocks. Equates are constant values assigned to symbols so that the symbol can be used in place of the value. Macros are series of statements that are assigned a symbolic name (and optionally parameters) so that the symbol can be used in place of the statements. Repeat blocks are a special form of macro used to do repeated statements.

Both equates and macros are processed at assembly time. They can simplify writing source code by allowing the user to substitute mnemonic names for constants and repetitive code. By changing a macro or equate, a programmer can change the effect of statements throughout the source code.

In exchange for these conveniences, the programmer loses some assembly-time efficiency. A program that uses macros and equates extensively may assemble slightly slower than the same program written without them. However, the program without macros and equates will usually take longer to write and will be more difficult to maintain.

11.1 Using Equates

The equate directives enable you to use symbols that represent numeric or string constants. **MASM** recognizes three kinds of equates: redefinable numeric equates, nonredefinable numeric equates, and string equates (also called text macros).

11.1.1 Using Redefinable Numeric Equates

Redefinable numeric equates are used to assign a numeric constant to a symbol. The value of the symbol can be redefined at any point during assembly time. Although the value of a redefinable equate may be different at different points in the source code, a constant value will be assigned for each use and that value will not change at run time.

Redefinable equates are often used for assembly-time calculations in macros and repeat blocks.

■ Syntax

name= *expression*

The equal-sign (=) directive creates or redefines a constant symbol by assigning the numeric value of *expression* to *name*. No storage is allocated for the symbol. The symbol can be used in subsequent statements as an immediate operand having the assigned value. It can be redefined at any time.

The *expression* can be an integer, a one- or two-character string constant, a constant expression, or an expression that evaluates to an address. The value of the expression must not exceed 65,535. The *name* must be either a unique name or a name previously defined using the equal-sign (=) directive.

Note

Redefinable equates must be assigned numeric values. String constants longer than two characters cannot be used.

■ Example

```
count       =       26                  ; Define "count"
letter      =       'z'                 ; Define "letter"

alphabet    LABEL   BYTE
            REPT    count               ; Use "count"
            DB      letter              ; Use "letter"
            DB      count               ; Use "count"
letter      =       letter - 1          ; Redefine "letter"
count       =       count - 1           ; Redefine "count"
            ENDM
```

This example redefines equates inside a repeat block to declare of series of variables initialized to alternating letters and numbers. See Section 11.3 for more information on repeat blocks.

11.1.2 Using Nonredefinable Numeric Equates

Nonredefinable numeric equates are used to assign a numeric constant to a symbol. The value of the symbol cannot be redefined.

Nonredefinable equates are often used for assigning mnemonic names to constant values. This can make the code more readable and easier to maintain. If a constant value used in numerous places in the source code needs to be changed, then the equate can be changed in one place rather than throughout the source code.

■ Syntax

name **EQU** *expression*

The **EQU** directive creates constant symbols by assigning *expression* to *name*. The assembler replaces each subsequent occurrence of *name* with the value of *expression*.

Note

String constants can also be defined with the **EQU** directive, but the syntax is different, as described in Section 11.1.3.

No storage is allocated for the symbol. Symbols defined with numeric values can be used in subsequent statements as immediate operands having the assigned value.

■ Examples

```
column      EQU     80                  ; Numeric constant 80
row         EQU     25                  ; Numeric constant 25
screenful   EQU     column * row        ; Numeric constant 2000

            .DATA
buffer      DW      screenful

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

11.1.3 Using String Equates

String equates (or text macros) are used to assign a string constant to a symbol. String equates can be used in a variety of contexts, including defining aliases and string constants.

■ **Syntax**

name **EQU** ⟦<⟧*string*⟦>⟧

The **EQU** directive creates constant symbols by assigning *string* to *name*. The assembler replaces each subsequent occurrence of *name* with *string*. Symbols defined to represent strings with the **EQU** directive can be redefined to new strings. Symbols cannot be defined to represent strings with the = directive.

A special kind of string equate is an alias. An alias is a symbol that represents another symbol or a keyword.

Note

The use of angle brackets to force string evaluation is a new feature of **MASM** Version 4.5. Previous versions of **MASM** first tried to evaluate equates as expressions. If the string did not evaluate to a valid expression, **MASM** evaluated it as a string. This behavior sometimes caused unexpected consequences.

For example, the statement

```
rt          EQU     run-time
```

would be evaluated as `run` minus `time`, even though the user might intend to define the string `run-time`. If `run` and `time` were not already defined as numeric equates, the statement would generate an error. Using angle brackets solves this problem. The statement

```
rt          EQU     <run-time>
```

is evaluated as the string `run-time`.

When maintaining existing source code, you can leave string equates that evaluate correctly alone, but for new source code that will not be used with previous versions of **MASM**, it is a good idea to enclose all string equates in angle brackets.

■ Examples

```
; String equate definitions
pi          EQU     <3.1415>            ; String constant "3.1415"
prompt      EQU     <'Type Name: '>     ; String constant "'Type Name: '"
query       EQU     prompt              ; Alias for "prompt"
WPT         EQU     <WORD PTR>          ; String constant for "WORD PTR"
arg1        EQU     <[bp+4]>            ; String constant for "[bp+4]"

; Use of string equates
            .DATA
message     DB      query               ; Allocate string "'Type Name:'"
pie         DQ      pi                  ; Allocate real number 3.1415

            .CODE
            mov     ax,WPT message[3]   ; Load byte elements
                                        ;   into word register
            inc     parm1               ; Increment value of first
                                        ;   argument passed on stack
```

11.2 Using Macros

Macros enable you to assign a symbolic name to a block of source statements, then use that name in your source file to represent the statements. Parameters can also be defined to represent arguments passed to the macro.

Macro expansion is a text-processing function that occurs at assembly time. Each time **MASM** encounters the text of a macro name, it replaces that text with the text of the statements in the macro definition. Similarly, the text of dummy parameter names is replaced with the text of the corresponding actual arguments.

A macro can be defined any place in the source file as long as the definition precedes the first source line that calls that macro. Macros and equates are often kept in a separate file and made available to the program through an **INCLUDE** directive (see Section 11.6.1) at the start of the source code.

Often a task can be done by either a macro or procedure. For example, the addup procedure shown in Section 17.4.3 does the same thing as the addup macro in Section 11.2.1. Macros are expanded on every occurrence of the macro name, so they can increase the length of the executable file if called repeatedly. Procedures are coded only once in the executable file, but the increased overhead of saving and restoring addresses and parameters can make them slower.

The next sections tell how to define and call macros. Repeat blocks, a special form of macro for doing repeated operations, are discussed separately in Section 11.3.

11.2.1 Defining Macros

The **MACRO** and **ENDM** directives are used to define macros. **MACRO** designates the beginning of the macro block and **ENDM** designates the end.

■ Syntax

name **MACRO** ⟦*parameter* ⟦*,parameter...*⟧⟧
.
.
.
ENDM

The *name* must be a valid symbol name and must be unique. The name can be used later in the source file to invoke the macro.

The *parameters* (sometimes called a dummy parameters) are names that act as placeholders for values to be passed as arguments to the macro when it is called. Any number of *parameters* can be specified, but they must all fit on one line. If you give more than one, you must separate them with commas.

Note

This manual uses the term "parameter" to refer to a placeholder for a value that will be passed to a macro or procedure. Parameters appear in macro or procedure definitions. The term "argument" is used to refer to an actual value passed to the macro or procedure when it is called.

Any valid assembler statements may be placed within a macro, including statements that call or define other macros. Any number of statements can be used. The *parameters* can be used any number of times in the statements. Macros can be nested, redefined, or used recursively, as explained in Section 11.5.

MASM assembles the statements in a macro only if the macro is called, and only at the point in the source file from which it is called. The macro definition itself is never assembled.

A macro definition can include the **LOCAL** directive, which lets you define labels used only within a macro, or the **EXITM** directive, which allows you to exit from a macro before all the statements in the block are expanded. These directives are discussed in Sections 11.2.3 and 11.2.4. Macro operators can also be used in macro definitions, as described in Section 11.4.

■ **Example**

```
addup       MACRO   ad1,ad2,ad3
            mov     ax,ad1      ;; First parameter in AX
            add     ax,ad2      ;; Add next two parameters
            add     ax,ad3      ;;   and leave sum in AX
            ENDM
```

The preceding example defines a macro named `addup`, which uses three parameters to add three values and leave their sum in the **AX** register. The three parameters will be replaced with arguments when the macro is called.

11.2.2 Calling Macros

A macro call directs **MASM** to copy the statements of the macro to the point of the call and to replace any parameters in the macro statements with the corresponding actual arguments.

■ **Syntax**

name ⟦*argument* ⟦**,***argument*...⟧⟧

The *name* must be the name of a macro defined earlier in the source file. The *arguments* can be any text. For example, symbols, constants, and registers are often given as arguments. Any number of arguments can be given, but they must all fit on one line. Multiple arguments must be separated by commas, spaces, or tabs.

MASM replaces the first parameter with the first argument, the second parameter with the second argument, and so on. If a macro call has more arguments than the macro has parameters, the extra arguments are ignored. If a call has fewer arguments than the macro has parameters, any remaining parameters are replaced with a null (empty) string.

You can use conditional statements to enable macros to check for null strings or other types of arguments. The macro can take then take appropriate action to adjust to different kinds of arguments. See Chapter 10, "Assembling Conditionally," for more information on using conditional-assembly and conditional-error directives to test macro arguments.

■ Example

```
addup       MACRO   ad1,ad2,ad3         ; Macro definition
            mov     ax,ad1              ;; First parameter in AX
            add     ax,ad2              ;; Add next two parameters
            add     ax,ad3              ;;   and leave sum in AX
            ENDM
            .
            .
            .
            addup   bx,2,count          ; Macro call
```

When the addup macro is called, **MASM** replaces the dummy parameters with the actual parameters given in the macro call. In the example above, the assembler would expand the macro call to the following code:

```
            mov     ax,bx
            add     ax,2
            add     ax,count
```

This code could be shown in an assembler listing, depending on whether the **.LALL**, **.XALL**, or **.SALL** directive was in effect (see Section 12.3.3).

11.2.3 Using Local Symbols

The **LOCAL** directive can be used within a macro to define symbols that are available only within the defined macro.

Note

In this context, the term "local" is not related to the public availability of a symbol, as described in Chapter 8, "Creating Programs from Multiple Modules," or to variables that are defined to be local to a procedure, as described in Section 17.4.4. Local simply means that the symbol is not known outside the macro where it is defined.

■ Syntax

LOCAL *localname* ⟦,*localname...*⟧

The *localname* is a temporary symbol name that is to be replaced by a unique symbol name when the macro is expanded. At least one *localname* is required for each **LOCAL** directive. If more than one local symbol is given, the names must be separated with commas. Once declared, a *localname* can be used in any statement within the macro definition.

MASM creates a new actual name for *localname* each time the macro is expanded. The actual name has the following form:

??*number*

The *number* is a hexadecimal number in the range 0000 to 0FFFF. You should not give other symbols names in this format, since doing so may produce a symbol with multiple definitions. In listings, the local name is shown in the macro definition, but the actual name is shown in expansions of macro calls.

Nonlocal labels may be used in macros, but if the macro is used more than once, the same label will appear in both expansions, and **MASM** will display an error message, indicating that the file contains a symbol with multiple definitions. To avoid this problem, use only local labels (or redefinable equates) in macros.

Note

The **LOCAL** directive can only be used in macro definitions, and it must precede all other statements in the definition. If you try another statement (such as a comment instruction) before the **LOCAL** directive, a warning error will be generated.

■ Example

```
power       MACRO   factor,exponent     ;; Macro definition
            LOCAL   again,gotzero       ;; Declare symbols for macro
            mov     cx,exponent         ;; Exponent is count for loop
            mov     ax,1                ;; Multiply by 1 first time
            jcxz    gotzero             ;; Get out if exponent is zero
            mov     bx,factor
again:      mul     bx                  ;; Multiply until done
            loop    again
gotzero:
            ENDM
```

In this example, the **LOCAL** directive defines the dummy names `again` and `gotzero` as labels to be used within the `power` macro.

These dummy names will be replaced with unique names each time the macro is expanded. For example, the first time the macro is called, `again` will be assigned the name `??0000` and `gotzero` will be assigned `??0001`. The second time through `again` will be assigned `??0002` and `gotzero` will be assigned `??0003`, and so on.

11.2.4 Exiting from a Macro

Normally, **MASM** processes all the statements in a macro definition and exits after it completes the last statement. However, you can use the **EXITM** directive to tell the assembler to terminate macro expansion and continue assembly with the next statement after the macro call.

When the **EXITM** directive is encountered, the assembler exits the macro or repeat block immediately. Any remaining statements in the macro or repeat block are not processed. If **EXITM** (see Section 11.2.4) is encountered in a nested macro or repeat block, **MASM** returns to expanding the outer block.

The **EXITM** directive is typically used with conditional directives to skip the last statements in a macro under specified conditions. Often macros using the **EXITM** directive contain repeat blocks or are called recursively.

■ Example

```
alloc       MACRO   times        ; Macro definition
x           =       0
            REPT    times        ;; Repeat up to 256 times
            IFE     x - OFFh     ;; Does x = 255 yet?
            EXITM                ;; If so, quit
            ELSE
            DB      x            ;; Else allocate x
            ENDIF
x           =       x + 1        ;; Increment x
            ENDM
            ENDM
```

This example defines a macro that creates no more than 255 bytes of data. The macro contains an **IFE** directive that checks the expression `x - OFFh`. When the value of this expression is true (x equals 255), the **EXITM** directive is processed and expansion of the macro stops.

11.3 Defining Repeat Blocks

Repeat blocks are a special form of macro that allows you to create blocks of repeated statements. They are different from macros in that they are not named, and thus cannot be called. However, like macros, they can have dummy parameters that are replaced by actual arguments during assembly. Macro operators, symbols declared with the **LOCAL** directive, and the **EXITM** directive can be used in repeat blocks. Like macros, repeat blocks are always terminated by an **ENDM** directive.

Repeat blocks are frequently placed in macros in order to repeat some of the statements in the macro. They can also be used independently, usually for declaring arrays with repeated data elements.

Three different kinds of repeat blocks can be defined using the **REPT**, **IRP**, and **IRPC** directives. The difference between them is in how the number of repetitions is specified.

11.3.1 Using the REPT and ENDM Directives

The **REPT** directive is used to create repeat blocks in which the number of repetitions is specified with a numeric argument.

■ **Syntax**

REPT *expression*

.

.

.

ENDM

The *expression* must evaluate to a numeric constant (a 16-bit unsigned number). It specifies the number of repetitions. Any valid assembler statements may be placed within the repeat block.

■ **Example**

```
x               =       0           ; Initialize

                REPT    100         ; Specify 100 repetitions
x               =       x + 1       ; Increment
                DB      x           ; Allocate
                ENDM
```

This example repeats the equal-sign (=) and **DB** directives 100 times. The resulting statements create 100 bytes of data whose values range from 1 to 100.

11.3.2 Using the IRP and ENDM Directives

The **IRP** directive is used to create repeat blocks in which the number or repetitions as well as parameters for each repetition are specified in a list of arguments.

■ **Syntax**

IRP *dummyname*,<*parameter*⟦,*parameter*...⟧>

.

.

.

ENDM

The assembler statements inside the block are repeated once for each *parameter* in the list enclosed by angle brackets (< >). The *dummyname* is a name for a placeholder to be replaced by the current argument. Each argument can be any text, such as a symbol, string, or numeric constant.

Any number of parameters can be given. If more than one parameter is given, they must be separated with commas. The angle brackets (< >) around the parameter list are required. The *dummyname* can be used any number of times in the *statements*.

When **MASM** encounters an **IRP** directive, it makes one copy of the statements for each argument in the enclosed list. While copying the statements, it substitutes the current argument for all occurrences of *dummyname* in these statements. If a null argument (< >) is found in the list, the dummy name is replaced with a null value. If the argument list is empty, the **IRP** directive is ignored and no statements are copied.

■ **Example**

```
IRP     x,<0,1,2,3,4,5,6,7,8,9>
DB      10 DUP(x)
ENDM
```

This example repeats the **DB** directive 10 times, duplicating the numbers in the list once for each repetition. The resulting statements create 100 bytes of data with the sequence 0–9 duplicated 10 times.

11.3.3 Using the IRPC and ENDM Directives

The **IRPC** directive is used to create repeat blocks in which the number of repetitions, as well as arguments for each repetition, are specified in a string.

■ **Syntax**

IRPC *dummyname,string*

.

.

.

ENDM

The assembler statements inside the block are repeated once for each character in *string*. The *dummyname* is a name for a placeholder to be replaced by the current character in *string*. The string can be any combination of letters, digits, and other characters. It should be enclosed with angle brackets (< >) if it contains spaces, commas, or other separating characters. The *dummyname* can be used any number of times in these statements.

When **MASM** encounters an **IRPC** directive, it makes one copy of the statements for each character in the string. While copying the statements, it substitutes the current character for all occurrences of *dummyname* in these statements.

■ **Example 1**

```
IRPC    x,0123456789
DB      x + 1
ENDM
```

Example 1 repeats the **DB** directive 10 times, once for each character in the string `0123456789`. The resulting statements create 10 bytes of data having the values 0–9.

■ **Example 2**

```
IRPC    letter,ABCDEFGHIJKLMNOPQRSTUVWXYZ
DB      '&letter'          ; Allocate uppercase letter
DB      '&letter'+20h      ; Allocate lowercase letter
DB      '&letter'-40h      ; Allocate number of letter
ENDM
```

Example 2 allocates the ASCII codes for uppercase, lowercase, and numeric versions of each letter in the string. Notice that the substitute operator (&) is required so that the `letter` will be treated as an argument rather than a string. See Section 11.4.1 for more information on the substitute operator.

11.4 Using Macro Operators

Macro and conditional directives use the following special set of macro operators:

Operator	Definition
&	Substitute operator

< >	Literal-text operator
!	Literal-character operator
%	Expression operator
;;	Macro comment

When used in a macro definition, a macro call, a repeat block, or as the argument of a conditional-assembly directive, these operators carry out special control operations, such as text substitution.

11.4.1 Using the Substitute Operator

The substitute operator (&) forces **MASM** to replace a parameter with its corresponding actual argument value.

■ Syntax

&parameter

The operator can be used when a parameter immediately precedes or follows other characters, or whenever the parameter appears in a quoted string.

■ Example

```
errgen      MACRO   y,x
            PUBLIC  err&y
err&y       DB      'Error &y: &x'
            ENDM
```

In the example, **MASM** replaces &x with the value of the argument passed to the macro `errgen`. If the macro is called with the statement

```
            errgen  5,<Unreadable disk>
```

the macro is expanded to

```
err5        DB      'Error 5: Unreadable disk'
```

Note

For complex, nested macros, you can use extra ampersands to delay the replacement of a parameter. In general, you need to supply as many ampersands as there are levels of nesting.

For example, in the following macro definition, the substitute operator is used twice with z to make sure its replacement occurs while the **IRP** directive is being processed:

```
alloc       MACRO   x
            IRP     z,<1,2,3>
x&&z        DB      z
            ENDM
            ENDM
```

In this example, the dummy parameter x is replaced immediately when the macro is called. The dummy parameter z, however, is not replaced until the **IRP** directive is processed. This means the parameter is replaced once for each number in the **IRP** parameter list. If the macro is called with

```
            alloc   var
```

the macro will be expanded as shown below:

```
var1    DB      1
var2    DB      2
var3    DB      3
```

11.4.2 Using the Literal-Text Operator

The literal-text operator directs **MASM** to treat a list as a single string rather than as separate arguments.

■ **Syntax**

<*text*>

The *text* is considered a single literal element regardless of whether it contains commas, spaces, or tabs. The operator is most often used in macro calls and with the **IRP** directive to ensure that values in a parameter list are treated as a single parameter.

The literal text operator can also be used to force **MASM** to treat special characters such as the semicolon or the ampersand literally. For example, the semicolon inside angle brackets <;> becomes a semicolon, not a comment indicator.

MASM removes one set of angle brackets each time the parameter is used in a macro. When using nested macros, you will need to supply as many sets of angle brackets as there are levels of nesting.

■ Example

```
alloc   1,2,3,4,5           ; Passes five parameters
                            ;   to "alloc"

alloc   <1,2,3,4,5>         ; Passes one five-element
                            ;   parameter to "alloc"
```

Note

When the **IRP** directive is used inside a macro definition and the argument list of the **IRP** directive is also a parameter of the macro, you must use the literal text operator (angle brackets) to enclose the macro parameter.

For example, in the following macro definition, the parameter x is used as the argument list for the **IRP** directive:

```
alloc       MACRO   x
            IRP     y,<x>
            DB      y
            ENDM
            ENDM
```

If this macro is called with

```
            alloc   <0,1,2,3,4,5,6,7,8,9>
```

the macro removes the angle brackets from the parameter so that is is expanded as 0,1,2,3,4,5,6,7,8,9. The brackets inside the repeat block are necessary to put the angle brackets back on. The repeat block is then expanded as shown below:

```
            IRP     y,<0,1,2,3,4,5,6,7,8,9>
            DB      y
```

```
          ENDM
```

11.4.3 Using the Literal-Character Operator

The literal-character operator forces the assembler to treat a specified character literally rather than as a symbol.

■ **Syntax**

!*character*

It is used with special characters such as the semicolon or ampersand when their special meaning must be suppressed. Using the literal character operator is the same as closing a single character in brackets. For example, `!!` is the same as <!>.

■ **Example**

```
errgen      MACRO   y,x
error&y     DB      'Error &y - &x'
            ENDM
            .
            .
            .
            errgen  103,<Expression !> 255>
```

The example macro call is expanded to allocate the string `Error 103 - Expression > 255`. Without the literal character operator, the greater-than symbol would be interpreted as the end of the argument and an error would result.

11.4.4 Using the Expression Operator

The expression operator (%) causes the assembler to treat the argument following the operator as an expression.

■ Syntax

%text

MASM computes the expression's value, using numbers of the current radix, and replaces *text* with this new value.

The expression operator is typically used in macro calls where the programmer needs to pass the result of an expression rather than the actual expression to a macro.

■ Example

```
printe  MACRO   msg,num
        IF2                     ;; On pass 2 only
        %OUT    * &msg&num *    ;; Display message and number
        ENDIF                   ;;   to screen
        ENDM

sym1    EQU     100
sym2    EQU     200

        printe  <sym1 + sym2 = >,%(sym1 + sym2) ; Macro call
```

In this example, the macro call

```
        printe  <sym1 + sym2 = >,%(sym1 + sym2)
```

passes the text literal `sym1 + sym2 =` to the dummy parameter `msg`. It passes the value 300 (the result of the expression `sym1 + sym2`) to the dummy parameter `num`. The result is that **MASM** displays the message

```
sym1 + sym2 = 300
```

when it reaches the macro call during the assembly. The **%OUT** directive, which sends a message to the screen, is described in Section 12.1 and the **IF2** directive is described in Section 10.1.2.

11.4.5 Specifying Macro Comments

A macro comment is any text in a macro definition that does not need to be copied in the macro expansion. A double semicolon (;;) is used to start a macro comment.

■ **Syntax**

;;*text*

All *text* following the double semicolon is ignored by the assembler and will appear only in the macro definition when the source listing is created.

The regular comment operator (;) can also be used in macros. However, regular comments may appear in listings when the macro is expanded. Macro comments will appear in the macro definition, but not in macro expansions. Whether or not regular comments are listed in macro expansions depends on the use of the **.LALL**, **.XALL**, and **.SALL** directives, as described in Section 12.3.3.

11.5 Using Recursive, Nested, and Redefined Macros

The concept of replacing macro names with predefined macro text is simple, but in practice it has many implications and potentially unexpected side effects. The following sections discuss advanced macro features, such as nesting, recursion, and redefinition, and point out some side effects of macros.

11.5.1 Using Recursion

Macro definitions can be recursive: that is, they can call themselves. Recursive macros are one way of doing repeated operations. The macro does a task, then calls itself to do the task again. The recursion is repeated until a specified condition is met.

■ **Example**

```
pushall     MACRO   reg1,reg2,reg3,reg4,reg5,reg6
            IFNB    <reg1>          ;; If parameter not blank
            push    reg1            ;;   push one register and repeat
            pushall reg2,reg3,reg4,reg5,reg6
            ENDIF
            ENDM
            .
            .
            .
pushall     ax,bx,si,ds
```

```
pushall     cs,es
```

In this example, the `pushall` macro repeatedly calls itself to push a register given in a parameter until there are no parameters left to push. A variable number of parameters (up to six) can be given.

11.5.2 Nesting Macro Definitions

One macro can define another. **MASM** does not process nested definitions until the outer macro has been called. Therefore, nested macros cannot be called until the outer macro has been called at least once. Macro definitions can be nested to any depth. Nesting is limited only by the amount of memory available when the source file is assembled.

Using a macro to create similar macros can make maintenance easier. If you want to change all the macros, you change the outer macro and it automatically changes the others.

■ Example

```
shifts      MACRO   opname              ; Define macro that defines macros
opname&s    MACRO  operand,rotates
            IF      rotates LE 3
            REPT    rotates
            opname  operand,1           ;; One at a time is faster
            ENDM                        ;;   for 3 or less on 8088/8086
            ELSE
            mov     cl,rotates          ;; Using CL is faster
            opname  operand,cl          ;;   for more than 3 on 8088/8086
            ENDIF
            ENDM
            ENDM

            shifts  ror                 ; Call macro
            shifts  rol                 ;   to new macros
            shifts  shr
            shifts  shl
            shifts  rcl
            shifts  rcr
            shifts  sal
            shifts  sar
            .
            .
            .
            shrs    ax,5                ; Call defined macros
            rols    bx,3
```

This macro, when called as shown, creates macros for multiple shifts with each of the shift and rotate instructions. All macros are identical except for the instruction. For example, the macro for the **SHR** instruction is called `shrs` while the macro for the **ROL** instruction is called `rols`. If you wanted to enhance the macros by doing more parameter checking, you could modify the original macro. It would change the created macros automatically. This macro uses the substitute macro operator, as described in Section 11.4.1.

11.5.3 Nesting Macro Calls

Macro definitions can contain calls to other macros. Nested macro calls are expanded like any other macro call, but only when the outer macro is called.

■ Example

```
ex          MACRO   text,val    ; Inner macro definition
            IF2
            %OUT    The expression (&text) has the value: &val
            ENDIF
            ENDM

express     MACRO   expression ; Outer macro definition
            ex      <expression>,%(expression)
            ENDM
            .
            .
            .
            express <((.TYPE ex) SHL 14) EQ 8000h>
```

The two sample macros enable you to print the result of a complex expression to the screen using the **%OUT** directive, even though that directive expects text rather than an expression (see Section 12.1). Being able to see the value of an expression is convenient during debugging.

Both macros are necessary. The `express` macro calls the `ex` macro, using operators to pass the expression both as text and as the value of the expression. With the call in the example, the assembler sends the following line to the standard output:

```
The expression (((.TYPE ex) SHL 14) EQ 8000h) has the value: 0
```

You could get the same output using only the `ex` macro, but you would have to type the expression twice and supply the macro operators in the correct places yourself. The `express` macro does this for you automatically. Notice that expressions containing spaces must still be enclosed in angle brackets. Section 11.4.2 explains why.

11.5.4 Redefining Macros

Macros can be redefined. You do not need to purge the macro before redefining it. The new definition automatically replaces the old definition. If you redefine a macro from within the macro itself, make sure there are no lines between the **ENDM** directive of the nested redefinition and the **ENDM** directive of the original macro. The following example may produce incorrect code:

```
dostuff      MACRO
             .
             .
             .
dostuff      MACRO
             .
             .
             .
             ENDM
             ;; Comments or statements not allowed
             ENDM
```

To correct the error, remove the line between the **ENDM** directives.

11.5.5 Avoiding Inadvertent Substitutions

MASM replaces all occurrences of a parameter with the corresponding argument, even if the argument is inappropriate. For example, if you use a register name such as **AX** or **BH** as a parameter, **MASM** replaces all occurrences of that name when it expands the macro. If the macro definition contains statements that use the register, not the parameter, the macro will be incorrectly expanded. **MASM** will not warn you about using reserved names used as macro parameters.

MASM does give a warning if you use a reserved name as a macro name. You can ignore the warning, but you should be aware that the reserved name will no longer have its original meaning. For example, if you define a macro called ADD, the **ADD** instruction will no longer be available. The ADD macro takes its place.

11.6 Managing Macros and Equates

Macros and equates are often kept in a separate file and read into the assembler source file at assembly time. In this way, libraries of related macros and equates can be used by many different source files.

The **INCLUDE** directive is used to read an include file into a source file. Memory can be saved by using the **PURGE** directive to delete the unneeded macros from memory.

11.6.1 Using Include Files

The **INCLUDE** directive inserts source code from a specified file into the source file from which the directive is given.

■ **Syntax**

INCLUDE *filespec*

The *filespec* must specify an existing file containing valid assembler statements. When the assembler encounters an **INCLUDE** directive, it opens the specified source file and begins processing its statements. When all statements have been read, **MASM** continues with the statement immediately following the **INCLUDE** directive.

The *filespec* can be given either as a file name, or as a complete file specification including drive or directory name.

If a complete file specification is given, **MASM** looks for the include file only in the specified directory. If a file name is given without a directory or drive name, **MASM** looks for the file in the following order:

1. If paths are specified with the **/I** option, **MASM** looks for the include file in the specified directory or directories. See Section 2.4.6 for more information on the **/I** option.
2. **MASM** looks for the include file in the current directory.
3. If an **INCLUDE** environment variable is defined, **MASM** looks for the include file in the directory or directories specified in the environment variable.

Nested **INCLUDE** directives are allowed. **MASM** marks included statements with the letter C in assembly listings.

Directories can be specified in **INCLUDE** path names with either the backslash (\) or the forward slash (/). This is for XENIX® compatibility.

Note

Any standard code can be placed in an include file. However, include files are usually used only for macros, equates, and standard segment definitions. Standard procedures are usually assembled into separate object files and linked with the main source modules.

■ **Examples**

```
INCLUDE fileio.inc                   ; File name only; use with
                                     ;   /I or environment
INCLUDE b:\include\keybd.inc         ; Complete file specification

INCLUDE /usr/jons/include/stdio.inc  ; Path name in XENIX format

INCLUDE masm_inc\define.inc          ; Partial path name in DOS format
                                     ;   (relative to current directory);
                                     ;   can use with /I or environment
```

11.6.2 Purging Macros from Memory

The **PURGE** directive can be used to delete a currently defined macro from memory.

■ **Syntax**

PURGE *macroname*⟦,*macroname*...⟧

Each *macroname* is deleted from memory when the directive is encountered at assembly time. Any subsequent call to that macro causes the assembler to generate an error.

The **PURGE** directive is intended to clear memory space no longer needed by a macro. If a macro has been used to redefine a reserved name, the reserved name is restored to its previous meaning.

The **PURGE** directive can be used to clear memory if a macro or group of macros is only needed for part of a source file.

It is not necessary to purge a macro before redefining it. Any redefinition of a macro automatically purges the previous definition. Also, a macro can purge itself as long as the **PURGE** directive is on the last line of the macro.

■ Examples

```
GetStuff
PURGE   GetStuff
```

This example calls a macro and then purges it. You might need to purge macros in this way if your system does not have enough memory to keep all the macros needed for a source file in memory at the same time.

Chapter 12
Controlling Assembly Output

MASM has two ways of communicating results of an assembly to the user. It can write information to a listing, cross-reference, or object file, or it can display messages to the standard output device (ordinarily the screen).

Both kinds of output can be controlled from the command line or from inside a source file. The command lines and options that affect information output are described in Chapter 2, "Using MASM." This chapter explains the directives that directly control output from inside source files.

12.1 Sending Messages to the Standard Output Device

The **%OUT** directive instructs the assembler to display text to the standard output device. This device is normally the screen, but you can redirect the output to a file or other device (see Section 2.3).

■ Syntax

%OUT *text*

The *text* can be any line of ASCII characters. If you want to display multiple lines, you must use a separate **%OUT** directive for each line.

The directive is useful for displaying messages at specific points of a long assembly. It can be used inside conditional assembly blocks to display messages when certain conditions are met.

The **%OUT** directive generates output for both assembly passes. The **IF1** and **IF2** directives can be used to control when the directive is processed. Macros that enable you to output the value of expressions are shown in Section 11.5.3.

■ Example

```
IF1
%OUT    First Pass - OK
ENDIF
```

This sample block could be placed at the end of a source file so that the message `First Pass - OK` would be displayed at the end of the first pass, but ignored on the second pass.

12.2 Controlling Page Format in Listings

MASM provides several directives for controlling the page format of listings. These directives include the following:

Directive	Meaning
TITLE	Sets title for listings
SUBTTL	Sets title for sections in listings
PAGE	Sets page length and width, and controls page and section breaks

12.2.1 Setting the Listing Title

The **TITLE** directive specifies a title to be used on each page of assembly listings.

■ **Syntax**

TITLE *text*

The *text* text can be any combination of characters up to 60 characters in length. The title is printed flush left on the second line of each page of the listing.

If no **TITLE** directive is given, the title will be blank. No more than one **TITLE** directive per module is allowed.

The first word of the title is used as the module name if the source file does not contain a **NAME** directive and if the **/ZI** and **/ZD** are not used during assembly. See Section 12.5 for more information on the **NAME** directive.

■ **Example**

```
TITLE Graphics Routines
```

This example sets the listing title. A page heading that reflects this title is shown below:

```
Microsoft (R) Macro Assembler Version 5.00              9/25/87 12:00:00
Graphics Routines                                       Page     1-2
```

12.2.2 Setting the Listing Subtitle

The **SUBTTL** directive specifies the subtitle used on each page of assembly listings.

■ **Syntax**

SUBTTL *text*

The *text* can be any combination of characters up to 60 characters. The subtitle is printed flush left on the third line of listing pages.

If no **SUBTTL** directive is used, or if no *text* is given for a **SUBTTL** directive, the subtitle line is left blank.

Any number of **SUBTTL** directives can be given in a program. Each new directive replaces the current subtitle with the new *text*. **SUBTTL** directives are often used just before a **PAGE +** statement that creates a new section (see Section 12.2.3).

■ **Example**

```
SUBTTL Point Plotting Procedure
PAGE    +
```

The example above creates a section title, then creates a page break and a new section. A page heading that reflects this title is shown below:

```
Microsoft (R) Macro Assembler Version 5.00              9/25/87 12:00:00
Graphics Routines                                       Page     3-1
Point Plotting Procedure
```

12.2.3 Controlling Page Breaks

The **PAGE** directive can be used to designate the line length and width for the program listing, to increment the section and adjust the section number accordingly, or to generate a page break in the listing.

■ Syntax

PAGE ⟦⟦*length*⟧,*width*⟧
PAGE +

If *length* and *width* are specified, the **PAGE** directive sets the maximum number of lines per page to *length*, and the maximum number of characters per line to *width*. The *length* must be in the range 10 to 255. The default page length is 50 lines. The *width* must be in the range 60 to 132. The default page width is 80 characters. To specify *width* without changing the default *length*, use a comma before *width*.

If no argument is given, **PAGE** starts a new page in the program listing by copying a form-feed character to the file and generating new title and subtitle lines.

If a plus sign follows **PAGE**, a page break occurs, the section number is incremented, and the page number is reset to 1. Program listing page numbers have the following format:

section-page

The *section* is the section number within the module and *page* is the page number within the section. By default, section and page numbers begin with 1-1. The **SUBTTL** directive and the **PAGE** directive can be used together to start a new section with a new subtitle. See Section 12.2.2 for an example.

■ Example 1

```
PAGE
```

Example 1 creates a page break.

■ Example 2

```
PAGE 58,90
```

Example 2 sets the maximum page length to 58 lines, and the maximum width to 90 characters.

■ Example 3

```
PAGE ,132
```

Example 3 sets the maximum width to 132 characters. The current page length (either the default of 50 or a previously set value) remains unchanged.

■ Example 4

```
PAGE +
```

Example 4 creates a page break, increments the current section number, and sets the page number to 1. For example, if the preceding page was 3-6, the new page would be 4-1.

12.3 Controlling the Contents of Listings

MASM provides several directives for controlling what text will be shown in listings. The directives the control the contents of listings are shown below:

Directive	Meaning
.LIST	List statements in program listing
.XLIST	Suppress listing of statements
.LFCOND	List false conditional in program listing
.SFCOND	Suppress false-conditional listing
.TFCOND	Toggle false-conditional listing

.LALL	Include macro expansions in program listing
.SALL	Suppress listing of macro expansions
.XALL	Exclude comments from macro listing

12.3.1 Suppressing and Restoring Listing Output

The **.LIST** and **.XLIST** directives specify which source lines are included in the program listing.

■ Syntax

.LIST
.XLIST

The **.XLIST** directive suppresses copying of subsequent source lines to the program listing. The **.LIST** directive restores copying. The directives are typically used in pairs, to prevent a particular section of a source file from being copied to the program listing.

The **.XLIST** directive overrides other listing directives such as **.SFCOND** or **.LALL**.

■ Example

```
.XLIST              ; Listing suspended here
.
.
.
.LIST               ; Listing resumes here
.
.
.
```

12.3.2 Controlling Listing of Conditional Blocks

The **.SFCOND**, **.LFCOND**, and **.TFCOND** directives control whether false conditional blocks should be included in assembly listings.

■ Syntax

.SFCOND
.LFCOND
.TFCOND

The **.SFCOND** directive suppresses the listing of any subsequent conditional blocks whose condition is false. The **.LFCOND** directive restores the listing of these blocks. Like **.LIST** and **.XLIST**, conditional-listing directives can be used to suppress listing of conditional blocks in sections of a program.

The **.TFCOND** directive toggles the current status of listing of conditional blocks. This directive can be used in conjunction with the **/X** option of the assembler. By default, conditional blocks are not listed on start-up. However, they will be listed on start-up if the **/X** is given. This means that using **/X** reverses the meaning of the first **.TFCOND** directive in the source file. The **/X** option is discussed in Section 2.4.14.

■ Example

```
test1       EQU     0           ; Defined to make all conditionals false

                                ; /X not used       /X used
            .TFCOND
            IFNDEF  test1       ; Listed            Not listed
test2       DB      128
            ENDIF
            .TFCOND
            IFNDEF  test1       ; Not listed        Listed
test3       DB      128
            ENDIF
            .SFCOND
            IFNDEF  test1       ; Not listed        Not listed
test4       DB      128
            ENDIF
            .LFCOND
            IFNDEF  test1       ; Listed            Listed
test5       DB      128
            ENDIF
```

In the example above, the listing status for the first two conditional blocks would be different, depending on whether the **/X** option was used. The blocks with **.SFCOND** and **.LFCOND** would not be affected by the **/X** option.

12.3.3 Controlling Listing of Macros

The **.LALL**, **.XALL**, and **.SALL** directives control the listing of the expanded macros calls. The assembler always lists the full macro definition. The directives only affect expansion of macro calls.

■ **Syntax**

.LALL
.XALL
.SALL

The **.LALL** directive causes **MASM** to list all the source statements in a macro expansion, including normal comments (preceded by a single semicolon) but not macro comments (preceded by a double semicolon).

The **.XALL** directive causes **MASM** to list only those source statements in a macro expansion that generate code or data. For example, comments, equates, and segment definitions are ignored.

The **.SALL** directive causes **MASM** to suppress listing of all macro expansions. The listing shows the macro call, but not the source lines generated by the call.

The **.XALL** directive is in effect when **MASM** first begins execution.

■ **Example**

```
tryout      MACRO   param
                                ;;Macro comment line
                                ; Normal comment line
it          EQU     3           ; No code or data
            ASSUME  es:_DATA    ; No code or data
            DW      param       ; Generates data
            mov     ax,it       ; Generates code
            ENDM
            .
            .
            .
            .XALL
            tryout  6           ; Call with .LALL

            .XALL
            tryout  6           ; Call with .XALL

            .SALL
            tryout  6           ; Call with .SALL
```

The macro calls in the example generate the following listing lines:

```
                                  .LALL
                                  tryout  6          ; Call with .LALL
                        1                            ; Normal comment line
 = 0003                 1 it      EQU     3          ; No code or data
                        1         ASSUME  es:_TEXT   ; No code or data
 0015  0006             1         DW      6          ; Generates data
 0017  B8 0003          1         mov     ax,it      ; Generates code


                                  .XALL
                                  tryout  6          ; Call with .XALL
 001A  0006             1         DW      6          ; Generates data
 001C  B8 0003          1         mov     ax,it      ; Generates code


                                  .SALL
                                  tryout  6          ; Call with .SALL
```

Notice that the macro comment line is never listed in macro expansions. Normal comment lines are listed only with the **.LALL** directive.

12.4 Controlling Cross-Reference Output

The **.CREF** and **.XCREF** directives control the generation of cross-references for the macro assembler's cross-reference file.

■ **Syntax**

.CREF
.XCREF ⟦*name*⟦**,***name*...⟧⟧

The **.XCREF** directive suppresses the generation of label, variable, and symbol cross-references. The **.CREF** directive restores generation of cross-references.

If *names* are specified with **.XCREF**, only the named labels, variables, or symbols will be suppressed. All other names will be cross-referenced. The named labels, variables, or symbols will also be omitted from the symbol table of the program listing.

■ **Example**

```
.XCREF              ; Suppress cross-referencing
.                   ;   of symbols in this block
.
.
.CREF               ; Restore cross-referencing
.                   ;   of symbols in this block
.
.
.XCREF test1,test2  ; Don't cross-reference test1 or test2
.                   ;   in this block
.
.
```

12.5 Naming Object Modules

The **NAME** directive specifies the module name that will be written to the object file. The module name is used by the linker when displaying error messages. It is also used by some debuggers.

■ **Syntax**

NAME *modulename*

The *modulename* can be any combination of letters and digits. It can be any length, but must fit on one line and must not contain spaces.

Every module has a module name whether the **NAME** directive is used or not. The name is based on the following factors:

1. If the **/ZI** or **/ZD** assembler options are used, the name of the source file becomes the module name. The CodeView debugger uses this module name to match the executable file with the corresponding source file. Any conflicting name specified with the **NAME** directive will be ignored.
2. If the **/ZI** and **/ZD** options are not used, the module name will be the name specified by the **NAME** directive.
3. If the **NAME** directive and the **/ZI** and **/ZD** options are not used, the assembler creates a default module name using the first word of any title specified with the **TITLE** directive.

4. If the **NAME** and **TITLE** directives and the **/ZI** and **/ZD** options are not used, the default name "A" is used.

By default, the name will be written to the object file with all uppercase letters regardless of how it is given in the source file. It will be case sensitive if **MASM** was started with the **/ML** option.

MASM specifies the module name by writing the default name to the **THEADDR** record of the object file.

Using Instructions

Chapter 13
Understanding 8086-Family Processors

This chapter introduces the 8086-family of processors. It describes their segmented-memory structure and their registers. Differences between the different chips in the family are also covered.

13.1 Using the 8086-Family Processors

The Intel Corporation manufactures the group of processors that is referred to in this manual as the 8086-family of processors. The MS-DOS and PC-DOS operating systems are designed to work under these processors and to take advantage of their features. The processors have several features in common:

- Memory is organized using a segmented architecture.
- The same set of registers is used in all versions.
- The instruction set is upwardly compatible—that is, all features available in the early versions of the processor are also available in the newer versions, but the new versions contain additional features that are not supported in the old versions.

13.1.1 Processor Differences

The main 8086-family processors are discussed below:

Processor	Description
8088 and 8086	These processors work in real mode. They are designed to run a single process. No provision is made to protect one part of memory from actions occurring in another part of memory. The processor can address up to 1 megabyte of memory. Addresses specified in assembly language correspond to physical memory addresses. The 8088 uses an 8-bit data bus, while the 8086 uses a 16-bit data bus. This makes the 8086 somewhat faster. However, from the programming standpoint, the two processors are identical except that the 8086 will handle certain data more efficiently if it is byte aligned using the **EVEN** or **ALIGN** directives (see Section 6.4).
80186	This processor is identical to the 8086 except that new instructions are added and some old instructions are

optimized. It runs significantly faster than the 8086. (There is also an enhanced version of the 8088 called the 80188.)

80286	This processor has the added instructions of the 80186. It can run in the real mode of the 8088 and 8086, but it also has an optional protected mode in which multiple processes may be run concurrently. Memory used by each process can be protected from memory used by other processes. In protected mode, the processor can address up to 16 megabytes of memory. However, when memory is accessed in protected mode, the addresses do not correspond to physical memory, since the processor automatically allocates and manages memory dynamically. Additional instructions for initializing protected mode and controlling multiple processes are implemented.
80386	This is a 32-bit processor. At the system level, it implements many new features, including paging code to disk, addressing up to 4 gigabytes of memory, and multiple 8086 processes. This manual does not explain how to implement these features. For the applications programmer running in DOS, the 80836 supports all the instructions of the 80286 and some additional instructions. It also allows limited use of 32-bit registers and addressing modes. Finally, the 80386 operates significantly faster than the 80286. Considerations for programming the 80386 under DOS are summarized in Section 13.4.
8087, 80287, and 80387	These are math coprocessors that work concurrently with the 8086-family processors. They do mathematical calculations faster and with more accuracy than can be done with the 8086-family processors. Although there are performance differences between the three coprocessors, the main difference to the programmer is that the 80287 and 80387 can operate in protected mode. The 80387 also implements several new instructions.

13.1.2 Real and Protected Modes

Real mode is the single-process mode used in current versions of DOS. Protected mode is the multiple-process mode that is used in Microsoft XENIX. It will also be used in future multitasking versions of DOS.

To the applications programmer, there is little difference between assembly-language programming in real and protected mode. Processes are managed at the system level by the operating system. The programmer does not deal with them except when interfacing with the operating system.

This manual does not address issues of interfacing with multitasking operating systems. If you are using a multitasking system, you must use the documentation for that operating system. However, applications programmers should be aware of the following differences between real and protected mode programming:

- Up to 1 megabyte of memory can be addressed in real mode, while up to 16 megabytes can be addressed in protected mode. This distinction may make a difference in the number and size of data structures created, but it should make no difference in the assembly-language syntax, since data is addressed in exactly the same way in either mode.
- In real mode, physical memory may be addressed directly, since segments represent physical addresses. This is sometimes done using **AT** segments (see Section 5.2.2.1). In protected mode, physical addresses cannot be addressed directly, since segments represent selectors which are allocated and managed by the operating system. Therefore, programmers working in protected mode should not attempt to manipulate memory directly. Future multitasking versions of DOS will provide functions for doing the kinds of tasks that are often done by manipulating physical memory under current versions of DOS.
- Future multitasking versions of DOS will use the Applications Program Interface (API) to access DOS functions. This system is different from the current DOS system of using interrupt 21h. The API is the same interface used under Microsoft Windows.

13.2 Segmented Addresses

When used with current versions of DOS, 8086-family processors can store addresses as 16-bit word values. Therefore, the maximum unsigned value that can be stored as an address is 65,635 (0FFFFh). Yet the processors are actually capable of accessing much larger addresses. The highest possible address is 1 megabyte (0FFFF0h) in real mode or 16 megabytes (0FFFFF0h) in protected mode.

Addresses larger than 65,535 bytes are specified by combining two segmented word addresses: a 16-bit segment and a 16-bit offset within the segment. A common syntax for showing segmented addresses is in the *segment:offset* format. For example, an address with a segment of 01111h and an offset of 01111h would be represented as `1111:1111`. This method of specifying addresses can be used directly in many debuggers, including the CodeView debugger, **SYMDEB**, and **DEBUG**, but not in assembler source code.

In the real-mode, the address `1111:1111` represents a physical 20-bit address. This address can be calculated by multiplying the *segment* portion of the address by 16 (10h), then adding the *offset* portion, as shown below:

```
   11110h        Segment times 10h
+   1111h        Offset
--------
   12221h        Physical address
```

In protected mode, the address `1111:1111` represents a movable address. The segment portion of the address is a selector that is assigned a physical address by the operating system. The applications programmer has no control (and needs none) over the physical address represented by the selector.

80386 Processor Only

The 80386 processor supports 48-bit addresses consisting of a 16-bit segment selector and a 32-bit offset. This enables it to access addresses of up to 4 gigabytes in protected mode. The processor can also run in modes that are compatible with the 16- bit real- and protected-mode addressing schemes of the other 8086-family processors.

Addresses cannot be represented directly in the *segment:offset* format in assembly language. Instead the *segment* portion of the address is specified symbolically, using a name assigned to the segment in the source code. The address represented by the symbol can then be assigned to one of the segment registers. Chapter 5, "Defining Segment Structure." describes the directives that assign symbols to segment addresses.

The *offset* portion of addresses can be specified in a number of ways, depending on the context. Directives that assign symbols to offsets are discussed in Chapter 4, "Writing Source Code."

In assembly-language programming, addresses can be near or far. A near address is simply the offset portion of the address. Any instruction that accesses a near address will assume that the segment address is the same as the current segment for the type of address being accessed (usually a code segment for code or a data segment for data).

A far address consists of both the segment and offset portions of the address. Far addresses can be accessed from any segment. Both the segment and offset must be provided for instructions that access far addresses. Far addresses are more flexible because they can be used for larger programs and larger data objects. However, near addresses are more efficient, since they produce smaller code and can be accessed more quickly.

13.3 Using 8086-Family Registers

Like most microprocessors, the 8086-family processors have special areas of memory called registers. Some registers control the behavior or status of the processor. Others are used as temporary storage places where data can be accessed and processed faster than if they were stored in regular memory.

All the 8086-family processors share the same set of 16-bit registers. Some registers can be accessed as two separate 8-bit registers. In the 80386, most registers can also be accessed as extended 32-bit registers.

Figure 13.1 shows the registers common to all the 8086-family processors. Each register and group of registers has its own special uses and limitations, as described in the next sections.

```
          General-Purpose Registers

Name                                 Special Functions
             15       7        0
             +--------+--------+
             |        |        |     Multiply, divide,
Accumulator  |   AH   |   AL   |     I/O, and optimized
             |        AX       |     moves
             +--------+--------+
             |        |        |     Multiply, divide,
Data         |   DH   |   DL   |     and I/O
             |        DX       |
             +--------+--------+
             |        |        |     Count for loops,
Count        |   CH   |   CL   |     repeats, shifts,
             |        CX       |     and rotates
             +--------+--------+
             |        |        |     Pointer to base
Base         |   BH   |   BL   |     address of data
             |        BX       |     segment
             +--------+--------+
Base         |                 |     Pointer to base
Pointer      |                 |     address of stack
             |        BP       |     segment
             +--------+--------+
Source       |                 |     String source and
Index        |                 |     index pointer
             |        DI       |
             +--------+--------+
Destination  |                 |     String destination
Index        |                 |     index pointer
             |        DI       |
             +--------+--------+
Stack        |                 |     Pointer to
Pointer      |                 |     top of stack
             |        SP       |
             +--------+--------+

          Segment Registers                      Other Registers

          +--------+--------+                 +--------+--------+
Code      |                 |                 |                 |
Segment   |                 |    Flags        |                 |
          |        CS       |                 |      Flags      |
          +--------+--------+                 +--------+--------+
Data      |                 |    Instruction  |                 |
Segment   |                 |    Pointer      |                 |
          |        DS       |                 |        IP       |
          +--------+--------+                 +--------+--------+
Stack     |                 |
Segment   |                 |
          |        SS       |
          +--------+--------+
Extra     |                 |
Segment   |                 |
          |        ES       |
          +--------+--------+
```

■ 80386 Only

The 80386 processor uses the same registers as the other processors in the 8086 family, but all except the segment registers can be extended to 32 bits. The 80386 also has two additional segment registers, **FS** and **GS**. Figure 13.2 shows the extended registers of the 80386.

```
                       General Purpose Registers            80386 Extensions
Name
             31      23      15       7       0
             +-------+-------+-------+-------+
             |              EAX      |       |
Accumulator  |               |  AH   |  AL   |
             |               |       AX      |
             +-------+-------+-------+-------+
Data         |              EDX      |       |
Register     |               |  DH   |  DL   |
             |               |       DX      |
             +-------+-------+-------+-------+
Count        |              ECX      |       |
Register     |               |  CH   |  CL   |
             |               |       CX      |
             +-------+-------+-------+-------+
Base         |              EBX      |       |
Register     |               |  BH   |  BL   |
             |               |       BX      |
             +-------+-------+-------+-------+
Base         |              EBP              |
Pointer      |               |               |
             |               |       BP      |
             +-------+-------+-------+-------+
Source       |              ESI              |
Index        |               |               |
             |               |       DI      |
             +-------+-------+-------+-------+
Destination  |              EDI              |
Index        |               |               |
             |               |       DI      |
             +-------+-------+-------+-------+
Stack        |              ESP              |
Pointer      |               |               |
             |               |       SP      |
             +-------+-------+-------+-------+

        Segment Registers                        Other Registers

        +-------+-------+                 +-------+-------+-------+-------+
Code    |               |                 |             Eflags            |
Segment |               |   Flags         |               |               |
        |      CS       |                 |               |     Flags     |
        +-------+-------+                 +-------+-------+-------+-------+
Data    |               |   Instruction   |              EIP              |
Segment |               |   Pointer       |               |               |
        |      DS       |                 |               |      IP       |
        +-------+-------+                 +-------+-------+-------+-------+
Stack   |               |
Segment |               |
        |      SS       |
        +-------+-------+
Extra   |               |
```

```
Segment |                 |
        |       ES        |
        +--------+--------+
Extra   |                 |
Segment |                 |
        |       FS        |
        +--------+--------+
Extra   |                 |
Segment |                 |
        |       GS        |
        +--------+--------+
```

13.3.1 Segment Registers

At run time, all addresses are relative to one of four segment registers: **CS**, **DS**, **SS**, or **ES**. These registers and the segments they correspond to are listed below:

Segment	Purpose
Code Segment (**CS**)	Addresses in the segment pointed to by this register contain the encoded instructions and operands specified by the program.
Data Segment (**DS**)	Addresses in the segment pointed to by this register normally contain data allocated by the program.
Stack Segment (SS)	Addresses in the segment pointed to by this register are available for instructions that store data on the program stack. A stack is an area reserved for storing temporary data on a first-in-first-out (FIFO) basis.
Extra Segment (**ES**)	Addresses in the segment pointed to by this register are available for string instructions. An additional segment can also be stored in the **ES** register. The 80386 has two additional extra segments: **FS** and **GS**.

13.3.2 General-Purpose Registers

The **AX**, **BX**, **CX**, **DX**, **BP**, **SI**, **DI**, and **SP** registers are 16-bit, general-purpose registers. They can be used to temporarily store data during processing. Data in registers can be accessed much more quickly than data in memory. Therefore, it is more efficient to keep the most frequently used values in registers.

Memory to memory operations are never allowed in 8086-family processors. As a result, data must often be moved into registers before doing calculations or other operations involving more than one variable.

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

In addition to their general use for storing data, each of the general-purpose registers has special uses in certain situations. Specific uses for each register are listed below:

Register	Description
AX	The **AX** (accumulator) register is most often used for storing temporary data. Many instructions are optimized so that they work slightly faster on data in the accumulator register than on data in other registers. With division instructions, the accumulator holds all or part of the dividend before the operation and the quotient afterward. With multiplication instructions, the accumulator holds one of the factors before the operation and all or part of the result afterward. In I/O operations to and from ports, the accumulator holds the data being transferred.
DX	The **DX** (data) register is most often used for storing temporary data. With division instructions operating on word values, **DX** holds the upper word of the dividend before the operation and the remainder afterward. With multiplication instructions operating on word values, **DX** holds the upper word of one of the factors before the operation and the upper word of the result afterward. In I/O operations to and from ports, **DX** holds the number of the port to be accessed.
CX	The **CX** register is used to hold the count for instructions that do looping or other repeated operations. These include the loop instructions, certain jump instructions, repeated string instructions, and shifts and rotates.

BX The **BX** register can be used to point to the base of a data object (see Section 14.3.2).

BP The **BP** register can be used for general data storage. It is more often used for pointing to the base of a stack frame. The Microsoft conventions for passing arguments to procedures have a specific use for **BP** as described in Section 17.4.3. The **SS** register is assumed as the segment register in operations using **BP**.

SI The **SI** register can be used for pointing to (indexing) an item within a data object. With some string instructions, **SI** is used to point to bytes or words within a source string.

DI The **DI** register can be used for pointing to (indexing) an item within a data object. With some string instructions, **DI** is used to point to bytes or words within a destination string.

SP The **SP** register points to the current location within the stack segment. Pushing a value onto the stack increases the value of **SP** by two, while popping from the stack decreases the value of **SP** by two. Call instructions store the calling address on the stack and decrease SP accordingly, while return instructions get the stored address and increase **SP**. (With 80386 32-bit segments, **SP** is increased or decreased by four instead of two.) Sections 15.4.2 and 17.4.3 discuss operation of the stack in more detail.

13.3.3 The Instruction Pointer

The instruction-pointer register always contains the address of the instruction about to be executed. The programmer cannot directly access or change the instruction pointer. However, process-control instructions, such as calls, jumps, loops, and interrupts, automatically change the instruction pointer.

13.3.4 The Flags Register

The flags register is a 16-bit register containing status bits that control various instructions and reflect the current status of the processor. In the 80386, 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, and 13 flags for the 80386.

Figure 13.3 shows the bits of the 32-bit flags register for the 808386. Only the lower word is used for the other 8086-family processors. The unmarked bits are reserved for processor use and should never be modified by the programmer.

```
                              + Virtual 8086 Mode
                              | + Resume
                              | |   + Nested Task
                              | |   |  + I/O Protection Level
                              | |   |  |  + Overflow
                              | |   |  |  | + Direction
                              | |   |  |  | | + Interrupt Enable
                              | |   |  |  | | | + Trap
                              | |   |  |  | | | | + Sign
                              | |   |  |  | | | | | + Zero
                              | |   |  |  | | | | | |   + Auxiliary Carry
                              | |   |  |  | | | | | |   |   + Parity
                              | |   |  |  | | | | | |   |   |   + Carry
 F E D C B A 9 8 7 6 5 4 3 2 1 0 F E D C B A 9 8 7 6 5 4 3 2 1 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| | | | | | | | | | | | | | |V|R| |N|IOP|O|D|I|T|S|Z| |A| |P| |C|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
                    ^         |   ^   |           ^
                  80386       | 80286 |          All
                  only        | 80386 |       Processors
```

The nine flags common to all 8086-family processors are summarized below, starting with the low-order flags. In these decriptions, the term "set" means the bit value is 1, while "clear" means the bit value is 0.

Flag	Description
Carry	Set if an addition operation uses a carry or if a subtraction operation uses a borrow.
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 addition operation uses a carry on the low-order four bits of **AL** or if a subtraction operation uses a borrow on the low-order four bits of **AL**. This flag is used for binary-coded decimal arithmetic.
Zero	Set if the result of an operation is zero.
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 debugger program can use this feature to execute a program one instruction at a time.
Interrupt Enable	If set, interrupts will be recognized and acted on as they are received. The bit can be cleared to temporarily turn off interrupt processing.
Direction	Can be set to make string operations process down from high addresses to low addresses, or can be cleared to make 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.
I/O Protection Level	This 2-bit flag indicates the protection level for input and output. Managing the protection level is a systems task that is not described in this manual.
Nested Task	Controls chaining of interrupted and called tasks. Controlling tasks in protected mode is a systems task that is not described in this manual.
Resume	If set, debug exceptions are temporarily disabled. Using 80386 debug exceptions is a systems task that is not described in this manual.
Virtual 8086 Mode	If set, the processor is running an 8086-family real mode program in a protected multitasking environment. If clear, the 80386 processor is in its normal mode. Running in virtual 8086 mode is a systems task that is not described in this manual.

13.3.5 8087-Family Registers

The 8087-family processors use a stack-based architecture to access up to eight 80-bit registers. See Chapter 19, "Calculating with a Math Coprocessor," for information on using 8087-family registers and instructions. The format of real numbers used by coprocessors is explained in Section 6.2.1.5.

13.4 Using the 80386 Processor Under DOS

Many of the added functions of the 80386 are not supported by versions of DOS available at release time for Version 4.5 of the Microsoft Macro Assembler. Although DOS runs under 80386 machines, it does not operate any different (except faster) than on an 80286 machine.

For example, DOS does not support protected mode or 8086 virtual mode. Since 32-bit segments are only available in protected mode, they cannot be used under DOS. Systems programmers may be able to overcome some of these limitations, but the techniques for doing so are beyond the scope of this manual.

Applications programmers can also use some 80386 enhancements. The following features of the 80386 can be used under current versions of DOS. Note that using any of these features means your code will not run on machines that do not have an 80386 processor.

- You can use the new 80386 instructions (except for those that manage protected mode). New instructions include: bit scan (**BSF** and **BFR**); bit test (**BT**, **BTC**, **BTR**, and **BTS**); move with sign and zero extend (**MOVSX** and **MOVZX**); set byte on condition (**SET***condition*); and double precision shift (**SHLD** and **SHRD**).
- You can use 80286 instructions that have been enhanced to work with 32-bit registers. These include: conversion instructions (**CWDE** and **CDQ**), string instructions (**CMPSD**, **LODSD**, **MOVSD**, **SCASD**, **STOSD**, **INSD**, **OUTSD**); 32-bit stack enhancements (**POPFD**, **PUS AD**, **PUSHFD**, and **IRETD**); and translate (**XLATB**).
- You can use 32-bit registers for calculations. For example, you can do addition and subtraction on doubleword integers without bit splicing, and you can do multiplication and division on 64-bit integers.
- You can use 32-bit registers to point into 16-bit segments. In previous processors, only **BX**, **BP**, **DI**, and **SI** could be used as pointers in indirect memory operands. The 80386 has the same limitations on 16-bit registers, but allows all general-purpose 32-bit registers in indirect memory operands. If you use this technique, you must make sure that 32-bit registers used as pointers actually contain 16-bit values.

Although significant, these new features fall short of using the full power that will be available under multiprocessing 80386 operating systems.

Chapter 14
Using Addressing Modes

Instruction operands can be given in different forms called addressing modes. Addressing modes tell the processor how to calculate the actual value of the operand at run time.

The three types of addressing modes are immediate, register, and memory. Memory operands are turn broken into two groups: direct and indirect. Finally, there are five kinds of indirect memory operands: register indirect, based, indexed, based indexed, and based indexed with displacement.

The value of operands is calculated at assembly time for immediate operands, at load time for direct memory operands, and at run time for register operands and indirect memory operands.

Although two statements may be similar and the instruction mnemonic the same, **MASM** actually assembles different code for an instruction when it is used with different addressing modes. For example, the statements

```
mov     ax,1
```

and

```
mov     ax,place[bx][di]
```

use the same instruction, but have different encoding, timing, and size. See the *Microsoft Macro Assembler Reference* for more information on the encoding, timing, and size of different instructions. Many instructions take two operands. For these instructions, the left operand is the destination operand. It specifies the data that will be operated on and possibly changed by the instruction. The right operand is the source operand. It specifies data that will be used, but not changed, in the operation.

14.1 Using Immediate Operands

Immediate operands consist of constant numeric data that are known or calculated at assembly time. Immediate values are coded into the executable program and processed the same way each time the program is run.

Some instructions have limits on the size of immediate values (usually 8-, 16-, or 32-bit). String constants longer than two characters cannot be immediate data. They must be stored in memory before they can be processed by instructions.

Many instructions permit immediate data in one operand and either memory or register data in another. The instruction combines or replaces the register or memory data with the immediate data in some way defined by the instruction. Examples of this type of instruction include **MOV**, **ADD**, **CMP**, and **XOR**.

A few instructions, such as **RET** and **INT**, take a single immediate operand.

Immediate data is never permitted in the destination operand. If the source operand is immediate, the destination operand must be either register or direct memory so that there will be a place to store the result of the operation.

■ Examples

```
            .DATA
five        DB      5           ; Memory data
nine        EQU     9           ; Constant data

            .CODE
; Source operand is immediate
            mov     bx,nine+3
            add     five,3
            or      bx,00100100b
            in      al,43h
            cmp     cx,200

; Only operand is immediate
            ret     6
            int     21h
```

14.2 Using Register Operands

Register operands consist of data stored in registers. Register-direct mode refers to using the actual value inside the register at the time the instruction is used. Registers can also be used indirectly to point to memory locations, as described in Section 14.3.2.

Most instructions allow register values in one or more operands. Some instructions can only be used with certain registers. Often instructions have shorter encoding (and faster operation) if the accumulator register (**AX** or **AL**) is specified. Use of segment registers in operands is limited to a few instructions and special circumstances.

The registers shown in Table 14.1 can be used in register direct mode.

Table 14.1
Register Operands

Register Operand Type	Register Name			
8-bit high registers	AH	BH	CH	DH
8-bit low registers	AL	BL	CL	DL
16-bit general purpose	AX	BX	CX	DX
32-bit general purpose[1]	EAX	EBX	ECX	EDX
16-bit pointer and index	SP	BP	SI	DI
32-bit pointer and index[1]	ESP	EBP	ESI	EDI
16-bit segment	CS	DS	SS	ES
Additional 80386 segment[1]	FS	GS		

[1] Available only if the 80386 processor is enabled

Registers are discussed in more detail in Section 13.3. Limitations on register use for specific instructions are discussed in sections on the specific instructions throughout Part 3, "Using Instructions to Control Run-Time Processing."

■ Examples

```
; Source and destination operands are register direct
            add     ax,bx
            mov     ds,ax
            xor     eax,ebx             ; 80386 only
            cmp     ah,bh

; Source operand is register direct
            and     stuff,dx
            sub     array[bx][si],ax

; Destination operand is register direct
            shl     ax,1
            cmp     cx,counter

; Only operand is register direct
            mul     bx
            pop     cx
            inc     ah
```

14.3 Using Memory Operands

Many instructions can work on data in memory. When a memory operand is given, processor must calculate the address of the data to be processed. This address is called the effective address.

The effective address is calculated differently depending on how the operand is specified. The relative addresses for direct memory operands are calculated at assembly time, then the actual addresses are calculated when the program is loaded. The addresses for indirect memory operands are calculated at run time.

The effective address is always relative to a segment register. By default, this register is **DS** for most addressing modes (it is **SS** when **BP** is used as the base register in indirect modes).

The default segment for the effective address can be overridden by specifying a segment using the segment-override operator (**:**). The segment can be specified as a segment register, a segment name, or a group name.

Note

Two memory operands can never be used together for the same instruction. If an instruction requires two operands and one is a memory operand, the other must be an immediate operand or a register direct operand. For example, the following statements are illegal because they attempt to do memory-to-memory operations:

```
mov     test1,test2         ; Illegal if both are variables
add     [bp],[bx]           ; Illegal
or      stuff[bx],stuff[si] ; Illegal
```

If you wish to do an operation involving two memory values, you must move one value into a register, do the operation, and then return the value to memory if it has been changed.

14.3.1 Using Direct Memory Operands

A direct memory operand is a symbol that represents the address (segment and offset) of an instruction or data. The offset address represented by a direct memory operand is calculated at assembly time. The address of each operand relative to the start of the program is calculated at link time. The actual (or effective) address is calculated at run time.

Direct memory operands can be any constant or symbol representing an address. This includes labels, procedure names, variables, structure variables, record variables, or the value of the location counter.

Direct memory operands are often specified as constant expressions using the index operator. For example, the operand `table[4]` refers to the byte having an offset 4 bytes from the address of `table`. This expression is equivalent to `table+4`.

Note

If the label is omitted from a direct memory operand used with a constant index, a segment must be specified. The offset of the operand is assumed to be the start of the specified segment plus the indexed offset. For example,

```
mov     ax,ds:[100h]
```

moves the value at address 100h in the data segment into the **AX** register. It is equivalent to

```
mov     ax,ds:100h
```

If the segment override is omitted, the constant value is used rather than the value it points to. For example

```
mov     ax, [100h]
```

moves the value 100h into the **AX** register. It is equivalent to the statement

```
mov     ax,100h
```

■ Examples

```
            .DATA
stuff       DW      here
            .CODE
            .
            .
            .
            mov     ax,stuff         ; Load value at address "stuff"
                                     ;   (address of "here") into AX
            mov     bx,OFFSET stuff  ; Load address of "stuff"
                                     ;   into BX
            jmp     stuff            ; Jump to value of "stuff"
                                     ;   (which is address of "here")
            jmp     here             ; Jump to the address of "here"

            jmp     ax               ; Jump to AX (value of "stuff")

            jmp     [bx]             ; Jump to [BX] (value at address
            .                        ;   of "stuff")
            .
here:       .
```

This example illustrates the difference between memory operands that represent addresses and memory operands that represent the value at an address. Labels and variable names in the data segment (such as `stuff`) represent the value at an address. Code labels (such as `here`) represent the address itself. The four jump statements at the end of the example use different kinds of operands to transfer control to the same address.

14.3.2 Using Indirect Memory Operands

Indirect memory operands enable you to use registers to point to values in memory. Since values in the registers can change at run time, you can use indirect memory operands to operate on data dynamically.

On all processors except the 80386, there are only four registers that can be used in indirect mode (see Section 14.3.3 for information on 80386 enhancements). The **BX** and **BP** registers can be used as base registers, and the **DI** and **SI** registers can be used as index registers. An attempt to use any other register in a statement that accesses memory indirectly will result in an error.

These four registers can be used separately or in pairs, with or without specifying a displacement. The five modes in which registers can be used are shown in Table 14.2.

Table 14.2

Indirect Addressing Modes

Mode	Syntax	Description
Register indirect	[BX] [BP] [DI] [SI]	Effective address is contents of register
Based	*displacement*[BX] *displacement*[BP]	Effective address is address of *displacement* plus contents of base register
Indexed	*displacement*[DI] *displacement*[SI]	Effective address is address of *displacement* plus contents of index register
Based indexed	[BX][DI] [BP][DI] [BX][SI] [BP][SI]	Effective address is contents of base register plus contents of index register
Based indexed with displacement	*displacement*[BX][DI] *displacement*[BP][DI] *displacement*[BX][SI] *displacement*[BP][SI]	Effective address is address of *displacement* plus contents of base register plus contents of index register

The *displacement* can be any expression that evaluates to a direct memory operand. For example, it can be a variable, a label, or a variable plus a constant.

Register indirect mode is used to point at a memory address within a segment. Based and indexed modes are used to point at a memory address relative to a table or one-dimensional array. Based indexed modes are useful for pointing to memory locations in complex data structures such as multidimensional arrays.

The choice of which registers to use depends on the context of the statement. String instructions require that specific registers be used in specific situations, as explained in Chapter 18, "Manipulating Strings." With other instructions, base and index registers can often be used interchangeably, depending on which registers are available. Even when based and indexed modes have the same effect, the processor will code them differently and they may take different lengths of time to execute, as shown in the *Microsoft Macro Assembler Reference.*

When calculating the effective address of an indirect operand, the processor uses a default segment register. The default segment register is **DS** if the base register is **BX** or if an index register is used without a base register. The default segment register is **SS** if **BP** is used as a base register. You can use the segment-override operator (**:**) to specify a different segment, group, or segment register.

A common syntax for indirect memory operands is to put each register within index operators (brackets). The register or registers must always be within brackets, but a variety of alternate syntaxes are possible. Any operator that indicates addition can be used to combine the displacement and multiple registers. For example, the following statements are equivalent:

```
mov     ax,table[bx][di]
mov     ax,table[bx+di]
mov     ax,[table+bx+di]
mov     ax,[bx][di]+table
```

When using based indexed modes, one of the registers must be a base register and the other an index register. The following statements are illegal:

```
mov     ax,table[bx][bp]    ; Illegal - two base registers
mov     ax,table[di][si]    ; Illegal - two index registers
```

■ Example 1

```
mov     dx,[bp]             ; Load the contents of SS:BP
                            ;   into DX
add     dx,[bx]             ; Add the contents of DS:BX
                            ;   to the contents of DX
sub     dx,12[bx]           ; Subtract the contents of DS:BX+12
                            ;   from the contents of DX
xor     red[bx],dx          ; XOR the contents of DX with
                            ;   the contents of DS:red+BX
and     dx,red[si]+3        ; AND the contents of DS:red+SI+3
                            ;   with the contents of DX
dec     [bx][si]            ; Decrement the contents
```

```
                                  ;   of DS:BX+SI
        cmp     cx,here[bp][si] ; Compare the contents of SS:here+BP+SI
                                  ;   to the contents of CX
        push    place[bx][di]+2 ; Save the contents of DS:place+BX+DI+2
                                  ;   on the stack
```

The statements in Example 1 illustrate how the various instructions can be used with indirect memory operands.

■ Example 2

```
          mov     ax,scrnbuff           ; Load address of screen buffer
          mov     es,ax                 ;   (value is hardware dependent)
                                        ; ES must point to buffer

          mov     ax,4                  ; Load column 4 as first argument
          push    ax
          mov     ax,6                  ; Load row 6 as second argument
          push    ax
          mov     ax,BYTE PTR "z"       ; Load "z" as third argument
          push    ax
          call    show                  ; Call the procedure
          add     sp,6                  ; Restore stack
          .
          .
          .
show      PROC    NEAR
          push    bp                    ; Save and set up
          mov     bp,sp                 ;   stack frame
          push    si                    ; Save SI

          mov     si,[bp+8]             ; Load column
          dec     si                    ; Adjust for zero
          shl     si,1                  ; Column times 2 bytes per character
          mov     bx,[bp+6]             ; Load row
          dec     bx                    ; Adjust for zero
          mov     ax,160                ; Multiply 160 bytes per line
          mul     bx                    ;   times current row
          mov     bx,ax                 ; Put result in index

          mov     dl,BYTE PTR [bp+4]    ; Load character
          mov     es:[bx][si],dl        ; Put character in buffer

          pop     si                    ; Restore SI
          mov     sp,bp                 ; Restore stack frame
          pop     bp
          ret     6                     ; Return and adjust stack for
show      ENDP                          ;   three 2-byte arguments
```

Example 2 shows two examples of indirect memory operands. Parameters are pushed onto the stack before calling a procedure. When the procedure is called, the parameters are removed using indirect memory operands.

The procedure writes a character to a screen buffer (a common technique with many computers and display adapters). The **BX** register points to the column position in the buffer, while the **SI** register points to the row position. In this example, the **ES** register must contain the address of the screen buffer (this address varies for different hardware).

Example 2 will work on any processor. Section 14.3.3 shows an enhanced version that uses 80386 instructions and addressing modes.

14.3.3 Using 80386 Indirect Memory Operands

Instructions for the 80386 can be given in two modes: 16 bit and 32 bit. Understanding these modes is crucial, since indirect-memory operands are completely different in each.

The 80386 instruction modes are controlled by the use type of the code segment in which the instructions are located. The mode is 16 bit if the use type is **USE16** or 32 bit if the use type is **USE32**. In 32-bit mode, segments are 32 bits wide, meaning that an offset address can be up to 4 gigabytes. In 16-bit mode, an offset address can be up to 64K. The 16-bit mode of the 80386 is the same as the mode used by all the other 8086-family processors.

If the 80386 processor is enabled (with the **.386** directive), 32-bit general purpose registers are always available. They can be used from 16-bit or 32-bit segments. When 32-bit registers are used, many of the limitations of 16-bit indirect-memory modes do not apply. The following extensions are available when 32-bit registers are used in indirect-memory operands:

- There are fewer limitations on the registers that can be used as base and index registers. With other 8086-family processors, only the **BX** and **BP** registers can be used as base registers and only the **DI** and **SI** registers can be used as index registers. With the 80386 any general-purpose 32-bit register can be used as the base register and any general purpose 32-bit register except **ESP** can be used as the index register. The same register can even be used as both the base and the index.

  If the **ESP** or **EBP** register is used as the base register, then the **SS** register is assumed as the segment register. The **DS** register is assumed with all other general-purpose registers.

- The index register can have a scaling factor of 1, 2, 4, or 8. The scaling factor is specified using the multiplication operator (*) adjacent to the index register. Scaling factors are shown in the following examples:

```
mov     eax,darray[edx*4]
mov     eax,[esi*8][edi]
mov     eax,wtable[ecx+2][edx*2]
```

Scaling can be used to index into arrays with different sizes of elements. For example, a scaling factor of 1 is used for byte arrays, 2 for word arrays, 4 for doubleword arrays, and 8 for quadword arrays. There is no performance penalty for using a scaling factor.

Since most 32-bit registers can be used as either the base or the index, it is not always clear which is the base and which is the index. However, if a scaling factor is used, it can only apply to one register and that register is defined to be the index register.

Statements can mix 16- and 32-bit registers. However, it is important to understand the implications of these statements. For example, the following statement is legal from either 16- or 32-bit segments:

```
mov     eax, [bx]
```

This moves the 32-bit value pointed to by **BX** into the **EAX** register. Although **BX** is a 16-bit pointer, it may still point into a 32-bit register. However, the following statement is never legal.

```
mov     eax, [cx]
```

The **CX** register may not be used as a 16-bit pointer (although **ECX** may be used as a 32-bit pointer).

The following statement is also legal in either mode:

```
mov     bx, [eax]
```

This moves the 16-bit value pointed to by **EAX** into the **BX** register. This works fine in 32-bit mode, but in 16-bit mode, a 32-bit pointer into a 16-bit segment may cause problems. If **EAX** contains a 16-bit value (the top half of the 32-bit register is zero), then the statement works. However, if the top half of the **EAX** register is not zero, the processor will probably generate an error.

Warning

It is possible to use both 16-bit and 32-bit modes in the same program by defining separate code segment for the two modes. However, this is a complex technique that involves special calculations to account for the differences between the two modes. Combining modes is generally done in systems programming, and is beyond the scope of this manual.

■ Example

```
                .CODE                       ; .CODE preceeds .386
                                            ;   to make 16-bit segments
                .386

                mov     ax,scrnbuff         ; Load address of screen buffer
                mov     es,ax               ;   (value is hardware dependent)
                                            ; ES must point to buffer

                push    4                   ; 4 is first argument
                push    6                   ; 6 is second argument
                push    BYTE PTR "z"        ; "z" is third argument
                call    show                ; Call the procedure
                .
                .
                .
show            PROC    NEAR
                push    bp                  ; Save and set up
                mov     bp,sp               ;   stack frame
                push    esi                 ; Save ESI

                movzx   ebx,[bp+8]          ; Load column
                dec     ebx                 ; Adjust for zero
                movzx   eax,[bp+6]          ; Load row
                dec     eax                 ; Adjust for zero
                imul    eax,160             ; Multiply 160 bytes per line

                mov     dl,BYTE PTR [bp+4]  ; Load character
                mov     es:[eax][ebx*2],dl  ; Put character in buffer

                pop     esi                 ; Restore ESI
                mov     sp,bp               ; Restore stack frame
                pop     bp
                ret     6                   ; Return and adjust stack
show            ENDP                        ;   for three 2-byte arguments
```

This example is the same as the one in Section 14.3.2 except that it uses enhanced 80386 instructions and addressing modes to make the code shorter and more efficient. Note that **EBX** is used with an index to save one instruction. Using **EAX** rather than **BX** as a base register allows other improvements.

The example also uses 80186 enhancements that allow immediate operands to be used with the **PUSH** and **IMUL** instructions. These enhancements are described in Sections 15.4.1 and 16.3 respectively.

Chapter 15
Loading, Storing, and Moving Data

The 8086-family instruction sets provide several instructions for loading, storing, or moving various kinds of data. Among the types of data that can be transferred are variables, pointers, and flags. Data can be moved to and from registers, memory, the stack, and ports. This chapter explains the instructions that move data from one location to another.

15.1 Transferring Data

Moving data is one of the most common tasks in assembly-language programming. Data can be moved between registers or between memory and registers. Immediate data can be loaded into registers or into memory.

15.1.1 Copying Data

The **MOV** instruction is the most common method of moving data. This instruction can be thought of as a "copy" instruction, since it always copies the source operand to the destination operand. Immediately after a **MOV** instruction, the source and destination both contain the same value. The old value in the destination operand is destroyed.

■ **Syntax**

MOV *register,immediate*
MOV *register,register*
MOV *register,memory*
MOV *memory,register*
MOV *segmentregister,memory*
MOV *memory,segmentregister*

The instruction has several variations:

Type | **Description**

Immediate to general register

Moves a constant value into a general register. For example:

```
mov     ax,7        ; Immediate to register
```

Immediate to memory

Moves a constant value into memory. For example:

```
mov     mem,7       ; Immediate to memory direct
mov     mem[bx],7   ; Immediate to memory indirect
mov     mem,ds      ; Segment register to memory
```

Memory to register — Moves a value from memory into a register. For example:

```
mov     ax,mem      ; Memory direct to register
mov     ax,mem[bx]  ; Memory indirect to register
mov     ds,mem      ; Memory to segment register
```

Register to memory — Moves a value from a register into memory. For example:

```
mov     mem,ax      ; Register to memory direct
mov     mem[bx],ax  ; Register to memory indirect
```

Register to register — Moves a value from one register to another register of the same size. For example:

```
mov     ax,bx       ; Register to register
mov     ds,ax       ; General to segment register
mov     ax,ds       ; Segment to general register
```

This takes care of most types of data transfer. However, two common transfer operations are not allowed and must be done in two steps.

Type — **Description**

Immediate or memory to segment register — This must be done by moving the data into a general register and then moving the general register into the segment register. For example:

```
mov     ax,DGROUP   ; Load immediate to general
mov     ds,ax       ; Move to segment register
```

Memory to memory — This must be done by moving the source-memory data into a register, then moving the register value into the destination memory. For example:

```
mov     ax,mem1     ; Load source into register
mov     mem2,ax     ; Move register to destination
```

15.1.2 Exchanging Data

The **XCHG** instruction exchanges the data in the source and destination operands. Data can be exchanged between registers or between registers and memory.

■ **Syntax**

XCHG *register,register*
XCHG *memory,register*
XCHG *register,memory*

■ **Examples**

```
xchg    ax,bx       ; Put AX in DX and DX in AX
xchg    memory,ax   ; Put "memory" in AX and AX in "memory"
```

15.1.3 Looking Up Data

The **XLAT** instruction is used to load data from a table in memory. The instruction is useful for translating bytes from one coding system to another.

The address of the table is specified as an operand to the instruction (only one operand is used). The **BX** register must contain the address of the start of the table. By default the **DS** register contains the segment of the table, but a segment override can be used to specify a different segment register. The ability to specify a segment override is the only reason for requiring an operand.

Before the **XLAT** instruction is called, the **AL** register should contain a value that points into the table (the start of the table is considered 0). After the instruction is called, **AL** will contain the table value pointed to. For example, if **AL** contains 7, the 8th byte of the table will be placed in **AL**.

■ Example

```
                .DATA
                ; Table of ALT-key extended codes
extended        DB      "QWERTYUIOP    ASDFGHJKL     ZXCVBNM"
extmsg          DB      "You pressed ALT-$"

                .CODE

                mov     ah,8                ; Get a key
                int     21h
                and     al,al               ; Is it null?
                jne     ascii               ; No? ASCII key pressed
                int     21h                 ; Yes? Extended key, so
                                            ;   get second key
                sub     al,16               ; Adjust relative to 0
                mov     bx,OFFSET extended ; Load table
                xlat    extended            ; Translate
                mov     bl,al               ; Save character
                mov     dx,OFFSET extmsg    ; Load message
                mov     ah,9                ; Display message
                int     21h
                mov     dl,bl               ; Load saved character
                mov     ah,2                ; Display character
                int     21h
ascii:
```

This example takes the extended codes used for ALT key sequences and looks them up in a table of equivalent ASCII codes.

■ 80386 Processor Only

The 80386 processor has an additional instruction, **XLATB**, which allows you to omit the operand as long as the register pair **DS:BX** or **DS:EBX** points to the table to be translated.

By default, **XLAT** and **XLATB** use **BX** to point to the table if the current code-segment size is 16 bits, or **EBX** if the segment size is 32 bits. You can use the keyword **WORD** or **DWORD** to override these assumptions.

■ Examples

```
                xlat    WORD table          ; Point to 16-bit table
                                            ;   in 32-bit segment
                xlat    DWORD table         ; Point to 32-bit table
                                            ;   in 16-bit segment
```

15.1.4 Transferring Flags

The 8086-family processors provide instructions for loading and storing flags in the **AH** register.

■ **Syntax**

LAHF
SAHF

The flag status at a given point can be saved, and then restored when needed later. Another use is to put the flags in a register where a particular flag can be modified directly. This is not usually necessary, since the most important flags can be modified directly with instructions.

■ **Example**

```
        lahf                        ; Load flags into AH
        xor     ah,010000b          ; Toggle auxiliary carry
        sahf                        ; Store flags from AH
```

15.2 Converting Data Sizes

Since moving data between registers of different sizes is illegal, the **CBW** instruction is provided to extend 8-bit values to 16-bit values, and the **CWD** instruction is provided to extend 16-bit values to 32-bit values.

■ **Syntax**

CBW
CWD

Both these instructions assume that numbers are signed. You should not use them with unsigned numbers.

The **CBW** instruction converts an 8-bit signed value in **AL** to a 16-bit signed value in **AX**. The **CWD** instruction is similar except that it sign-extends a 16-bit value in **AX** to a 32-bit value in the **DX:AX** register pair.

■ Example 1

```
            .DATA
mem8        DB      -5
mem16       DW      -5
            .CODE
            mov     al,mem8     ; Load 8-bit -5 (FBh)
            cbw                 ; Convert to 16-bit -5 (FFFBh) in AX

            mov     ax,mem16    ; Load 16-bit -5 (FFFBh)
            cwd                 ; Convert to 32-bit -5 (FFFF:FFFBh)
                                ;   in DX:AX
```

The **CBW** instruction will not work correctly if the value to be extended is unsigned. For example, if **AL** contains the unsigned value 251 (FBh), **CBW** extends FBh to FFFBh instead of to 00FBh. The processor cannot tell the difference between a signed number and an unsigned number. The programmer has to understand this difference and program accordingly.

To extend an unsigned 8-bit number, you can clear the upper half of the register, as in Example 2.

■ Example 2

```
            .DATA
mem8        DB      -5
            .CODE
            mov     al,mem8     ; Load 251 (FBh) from 8-bit memory
            xor     ah,ah       ; Zero upper half of register
```

■ 80386 Processor Only

The 80386 processor provides additional conversion instructions for 32-bit values.

■ Syntax

CWDE
CDQ

The **CWDE** instruction converts a signed 16-bit value in **AX** to a signed 32-bit signed value in **EAX**. The **CDQ** instruction converts a 32-bit signed value in **EAX** to a signed 64-bit value in the **EDX:EAX** register pair.

■ Example 4

```
             .DATA
mem16        DW      -5
mem32        DD      -5
             .CODE
             mov     ax,mem16    ; Load 16-bit -5 (FFFBh)
             cwde                ; Convert to 32-bit -5 (FFFFFFFBh)
                                 ;   in EAX
             mov     eax,mem32   ; Load 32-bit -5 (FFFFFFFBh)
             cdw                 ; Convert to 64-bit -5
                                 ;   (FFFFFFFF:FFFFFFFBh) in EDX:EAX
```

In addition, the 80386 processor provides instructions that move and extend a value in a single step. The same thing can be done in two steps with 80286 instructions, but the new 80386 instructions are faster.

■ Syntax

MOVSX *register,register/memory*
MOVZX *register,register/memory*

MOVSX moves a signed value into a register and sign-extends it. **MOVZX** moves an unsigned value into a register and zero-extends it.

■ Example 5

```
; Enhanced 80386 instructions

             movzx   ax,bl       ; Load unsigned 8-bit value into
                                 ;   16-bit register and zero extend

; Equivalent to these 80286 instructions

             mov     al,bl       ; Load 8-bit unsigned value
             xor     ah,ah       ; Clear the top of register

; Enhanced 80386 instructions

             movsx   ax,bl       ; Load unsigned 8-bit value into
                                 ;   16-bit register and sign extend

; Equivalent to these 80286 instructions

             mov     al,bl       ; Load 8-bit unsigned value
             cbw                 ; Sign extend to 16-bit register
```

15.3 Loading Pointers

The 8086-family processors provide several instructions for loading pointer values into registers or register pairs. They can be used to load either near or far pointers.

15.3.1 Loading Near Pointers

The **LEA** instruction loads a near pointer into a specified register.

■ Syntax

LEA *register,memory*

The destination register may be any general purpose register. The source operand may be any memory operand. The effective address of the source operand is placed in the destination.

The **LEA** instruction can be used to calculate the effective address of a direct memory operand, but this is usually not efficient, since the address of a direct memory operand is a constant known at assembly time. For example, the following statements are equivalent, but the second version is faster:

```
lea     dx,string         ; Load effective address - slow
mov     dx,OFFSET string ; Load offset - fast
```

The **LEA** instruction is more useful for calculating the address of indirect memory operands:

```
lea     dx,string[si]     ; Load effective address
```

■ 80386 Processor Only

Scaling gives the **LEA** instruction some interesting side effects with the 80386 processor. By using a 32-bit value as both the index and the base register in an indirect memory operand, you can multiply by the constants 2, 3, 4, 5, 8, and 9 more quickly than you could using the **mul** operator.

```
lea     ebx, [eax*2]        ; EBX = 2 * EAX
lea     ebx, [eax*2+eax]    ; EBX = 3 * EAX
```

```
lea     ebx, [eax*4]        ; EBX = 4 * EAX
lea     ebx, [eax*4+eax]    ; EBX = 5 * EAX
lea     ebx, [eax*8]        ; EBX = 8 * EAX
lea     ebx, [eax*8+eax]    ; EBX = 9 * EAX
```

Multiplication by constants can also sometimes be made faster using shift instructions, as described in Section 16.9.1.

15.3.2 Loading Far Pointers

The **LDS** and **LES** instructions load far pointers.

■ Syntax

LDS *register,memory*
LES *register,memory*

The memory address being pointed to is specified in the source operand, and the register where the offset is to be stored is specified in the destination operand. The segment register where the segment will be stored is specified in the instruction name. For example, **LDS** puts the segment in **DS** and **LES** puts the segment in **ES**. These instructions are often used with string instructions, as explained in Chapter 18, "Manipulating Strings."

■ Examples

```
lds     dx,array[bx+di]  ; Put address in DS:SI pair
les     dx,array[bx+si]  ; Put address in ES:DI pair
```

■ 80386 Processor Only

The 80386 processor has additional instructions for loading far pointers. These instructions are exactly like **LDS** and **LES**, except for the segment register where they put the segment address.

■ **Syntax**

LSS *register,memory*
LFS *register,memory*
LGS *register,memory*

The **LSS**, **LFS**, and **LGS** instructions loads the segment address into **SS**, **FS**, and **GS** respectively.

15.4 Transferring Data to and from the Stack

A stack is an area of memory where data is stored on a first-in-first-out (FIFO) basis. The stack has several purposes on the 8086-family processors. The **CALL**, **INT**, **RET**, and **IRET** instructions automatically use the stack to store the calling addresses of procedures and interrupts (see Sections 17.4 and 17.5). You can also use the **PUSH** and **POP** instructions and their variations to store values on the stack.

15.4.1 Pushing and Popping

On 8086-family processors, the stack follows strict register conventions. The **SP** instruction always points to the current location in the stack. The **PUSH** and **POP** instructions use the **SP** register to keep track of the current position in the stack.

When used in indirect memory operands, the **BP** register is referenced to the stack segment (SS register). This makes it convenient as the base of a frame of reference (a stack frame) within the stack.

■ **Syntax**

PUSH *register/memory*
POP *register/memory*

The **PUSH** instruction is used to store a 16-bit operand on the stack. The **POP** instruction is used to retrieve a previously pushed value. When a value is pushed onto the stack, the **SP** is decreased by two. When a value is popped off the stack, the **SP** register is increased by two. Although the stack always contains word values, the **SP** register points to bytes. Thus **SP** changes in multiples of two.

The value transferred by the **PUSH** and **POP** instructions can be a memory value, or a general-purpose or segment register.

80186-80386 Processors Only

Starting with the 80186, the **PUSH** instruction can also be given with an immediate operand. For example, the following statement is legal on the 80186, 80286, and 80386 processors:

```
        push    8               ; 3 clocks on 80286
```

This statement is faster than the following equivalent statements, which are required on the 8088 or 8086:

```
        mov     ax,8            ; 2 clocks on 80286
        push    ax              ; 3 clocks on 80286
```

Figure 15.1 illustrates how pushes and pops change the **SP** register. Note that the value pushed onto the stack remains in stack memory even after it has been popped. However, since the stack pointer is above it, the value is no longer known and may be overwritten the next time the stack is used.

```
                    Pushing Words onto the Stack
High memory

          Before push ax                 After push ax
         |                 |            |                 |
         +--------+--------+            +--------+--------+
         |                 |            |                 |
         +--------+--------+ <- SP      +--------+--------+
         |                 |            |   word from ax  |
         +--------+--------+            +--------+--------+ <- SP
         |                 |            |                 |
         +--------+--------+            +--------+--------+
         |                 |            |                 |
Low memory

                    Popping Words off the Stack
High memory

          Before pop ax                  After pop ax
         |                 |            |                 |
         +--------+--------+            +--------+--------+
         |                 |            |                 |
         +--------+--------+            +--------+--------+ < - SP
```

```
|   word from ax  |              |   word from ax  |
+--------+--------+ <- SP        +--------+--------+
|                 |              |                 |
+--------+--------+              +--------+--------+
|                 |              |                 |
Low memory
```

The **PUSH** and **POP** instructions are almost always used in pairs. Words are popped off the stack in the opposite order from what they were pushed on. You should normally do the same number of pops as pushes to return the stack to its original status. However, it is possible to return the stack to its original status by subtracting the correct number of words from the **SP** registrs.

Values on the stack can be accessed using indirect memory operands with **BP** as the base register.

■ Example

```
mov     bp,sp               ; Set stack frame
push    ax                  ; Push first;  SP = BP + 2
push    bx                  ; Push second; SP = BP + 4
push    cx                  ; Push third;  SP = BP + 6
.
.
.
mov     ax, [bp+6]          ; Put third in AX
mov     bx, [bp+4]          ; Put second in BX
mov     cx, [bp+2]          ; Put first in CX
.
.
.
sub     sp,6                ; Restore stack pointer
                            ;   two bytes per push
```

■ Processor Differences

The 8088 and 8086 processors differ from later Intel processors in how they push and pop the **SP** register. If you give the statement `push sp` with the 8088 or 8086, the word pushed will be the word **SP** has after the push operation. The same statement under the **80186**, **80286**, or **80386** processor pushes the word **SP** has before the push operation.

■ 80386 Processor Only

When a **PUSH** or **POP** instruction is used in a 32-bit code segment (one with **USE32** use type), the value transferred is a 32-bit value. The size of the stack segment is not important. Only the code size matters.

15.4.2 Using the Stack

The stack can be used to store temporary data. For example, in the Microsoft calling convention, the stack is used to pass arguments to a procedure. The arguments are pushed on to the stack before the call. The procedure retrieves and uses them. Then the stack is restored to its original position at the end of the procedure. The stack can also be used to store variables that are local to a procedure. Both these techniques are discussed in Section 17.4.3.

Another common use of the stack is used to store temporary data when there are no free registers available or when a particular register must hold more than one value. For example, the **CX** register usually holds the count for loops. If two loops are nested, the outer count is loaded into **CX** at the start. When the inner loop starts, the outer count is pushed onto the stack and the inner count loaded into **CX**. When the inner loop finishes, the originally count is popped back into **CX**.

■ Example

```
        mov     cx,10       ; Load outer loop counter
outer:  .
        .                   ; Start outer loop task
        .
        push    cx          ; Save outer loop value
        mov     cx,20       ; Load inner loop counter
inner:  .
        .                   ; Do inner loop task
        .
        loop    inner
        pop     cx          ; Restore outer loop counter
        .
        .                   ; Continue outer loop task
        .
        loop    outer
```

15.4.3 Saving Flags and Registers on the Stack

The flags word can also be pushed and popped onto the stack using the **PUSHF** and **POPF** instructions.

■ **Syntax**

PUSHF
POPF

These instructions are sometimes used to save the status of flags before a procedure call and then to restore the same status after the procedure.

■ **Example**

```
pushf
call    systask
popf
```

■ **80386 Processor Only**

When used from a 32-bit code segment, the **PUSHF** and **POPF** instructions do not automatically transfer 32-bit values. You must append the letter D (for doubleword) to the instruction name. Thus the 32-bit versions of these instructions are **PUSHFD** and **POPFD**.

15.4.4 Saving All Registers on the Stack

■ **80186-80386 Processors Only**

Starting with the 80186 processor, the **PUSHA** and **POPA** instructions were implemented to push or pop all the general-purpose registers with one instruction.

■ **Syntax**

PUSHA
POPA

These instructions can be used to save the status of all registers before a procedure call and then to restore them after the return. Using **PUSHA** and **POPA** is significantly faster and takes fewer bytes of code than pushing and popping each register individually.

The registers are pushed in the following order: **AX**, **CX**, **DX**, **BX**, **SP**, **BP**, **SI**, **DI**. The **SP** word pushed is the value before the first register is pushed. The registers are popped in the opposite order.

■ **Example**

```
        pusha
        call    systask
        popa
```

■ **80386 Processor Only**

When used from a 32-bit code segment, the **PUSHA** and **POPA** instructions do not automatically transfer 32-bit values. You must append the letter D (for doubleword) to the instruction name. Thus the 32-bit versions of these instructions are **PUSHAD** and **POPAD**.

15.4.5 Transferring Data to and from Ports

Ports are the gateways between hardware devices and the processor. Each port has a unique number through which it can be accessed. Ports are sometimes used for low-level communications with devices such as disks, the video display, or the keyboard. The **IN** and **OUT** instructions can be used to send data to or receive data from ports.

In applications programming, most communications with hardware is done with DOS or BIOS calls. Ports are more often used in systems programming. Since systems programming is beyond the scope of this manual, and since ports differ greatly depending on hardware, the **IN** and **OUT** instructions are not explained in detail here.

■ Syntax

IN *accumulator,portnumber*
IN *accumulator*,**DX**
OUT *portnumber,accumulator*
OUT DX,*accumulator*

When using the **IN** instruction, the number of the port is given as the source operand. It can either be an immediate value or the **DX** register. The value to be received from the port must be given as the destination operand. It must be the accumulator register (**AX** for word values or **AL** for byte values).

When using the **OUT** instruction, the number of the port is given as the destination operand. It can either be an immediate value or the **DX** register. The value to be sent to the port must be given as the source operand. It must be the accumulator register (**AX** for word values or **AL** for byte values).

■ Example

```
mov     dx,03DAh      ; Load port 03DAh
in      al,dx         ; Get a byte from port

out     43h,al        ; Send a byte to port 43h
```

■ 80186-80386 Processors Only

Starting with the 80186 processor, instructions were implemented to send strings of data to and from ports. The instructions are **INS**, **INSB**, **INSW**, **OUTS**, **OUTSB**, and **OUTSW**. The operation of these instructions is much like the operation of string instructions. They are discussed in Section 18.7.

■ 80386 Processor Only

The 80386 processor adds two additional instructions for sending doubleword strings of data to and from ports. The **INSD** and **OUTSD** instructions are discussed in Section ?.?.

Chapter 16

Doing Arithmetic and Bit Manipulations

The 8086-family processors provide instructions for doing calculations on byte, word, and doubleword values. Operations include addition, subtraction, multiplication, and division. You can also do calculations at the bit level. This includes the AND, OR, XOR, and NOT logical operations. Bits can also be shifted or rotated to the right or left.

This chapter explains how to use the instructions that do calculations on numbers and bits.

16.1 Adding

The **ADD**, **ADC**, and **INC** instructions are used for adding and incrementing values.

■ Syntax

ADD *register/memory,immediate*
ADD *register/memory,register*
ADD *register,register/memory*
ADC *register/memory,immediate*
ADC *register/memory,register*
ADC *register,register/memory*
INC *register/memory*

These instructions can work directly on 8-bit or 16-bit values (32-bit values on the 80386). They can be also be used in combination to do calculations on values that are too large to be held in a single register (such as 32-bit values on the 80286 or 64-bit values on the 80386). When used with **AAA** and **DAA**, they can be used to do calculations on BCD numbers, as described in Section 16.5.

16.1.1 Adding Values Directly

The **ADD** and **INC** instructions are used for adding to values in registers or memory.

The **INC** instruction takes a single register or memory operand. The value of the operand is incremeneted.

The **ADD** instruction adds values given in source and destination operands. The destination must be a register or memory operand. It will be destroyed by the operation. The source operand can be an immediate, memory, or register operand. It will not be destroyed by the operation. Since memory to memory operations are never allowed, the source and destination operands can never both be memory operands.

The result of the operation is stored in the source operand. The operands can be either 8 bit or 16 bit (32 bit on the 80386), but both should be the same.

An addition operation can be interpreted either as addition of signed numbers or of unsigned numbers. It is the programmer's responsibility to decide how the addition should be interpreted and to make take appropriate action if the sum is too large for the destination operand. When an addition overflows the possible range for signed numbers, the overflow flag is set. When an addition overflows the range for unsigned numbers, the carry flag is set.

■ Examples

```
               .DATA
mem8           DB       39        ;                         unsigned  signed
               .CODE
               mov      al,26     ; Start with register       26        26
               inc      al        ; Increment                  1         1
               add      al,76     ; Add immediate           + 76        76
                                  ;                         ----      ----
                                  ;                          103       103
               add      al,mem8   ; Add memory              + 39        39
                                  ;                         ----      ----
               mov      ah,al     ; Copy to AH               142      -114+overflow
               add      al,ah     ; Add register             142
                                  ;                         ----
                                  ;                           28+carry
```

This example shows 8-bit addition. When the sum exceeds 127, the overflow flag is set. A **JO** (Jump on Overflow) instruction at this point could transfer control to error-recovery statements. When the sum exceeds 255, the carry flag is set. A **JC** (Jump on Carry) instruction at this point could transfer control to error-recovery statements.

16.1.2 Adding with Bit Splicing

The **ADC** (add with carry) instruction makes it possible to add numbers larger than can be held in a single register. This technique is called "bit splicing."

The **ADC** instruction adds two numbers in the same fashion as the **ADD** instruction, except that the value of the carry flag is included in the addition. If a previous calculation has set the carry flag, then 1 will be added to the sum of the numbers. If the carry flag is not set, the **ADC** instruction has the same effect as the **ADD** instruction.

When adding numbers that must be placed in multiple registers, the carry flag should be ignored when adding the portion in the least significant register, but taken into account when adding portions in more significant registers. This can be done by using the **ADD** instruction for the least significant portion and the **ADC** instruction for more significant portions. If the operation is being done inside a loop, you can use the **ADC** instruction in each iteration, but you must specifically turn off the carry flag with the **CLC** (Clear Carry Flag) instruction before entering the loop so that it will not be used for the first iteration.

■ **Example**

```
            .DATA
mem32       DD      316423

            .CODE
            mov     ax,43981             ; Load immediate     43981
            cwd                          ;   into DX:AX
            add     ax,WORD PTR mem32    ; Add to both      + 316423
            adc     dx,WORD PTR mem32[2];   memory words     ------
                                         ; Result in DX:AX   460404
```

16.2 Subtracting

The **SUB**, **SBB**, **DEC**, and **NEG** instructions are used for subtracting and decrementing values.

■ Syntax

SUB *register/memory,immediate*
SUB *register/memory,register*
SUB *register,register/memory*
SBB *register/memory,immediate*
SBB *register/memory,register*
SBB *register,register/memory*
DEC *register/memory*
NEG *register/memory*

These instructions can work directly on 8-bit or 16-bit values (32-bit values on the 80386). They can be also be used in combination to do calculations on values that are to large to be held in a single register (such as 32-bit values on the 80286 or 64-bit values on the 80386). When used with the **AAA** and **DAA**, they can be used to do calculations on BCD numbers, as described in Section 16.5.

16.2.1 Subtracting Values Directly

The **SUB** and **DEC** instructions are used for subtracting from values in registers or memory. A related instruction, **NEG**, reverses the sign of a two's complement number.

The **DEC** instruction takes a single register or memory operand. The value of the operand is decremented.

The **NEG** instruction takes a single register or memory operand. The sign of the value of the operand is reversed. The **NEG** instruction should normally be used only on signed numbers. The result is usually meaningless if it is used on unsigned numbers.

The **SUB** instruction subtracts the values given in the source operand from the value of the destination operand. The destination must be a register or memory operand. It will be destroyed by the operation. The source operand can be an immediate, memory, or register operand. It will not be destroyed by the operation. Since memory-to-memory operations are never allowed, the source and destination operands cannot both be memory operands.

The result of the operation is stored in the source operand. The operands can be either 8 bit or 16 bit (32 bit on the 80386), but both should be the same.

A subtraction operation can be interpreted either as subtraction of signed numbers or of unsigned numbers. It is the programmer's responsibility to decide how the subtraction should be interpreted and to make take appropriate action if the result is too small for the destination operand. When a subtraction overflows the possible range for signed numbers, the carry flag is set. When a subtraction underflows the range for unsigned numbers (becomes negative), the sign flag is set.

■ Examples

```
              .DATA
mem8          DB        139          ;                          signed  unsigned
              .CODE
              mov       ah,142       ; Start with
              mov       al,95        ;   registers                 95       95
              dec       al           ; Decrement               -    1  -     1
              sub       al,76        ; Subtract immediate -   76  -    76
                                     ;                           ----     ----
                                     ;                             18       18
              neg       mem8         ; Reverse sign and
              sub       al,mem8      ;   subtract memory    - 139   - 139
                                     ;                         ----     ----
                                     ;                         -121    135+sign
              sub       al,ah        ; Subtract register    - 142
                                     ;                         ----
                                     ;                          249+carry
```

This example shows 8-bit subtraction. When the result goes below 0, the sign flag is set. A **JS** (Jump on Sign) instruction at this point could transfer control to error-recovery statements. When the result goes below –128, the carry flag is set. A **JC** (Jump on Carry) instruction at this point could transfer control to error recovery statements.

16.2.2 Subtracting with Bit Splicing

The **SBB** (subtract with borrow) instruction makes it possible to subtract from numbers larger than can be held in a single register.

The **SBB** instruction subtracts two numbers in the same fashion as the **SUB** instruction except that the value of the carry flag is included in the subtraction. If a previous calculation has set the carry flag, then 1 will be subtracted from the result. If the carry flag is not set, the **SBB** instruction has the same effect as the **SUB** instruction.

When subtracting numbers that must be placed in multiple registers, the carry flag should be ignored when subtracting the portion in the least significant registers, but taken into account when adding portions in more significant registers. This can be done by using the **SUB** instruction for the least significant portion and the **SBB** instruction for more significant portions. If the operation is being done inside a loop, you can use the **SBB** instruction in each iteration, but you must specifically turn off the carry flag with the **CLC** (Clear Carry Flag) instruction before entering the loop so that it will not be used for the first iteration.

■ Example

```
                .DATA
mem32           DD      316423

                .CODE
                mov     ax,WORD PTR mem32[0]; Load mem32       + 316423
                mov     dx,WORD PTR mem32[2];   into CX:BX
                mov     bx,43981            ; Load immediate     43981
                sub     ax,bx               ; Subtract DX:AX    ------
                sbb     dx,0                ; Result in CX:BX   272642
```

16.3 Multiplying

The **MUL** and **IMUL** instructions are used to multiply numbers. The **MUL** instruction should be used for unsigned numbers, while the **IMUL** instruction should be used for signed numbers. This is the only difference between the two.

■ Syntax

MUL *register/memory*
IMUL *register/memory*

The multiply instructions require that one of the factors be in the accumulator register (**AL** for 8-bit numbers, **AX** for 16-bit numbers, or **EAX** for 32-bit numbers). This register is implied; it should not be specified in the source code. It will be destroyed by the operation.

The other factor to be multiplied must be specified in a single register or memory operand. The operand will not be destroyed by the operation unless it is **DX**, **AH**, or **AL**.

Note that multiplying two 8-bit numbers will produce a 16-bit number in **AX**. If the product is a 16-bit number, it will be placed in **AX** and the overflow and carry flags will be set.

Similary, multiplying two 16-bit numbers will produce a 32-bit number in the **DX:AX** register pair. If the product is a 32-bit number, the most significant bits will be in **AX**, the least significant bits will be in **DX**, and the overflow and carry flags will be set. (The 80386 handles 64-bit products in the same way in the **EDX:EAX** register pair.)

Note

Multiplication is one of the slowest operations on 8086-family processors, and in particular on the 8088 and 8086. Multiplying by certain common constants can be done faster by shifting bits (Section 16.9.1) or by using 80386 scaling (Section 15.3.1).

■ Examples

```
        .DATA
mem16   DW      -30000
        .CODE
                        ; 8-bit unsigned multiply
        mov     al,23   ; Load AL                  23
        mov     bl,24   ; Load BL                * 24
        mul     bl      ; Multiply BL            -----
                        ; Product in AX           552
                        ;   overflow and carry set

                        ; 16-bit signed multiply
        mov     ax,50   ; Load AX                  50
                        ;                      -30000
        imul    mem16   ; Multiply memory       -----
                        ; Product in DX:AX   -1500000
                        ;   overflow and carry set
```

■ 80186-80386 Processors Only

Starting with the 80186, the **IMUL** instruction has two additional syntaxes that allow for 16-bit multiplies that produce a 16-bit product. (These instructions can be extended to 32 bits on the 80386.)

■ Syntax

IMUL *register16,immediate*
IMUL *register16,memory16,immediate*

You can specify a 16-bit immediate value as the source instruction and a word register as the destination operand. The product appears in the destination operand. The 16-bit result will be placed in the destination operand. If the product is too large to fit in 16 bits, the carry and overflow flags will be set. In this context, **IMUL** can be used for either signed or unsigned multiplication, since the 16-bit product is the same.

You can also specify three operands for **IMUL**. The first must be a 16-bit register operand. The second operand must be a 16-bit memory operand and the third operand must be a 16-bit immediate operand. The second and third operands are multiplied and the product stored in the first operand.

With both these syntaxes, the carry and overflow flags will be set if the product is too large to fit in 16 bits. The **IMUL** instruction with multiple operands can be used for either signed or unsigned multiplication, since the 16-bit product is the same in either case. If you need to get a 32-bit result, you must use the single-operand version of **MUL** or **IMUL**.

■ Examples

```
imul    dx,456     ; Multiply DX times 456
imul    ax,[bx],6  ; Multiply the value pointed to by BX
                   ;   times 6 and put the result in AX
```

16.4 Dividing

The **DIV** and **IDIV** instructions are used to divide integers. Both a quotient and a remainder are returned. The **DIV** instruction should be used for unsigned integers, while the **IDIV** instruction should be used for signed integers. This is the only difference between the two.

■ Syntax

DIV *register/memory*
IDIV *register/memory*

To divide a 16-bit number by an 8-bit number, put the number to be divided (the dividend) in the **AX** register. This register will be destroyed by the operation. Specify the number to be divided (the divisor) in any 8-bit memory or register operand (except **AL** or **AH**). This operand will not be changed by the operation. After the multiplication, the result (quotient) will be in **AL** and the remainder will be in **AH**.

To divide a 32-bit number by a 16-bit number, put the dividend in the **DX:AX** register pair. The most significant bits go in **AX**. These registers will be destroyed by the operation. Specify the divisor in any 16-bit memory or register operand (except **AX** or **DX**). This operand will not be changed by the operation. After the division, the quotient will be in **AX** and the remainder will be in **DX**. (The 80386 handles 64-bit division in the same way using the **EDX:EAX** register pair.)

To divide a 16-bit number by a 16-bit number, you must sign-extend or zero-extend (see Section 15.2) the dividend to 32 bits. Then do the division as described above. You cannot divide a 32-bit number by another 32-bit number (except on the 80386).

If division by zero is specified, or if the quotient exceeds the capacity of its register (**AL** or **AX**), the processor automatically generates an interrupt 0. By default, the program will terminate and return to DOS. This problem can be handled in two ways: You can check the divisor before division and go to an error routine if it can be determined to be invalid, or you can also write your own interrupt routine to replace the processor's interrupt 0 routine. See Section 17.5 for more information in interrupts.

Note

Division is one of the slowest operations on 8086-family processors, and in particular on the 8088 and 8086. Dividing by certain common constants can be done faster by shifting bits as described in Section 16.9.1.

■ Examples

```
            .DATA
mem16       DW      -2000
mem32       DD      500000
            .CODE
                                        ; Divide 16-bit unsigned by 8-bit
            mov     ax,700              ; Load dividend          700
            mov     bl,36               ; Load divisor      DIV   36
            div     bl                  ; Divide BL             -----
                                        ; Quotient in AL          19
                                        ; Remainder in AH            16

                                        ; Divide 32-bit signed by 16-bit

            mov     ax,WORD PTR mem32[0] ; Load into DX:AX
            mov     dx,WORD PTR mem32[2] ;                     500000
            idiv    mem16               ;                   DIV -2000
                                        ; Divide memory          ------
                                        ; Quotient in AX          -250
                                        ; Remainder in DX               0

                                        ; Divide 16-bit signed by 16-bit

            mov     ax,WORD PTR mem16   ; Load into AX            2000
            cwd                         ; Extend to DX:AX
            mov     bx,-421             ;                    DIV -421
            idiv    bx                  ; Divide by BX           -----
                                        ; Quotient in AX            -4
                                        ; Remainder in DX              316
```

16.5 Calculating with Binary Coded Decimals

The 8086-family processors provide several instructions for adjusting BCD numbers. The BCD format is seldom used in assembly-language applications programming. Programmers who wish to use BCD numbers usually use a high-level langugage. However, BCD instructions are used to develop compilers, function libraries, and other systems tools.

Since systems programming is beyond the scope of this manual, this section provides only a brief overview of calculations on the two kinds of BCD numbers: unpacked and packed.

Note

Intel mnemonics use the term "ASCII" to refer to unpacked BCD numbers and "decimal" to refer to packed BCD numbers. Thus **AAA** (ASCII Adjust for Addition) adjusts unpacked numbers, while **DAA** (Decimal Adjust for Addition) adjusts packed numbers.

16.5.1 Calculating with Unpacked BCD Numbers

Unpacked BCD numbers are made up of bytes containing a single decimal digit in the lower 4 bits of each byte. The 8086-family processors provide instructions for adjusting unpacked values with the four arithmetic operations: addition, subtraction, multiplication, and division.

To do arithmetic on unpacked BCD numbers, you must do the 8-bit arithmetic calculations on each digit separately. The result should always be in the **AL** register. After each operation, use the corresponding BCD instruction to adjust the result. The ASCII adjust instructions do not take an operand. They always work on the value in the **AL** register.

When a calculation using two one-digit values produces a two-digit result, the ASCII adjust instructions put the first digit in **AL** and the second in **AH**. If the digit in **AL** needs to carry to or borrow from the digit in **AH**, the carry and auxiliary carry flags are set.

The four ASCII adjust instructions are described below:

Instruction | **Description**

AAA

Adjusts after an addition operation. For example, to add 9 and 3, put 9 in **AL** and 3 in **BL**. Then use the following lines to add them:

```
add     al,bl     ; Add 09h and 03h to get 0Ch
aaa               ; Adjust 0Ch in AL to 02h and
                  ;   put carried digit (1) in AH
                  ;   set carry and auxiliary carry
```

AAS

Adjust after a subtraction operation. For example, to subtract 4 from 3, put 3 in **AL** and 4 in **BL**. Then use the following lines to subtract them:

```
sub     al,bl      ; Subtract 04h from 03h to get 0FFh
aas                ; Adjust 0FFh in AL to 09h and
                   ;   put carried digit (0FFh) in AH;
                   ;   set carry and auxiliary carry
```

AAM

Adjust after a multiplication operation. Always use **mul**, not **imul**. For example, to multiply 9 times 3, put 9 in **AL** and 3 in **BL**. Then use the following lines to multiply them:

```
mul     bl         ; Multiply 09h and 03h to get 01Bh
aaa                ; Adjust 01Bh in AL to 027h
                   ;   with 2 in AH and 7 in AL
```

AAD

Adjust before a division operation. Unlike other BCD instructions, this one converts a BCD value to a binary value before the operation. After the operation, the quotient must still be adjusted using **AAM**. For example, to divide 25 by 2, put 25 in **AX** in unpacked BCD format: 2 in **AH** and 5 in **AL**. Put 2 in **BL**. Then use the following lines to divide them:

```
aad                ; Adjust 0205h in AX
                   ;   to 19h in AX
div     bl         ; Divide by 3 to get
                   ;   quotient 0Ch in AL
                   ;   remainder 1 in AH
aam                ; Adjust 0Ch in AL to 012h
                   ;   with 1 in AH and 2 in AL
                   ;   (remainder destroyed)
```

Notice that the remainder is lost. If you need the remainder, save it in another register before adjusting the quotient. Then move it back to **AL** and adjust if necessary.

Multidigit BCD numbers are usually processed in loops. Each digit is processed and adjusted in turn.

In addition to its use for processing unpacked BCD numbers, the **AAM** instruction can be used in routines that convert binary numbers to decimal.

■ Example

```
            .DATA
number      DB      79
            .CODE
            mov     al,number   ; Load 79 (04Fh)
            aam                 ; Adjust to BCD format
                                ;   7 in AH; 9 in AL
            add     ah,48       ; Adjust to ASCII characters
            add     al,48
            mov     dx,ax       ; Move to DX
            xchg    dl,dh       ;   and trade digits
            mov     ah,2        ; Display most significant digit
            int     21h
            mov     dl,dh       ; Load least significant digit
            int     21h         ;
```

The example only handles two-digit numbers, but could be enhanced to handle large numbers.

16.5.2 Calculating with Packed BCD Numbers

Packed BCD numbers are made up of bytes containing two decimal digits: one in the upper 4 bits and one in the lower 4 bits. The 8086-family processors provide instructions for adjusting packed BCD numbers after addition and subtraction. You must write your own routines to adjust for multiplication and division.

To do arithmetic on packed BCD numbers, you must do the 8-bit arithmetic calculations on each byte separately. The result should always be in the **AL** register. After each operation, use the corresponding BCD instruction to adjust the result. The decimal adjust instructions do not take an operand. They always work on the value in the **AL** register.

Unlike the ASCII adjust instructions, the decimal adjust instructions never affect **AH**. The auxiliary carry flag is set if the digit in the lower 4 bits carries to or borrows from the digit in the upper 4 bits. The carry flag is set if the digit in the upper 4 bits needs to carry to or borrow from another byte.

The decimal adjust instructions are described below:

Instruction	Description
DAA	Adjust after an addition operation. For example, to add 88 and 33, put 88 in **AL** and 33 in **BL**. Then use the following lines to add them:

```
add     al,bl   ; Add 88h and 33h to get OBBh
daa             ; Adjust OBBh in AL to 21h
                ;   set carry and auxiliary carry
```

DAS Adjust after a subtraction operation. For example, to subtract 38 from 83, put 83 in **AL** and 38 in **BL**. Then use the following lines to subtract them:

```
sub     al,bl   ; Subtract 38h from 83h to get 04Bh
das             ; Adjust 04Bh in AL to 45h
                ;   set auxiliary carry
```

Multidigit BCD numbers are usually processed in loops. Each byte is processed and adjusted in turn.

16.6 Doing Logical Bit Manipulations

The logical instructions do boolean operations on individual bits. The AND, OR, XOR, and NOT operations are supported by the 8086-family instructions.

AND compares two bits and sets the result if both bits are set. OR compares two bits and sets the result if either bit is set. XOR compares two bits and sets the result if the bits are different. NOT reverses a single bit. Table 16.1 shows a truth table for the logical operations.

Table 16.1

Values Returned by Logical Operations

X	Y	NOT X	X AND Y	X OR Y	X XOR Y
1	1	0	1	1	0
1	0	0	0	1	1
0	1	1	0	1	1
0	0	1	0	0	0

The **AND**, **OR**, and **XOR** instructions work exactly the same except for the operation performed. The target value to be changed by the operation is placed in one operand. A mask showing the positions of bits that will be changed is placed in the other operand. The format of the mask differs for each logical instruction. The destination operand can be register or memory. The source operand can be register, memory, or immediate. However, the source and destination operands cannot both be memory.

Either of the values can be in either operand. However, the source operand will be unchanged by the operation, while the destination operand will be destroyed by it. Your choice of operands depends on whether you want to save a copy of the mask or of the target value.

Note

> The logical instructions should not be confused with the logical operators. They specify completely different behavior. The instructions control run-time bit calculations. The operators control assembly-time bit calculations. Although the instructions and operators have the same name, the assembler can distinguish them from context.

16.6.1 Doing AND Operations on Bits

The **AND** instruction does an AND operation on the bits of the source and destination operands. The original destination operand is replaced by the resulting bits.

■ Syntax

AND *register/memory,immediate*
AND *register/memory,register*
AND *register,register/memory*

The **AND** instruction can be used to clear the value of specific bits regardless of their current settings. To do this, put the target value in one operand and a mask of the bits you want to clear in the other. The bits of the mask should be 0 for any bit positions you want to clear and 1 for any bit positions you want to remain unchanged.

■ **Example**

```
mov     ax,035h     ; Load value                00110101
and     ax,0FBh     ; Mask off bit 2        AND 11111011
                    ;                           --------
                    ; Value is now 31h          00110001
and     ax,F8h      ; Mask off bits 2,1,0   AND 11111000
                    ;                           --------
                    ; Value is now 30h          00110000
```

16.6.2 Doing OR Operations on Bits

The **OR** instruction does an OR operation on the bits of the source and destination operands. The original destination operand is replaced by the resulting bits.

■ **Syntax**

OR *register/memory,immediate*
OR *register/memory,register*
OR *register,register/memory*

The **OR** instruction can be used to set the value of specific bits regardless or their current settings. To do this, put the target value in one operand and a mask of the bits you want to clear in the other. The bits of the mask should be 1 for any bit positions you want to set and 0 for any bit positions you want to remain unchanged.

■ **Example**

```
mov     ax,035h     ; Move value to register     00110101
or      ax,0FBh     ; Mask on bit 3           OR 00001000
                    ;                            --------
                    ; Value is now 3Dh           00111101
or      ax,F8h      ; Mask on bits 2,1,0      OR 00000111
                    ;                            --------
                    ; Value is now 3Fh           00111111
```

Another common use for **OR** is to compare an operand to 0. For example:

```
or      bx,bx       ; Compare to 0
                    ;   2 bytes, 2 clocks on 8088
jg      positive    ; BX is positive
jl      negative    ; BX is negative
                    ; BX is zero
```

The first statement has the same effect as the following statement, but is faster and smaller:

```
cmp     bx,0        ; 3 bytes, 3 clocks on 8088
```

16.6.3 Doing XOR Operations on Bits

The **XOR** (Exclusive OR) instruction does an XOR operation on the bits of the source and destination operands. The original destination operand is replaced by the resulting bits.

■ Syntax

XOR *register/memory,immediate*
XOR *register/memory,register*
XOR *register,register/memory*

The **XOR** instruction can be used to toggle the value of specific bits (reverse them from their current settings). To do this, put the target value in one operand and a mask of the bits you want to toggle in the other. The bits of the mask should be 1 for any bit positions you want to toggle and 0 for any bit positions you want to remain unchanged.

■ Example

```
mov     ax,035h     ; Move value to register       00110101
xor     ax,0FBh     ; Mask on bit 3            XOR 00001000
                    ;                              --------
                    ; Value is now 3Dh             00111101
xor     ax,F8h      ; Mask on bits 2,1,0       XOR 00000111
                    ;                              --------
                    ; Value is now 3Ah             00111010
```

Another common use for the **XOR** instruction is to set a register to 0. For example:

```
xor     cx,cx       ; 2 bytes, 3 clocks on 8088
```

This sets the **CX** register to 0. When the identical operands are XORed, each bit cancels itself, producing 0. The statement

```
mov     cx,0        ; 3 bytes, 4 clocks on 8088
```

is the obvious way of doing this, but it is larger and slower. The statement

```
sub     cx,cx       ; 2 bytes, 3 clocks on 8088
```

is also smaller than the **MOV** version. The only advantage of using **MOV** is that it does not affect any flags.

16.6.4 Doing NOT Operations on Bits

The **NOT** instruction does a NOT operation on the bits of a single operand. It is used to toggle the value of all bits at once.

■ Syntax

NOT *register/memory*

The **NOT** instruction is often used to reverse the sense of a bit mask from masking certain bits on to masking them off.

■ Example

```
            .DATA
bitcount    DB      2
            .CODE
            mov     bx,035h    ; Move value to register      00110101
            or      ax,1       ; Load single bit             00000001
            mov     cl,bitcount; Might change at run time
            shl     ax,cl      ; Mask on bit 3               00000100

            not     ax         ; Mask off for AND            11111011
            and     bx,ax      ; Clear masked bit        AND 00110101
                               ;                             --------
                               ; Value is now 31h            00110001
```

16.7 Testing Bits

■ 80386 Processor Only

The 80386 processor has bit test instructions that were not available in earlier versions. These instructions have two purposes. They can test the status of a bit to control program flow. Some of them can also change the value of a specified bit.

■ Syntax

BT *register/memory,register/immediate*
BTC *register/memory,register/immediate*
BTR *register/memory,register/immediate*
BTS *register/memory,register/immediate*

For each of the instructions, the memory or register destination operand is the target value that will be tested. The register or immediate source operand specifies the number of the bit to be tested in the destination operand. The four bit testing instructions are described below:

Instruction	Description
BT	The Bit Test instruction examines the specified bit in the target value and puts a copy in the carry flag. The carry flag can then be used by another instruction such as a conditional jump. For example, assume **CX** contains 4 in the following statements:

```
mov     ax,3Dh      ; Load 00111101b
bt      ax,cx       ; Put bit 4 in carry
jc      somewhere
```

The same thing could be done less efficiently on other 8086-family processors with the following statements:

```
mov     ax,3Dh      ; Load          00111101b
and     ax,1        ; Create mask   00000001b
shl     ax,cl       ; Adjust        00010000b
test    ax,cx       ; Put bit 4 in zero
jnz     somewhere
```

This instruction is only useful if the source operand is only known at run time, since the **TEST** instruction (see Section 17.1.1.2) is more efficient if the source is known at assembly time.

BTC

The Bit Test and Complement instruction examines the specified bit in the target value and puts a copy in the carry flag. It then reverses the value of the bit. For example, assume **CX** contains 4 in the following statements:

```
mov     ax,3Dh      ; Load 00111101b
btc     ax,cx       ; Put bit 4 in carry
                    ;   and toggle bit 4
jc      somewhere
```

BTR

The Bit Test and Reset instruction examines the specified bit in the target value and puts a copy in the carry flag. It then clears the bit. For example, assume **CX** contains 4 in the following statements:

```
mov     ax,3Dh      ; Load 00111101b
btr     ax,cx       ; Put bit 4 in carry
                    ;   and clear bit 4
jc      somewhere
```

BTC

The Bit Test and Set instruction examines the specified bit in the target value and puts a copy in the carry flag. It then sets the bit. For example, assume **CX** contains 4 in the following statements:

```
mov     ax,3Dh      ; Load 00111101b
btc     ax,cx       ; Put bit 4 in carry
                    ;   and set bit 4
jc      somewhere
```

■ Example

```
          .DATA
errors    RECORD  a:1=0,b:2=0,c:1=0,d:2=0,e:1=0,f:1=0
error     errors  <>
          .CODE
          .
          .
          .
          btr     error,c
          jc      fixc
          .
          .
fixa:     .
```

In this example, a bit field made up of error flags is tested. If the bit flag being tested is set, indicating an error, the flag is turned off and control is directed to a label where the error is corrected.

16.8 Scanning for Set Bits

■ 80386 Processor Only

The 80386 processor has instructions for scanning bits to find the first or last set bit in a register value. These instructions can be used to find the position of a set bit in a mask or other value. They can also check to see if a register value is zero.

The bit scan instructions work only on 16-bit or 32-bit registers. They cannot be used on memory operands or 8-bit registers. The source register contains the value to be scanned. The destination register should be the register where you want to store the position of the first or last set bit.

The **BSF** (Bit Scan Forward) instruction scans the bits of the source register starting with the 0 bit and working toward the most significant bit. The **BSR** (Bit Scan Reverse) instruction scans the bits of the source register starting with the most significant bit and working toward the 0 bit.

■ Example

```
                .DATA
widfield        EQU     200
bitfield        DD      widfield DUP (?)
                .CODE
                .
                .
                .
                cld
                mov     ax,ds                   ; Load segment of bitfield
                mov     es,ax                   ;   into ES
                mov     ecx,widfield            ; Load maximum count
                xor     eax,eax                 ; Set search value to 0
                mov     edi,OFFSET bitfield; Load bitfield address
                repe    scasd                   ; Find first nonzero bit
                jecxz   none                    ; If none found, get out
                mov     eax,[edi]               ; Else load first nonzero
                bsr     ecx,eax                 ; Find first set bit
                .                               ; ECX now contains bit position
                .
                .
```

This example scans a large bitfield. Starting at the beginning of the field, it finds the first nonzero doubleword. Then it finds the first set bit within the doubleword. See Chapter 18, "Manipulating Strings," for more information on the string instructions used in this example.

16.9 Shifting and Rotating Bits

The 8086-family processors provide a complete set of instructions for shifting and rotating bits. Bits can be moved right (toward the most significant bits) or left (toward the 0 bit). Values shifted off the end go into the carry flag.

Shift instructions move bits a specified number of places to the right or left. The last bit in the direction of the shift goes into the carry flag, and the first bit is filled with 0 or with the previous first bit.

Rotate instructions move bits a specified number of places to the right or left. For each bit rotated, the last bit in the direction of the rotate is moved into the first bit position at the other end of the operand. With some variations, the carry bit is used as an additional bit of the operand.

Figure 16.1 illustrates the eight variations of shift and rotate instructions for 8-bit operands. Notice that **SHL** and **SAL** are exactly the same.

```
SHL (Shift Left)

  C     7  6  5  4  3  2  1  0
+--+  +--+--+--+--+--+--+--+--+
|  |< |< |< |< |< |< |< |< |< 0
+--+  +--+--+--+--+--+--+--+--+
```

```
SHR (Shift Right)

  7  6  5  4  3  2  1  0    C
+--+--+--+--+--+--+--+--+  +--+
0 >| >| >| >| >| >| >| >| >|  |
+--+--+--+--+--+--+--+--+  +--+
```

```
SAL (Shift Arithmetic Left)

  C     7  6  5  4  3  2  1  0
+--+  +--+--+--+--+--+--+--+--+
|  |< |< |< |< |< |< |< |< |< 0
+--+  +--+--+--+--+--+--+--+--+
```

```
SAR (Shift Arithmetic Right)

  7  6  5  4  3  2  1  0    C
+--+--+--+--+--+--+--+--+  +--+
| >| >| >| >| >| >| >| >| >|  |
+--+--+--+--+--+--+--+--+  +--+
 ^_|
```

```
ROL (Rotate Left)

  C     7  6  5  4  3  2  1  0
+--+  +--+--+--+--+--+--+--+--+
|  |< |< |< |< |< |< |< |< |< |
+--+ /+--+--+--+--+--+--+--+--+
    v__________________________^
```

```
ROR (Rotate Right)

  7  6  5  4  3  2  1  0    C
+--+--+--+--+--+--+--+--+  +--+
| >| >| >| >| >| >| >| >| >|  |
+--+--+--+--+--+--+--+--+ +--+
 ^_______________________v
```

```
RCL (Rotate Through Carry Left)

  C     7  6  5  4  3  2  1  0
+--+  +--+--+--+--+--+--+--+--+
|  |< |< |< |< |< |< |< |< |< |
+--+  +--+--+--+--+--+--+--+--+
 v_____________________________^
```

```
RCR (Rotate Through Carry Right)

  7  6  5  4  3  2  1  0    C
+--+--+--+--+--+--+--+--+  +--+
| >| >| >| >| >| >| >| >| >|  |
+--+--+--+--+--+--+--+--+  +--+
 ^___________________________v
```

■ Syntax

SHL *register/memory*,1
SHL *register/memory*,CL
SHR *register/memory*,1
SHR *register/memory*,CL
SAL *register/memory*,1
SAL *register/memory*,CL
SAR *register/memory*,1
SAR *register/memory*,CL
ROL *register/memory*,1
ROL *register/memory*,CL
ROR *register/memory*,1
ROR *register/memory*,CL
RCL *register/memory*,1
RCL *register/memory*,CL
RCR *register/memory*,1
RCR *register/memory*,CL

The format of all the shift instructions is the same. The destination operand should contain the value to be shifted. It will contain the shifted operand after the instruction. The source operand should contain the number of bits to shift or rotate. It can be the immediate value 1 or the **CL** register. No other value or register is accepted on the 8088 and 8086 processors.

80186-80386 Processors Only

Starting with the 80186 processor, 8-bit immediate values larger than 1 can be given as the source operand for shift or rotate instructions. For example:

```
shr     bx,4        ; 9 clocks on 80286
```

The following statements would be equivalent if the program must run the 8088 or 8086:

```
mov     cl,4        ; 2 clocks on 80286
shr     bx,cl       ; 9 clocks on 80286
```

16.9.1 Multiplying and Dividing by Constants

Shifting right by one has the effect of dividing by two, while shifting left by one has the effect of multiplying by two. You can take advantage of this to do fast multiplication and division by common constants. The easiest constants are the powers of two. Shifting left twice multiplies by 4, shifting left three times multiplies by 8, and so on.

SAR should be used to divide signed numbers, while **SHR** is used to divide unsigned numbers. **SAL** and **SHL** synonyms for the same instruction. Multiplication by shifting is the same for signed and unsigned numbers, so either mnemonic can be used.

Since the multiply and divide instructions are the slowest on 8086-family processors, using shifts instead can often speed operations by a factor of 10 or more. For example, on the 8088 or 8086 processor, the following statement takes 2 clocks:

```
        shl     al,1
```

The following statements have the same effect, but take between 74 and 81 clocks on the 8088 or 8086:

```
        mov     bh,2
        mul     bh
```

The same statements take 15 clocks on the 80286 or between 11 and 16 clocks on the 80386. See the *Microsoft Macro Assembler Reference* for complete information on timing of instructions.

Shift instructions can be combined with add or subtract instructions to do multiplication by common constants. These operations are best put in macros so that they can be changed if the constants in a program change.

■ **Example 1**

```
mul_10      MACRO   factor
            mov     ax,factor    ; Load into AX
            shl     ax,1         ; AX = factor * 2
            mov     bx,ax        ; Save copy in BX
            shl     ax,1         ; AX = factor * 4
            shl     ax,1         ; AX = factor * 8
            add     ax,bx        ; AX = (factor * 8) + (factor * 2)
            ENDM                 ; AX = factor * 10
```

■ **Example 2**

```
div_u512    MACRO   dividend    ; Unsigned only
            mov     ax,dividend; Load into AX
            shr     ax,1        ;   AX = dividend / 2 (unsigned)
            xchg    al,ah       ; xchg is like rotate right 8)
                                ;   AL = (dividend / 2) / 256
            cbw                 ; Clear upper byte
            ENDM                ;   AX = (dividend / 512
```

16.9.2 Moving Bits to the Most Significant Position

Sometimes a group of bits within an operand needs to be treated as a single unit (for example, to do an arithmetic operation on those bits without affecting other bits). This can be done by masking off the bits, then shifting them into the most significant positions. After the arithmetic operation is done, they are shifted back to the original position and merged with the original bits using OR. An example of this is shown in Section 7.2.5.1.

16.9.3 Adjusting Masks

Masks for logical instructions can be shifted to new bit positions. For example, an operand that masks off a bit or group of bits can be shifted to move the mask to a different position.

■ **Example**

```
                        ; Assume AX has run-time value  10100100b
        shr     ax,1    ; Adjust mask right             01010010b
        or      bx,ax   ; Set bits 1, 4, and 6
        shl     ax,1    ; Adjust mask left              10100100b
        shl     ax,1                                    01001000b
        not     ax      ; Reverse mask                  10110111b
        and     bx,ax   ; Clear bits 3 and 6
```

This technique is only useful if the mask value is not known until run time.

16.9.4 Shifting Multiword Values

Sometimes it is necessary to shift a value that is too large to fit in a register. In this case, you can shift each part separately, passing the shifted bits through the carry flag. The **RCR** or **RCL** instructions must be used to move the carry value from the first register to the second.

RCR and **RCL** can also be used to initialize the high or low bit of an operand. Since the carry flag is treated as part of the operand (like using a 9-bit operand), the flag value before the operation is crucial. The carry flag may be set by a previous instruction, or you can set it directly using the **CLC** (Clear Carry Flag), **CMC** (Complement Carry Flag), and **STC** (Set Carry Flag) instructions.

■ Example

```
                .DATA
mem32           DD      500000
                .CODE
                mov     ax,WORD PTR mem32[0] ; Load into DX:AX
                mov     dx,WORD PTR mem32[2]
                mov     cx,4                 ; Shift right 4 (divide by 16)
again:          shr     dx,1                 ; Shift once into carry
                rcr     ax,1                 ; Rotate carry in
                loop    again
```

■ 80386 Processor Only

The 80836 processor has new instructions for shifting multiple bits into an operand. The **SHLD** (Double Precision Shift Left) instruction shifts a specified group of bits left and into an operand. The **SHRD** (Double Precision Shift Right) instruction shifts a specified group of bits right and into an operand.

■ Syntax

SHRD *register*,*register*,*immediate*
SHRD *register*,*register*,**CL**
SHLD *register*,*register*,*immediate*
SHLD *register*,*register*,**CL**

These instructions take three operands. The first (leftmost) contains the value to be shifted. It must be a 16-bit or 32-bit register. The second operand contains the bits to be shifted into the value. It must be a register of the same size as the first operand. The third operand contains the number of bits to shift. It may be an immediate operand or the **CL** register.

■ Example

```
mov     ax,3AF2h     ; Load    AX=00111010 11110010
mov     bx,09Ch      ; Load    BX=                  10011100
shrd    ax,bx,7      ; Shift 7       <- 0 11110010 10011100
                     ;            -----------------
                     ;         AX=01111001 01001110 (794Eh)
```

Chapter 17
Controlling Program Flow

The 8086-family processors provide a variety of instructions for controlling the flow of a program. The four major types of program-flow instructions are jumps, loops, procedure calls, and interrupts.

This chapter tells how to use these instructions and how to test conditions for the instructions that change program flow conditionally.

17.1 Jumping

Jumps are the most direct method of moving from one location in code to another. At the internal level, jumps work by changing the value of the **IP** (instruction pointer) register from the address of the current instruction to a target address.

17.1.1 Jumping Conditionally

The most common way of transferring control in assembly language is with conditional jumps. This is a two-step process: first test the condition, then jump if the condition is true or continue if it is false.

■ Syntax

J*condition register/memory*

Conditional-jump instructions take a single operand containing the address to be jumped to. The distance from the jump instruction to the specified address must be "short" (less than 128 bytes). If a longer distance is specified, an error will be generated telling the distance of the jump in bytes. See Section 17.1.2 for information on arranging longer conditional jumps.

Conditional jump instructions (except **JCXZ**) use as their condition the status of one or more flags. Thus any statement that sets a flag under specified conditions can be the test statement. The most common test statements use the **CMP** or **TEST** instructions. The jump statement can be any one of 31 conditional jump instructions.

17.1.1.1 Comparing and Jumping

The **CMP** instruction is specifically designed to test for conditional jumps. It does not change either operand, and so can be used to compare two values nondestructively. Destructive instructions can also be used to test conditions.

The **CMP** instruction compares two operands and sets a flag based on the result. It is used to test the following relationships: equal, not equal, greater than, less than, greater than or equal, less than or equal.

■ Syntax

CMP *register/memory,immediate*
CMP *register/memory,register*
CMP *register,register/memory*

The destination operand can be memory or register. The source operand can be immediate, memory, or register. However, they cannot both be memory operands.

The jump instructions that can be used with **CMP** are made up of mnemonic letters that are combined to indicate the type of jump. The letters are shown below:

Letter	Meaning
J	Jump
G	Greater than (for unsigned comparisons)
L	Less than (for unsigned comparisons)
A	Above (for signed comparisons)
B	Below (for signed comparisons)
E	Equal
N	Not

The mnemonic names always refer to the relationship the first operand of the **CMP** instruction has to the second operand of the **CMP** instruction. For example, **JG** tests whether the first operand is greater than the second. Several conditional instructions have two names. You can use whichever name seems more mnemonic in context.

Comparisons and conditional jumps can be thought of as statements in the following format:

IF (*value1 relationship value2*) **THEN GOTO** *truelabel*

Statements of this type can be coded in assembly language using the following syntax:

CMP *value1*,*value2*
J*relationship truelabel*
.
.
.
truelabel:

Table 17.1 lists conditional jump instruction for each *relationship*, and shows the flags that are tested to see if the *relationship* is true.

Table 17.1

Conditional Jump Instructions Used after Compare

Jump Condition		Signed Compare	Jump if:	Unsigned Compare	Jump if:
Equal	$=$	**JE**	**ZF**=1	**JE**	**ZF**=1
Not equal	$\neq$	**JNE**	**ZF**= **1**	**JNE**	**ZF**=1
Greater than	$>$	**JG** or **JNLE**	**ZF**=0 and **SF**= **OF**	**JA** or **JNBE**	**CF**=0 and **ZF**=0
Less than or equal	$\leq$	**JLE** or **JNG**	**ZF**=1 and **SF**$\neq$ **OF**	**JBE** or **JNA**	**CF**=1 or **ZF**=1
Less than	$<$	**JL** or **JNGE**	**SF**$\neq$ **OF**	**JB** or **JNAE**	**CF**=1
Greater than or equal	$\geq$	**JGE** or **JNL**	**SF**= **OF**	**JAE** or **JNB**	**CF**=0

The other conditional jump instructions described in Section 17.1.3 can be used after **CMP**, although they are less useful in this context.

Internally, the **CMP** instruction is exactly the same as the **SUB** instruction, except that the destination operand is not changed. The flags are set according to the result that would have been generated by a subtraction.

■ Example 1

```
; If CX is less than -20, then make DX 30, else make DX 20

            cmp     cx,-20      ; If signed CX is smaller than -20
            jl      less        ;    Then do stuff at "less"
            mov     dx,20       ; Else set DX to 20
            jmp     continue    ; Finished
less:       mov     dx,30       ; Then set DX to 30
continue:
```

Example 1 shows the basic form of conditional jumps. Notice that in assembly language, if-then-else constructions are usually written in the form if-else-then.

This theme has many variations. For example, you may find it more mnemonic to code in the if-then-else format. However, you must then use the opposite jump condition, as shown in Example 2.

■ Example 2

```
; If CX is greater than or equal to 40, then make DX 60, else make DX 10

            cmp     cx,40       ; If unsigned CX is smaller than 40
            jnl     notless     ;    else do stuff at "abovequal"
            mov     dx,60       ; Then set DX to 20
            jmp     continue    ; Finished
notless:    mov     dx,10       ; Else actions
continue:
```

The then-if-else format shown in Example 3 is often more efficient. Do the work for the most likely case, then compare for the opposite condition. If the condition is true, you are finished.

■ Example 3

```
; DX is 30, unless CX is less than -20, then make DX 20

            mov     dx,30       ; Then set DX to 30
            cmp     cx,-20      ; If signed CX is not greater than -20
            jge     greatequal  ;    Then done
            mov     dx,20       ; Else set DX to zero
greatequal:
```

This example avoids the unconditional jump used in Examples 1 and 2, and is thus faster even if the less likely condition is true.

17.1.1.2 Testing Bits and Jumping

Like the **CMP** instruction, the **TEST** instruction is designed to test for conditional jumps. However, specific bits are compared rather than entire operands.

■ Syntax

TEST *register/memory,immediate*
TEST *register/memory,register*
TEST *register,register/memory*

The destination operand can be memory or register. The source operand can be immediate, memory, or register. However, the operands cannot both be memory.

Normally one of the operands is a mask in which the bits to be tested are the only bits set. The other operand contains the value to be tested. If all the bits set in the mask are clear in the operand being tested, the zero flag will be set. If any of the flags set in the mask are also set in the operand, the zero flag will be cleared.

The **TEST** instruction is actually exactly the same as the **AND** instruction, except that neither operand is changed. If the result of the operation would be 0, the zero flag is set, but the 0 is not actually written to the destination operand.

You can use the **JZ** and **JNZ** instructions to jump after the test. **JE** and **JNE** are exactly the same and can be used if you find them more mnemonic.

■ Example

```
; If bit 2 or bit 4 is set, then set bit 3

          mov     ax,C3h     ; Load number to be tested    11000011
          test    ax,10100b  ; If 2 or 4 is set        AND 00010100
          jz      continue   ; (Else continue)             --------
          or      ax,1000b   ;    Then set bit 3           00000000
continue:                    ;                        Jump not taken
```

```
; If bit 2 and bit 4 are clear, then set bit 3

           mov     ax,D7h      ; Load number to be tested     11010111
           test    ax,10100b   ; If 2 and 4 are clear    AND 00010100
           jnz     next        ; (Else continue)              --------
           or      ax,1000b    ;    Then set bit 3            00010100
next:                          ; Continue                   Jump taken
```

17.1.1.3 Jumping Based on Flag Status

The **CMP** instruction is the most mnemonic way to test for conditional jumps, but any instruction that changes flags can be used as the test condition. The conditional-jump instructions listed below enable you to jump based on the condition of flags rather than relationships of operands. Some of these instructions are the same as instructions in Table 17.1.

Instruction	Description
JO	Jump if the overflow flag is set
JNO	Jump if the overflow flag is clear
JC	Jump if the carry flag is set
JNC	Jump if the carry flag is clear
JZ	Jump if the zero flag is set
JNZ	Jump if the zero flag is clear
JS	Jump if the sign flag is set
JNS	Jump if the sign flag is clear
JP	Jump if the parity flag is set
JNP	Jump if the parity flag is clear
JPE	Jump if parity is even (parity flag set)
JPO	Jump if parity is odd (parity flag clear)
JCXZ	Jump if **CX** is 0

Notice that the **JCXZ** is the only conditional jump that is based on the condition of a register (**CX**) rather than flags.

80386 Processor Only

The 80386 processor has an additional instruction, **JEXCZ**, that tests

ECX for 0.

■ **Example 1**

```
            add     ax,bx       ; Add two values
            jo      overflow    ; If value too large, adjust
            .
            .
            .
overflow:                       ; Adjustment routine here
```

■ **Example 2**

```
            and     ax,ax       ; Add two values
            jnz     go_on       ; If AX is not zero, continue
            .                   ;    else do some task
            .
            .
go_on:
```

17.1.2 Jumping Unconditionally

The **JMP** instruction is used to jump unconditionally to a specified address.

■ **Syntax**

JMP *register/memory*

The operand should contain the address to be jumped to. Unlike conditional jumps, whose target address must be short (within 128 bytes), the target address for unconditional jumps can be near or far. If a conditional jump must be greater than 128 bytes, the construction must be reorganized. This is usually done by reversing the sense of the jump, as shown in Example 1.

■ Example 1

```
          cmp     ax,7          ; If AX is 7 and jump is short
          je      close         ;   then jump close

          cmp     ax,6          ; If AX is 6 and jump is near
          jne     close         ;   then test opposite and skip over
          jmp     distant       ; Now jump
close:                          ; Less than 128 bytes from jump
          .
          .
          .
distant:                        ; More than 128 bytes from jump
```

Unconditional jumps can be used as a form of conditional jump by specifying the address in a register or indirect memory operand. The value of the operand can be calculated at run time, based on user interaction or other factors. You can use indirect memory operands to construct jump tables that work like C **switch** statements, BASIC **ON GOSUB** statements, or Pascal **case** statements.

■ Example 2

```
          .DATA
ctl_tbl   LABEL   WORD
          DW      extended          ; Null key (extended code)
          DW      ctrla             ; Address of CONTROL-A key routine
          DW      ctrlb             ; Address of CONTROL-B key routine
          .
          .
          .
          .CODE
          mov     ah,8h             ; Get a key
          int     21h
          cbw                       ; Convert AL to AX
          mov     bx,ax             ; Copy
          shl     bx,1              ; Convert to address

          jmp     ctl_tbl[bx]       ; Jump to key routine

extended: mov     ah,8h             ; Get second key of extended
          int     21h
          .                         ; Use another jump table
          .                         ;   for extened keys
          .
ctrla:    .                         ; CONTROL-A routine here
          .
          .
          jmp     next

ctrlb:    .                         ; CONTROL-B routine here
          .
          .
          jmp     next
```

```
            .
next:       .                     ; Continue
```

In Example 2, an indirect memory operand points to addresses of routines for handling different keystrokes.

17.2 Setting Bytes Conditionally

■ 80386 Processor Only

■ 80386 only>setting bytes conditionally

The 80386 processor has a new group of instructions for setting bytes conditionally. These instructions test the condition of specified flags, and depending on the result, set a memory operand either to 1 or to 0. They can be used to set byte variables that are used as boolean flags.

■ Syntax

SET*condition register/memory*

Conditional-set instructions test conditions in the same way as conditional-jump instructions, except that instead of jumping if the condition is met, they set a specified byte. For example, **SETZ** is similar to **JZ**, **SETNE** is similar to **JNE**, and so on. See Section 17.1.1 for more information on how flags are tested for conditional jumps.

Conditional-set instructions require one 8-bit operand, which can be either a register or a memory operand. If the condition tested by the instruction is true, the value of the operand is set to 1. Otherwise the value is set to 0.

Conditional set instructions are usually preceeded by a **CMP** or **TEST** instruction, although any instruction that sets flags can be used to test for the condition.

■ Example

```
            .DATA
bigflag     DB      ?           ; Boolean flag
size        DW      ?           ; Size variable to be set at run time
            .CODE
            .
            .                   ; Size is set
            .
; bigflag = size > 1000

            cmp     size,1000   ; Is "size" greater than 1000?
            setg    bigflag     ; If greater, "bigflag" = 1
                                ;   else "bigflag" = 0
```

In the example, the boolean variable `bigflag` is set according to a comparison of two other values. Some languages (such as BASIC) set the result of true relational statements to –1 rather than 1. To be compatible with such compilers, you should negate the value after setting it, as shown below:

```
            neg     bigflag     ; Negate result
```

This statement would be necessary for BASIC, since the expression `BIGFLAG=SIZE>1000` evaluates to –1. It would not be needed for C, since the expression `bigflag=size>1000` evaluates to 1.

17.3 Looping

The 8086-family of processors has several instructions specifically designed for creating loops of repeated instructions. In addition, you can create loops using conditional jumps.

■ Syntax

LOOP *register/memory*
LOOPE *register/memory*
LOOPZ *register/memory*
LOOPNE *register/memory*
LOOPNZ *register/memory*

The **LOOP** instruction is used for loops with a set number of times to go through the loop. For example, it can be used in constructions similar to the "for" loops of C, BASIC, and Pascal or the "do" loops of FORTRAN.

A single operand specifies the address to jump to each time through the loop. The **CX** register is used as a counter for the number of times to go through the loop. On each iteration, **CX** is decremented. When **CX** reaches 0, control passes to the instruction after the loop.

The **LOOPE**, **LOOPZ**, **LOOPNE**, and **LOOPNZ** instructions are used in loops that check for a condition. For example, they can be used in constructions similar to the **while** loops of BASIC, and Pascal, the **repeat** loops of Pascal, or the **do** loops of C.

The **LOOPE** (also called **LOOPZ**) instruction can be thought of as meaning "loop while equal." Similarly, **LOOPNE** (also called **LOOPNZ**) instruction can be thought of as meaning "loop while not equal." A single short memory operand specifies the address to loop to each time through. The **CX** register can specify a maximum number of times to go through the loop. The **CX** register can be set to a number that is out of range if you don't want a maximum count.

80386 Processor Only

The **LOOP** instruction and its variations assume that the loop will take a short jump to a value within 128 bytes of the loop instruction. However, if the loop instruction is in a 32-bit segment, a short jump can be up to 32768 bytes distant. To take advantage of this, you must use **DWORD** to override the default maximum size of the jump:

```
loop    DWORD next          ; Do 32-bit short jump
                            ;   (a signed 16-bit value)
```

■ Example 1

```
; For 0 to 200 do task

          mov     cx,200            ; Set counter
next:     .                         ; Do the task here
          .
          .
          loop    next              ; Do again
                                    ; Continue after loop
```

This loop has the same effect as the following statements:

```
; For 0 to 200, do task

            mov     cx,200              ; Set counter
next:       .                           ; Do the task here
            .
            .
            dec     cx
            or      cx,cx
            jne     next                ; Do again
                                        ; Continue after loop
```

The first version is more efficient as well as easier to understand. However, there are situations in which you must use conditional-jump instructions rather than loop instructions. For example, conditional jumps are required for loops that test more than one condition.

If the counter in **CX** is variable depending on previous instructions, you should use the **JCXZ** instruction to check for 0, as shown in Example 2. Otherwise, if **CX** is 0, it will be incremented in the first iteration and will continue through 65,535 iterations before it reaches 0 again.

■ Example 2

```
; For 0 to CX do task

                                        ; CX counter set previously
            jcxz    done                ; Check for 0
next:       .                           ; Do the task here
            .
            .
            loop    next                ; Do again
done:                                   ; Continue after loop
```

Example 3 illustrates a conditional loop.

■ Example 3

```
; While AX is not 128, do task

            mov     cx,0FFFFh           ; Set count too high to interfere
wend:       .                           ; Do the task here
            .
            .
            cmp     ax,128              ; Is it 128?
            loopne  wend                ; No? Repeat
                                        ; Yes? Continue
```

17.4 Using Procedures

Procedures are units of code that do a specific task. They provide a way of modularizing code so that a task can be accomplished from any point in the program without using the same code in each place. Assembly-language procedures are comparable to functions in C, subprograms and subroutines in BASIC, procedures or functions in Pascal, or routines in FORTRAN.

Two instructions and two directives are usually used in combination to define and use assembly-language procedures. The **CALL** instruction is used to call procedures that have been defined elsewhere. The **RET** instruction is used to return control from a called procedure to the code that called it. The **PROC** and **ENDP** directives normally mark the beginning and end of a procedure definition, as described in Section 17.4.2.

The **CALL** and **RET** instructions use the stack to keep track of the location of the procedure. The **CALL** instruction pushes the calling address onto the stack and then jumps to the starting address of the procedure. The **RET** instruction pops the address pushed by the **CALL** instruction and returns control to the instruction following the call.

Every **CALL** must have a **RET** to restore the stack to its status before the **CALL**. Calls may be nested, but the inside call always returns before the outside call.

17.4.1 Calling Procedures

The **CALL** instruction saves the address following the instruction on the stack and passes control to a specified address.

■ Syntax

CALL *register/memory*

The address is usually specified as a direct memory operands. However, the operand can also be a register or indirect memory operand containing a value calculated at run time. This enables you to write call tables similar to the jump table illustrated in Section 17.1.2.

Calls can be near or far. Near calls push only the offset portion of the calling address. Far calls push both the segment and offset. With **MASM**, the type of call is normally specified in the procedure definition rather than in the procedure call (this does not necessarily apply to other assemblers). If a procedure is defined to be near, then a call to that procedure will be a near call. The size of a call can be given specifically using **NEAR PTR** or **FAR PTR**.

17.4.2 Defining Procedures

Procedures are defined by labeling the start of the procedure and placing a **RET** instruction at the end. There are several variations on this syntax.

■ Syntax 1

label **PROC** ⟦**NEAR** | **FAR**⟧
.
.
.
RET ⟦*immediate*⟧
label **ENDP**

Procedures are normally defined using the **PROC** directive at the start of the procedure and the **ENDP** directive at the end. The **RET** instruction is normally placed immediately before the **ENDP** directive. The size of the **RET** automatically matches the size defined by the **PROC** directive.

■ Syntax 2

*label***:**
.
.
.
RETN ⟦*immediate*⟧

■ **Syntax 3**

label **LABEL FAR**

.
.
.

RETF ⟦*immediate*⟧

Starting with Version 4.5 of **MASM**, the **RET** instruction can be extended to **RETF** (return far) or **RETN** (return near) to override the default size. This enables you to define and use procedures without the **PROC** and **ENDP** directives, as shown in Syntax 2 and Syntax 3 above. However, with this method, the programmer is responsible for making sure the size of the **CALL** matches the size of the **RET**.

The **RET** instruction (and its **RETF** and **RETN** variations) allow an immediate operand that specifies a number of bytes to be added to the value of the **SP** register after the return. This operand can be used to adjust for arguments passed to the procedure before the call, as described in Section 17.4.3.

■ **Example 1**

```
            call    task            ; Call is near because procedure is near
            .                       ; Return comes to here
            .
            .
task        PROC    NEAR            ; Define "task" to be near
            .
            .                       ; Instructions of "task" go here
            .
            ret                     ; Return to instruction after call
task        ENDP                    ; End "task" definition
```

Example 1 shows the recommended way of making calls with **MASM**. Example 2 shows another method that programmers who are used to other assemblers may find more familiar.

■ **Example 2**

```
            call    NEAR PTR task ; Call is declared near
            .                     ; Return comes to here
            .
            .
task:                             ; Procedure begins with near label
            .
            .                     ; Instructions go here
```

```
        .
        retn                    ; Return declared near
```

This method gives more direct control over procedures, but the programmer must make sure that calls have the same size as corresponding returns.

For example, if a call is made with the statement

```
call NEAR PTR task
```

the assembler does a near call. This means that one word (the offset following the calling address) is pushed onto the stack. If the return is made with the statement

```
retf
```

two words are popped off the stack. The first will be the offset, but the second will be whatever happened to be on the stack before the call. Not only will the popped value not make sense, but the stack status will be incorrect. The system will probably crash.

17.4.3 Passing Arguments on the Stack

Procedure arguments can be passed in various ways. For example, values can be passed to a procedure in unused registers or in variables. However, the most common method of passing arguments is on the stack. Microsoft languages have a specific convention for doing this.

The arguments are pushed onto the stack before the call. The procedure can then retrieve and process them. At the end of the procedure, the stack is adjusted to account for the arguments.

Although the method is the same for Microsoft high-level languages, the details vary. For example, in some languages, pointers to the arguments are passed the the procedure, while in others the arguments themselves are passed. The order in which arguments are passed (whether the first argument is pushed first or last) also varies according the language. Finally in some languages the stack is adjusted by the **RET** instruction in the called procedure; in others the code immediately following the **CALL** instruction adjusts the stack.

■ Example 1

```
; C style procedure call and definition

              mov     ax,10       ; Load and
              push    ax          ;   push constant as third argument
              push    arg2        ; Push memory as second argument
              push    cx          ; Push register as first argument
              call    addup       ; Call the procedure
              add     sp,6        ; Destroy the pushed arguments
              .                   ;   (equivalent to three pops)
              .
              .
addup         PROC    NEAR        ; Return address for near call
                                  ;   takes two bytes
              push    bp          ; Save base pointer - takes two bytes
                                  ;   so arguments start at 4th byte
              mov     bp,sp       ; Load stack into base pointer
              mov     ax, [bp+4]  ; Get first argument from
                                  ;   4th byte above pointer
              add     ax, [bp+6]  ; Add second argument from
                                  ;   6th byte above pointer
              add     ax, [bp+8]  ; Add third argument from
                                  ;   8th byte above pointer
              mov     sp,bp       ; Restore stack pointer
              pop     bp          ; Restore base
              ret                 ; Return result in AX
addup         ENDP
```

Example 1 shows a C-style procedure. The method of passing and retrieving arguments is similar for other Microsoft languages. However, note the following points where this convention differs from the conventions for other languages:

- The first argument is pushed last.
- The value of the argument is passed, not its address.
- The stack space taken up by the arguments is restored by the calling code, not by the procedure.

Figure 17.1 shows the stack condition at key points in the process.

```
High memory
  Before "call addup"       After "call addup"        After "move bp,sp"
 |                 |       |                 |       |                 |
 +--------+--------+       +--------+--------+       +--------+--------+
 |   argument 3    |       |   argument 3    |       |   argument 3    | <-BP+8
 +--------+--------+       +--------+--------+       +--------+--------+
 |   argument 2    |       |   argument 2    |       |   argument 2    | <-BP+6
 +--------+--------+       +--------+--------+       +--------+--------+
 |   argument 1    | <-SP  |   argument 1    |       |   argument 1    | <-BP+4
 +--------+--------+       +--------+--------+       +--------+--------+
 |                 |       |  offset of call | <-SP  |  offset of call |
 +--------+--------+       +--------+--------+       +--------+--------+
 |                 |       |                 |       | old value of BP | <-BP/SP
 +--------+--------+       +--------+--------+       +--------+--------+
```

```
|                       |       |                   |       |                   |
Low memory

High memory
  After "pop bp"                After "ret"                 After "add sp,6"
 |                      |      |                   |       |                   | <-SP
 +---------+---------+         +---------+---------+       +---------+---------+
 |   argument 3         |      |   argument 3      |       |   argument 3      |
 +---------+---------+         +---------+---------+       +---------+---------+
 |   argument 2         |      |   argument 2      |       |   argument 2      |
 +---------+---------+         +---------+---------+       +---------+---------+
 |   argument 1         |      |   argument 1      | <-SP  |   argument 1      |
 +---------+---------+         +---------+---------+       +---------+---------+
 |  offset of call      | <-SP |  offset of call   |       |  offset of call   |
 +---------+---------+         +---------+---------+       +---------+---------+
 | old value of BP      |      | old value of BP   |       | old value of BP   |
 +---------+---------+         +---------+---------+       +---------+---------+
 |                      |      |                   |       |                   |
Low memory
```

■ Example 2

```
; FORTRAN style procedure call and definition

            mov     arg1,10          ; Load constant into memory
            push    OFFSET arg1      ;   push address of first argument
            push    OFFSET arg2      ; Push address of second argument
            mov     arg1,cx          ; Load register into memory and
            push    OFFSET arg2      ;   push address of third argument
            call    addup            ; Call the procedure
            .
            .
            .
addup       PROC    FAR        ; Return address for far call
                               ;   takes four bytes
            push    bp         ; Save base pointer - takes two bytes
                               ;   so arguments start at 6th byte
            mov     bp,sp      ; Load stack into base pointer
            mov     bx,[bp+10] ; Get address of first argument from
                               ;   10th byte above pointer
            mov     ax,[bx]    ; Load argument from address
            mov     bx,[bp+8]  ; Get second argument from
                               ;   8th byte above pointer
            add     ax,[bx]    ; Add argument from address
            mov     bx,[bp+6]  ; Add third argument from
                               ;   6th byte above pointer
            add     ax,[bx]    ; Add argument from address
            mov     sp,bp      ; Restore stack pointer
            pop     bp         ; Restore base
            ret     6          ; Return result in AX and pop
addup       ENDP               ;   six bytes to adjust stack
```

Example 2 shows a FORTRAN-style procedure. The convention is similar for Pascal and BASIC. Note the following points where this convention differs from the conventions for C:

- The first argument is pushed first.
- The address of the argument is passed, not its value. If the argument is not already in memory, it must be loaded into memory.
- The stack space taken up by the arguments is restored by a **RET** instruction at the end of the procedure.

17.4.4 Using Local Variables

In high-level languages, local variables are variables that are only known within a procedure. In Microsoft languages, these variables are stored on the stack. Assembly-language programs can use the same concept. These variables should not be confused with labels or variable names that are local to a module, as described in Chapter 8, "Creating Programs from Multiple Modules."

Local variables are created by saving stack space for the variable at the start of the procedure. The variable can then be accessed by position in the stack. At the end of the procedure, the stack space is returned.

■ Example

```
                push    ax          ; Push one argument
                call    task        ; Call
                add     sp,2        ; Destroy one argument

arg             EQU     <[bp+6]>
loc             EQU     <[bp-2]>    ; Name of variable
task            PROC    NEAR
                push    bp          ; Save base pointer
                mov     bp,sp       ; Load stack into base pointer
                sub     sp,2        ; Save two bytes for local variable
                .
                .
                .
                add     ax,loc      ; Add local variable
                sub     arg,bx      ; Subtract local from argument
                .                   ; Use "loc" and "arg" in other operations
                .
                .
                mov     sp,bp       ; Restore stack pointer
                                    ;   (also adjust for stack variable)
                pop     bp          ; Restore base
                ret                 ; Return result in AX and pop
task            ENDP                ;   two bytes to adjust stack
```

In this example, two bytes are subtracted from the **SP** register to make room for a local word variable. This variable can then be accessed as `[bp-2]`. In the example, this value is given the name `loc` with a text equate. Figure 17.2 shows the state of the stack at key points in this process.

```
High memory
  Before "call task"              After "call task"               After "move bp,sp"
 |                 |              |                 |             |                 |
 +--------+--------+              +--------+--------+             +--------+--------+
 |    argument     | <-SP         |    argument     |             |    argument     | <- BP+4
 +--------+--------+              +--------+--------+             +--------+--------+
 |                 |              |  offset of call | <-SP        |  offset of call |
 +--------+--------+              +--------+--------+             +--------+--------+
 |                 |              |                 |             | old value of BP | <- BP/SP
 +--------+--------+              +--------+--------+             +--------+--------+
 |                 |              |                 |             |                 |
 +--------+--------+              +--------+--------+             +--------+--------+
 |                 |              |                 |             |                 |
Low memory

High memory
  After "sub bp,2"                As used in procedure            After "pop bp"
 |                 |              |                 |             |                 | <-SP
 +--------+--------+              +--------+--------+             +--------+--------+
 |    argument     | <-BP+4       |    argument     | <-BP+6      |    argument     |
 +--------+--------+              +--------+--------+   (arg)     +--------+--------+
 |  offset of call |              |  offset of call |             |  offset of call | <-BP/SP
 +--------+--------+              +--------+--------+             +--------+--------+
 | old value of BP | <-BP         | old value of BP | <-BP        | old value of BP |
 +--------+--------+              +--------+--------+             +--------+--------+
 | space for local | <-SP         |      local      | <-BP-2      |                 |
 +--------+--------+              +--------+--------+   (loc)     +--------+--------+
 |                 |              |                 |             |                 |
Low memory
```

Note

Local variables created in assembler routines cannot be accessed with the CodeView debugger.

17.4.5 Setting Up Stack Frames

■ 80186-80386 Processors Only

Starting with the 80186 processor, the **ENTER** and **LEAVE** instructions are provided for setting up a stack frame. These instructions do exactly the same thing as the multiple instructions at the start and end of procedures in the Microsoft calling conventions (see the examples in Section 17.4.3).

The **ENTER** instruction takes two immediate operands. The first operand (with a value up to 16 bits) specifies how many bytes to reserve for local variables. The second operand (with a value of up to 32 bits) specifies the level of nesting for the procedure. This operand should always be 0 when writing procedures for C, BASIC, and FORTRAN. The nesting level is only used for Pascal and other languages that enable procedures to access the local variables of calling procedures.

The **LEAVE** instruction reverses the effect of the last **ENTER** instruction.

■ Example 1

```
task        PROC    NEAR
            enter   6,0         ; Set stack frame and reserve 6
            .                   ;   bytes for local variables
            .                   ; Do task here
            .
            leave               ; Restore stack frame
            ret                 ; Return
task        ENDP
```

Example 1 has the exact effect as the code in Example 2.

■ Example 2

```
task        PROC    NEAR
            push    bp          ; Save base pointer
            mov     bp,sp       ; Load stack into base pointer
            sub     sp,6        ; Reserve 6 bytes for local variables
            .
            .                   ; Do task here
            .
            mov     sp,bp       ; Restore stack pointer
            pop     bp          ; Restore base
            ret                 ; Return
task        ENDP
```

The code in Example 1 takes fewer bytes, but is slightly slower. See the Microsoft CodeView and Utilities manual for exact comparisons of size and timing.

17.5 Using Interrupts

Interrupts are a special form of routine. They differ from procedures in two important ways. They can be called by number instead of by address, and they can be initiated by hardware devices as well as by software. Hardware interrupts are called automatically whenever certain events occur in the hardware.

Interrupts can have any number from 0 to 255. Most of the interrupts with lower numbers are reserved for use the processor, DOS, or the BIOS.

The programmer can use interrupts in two ways: First, existing interrupts can be called with the **INT** instruction. Second, interrupts can be defined or redefined to be called later. For example, an interrupt that is called automatically by a hardware device can be redefined so that its action is different.

The processor defines several internal interrupts. The two that are sometimes used by applications programmers are listed below:

Interrupt	Description
0h	Divide overflow. Called automatically when the quotient of a divide operation is too large for the source operand or when a divide by zero is attempted.
04h	Overflow. Called automatically when an overflow occurs.

Interrupt 21h is the current method of using DOS functions. To call a function, place the function number in **AH**, put arguments in registers as appropriate, then call the interrupt. For complete documentation of DOS functions, see the *Microsoft MS-DOS Programmer's Reference* or one of the many other books on DOS functions.

DOS has several other interrupts, but they should not normally be called. Some (such as 20h and 27h) have been replaced by DOS functions. Others are used internally by DOS.

Note

Future multitasking versions of DOS will not use interrupt 21h to call DOS. The Application Program Interface (API) will be used instead. This is the system currently used for Microsoft Windows applications.

The BIOS of most computers can also be accessed by interrupts. BIOS interrupts are not documented here, since they vary for different computers. See the technical reference documents for your hardware.

17.5.1 Calling Interrupts

Interrupts are called with the **INT** instruction.

- **Syntax**

INT *interruptnumber*
INTO

The **INT** instruction takes an immediate operand with a value between 0 and 255.

When calling DOS and BIOS interrupts, a function number is usually placed in the **AH** register. Other registers may be used to pass arguments to functions. Some interrupts and functions return values in certain registers. Register use varies for each interrupt.

When the instruction is called, the processor takes the following steps:

1. Looks up the address of the interrupt routine in the interrupt descriptor table. In real mode, this table starts at the lowest point in memory (segment 0, offset 0) and consists of two words (a segment and offset) for each interrupt.
2. Pushes the flags register, the current code segment (**CS**), and the current instruction pointer (**IP**).
3. Clears the trap (**TF**) and interrupt enable (**IF**) flags.
4. Jumps to the address of the interrupt routine, as specified in the interrupt description table.
5. Executes the code of the interrupt routine until it encounters an **IRET** instruction.
6. Pops the instruction pointer, code segment, and flags.

Figure 17.3 shows the status of the stack immediately after the **INT** instruction has been executed.

```
  Before INT                 Inside INT routine         After IRET

Interrupt Descriptor Table

 +--------+--------+
 |   INT segment   | >-----+
 +--------+--------+       |
 |   INT offset    | >-----+
 +--------+--------+------  | ---- Segment = 0
                           |      Offset  = INT number * 4
                           v
                           |    Flags Status
 +--------+--------+       | +--------+--------+        +--------+--------+
 |  program flags  | >---+ | |  program flags  |    +-->|  program flags  |
 +--------+--------+     | | +--------+--------+    |   +--------+--------+
                         | | (changes in routine)   |
                         v v                        ^
                         | |                        |
                         | |   Code Segment         |
 +--------+--------+     | | +--------+--------+    |   +--------+--------+
 |    program CS   | >---+ +>|new CS from table|    +-->|    program CS   |
 +--------+--------+     | | +--------+--------+    |   +--------+--------+
                         | | (changes in routine)   |
                         v |                        ^
                         | |                        |
                         | | Instruction Pointer    |
 +--------+--------+     | | +--------+--------+    |   +--------+--------+
 |    program IP   | >---+ +>|new IP from table|    +-->|    program IP   |
 +--------+--------+     |   +--------+--------+    |   +--------+--------+
                         |   (changes in routine)   |
                         v                          ^
                         |                          |
High memory              |         Stack            |
 |                 |     |   |                 |    |   |                 |
 +--------+--------+--SP |   +--------+--------+    |   +--------+--------+--SP
 |                 |     +-->| previous flags  |>---+   | previous flags  |
 +--------+--------+     |   +--------+--------+    |   +--------+--------+
 |                 |     +-->|   previous CS   |>---+   |   previous CS   |
 +--------+--------+     |   +--------+--------+    |   +--------+--------+
 |                 |     +-->|   previous IP   |>---+   |   previous IP   |
 +--------+--------+         +--------+--------+--SP    +--------+--------+
 |                 |         |                 |        |                 |
Low memory
```

The **INTO** (Interrupt on Overflow) instruction is a variation of the **INT** instruction. It calls interrupt 04h whenever an instruction sets the overflow flag. By default, the interrupt routine simply consists of an **IRET** so that it returns without doing anything. However, you can write your own overflow interrupt routine. This is not usually necessary since you can simply use the **JO** (Jump on Overflow) instruction to jump to an overflow routine.

The **CLI** (Clear Interrupt Flag) and **STI** (Set Interrupt Flag) instructions can be used to turn interrupts on or off. You can use **CLI** to turn interrupt processing off so that an important routine cannot be stopped by a hardware interrupt. After the routine has finished, use **STI** to turn interrupt processing back on.

■ Example 1

```
; DOS call (Display String)

            mov     ah,09h              ; Load function number
            mov     dx,OFFSET string    ; Load argument
            int     21h                 ; Call DOS
```

■ Example 2

```
; BIOS call (Read Cursor Position and Size)

            xor     ah,ah               ; Load function number 0 in AH
            int     16h                 ; Call BIOS
                                        ; Return scan code in AH
                                        ; Return ascii code in AL
```

Example 2 is a BIOS call that works on IBM Personal Computers and IBM-compatible computers. See the reference manuals for your hardware for complete information on BIOS calls.

17.5.2 Defining and Redefining Interrupts

You can write your own interrupt routines, either to replace an existing routine, or to use an undefined interrupt number.

■ Syntax

label **PROC FAR**
.
.
.
IRET
label **ENDP**

An interrupt routine can be written like a procedure using the **PROC** and **ENDP** directives. The only differences are that the routine should always be defined as far and it should be terminated by an **IRET** instruction instead of a **RET** instruction.

Your program should replace the address in the interrupt descriptor table with the address of your routine. DOS calls are provided for this task. It is usually a good idea to save the old address and restore it before your program ends.

Interrupt routines you may want to replace include the processor's divide-overflow (0h) and overflow (04h) interrupts. You can also replace DOS interrupts such as the critical-error (24h) and CONTROL-C (23h) handlers. Interrupt routines can be part of device drivers. Writing interrupt routines is usually a systems tasks. The example below illustrates a simple routine. For complete information see the *Microsoft MS-DOS Programmer's Guide* or one of the other reference books on DOS.

80386 Processor Only

The **INT** instruction automatically pushes the a 32-bit instruction pointer for 32-bit segments or a 16-bit instruction pointer for 16-bit segments. However, the **IRET** instruction always pops a 16-bit instruction pointer before returning. To pop a 32-bit instruction pointer, you must append the letter D (for doubleword) to the instruction (**IRETD**).

■ Example

```
            .DATA
message     DB      "Illegal division; try again",13,10,"$"
vector      DW      ?,?
            .CODE

diverror    PROC    FAR                   ; Enable interrupts
                                          ;   (disabled by int instruction)
            pusha                         ; Save the registers
            mov     ah,09h                ; Display message
            mov     dx,OFFSET message
            int     21h
            popa                          ; Restore registers
            iret
diverror    ENDP

            mov     ax,3500h              ; Load interrupt number 0
            int     21h                   ;   and DOS function to get vector
            mov     ax,es                 ; Save segment
            mov     vector[0],ax
```

```
mov     vector[2],bx       ;   and offset

lds     dx,diverror        ; Load address of new routine
mov     ax,2500h           ; Load interrupt number 0
int     21h                ;   and DOS function to set vector
.
.
.
div     bx                 ; If overflow, our routine is used
.
.
.
lds     dx,vector0         ; Load original interrupt address
mov     ax,2500h           ; Load interrupt number 0
int     21h                ;   and DOS function to set vector
mov     ax,4C00h           ; Load return code 0
int     21h                ;   and DOS function to terminate
```

Notice that the original interrupt number is restored before the program terminates. For brevity, the example uses the **PUSHA** and **POPA** instructions, which are not available on the 8086 and 8088 processors. You would need to push and pop each register individually on these processors.

Chapter 18
Processing Strings

The 8086-family processors have a full set of instructions for manipulating strings. In discussing these instructions, the term "string" refers not only to the common definition of a string—a sequence of bytes containing characters—but to any sequence of bytes or words (or doublewords on the 80386).

The following instructions are provided for 8086-family string functions:

Instruction	Purpose
MOVS	Move string from one location to another
SCAS	Scan string for specified values
CMPS	Compare values in one string with values in another
LODS	Load values from a string to accumulator register
STOS	Store values from accumulator register to a string
INS	Transfers values from a port to memory
OUTS	Transfers values from memory to a port

All these instruction use registers in the same way and have a similar syntax. Most are used with the repeat instruction prefixes: **REP**, **REPE**, **REPNE**, **REPZ**, and **REPNZ**.

This chapter first explains the general format for string instructions, then tells how to use each instruction.

18.1 Setting Up String Operations

The string instructions all work in a similar way. Once you understand the general procedure, it is easy to adapt the format for a particular string operation. The steps are listed below:

1. Make sure the direction flag indicates the direction in which you want the string to be processed. If the direction flag (**DF**) is clear, the string will be processed up (from low addresses to high addresses). If the direction flag is set, the string will be processed down (from high addresses to low addresses). The **CLD** instruction clears the flag, while **STD** sets it. The direction flag will normally be cleared if your program has not changed it.

2. Load the number of iterations for the string instruction into the **CX** register. For example, if you want to process a 100-byte string, load 100. If string instruction will be terminated conditionally, load the maximum number of iterations that could be done without an error.

3. Load the starting offset address of the source string into **DS:SI** and the starting address of the destination string into **ES:DI**. Some string instructions take only a destination or only a source, as shown in Table 18.1. Normally the segment address of the source string should be **DS**, but you can use a segment override with the string instruction to specify a different segment. You cannot override the segment address for the destination string. Therefore you may need to change the value of **ES**.

4. Choose the appropriate repeat prefix instruction. Table 18.1 shows the repeat prefixes that can be used with each instruction.

5. Put the appropriate string instruction immediately after the repeat prefix (on the same line).

String instructions have two basic forms, as shown below:

■ Syntax 1

⟦*repeatprefix*⟧ *stringinstruction* ⟦*destination*,⟧⟦⟦*segmentregister*:⟧*source*⟧

The instruction can be given with the source and/or destination as operands. In this case, the size of the operand or operands indicates the size of the objects to be processed by the string. Note that the operands only specify the size. The actual values to be worked on are the ones pointed to by **DS:SI** and/or **ES:DI**. No error will be generated if the operand is not the same as the actual source or destination. One important advantage of this syntax is that the source operand can have a segment override. The destination operand is always relative to **ES** and cannot be overridden.

■ Syntax 2

⟦*repeatprefix*⟧ *stringinstruction***B**
⟦*repeatprefix*⟧ *stringinstruction***W**
⟦*repeatprefix*⟧ *stringinstruction***D** ; 80386 processor only

String instruction can be given with no operands, but with the letter B or

W appended to indicate bytes or words. (D indicates double words on the 80836.)

For example, **MOVS** can be given with byte operands to move bytes or with word operands to move words. Or **MOVSB** can be given with no operands to move bytes or **MOVSW** can be given with no operands to move words.

The repeat prefix can be one of the following instructions:

Instruction	Meaning
REP	Repeat for a specified number of iterations. The number is given in **CX**.
REPE or **REPZ**	Repeat while equal. The maximum number of iterations should be specified in **CX**.
REPNE or **REPNZ**	Repeat while not equal. The maximum number of iterations should be specified in **CX**.

REPE is the same as **REPZ**, and **REPNE** is the same as **REPNZ**. You can use whichever name you find more mnemonic. For brevity, the prefixes ending with E are used in syntax listings and tables in the rest of this chapter.

Table 18.1 lists each string instruction with the type of repeat prefix it uses and whether it works on a source, a destination, or both.

Table 18.1

Requirements for String Instructions

Instruction	Repeat Prefix	Source/Destination	Register Pair
MOVS	**REP**	Both	**DS:SI** and **ES:DI**
SCAS	**REPE/REPNE**	Destination	**ES:DI**
CMPS	**REPE/REPNE**	Both	**DS:SI** and **ES:DI**
LODS	None	Source	**DS:SI**
STOS	**REP**	Destination	**ES:DI**
INS	**REP**	Destination	**ES:DI**
OUTS	**REP**	Source	**DS:SI**

At run time, a string instruction preceded by a repeat sequence causes the processor to take the following steps:

1. Check the **CX** register and exit from the string instruction if **CX** is 0.
2. Perform the string operation once.
3. Increase **SI** and/or **DI** if the direction flag is clear. Decrease **SI** and/or **DI** if the direction flag is set. The amount of increase or decrease is one for byte operations, two for word operations, or four for doubleword operations (80386 only).
4. Decrement **CX** (no flags are modified).
5. If the string instruction is **SCAS** or **CMPS**, check the zero flag and exit if the repeat condition is false (if the flag is set with **REPE** or **REPZ** or if it is clear with **REPNE** or **REPNZ**).
6. Go to the next iteration.

Although string instructions (except **LODS**) are usually used with repeat prefixes, they can also be used by themselves. In this case, the **SI** and/or **DI** registers are adjusted as specified by the direction flag and the size of operands. However, the programmer is responsible for decrementing the **CX** register and setting up a loop for the repeated action.

18.2 Moving Strings

The **MOVS** instruction is used to move data from one area of memory to another.

■ **Syntax**

⟦**REP**⟧ **MOVS** *destination*,⟦*segmentregister*:⟧*source*
⟦**REP**⟧ **MOVSB**
⟦**REP**⟧ **MOVSW**
⟦**REP**⟧ **MOVSD** ; 80386 processor only

To move the data, load the count and the source and destination addresses into the appropriate registers, as discussed in Section 18.1. Then use the **REP** instruction with the **MOVS** instruction.

■ Example 1

```
        .MODEL  small
        .DATA
source  DB      10 DUP ('0123456789')
destin  DB      100 DUP (?)
        .CODE
        mov     ax,DGROUP          ; Load same segment
        mov     ds:ax              ;   to both DS
        mov     es:ax                  and ES
        .
        .
        .
        cld                        ; Work upward
        mov     cx,100             ; Set iteration count to 100
        mov     si,OFFSET source   ; Load address of source
        les     di,destin          ; Load address of destination
        rep     movsb              ; Move 100 bytes
```

Example 1 shows how to move a string using string instructions. For comparison, Example 2 shows a much less efficient way of doing the same operation without string instructions.

■ Example 2

```
        .MODEL  small
        .DATA
source  DB      10 DUP ('0123456789')
destin  DB      100 DUP (?)
        .CODE
        .                          ; Assume ES = DS
        .
        .
        mov     cx,100             ; Set iteration count to 100
        mov     si,OFFSET source   ; Load offset of source
        mov     di,OFFSET destin   ; Load offset of destination
repeat: mov     ax,es:[di]         ; Get a byte from source
        mov     [si],ax            ; Put it in destination
        inc     si                 ; Increment source pointer
        inc     di                 ; Increment destination pointer
        dec     cx                 ; Decrement count
        loop    repeat             ; Do it again
```

Both instructions illustrate moving byte strings in a small model program where **DS** already points to the segment containing the variables. In such programs, **ES** can be set to the same value as **DS**.

There are several variations on this. If the source string was not in the current data segment, you could load the address using the following statement:

```
        lds     si,source          ; Load into DS:SI
```

Another option would be to use the **MOVS** instruction with operands and give a segment override on the source operand. For example, you could use the following statement if **ES** pointed to both the source and the destination strings:

```
          rep     movs destin,es:source
```

It is sometimes faster to move a string of bytes as words (or as doublewords on the 80386). You must adjust for any odd bytes, as shown in Example 3. Assume the source and destination are already loaded.

■ Example 3

```
          mov     cx,count              ; Load count
          shr     cx,1                  ; Divide by 2 (carry will be set
                                        ;   if count is odd)
          rep     movsw                 ; Move words
          rcl     cx,1                  ; If odd, make CX 1
          rep     movsb                 ; Move odd byte if there
is one
```

18.3 Searching Strings

The **SCAS** instruction is used to scan a string for a specified value.

■ Syntax

[[**REPE** | **REPNE**]] **SCAS** *destination*
[[**REPE** | **REPNE**]] **SCASB**
[[**REPE** | **REPNE**]] **SCASW**
[[**REPE** | **REPNE**]] **SCASD** ; 80386 processor only

This instruction only works on a destination string, which must be pointed to by **ES:DI**. The value to scan for must be in the accumulator register—**AL** for bytes, **AX** for words, or **EAX** (80386 only) for doublewords.

The **SCAS** instruction works by comparing the value pointed to by **DI** to the value in the accumulator. If the values are the same, the zero flag is set. Thus the instruction only makes sense when used with one of the repeat prefixes that checks the zero flag.

If you want to search for the first occurrence of a specified value, use the **REPNE** or **REPNZ** instruction. If the value is found, **ES:DI** will point to its position in the string.

If you want to search for the first value that does not have a specified value, use **REPE** or **REPZ**. If the value is found, **ES:DI** will point to the position after the first nonmatching value. You can decrement **DI** to make it point to the first nonmatching value.

If the value is not found, the **CX** register will contain 0. You can use the **JCXZ** instruction to handle cases where the value is not found.

■ Example

```
            .DATA
string      DB      'The quick brown fox jumps over the lazy dog"
lstring     EQU     $-string
            .CODE
            cld                             ; Work upward
            mov     cx,lstring              ; Load length of string
            les     di,string               ; Load address of string
            mov     al,'z'                  ; Load character to find
            repne   scasb                   ; Search
            jcxz    notfound                ; CX is 0 if not found
            .
            .                               ; ES:SI points to first 'z'
            .
notfound:                                   ; Special case for not found
```

This example assumes that **ES** is not the same as **DS**. The **LES** instruction is used to load the far address of the string into **ES:DI**.

18.4 Comparing Strings

The **CMPS** instruction is used to compare two strings, and point to the address where a match or nonmatch occurs.

■ Syntax

⟦REPE | REPNE⟧ CMPS *destination,*⟦*segmentregister*:⟧*source*
⟦REPE | REPNE⟧ CMPSD
⟦REPE | REPNE⟧ CMPSW
⟦REPE | REPNE⟧ CMPSD ; 80386 processor only

The count and the addresses of the strings are loaded into registers as described in Section 18.1. Either string can be considered the destination or source string.

The **CMPS** instruction works by comparing in turn each value pointed to by **DI** to the value pointed to by **SI**. If the values are the same, the zero flag is set. Thus the instruction only makes sense when used with one of the repeat prefixes that checks the zero flag.

If you want to search for the first match between the strings, use the **REPNE** or **REPNZ** instruction. If a match is found **ES**:**DI** and **DS**:**SI** will point to the position of the match in the respective strings.

If you want to search for a nonmatch, use **REPE** or **REPZ**. If a nonmatch is found, **ES**:**DI** and **DS**:**SI** will point to the position after the nonmatch in the respective strings. You can decrement **DI** or **SI** to point to the nonmatch.

If the specified condition (match or nonmatch) never occurs, the **CX** register will contain zero. You can use the **JCXZ** instruction to handle cases where the entire string is processed.

■ Example

```
            .MODEL   large
            .DATA
string1     DB       'The quick brown fox jumps over the lazy dog"
            .FARDATA
string2     DB       'The quick brown dog jumps over the lazy fox"
lstring     EQU      $-string2
            .CODE
            mov      ax,DGROUP            ; Load DGROUP
            mov      ds,ax                ;   into DS
            mov      ax,SEG string2       ; Load far data segment
            mov      es,ax                ;   into ES
            .
            .
            .
            cld                           ; Work upward
            mov      cx,lstring           ; Load length of string
            mov      di,OFFSET string1    ; Load offset of string1
            mov      si,OFFSET string2    ; Load offset of string2
            repe     cmpsb                ; Compare
            jcxz     notfound             ; CX is 0 if no match
            .
            .                             ; ES:DI points to match in string1
            .                             ; DS:SI points to match in string2
notfound:                                 ; Special case for no match
```

This example assumes that the strings are in different segments. Each segment must be initialized to the segment register.

18.5 Filling Strings

The **STOS** instruction is used to store a specified value in each position of a string.

■ Syntax

[REP] STOS *destination*
[REP] STOSB
[REP] STOSW
[REP] STOSD ; 80386 processor only

The string is considered the destination, so it must be pointed to by **ES:DI**. The length and address of the string must be loaded into registers, as described in Section 18.1. The value to store must be in the accumulator register—**AL** for bytes, **AX** for words, or **EAX** (80386 only) for doublewords.

For each iteration specified by the **REP** instruction prefix, the value in the accumulator is loaded into the string.

■ Example

```
                .MODEL  small
                .DATA
destin          DB      100 DUP ?
                .CODE
                .                           ; Assume ES = DS
                .
                .
                cld                         ; Work upward
                mov     ax,'aa'             ; Load character to fill
                mov     cx,50               ; Load length of string
                mov     di,OFFSET destin    ; Load address of destination
                rep     stosw               ; Store 'a' into array
```

This example loads 100 bytes containing the character "a." Notice that this is done by storing 50 words rather than 100 bytes. This makes the code faster by reducing the number of iterations. You would have to adjust for the last byte if you wanted to fill an odd number of bytes.

18.6 Loading Values from Strings

The **LODS** instruction is used to load a value from a string into a register.

■ Syntax

LODS [[*segmentregister*:]]*source*
LODSB
LODSW
LODSD ; 80386 processor only

The string is considered the source, so it must be pointed to by **DS:SI**. The value is always loaded from the string into the accumulator register—**AL** for bytes, **AX** for words, or **EAX** (80386 only) for doublewords.

Unlike other string instructions, **LODS** is not normally used with a repeat prefix, since there is no reason to repeatedly move a value to a register. However, **LODS** does adjust the **DI** register as specified by the direction flag and the size of operands. The programmer must code the instructions to use the value after it is loaded.

■ Example 1

```
        .DATA
string  DB      0,1,2,3,4,5,6,7,8,9
        .CODE
        cld                             ; Work upward
        mov     cx,10                   ; Load length of string
        mov     si,OFFSET string        ; Load offset of source
get:    lodsb                           ; Get a character
        dec     cx                      ; Decrement count
        add     al,48                   ; Convert to ASCII
        mov     dl,al                   ; Move to DL
        int     21h                     ; Call DOS to display character
        loop    get                     ; Repeat
```

Example 1 loads and processes a string of bytes.

■ Example 2

```
            .DATA
args        EQU     82h
lbuffer     EQU     80
buffer      DB      lstring DUP(?)      ; Create buffer for argument string
            .CODE
            mov     ax,DGROUP           ; Initialize DS
            mov     ds,ax
                                        ; On start-up ES points to PSP
            cld                         ; Work upward
            mov     cx,lbuffer          ; Load length of string
            mov     si,OFFSET buffer    ; Load offset of source
            mov     di,args             ; Load position of argument string
get:        lodsb                       ; Get a character
            dec     cx                  ; Decrement count
            cmp     al,97               ; Is it high enough to be upper?
            jb      noway               ; No? Check
            cmp     al,122              ; Is it low enough to be letter?
            ja      noway
            sub     al,32               ; Yes! Convert to uppercase
noway:      stosb
            loop    get                 ; Repeat
```

Example 2 copies the command arguments from position 82h in the DOS Program Segment Prefix (PSP) while converting them to uppercase. See the *Microsoft MS-DOS Programmer's Reference* or one of the many other books on DOS for information about the PSP. Notice that both **LODSB** and **STOSB** are used without repeat prefixes.

18.7 Transferring Strings to and from Ports

The **INS** instruction reads a string from a port to memory, while the **OUTS** instruction writes a string from memory to a port.

■ Syntax

OUTS DX,⟦*segmentregister:*⟧*source*
OUTSB
OUTSW
OUTSD ; 80386 processor only
INS *destination*,**DX**
INSB
INSW
INSD ; 80386 processor only

The **INS** and **OUTS** instructions require that the number of the port be in **DX**. The port cannot be specified as an immediate value, as it can be with **IN** and **OUT**.

To move the data, load the count into **CX**. The string to be transferred by **INS** is considered the destination string, so it must be pointed to by **ES**:**DI**. The string to be transferred by **OUTS** is considered the source string, so it must be pointed to by **DS**:**SI**.

If you specify the source or destination as an operand, **DX** must be specified. Otherwise **DX** is assumed and should be omitted.

If you need to process the string as it is transferred (for example, to check for the end of a null-terminated string), you must set up the loop yourself instead of using the **REP** instruction prefix.

■ Example

```
            .DATA
count       EQU     100
buffer      DB      count DUP (?)
inport      DW      ?
            .CODE
            .                           ; Assume ES = DS
            .
            .
            cld                         ; Work upward
            mov     cx,count            ; Load length to transfer
            mov     di,OFFSET buffer    ; Load address of destination
            mov     dx,inport           ; Load port number
            rep     insb                ; Transfer the string
                                        ;    from port to buffer
```

Chapter 19
Calculating with a Math Coprocessor

The 8087-family coprocessors are used to do fast mathematical calculations. When used with real numbers, packed BCD numbers, or long integers, they can do calculations many times faster than the same operations done with 8086-family processors.

This chapter explains how to use the 8087-family processors to transfer and process data. The approach is from an applications standpoint. Features that would be used by systems programmers (such the flags used when writing exception handlers) are not explained. The chapter is intended as a reference, not a tutorial.

Note

> This manual does not attempt to explain the mathematical concepts involved in using certain coprocessor features. It assumes that you would not need to use the feature unless you understood the mathematics involved. For example, you need to understand logarithms to use the **FYL2X** and **FYL2XP1** instructions.

19.1 Coprocessor Architecture

The math coprocessor works simultaneously with the main processor. However, since the coprocessor cannot handle input or output, most data originates in the main processor.

The main processor and the coprocessor each have their own registers, which are completely separate and inaccessible to each other. They exchange data through memory, since it is available to both.

Using the coprocessor ordinarily involves three steps:

1. Load data from memory to coprocessor registers.
2. Process the data.
3. Store the data from coprocessor registers back to memory.

Step 2, processing the data, can occur while the main processor is handling other tasks. Steps 1 and 3 must be coordinated with the main processor so that the processor and coprocessor do not try to access the same memory at the same time, as explained in Section 19.3.

19.1.1 Coprocessor Data Registers

The 8087-family coprocessors have eight 80-bit data registers. Unlike 8086-family registers, the coprocessor data registers are organized as a stack. As data is pushed into the top register, previous data items move into higher-numbered registers. Register 0 is the top of the stack while register 7 is the bottom. The syntax for specifying registers is shown below:

ST⟦(*number*)⟧

The *number* must be between 0 and 7. If *number* is omitted, register 0 (top of stack) is assumed.

All coprocessor data is stored in registers in the temporary real format. This is the 10-byte IEEE format described in Section 6.2.1.5. The registers and the register format is shown in Figure 19.1.

```
              | Sign
Register      v    Exponent                      Mantissa
             +-+-----------------+-------------------------------+
ST           | |                 |                               |
             +-+-----------------+-------------------------------+
ST(1)        | |                 |                               |
             +-+-----------------+-------------------------------+
ST(2)        | |                 |                               |
             +-+-----------------+-------------------------------+
ST(3)        | |                 |                               |
             +-+-----------------+-------------------------------+
ST(4)        | |                 |                               |
             +-+-----------------+-------------------------------+
ST(5)        | |                 |                               |
             +-+-----------------+-------------------------------+
ST(6)        | |                 |                               |
             +-+-----------------+-------------------------------+
ST(7)        | |                 |                               |
             +-+-----------------+-------------------------------+
            79  78             64 63                              0
```

Internally, all calculations are done on numbers of the same type. Since temporary real numbers have the greatest precision, lower precison numbers are guaranteed not to lose precision as a result of calculations. The instructions that transfer values between the main processor and the coprocessor automatically convert numbers to and from the temporary real format.

19.1.2 Coprocessor Control Registers

The 8087-family coprocessors have seven 16-bit control registers. The most useful control registers are made up of bit fields or flags. Some flags control coprocessor operations, while others maintain the current status of the coprocessor. In this sense, they are much like the 8086-family flags registers.

You do not need to understand these registers to do most coprocessor operations. Control flags are set by default to the values appropriate for most programs. Status registers report errors and exceptions, but the coprocessor already has a default system for handling exceptions. Applications programmers can usually accept the defaults. Systems programmers may want to use status and control registers when writing exception handlers, but such problems are beyond the scope of this manual.

Figure 19.2 shows the overall layout of the control registers. The format of each the registers is not shown, since these registers are generally of use only to systems programmers. The exception is the condition-code bits of the status register. These bits are explained in Section 19.6.1.

```
                        +-----------------------+
Control word            |                       |
                        +-----------------------+
Status word             |                       |
                        +-----------------------+
Tag word                |                       |
                        +-----------------------+
                       /|                       |
Instruction pointer   < +-----------------------+
                       \|                       |
                        +-----------------------+
                       /|                       |
Operand pointer       < +-----------------------+
                       \|                       |
                        +-----------------------+
```

19.2 Using Coprocessor Instructions

Coprocessor instructions are readily recognizable because, unlike all 8086-family instruction mnemonics, they always start with the letter F.

Most coprocessor instructions have two operands, but in many cases one or both operands are implied. Often, one operand can be a memory operand; in this case the other operand is always implied as the stack-top register. Coprocessor instructions can never have immediate operands, and with the exception of the **FSTSW** instruction (see Section 19.4.2), they cannot have processor registers as operands. As with 8086-family instructions, memory to memory operations are never allowed. One operand must be a coprocessor register.

Instructions usually have a source and a destination operand. The source specifies one of the values to be processed. It is never changed by the operation. The destination specifies the value to be operated on and replaced with the result of the operation. If operands are specified, the first is the destination and the second is the source.

The stack organization of registers gives the programmer flexibility to think of registers either as elements on a stack, or as registers much like 8086-family registers. The variations of coprocessor instructions are listed below with the syntax for each:

Table 19.1

Coprocessor Operand Forms

Instruction Form	Syntax	Implied Operands	Example
Classical Stack	F*action*	ST(1),ST	`fadd`
Memory	F*action* *memory*	ST	`fadd memloc`
Register	F*action* ST(*num*),ST F*action* ST,ST(*num*)		`fadd st(5),st` `fadd st,st(3)`
Register Pop	F*action*P ST(*num*),ST		`faddp st(4),st`

Not all instructions accept all operand variations. For example, load and store instructions always require the memory form. Load constant instructions always take the classical stack form. Arithmetic instructions can usually take any form.

Some instructions that accept the memory form can have the letter I (integer) or B (BCD) following the initial F to specify how a memory operand is to be interpreted. For example, **FILD** interprets its operand as an integer and **FBLD** interprets its operand as a BCD number. If no type letter is included in the instruction name, the instruction works on real numbers.

19.2.1 Using Implied Operands in the Classical Stack Form

The stack form treats coprocessor registers like items on a stack. Items are pushed onto or popped off the top elements of the stack. Since only the top item can be accessed on a traditional stack, there is no need to specify operands. The first register (and the second if there are two operands) is always assumed.

In arithmetic operations (see Section 19.5), the top of the stack (**ST**) is the source operand and the second register (**ST(1)**) is the destination. The result of the operation goes into the destination operand and the source is popped off the stack. The effect is that both of the values used in the operation are destroyed and the result is left at the top of the stack.

Instructions that load constants always use the stack form (see Section 19.4.1). In this case the constant created by the instruction is the implied source, and the top of the stack (**ST**) is the destination. The source is pushed into the destination.

Note

The classical stack form with its implied operands is similar to the register-pop form, not to the register form. For example, `fadd`, with the implied operands **ST(1),ST**, is equivalent to `faddp st(1),st`, rather than to `fadd st(1),st`.

■ Example

```
fld1                ; Push 1 onto stack
fldpi               ; Push pi onto stack
fadd                ; Add pi and 1 and pop
```

Figure ?.? shows how these instructions affect the stack.

```
          +------+          +------+          +------+
ST        | 1.0  |          | 3.14|           | 4.14|
   fld1   +------+   fldpi  +------+   fadd   +------+
ST(1)     |      |          | 1.0  |          |      |
          +------+          +------+          +------+
          .      .          .      .          .      .
          .      .          .      .          .      .
          .      .          .      .          .      .
```

19.2.2 Using Memory Operands

The memory form treats coprocessor registers like items on a stack. Items are pushed from memory onto the top element of the stack, or popped from the top element to memory. Since only the top item can be accessed on a traditional stack, there is no need to specify the stack operand. The top register (**ST**) is always assumed. However, the memory operand must be specified.

Memory operands can be used in load and store instructions (see Section 19.4.1). Load instructions push source values from memory to an implied destination register (**ST**). Store instructions pop source values from an implied source register (**ST**) to the destination in memory. Some versions of store instructions pop the register stack so that the source is destroyed. Others simply copy the source without changing the stack.

Memory operands can also be used in calculation instructions that operate on two values (see Section 19.5). The memory operand is always the source. The stack top (**ST**) is always the implied destination. The result of the operation replaces the destination without changing its stack position.

■ **Example**

```
             .DATA
m1           DD       1.0
m2           DD       2.0
             .CODE
             fld      m1           ; Push m1 onto stack
             fld      m2           ; Push m1 onto stack
             fadd     m1           ; Add m2 to stack top
             fstp     m1           ; Pop top into m1
             fst      m2           ; Copy top to m2
```

Figure ?.? shows how these instructions affect the stack and the memory locations used in the instructions.

```
         +------+       +------+        +------+        +------+       +------+
m1       | 1.0  |       | 1.0  |        | 1.0  |        | 3.0  |       | 3.0  |
         +------+       +------+        +------+        +------+       +------+
m2       | 2.0  |       | 2.0  |        | 2.0  |        | 2.0  |       | 1.0  |
         +------+       +------+        +------+        +------+       +------+


         +------+       +------+        +------+        +------+       +------+
ST       | 1.0  |       | 2.0  |        | 3.0  |        | 3.0  |       | 1.0  |
 fld m1  +------+ fld m2 +------+ fadd m1 +------+ fstp m1 +------+ fst m2 +------+
ST(1)    |      |       | 1.0  |        | 1.0  |        | 1.0  |       |      |
         +------+       +------+        +------+        +------+       +------+
         .      .       .      .        .      .        .      .       .      .
         .      .       .      .        .      .        .      .       .      .
```

. . . .

19.2.3 Specifying Operands in the Register Form

The register form treats coprocessor registers as traditional registers. Instructions are specified exactly like 8086-family instructions with two register operands. The only limitation is that one of the two registers must be the stack top (**ST**).

In the register form, operands are specified by name. The second operand is the source; it is not affected by the operation. The first operand is the destination; its value is replaced with the result of the operation. The stack position of the operands does not change.

The register form can only be used with the **FXCH** instruction and with arithmetic instructions that do calculations on two values. With the **FXCH** instruction, the stack top is implied and need not be specified.

■ Example

```
        fadd    st(1),st    ; Add second stack item to first
                            ;   result goes in second item
        fadd    st,st(2)    ; Add first stack item to second
                            ;   result goes in first item
        fxch    st(1)       ; Exchange first and second stack items
```

Figure ?.? shows how these instructions would affect the stack if the stack elements were already initialized to 1.0, 2.0, and 3.0.

```
      +------+                   +------+                  +------+              +------+
ST    | 1.0  |                   | 1.0  |                  | 4.0  |              | 3.0  |
      +------+                   +------+                  +------+              +------+
ST(1) | 2.0  | fadd st(1),st     | 3.0  | fadd st,st(2)    | 3.0  | fxch st(1)   | 4.0  |
      +------+                   +------+                  +------+              +------+
ST(2) | 3.0  |                   | 3.0  |                  | 3.0  |              | 3.0  |
      +------+                   +------+                  +------+              +------+
      .      .                   .      .                  .      .              .      .
      .      .                   .      .                  .      .              .      .
      .      .                   .      .                  .      .              .      .
```

19.2.4 Specifying Operands in the Register-Pop Form

The register-pop form treats coprocessor registers as a modified stack. This form has some of the aspects of both a stack and of registers. The destination register can be specified by name, but the source register must always be the stack top.

The result of the operation will be placed in the destination operand, and the stack top will be popped off the stack. The effect is that both values being operated on will be destroyed and the result of the operation will be saved in the specified destination register. The stack register form is only used for instructions that do calculations on two values.

■ Example

```
        faddp   st(2),st    ; Add first and third items and pop
                            ;   first item destroyed
                            ;   third moves to second and holds result
```

Figure ?.? shows how this instructions would affect the stack if the stack elements were already initialized to 1.0, 2.0, and 3.0.

```
      +------+                   +------+
ST    | 1.0  |                   | 2.0  |
      +------+                   +------+
ST(1) | 2.0  | faddp st(2),st    | 4.0  |
      +------+                   +------+
ST(2) | 3.0  |                   |      |
      +------+                   +------+
      .      .                   .      .
      .      .                   .      .
      .      .                   .      .
```

19.3 Coordinating Memory Access

Problems can occur when the coprocessor and the main processor both try to access a memory location at the same time. Since the processor and coprocessor work independently, they may not finish working on memory in the order in which you give instructions. There are two separate cases, and they are handled in different ways.

If a processor instruction is given first followed by a coprocessor instruction, the coprocessor must wait until the processor is finished before it can start the next instruction. This is handled automatically by **MASM** for the 8088 and 8086 or by the processor for the 80186, 80286, and 80386.

Processor Differences

In order to synchronize operations between the 8088 or 8086 processors and the 8087 coprocessor, each 8087 instruction must be preceded by a **WAIT** instruction. This is not necessory for the 80287 and 80387. If you use the **.8087** directive or the **/R** option, **MASM** inserts **WAIT** instructions automatically. However, if you use the **.286** or **.386** directive, **MASM** assumes the instructions are for the 80287 or 80387 and does not insert the **WAIT** instructions. If your code will never need to run on an 8086 or 8088 processor, you can make your programs shorter and more efficient by using the **.286** or **.386** directive.

If a coprocessor instruction that accesses memory is followed by a processor instruction that attempts to access the same memory location, memory access is not automatically synchronized. For example, if you store a coprocessor register to a variable, then try to load that variable into a processor register, the coprocessor may not be finished. Thus the processor gets the value that was in memory before the coprocessor instruction rather than the value stored by the coprocessor.

Use the **WAIT** or **FWAIT** instruction (they are the same) to ensure that the coprocessor finishes.

■ Example

```
; Processor instruction first - No wait needed
            mov     WORD PTR mem32,ax   ; Load memory
            mov     WORD PTR mem32[2],dx
            fild    mem32               ; Load to register

; Coprocessor instruction first - Wait needed

            fist    mem32               ; Store to memory
            fwait                       ; Wait until coprocessor is done
            mov     ax,WORD PTR mem32   ; Move to register
            mov     dx,WORD PTR mem32[2]
```

19.4 Transferring Data

The 8087-family coprocessors have separate instructions for each of the following types of transfers:

- Transferring data between memory and registers, or between different registers
- Loading certain common constants into registers
- Transferring control data to and from memory

19.4.1 Transferring Data to and from Registers

Data-transfer instructions transfer data between main memory and the coprocessor registers, or between different coprocessor registers. Two basic principles govern data transfers:

- The instruction determines whether a value in memory will be considered an integer, BCD number, or a real number. The value is always considered a temporary real number once it is transferred to the coprocessor.
- The size of the operand determines the size of a value in memory. Values in the coprocessor always takes up 10 bytes.

The adjustments between formats are made automatically. Note that floating-point numbers must be stored in the IEEE format, not in the Microsoft Binary Real format. Data is automatically stored correctly when you use the **.8087**, **.287**, or **.387** directive. Coprocessor instructions are disabled until one of these directives has been used, so there is no danger of accidentally using the wrong format.

Data are transferred to stack registers using load commands. These push data onto the stack from memory or coprocessor registers. Data are removed using store commands. Some store commands pop data off the register stack into memory or coprocessor registers, while others simply copy the data without changing it on the stack.

The data transfer instructions are explained below:

Real Transfers

Syntax	Description
FLD *source*	Pushes a copy of the *source* into the stack-top register. The *source* may be a coprocessor register or a 4-, 8-, or 10-byte memory operand. If it is a memory operand, the value is automatically converted to the temporary-real format.
FST *destination*	Copies the value in the stack-top register into *destination* without affecting the register stack. The *destination* may be a coprocessor register or a 4- or 8-byte memory operand. If it is a memory operand, the value is automatically converted from temporary-real format to short real or long real, depending on the size of the operand. It cannot be converted to the 10-byte-real format.
FSTP *destination*	Pops a copy of the value in the stack-top register into *destination*. The *destination* may be a coprocessor register or a 4-, 8-, or 10-byte memory operand. If it is a memory operand, the value is automatically converted from temporary-real format to the appropriate real-number format, depending on the size of the operand.

Integer Transfers

Syntax	Description
FILD *source*	Pushes a copy of the *source* into the stack-top register. The *source* must be a 4-, 8-, or 10-byte memory operand. It will be interpreted as an integer and converted to temporary real format.
FIST *destination*	Copies the value in the stack-top register into *destination*. The *destination* must be a 2- or 4-byte memory operand. It is automatically converted from temporary-real format to a word or a doubleword, depending on the size of the operand. It cannot be converted to an 8-byte integer.

FISTP *destination*	Pops a copy of the value in the stack-top register into *destination*. The *destination* must be a 2-, 4-, or 8-byte memory operand. It is automatically converted from temporary-real format to a word, doubleword, or quadword, depending on the size of the operand.

Packed BCD Transfers

Syntax	Description
FBLD *source*	Pushes a copy of the *source* into the stack-top register. The *source* must be a 10-byte memory operand. It should contain a packed BCD value, although no check is made to see that the data is valid.
FBSTP *destination*	Pops a copy of the value in the stack-top register into *destination*. The *destination* must be a 10-byte memory operand. The value will be rounded to an integer, if necessary, and converted to a packed BCD value.

Register Exchange

Syntax	Description
FXCH *destination*	Exchanges the value in the stack-top register with the value in *destination*. The *destination* must be a coprocessor register. If no *destination* is specified, **SP(0)** and **SP(1)** are exchanged.

■ Example 1

```
fld     m1                 ; Push m1 into stack top
fld     st(0)              ; Push st(0) into stack top
fst     m2                 ; Copy stack top to m2
fxch    st(2)              ; Exchange stack and st(2)
fstp    m1                 ; Pop stack top into m1
```

Figure ?.? illustrates how the instructions in Example 1 affect the stack. Assume that stack registers **ST** and **ST(1)** have been initialized to 3.0 and 4.0 respectively.

```
          Main Memory
          +------+     +------+     +------+     +------+     +------+     +------+
m1        | 1.0  |     | 1.0  |     | 1.0  |     | 1.0  |     | 1.0  |     | 3.0  |
          +------+     +------+     +------+     +------+     +------+     +------+
m2        | 2.0  |     | 2.0  |     | 2.0  |     | 4.0  |     | 4.0  |     | 4.0  |
          +------+     +------+     +------+     +------+     +------+     +------+

              fld mem1     fld ST(2)    fst mem2     fxch ST(2)   fstp mem1

          +------+     +------+     +------+     +------+     +------+     +------+
ST(0)     | 3.0  |     | 1.0  |     | 4.0  |     | 4.0  |     | 3.0  |     | 1.0  |
          +------+     +------+     +------+     +------+     +------+     +------+
ST(1)     | 4.0  |     | 3.0  |     | 1.0  |     | 1.0  |     | 1.0  |     | 4.0  |
          +------+     +------+     +------+     +------+     +------+     +------+
ST(2)     |      |     | 4.0  |     | 3.0  |     | 3.0  |     | 4.0  |     | 4.0  |
          +------+     +------+     +------+     +------+     +------+     +------+
ST(3)     |      |     |      |     | 4.0  |     | 4.0  |     | 4.0  |     |      |
          +------+     +------+     +------+     +------+     +------+     +------+
          .      .     .      .     .      .     .      .     .      .     .      .
          .      .     .      .     .      .     .      .     .      .     .      .
          .      .     .      .     .      .     .      .     .      .     .      .
          Coprocessor Registers
```

■ **Example 2**

```
            .DATA
shortreal   DD      100 DUP (?)
longreal    DQ      100 DUP (?)
            .CODE
            .                          ; Assume array shortreal has been
            .                          ;   filled by previous code
            .
            mov     cx,100             ; Initialize loop
            xor     si,di              ; Clear pointer into shortreal
            xor     di,di              ; Clear pointer into longreal
again:      fld     shortreal[si]      ; Push shortreal
            fstp    longreal[di]       ; Pop longreal
            add     si,4               ; Increment source pointer
            add     di,8               ; Increment destination pointer
            loop    again              ; Do it again
```

Example 2 illustrates one way of doing run-time type conversions.

19.4.2 Loading Constants

Constants cannot be given as operands and loaded directly into coprocessor registers. You must allocate memory and initialize the variable to constant value. The variable can then be loaded using one of the load instructions described in Section 19.4.1.

However, special instructions are provided for loading certain constants. You can load 0, 1, pi, and several common logarithmic values directly. Using these instructions is faster and often more precise than loading the values from initialized variables.

The instructions that load constants all have the stack top as the implied destination operand. The constant to be loaded is the implied source operand. The instructions are listed below:

Syntax	Description
FLDZ	Pushes 0 onto top of stack
FLD1	Pushes 1 onto top of stack
FLDPI	Pushes the value of pi onto top of stack
FLDL2E	Pushes the value of $\log_2 e$ onto top of stack
FLDL2T	Loads $\log_2 10$ onto top of stack
FLDLG2	Loads $\log_{10} 2$ onto top of stack
FLDLN2	Loads $\log_e 2$ onto top of stack

19.4.3 Transferring Control Data

The coprocessor data area, or parts of it, can be stored to memory and later loaded back. One reason for doing this is to save a snapshot of the coprocessor state before going into a procedure, and restore the same status after the procedure. Another reason is to modify coprocessor behavior by storing certain data to main memory, operating on the data with 8086-family instructions, and then loading it back to the coprocessor data area.

You can choose to transfer the entire coprocessor data area, the control registers, or just the status or control word. Applications programmers will seldom need more than the control word.

All the control transfer instructions take a single memory operand. Load instructions use the memory operand as the destination; store instruction use it as the source. The coprocessor data area is the implied source for load instructions and the implied destination for store instructions.

Each store instruction has two forms: The wait form checks for unmasked numeric error exceptions and waits until they have been handled. The no-wait form (which always begins with FN) ignores unmasked exceptions. The instructions are listed below:

Syntax	Description
FLDCW *mem2byte*	Load control word
F[[**N**]]**STCW** *mem2byte*	Store control word
F[[**N**]]**STSW** *mem2byte*	Store status word
FLENV *mem14byte*	Load environment
F[[**N**]]**STENV** *mem14byte*	Store environment
FRSTOR *mem94byte*	Restore state
F[[**N**]]**SAVE** *mem94byte*	Save state

■ **80287-80387 Processors Only**

Starting with the 80287, the **FSTSW** and **FNSTSW** instructions can store data directly to the **AX** register. This is the only case in which data can be transferred directly between processor and coprocessor registers. For example:

```
        fstsw   ax
```

■ **80387 Processor Only**

In 32-bit mode, the 80387 stores 32-bit addresses in the instruction and operand pointers. Therefore, the **FSAVE** instruction stores 98 bytes instead of 94, and the **FSTENV** stores 18 bytes instead of 14.

19.5 Doing Arithmetic Calculations

The math coprocessors offer a rich set of instructions for doing arithmetic. Most arithmetic instructions accept operands in any of the formats discussed in Section 19.2.

When using memory operands with an arithmetic instruction, make sure you indicate in the name whether you want the memory operand to be treated as a real number or an integer. For example, use **FADD** to add a real number to the stack top or **FIADD** to add an integer to the stack top. You don't need to specify the operand type in the instruction if both operands are stack registers, since register values are always real numbers.

You cannot do arithmetic on BCD numbers in memory. You must use **FBLD** to load the numbers into stack registers.

The arithmetic instructions are listed below:

Addition

These instructions add the source and destination and put the result in the destination.

Syntax	Description
FADD	Classical stack form. **ST** and **ST(1)** are the implied source and destination.
FADD ST(*num*),ST	Register form. Stack top is source.
FADD ST,ST(*num*)	Register form. Stack top is destination.
FADD *mem*	Real-memory form. Memory operand is interpreted as real number. Stack top is implied destination.
FIADD *mem*	Integer-memory form. Memory operand is interpreted as integer. Stack top is implied destination.
FADDP ST(*num*),ST	Register-pop form.

Normal Subtraction

These instructions subtract the source from the destination and put the difference in the destination. Thus the number being subtracted from is replaced by the result.

Syntax	Description
FSUB	Classical stack form. **ST** and **ST(1)** are the implied source and destination.
FSUB ST(*num***),ST**	Register form. Stack top is source.
FSUB ST,ST(*num***)**	Register form. Stack top is destination.
FSUB *mem*	Real-memory form. Memory operand is interpreted as real number. Stack top is implied destination.
FISUB *mem*	Integer-memory form. Memory operand is interpreted as integer. Stack top is implied destination.
FSUBP ST(*num***),ST**	Register-pop form.

Reversed Subtraction

These instructions subtract the destination from the source and put the difference in the destination. Thus the number subtracted is replaced by the result.

Syntax	Description
FSUBR	Classical stack form. **ST** and **ST(1)** are the implied source and destination.
FSUBR ST(*num***),ST**	Register form. Stack top is source.
FSUBR ST,ST(*num***)**	Register form. Stack top is destination.
FSUBR *mem*	Real memory form. Memory operand is interpreted as real number. Stack top is implied destination.
FISUBR *mem*	Integer-memory form. Memory operand is interpreted as integer. Stack top is implied destination.

FSUBRP ST(*num*),ST	Register-pop form.

Multiplication

These instructions multiply the source and destination and put the product in the destination.

Syntax	Description
FMUL	Classical stack form. **ST** and **ST(1)** are the implied source and destination.
FMUL ST(*num*),ST	Register form. Stack top is source.
FMUL ST,ST(*num*)	Register form. Stack top is destination.
FMUL *mem*	Real-memory form. Memory operand is interpreted as real number. Stack top is implied destination.
FIMUL *mem*	Integer-memory form. Memory operand is interpreted as integer. Stack top is implied destination.
FMULP ST(*num*),ST	Register-pop form.

Normal Division

These instructions divide the destination by the source and put the quotient in the destination. Thus the dividend is replaced by the quotient.

Syntax	Description
FDIV	Classical stack form. **ST** and **ST(1)** are the implied source and destination.
FDIV ST(*num*),ST	Register form. Stack top is source.
FDIV ST,ST(*num*)	Register form. Stack top is destination.
FDIV *mem*	Real-memory form. Memory operand is interpreted as real number. Stack top is implied destination.

FIDIV *mem*	Integer-memory form. Memory operand is interpreted as integer. Stack top is implied destination.
FDIVP ST(*num*),ST	Register-pop form.

Reversed Division

These instructions divide the source by the destination and put the quotient in the destination. Thus the divisor is replaced by the quotient.

Syntax	Description
FDIVR	Classical stack form. ST and ST(1) are the implied source and destination.
FDIVR ST(*num*),ST	Register form. Stack top is source.
FDIVR ST,ST(*num*)	Register form. Stack top is destination.
FDIVR *mem*	Real-memory form. Memory operand is interpreted as real number. Stack top is implied destination.
FIDIVR *mem*	Integer-memory form. Memory operand is interpreted as integer. Stack top is implied destination.
FDIVRP ST(*num*),ST	Register-pop form.

Other Operations

These instructions all use the stack top (ST) as an implied destination operand. The result of the operation replaces the value in the stack top. No operand should be given.

Syntax	Description
FABS	Sets the sign of the stack-top value to positive.
FCHS	Changes the sign of the stack-top value.
FRNDINT	Rounds the stack-top element to an integer.

FSQRT	Replaces the contents of the stack-top element with its square root.
FSCALE	Scales by powers of two by adding the value of **ST(1)** to the exponent of the value in **ST**. This effectively multiplies the stack top-value by two to the power contained in **ST(1)**. Since the exponent field is an integer, the value in **ST(1)** should normally be an integer.
FPREM	Calculates the partial remainder by performing modulo division on the top two stack registers. The value in **ST** is divided by the value in **ST(1)**. The remainder replaces the value in **ST**. The value in **ST(1)** is unchanged. Since this instruction works by repeated subtractions, it can take a lot of execution time if the operands are greatly different in magnitude. **FPREM** is sometimes used with trigonometric functions.
FXTRACT	Breaks a number down into its exponent and mantissa and pushes the mantissa onto the register stack. Following the operation, **ST** contains the value of the original mantissa and **ST(1)** contains the value of the unbiased exponent.

80387 Processor Only

The 80387 has a new instruction **FPREM1**. It is similar to **FPREM** except that its result is calculated differently. Unlike **FPREM**, it conforms to the IEEE standard.

■ Examples

```
; Macro to solve quadratic equations  -  no error checking

quadratic   MACRO   a,b,c,posx,negx
            fld1        ; Get constants 2 and 4
            fadd    st,st       ; 2 at bottom
            fld     st          ; Copy
            fadd    st,st(1)    ; 4 next
```

```
fmul    a           ; 4 * a
fmul    c           ;   * c

fld     b           ; Load b
fmul    b           ; Square it
fsubr               ; (b squared) - 4ac
                    ; Negative value here produces error
fsqrt               ; Get square root
fld     b           ; Get b
fchs                ; Make it negative
fld     st          ; Copy it
fadd    st,st(2)    ; Do plus version
fxch                ; Exchange
fsub    st,st(2)    ; Do minus version
fld     st(3)       ; Get 2
fmul    a           ; 2 * a
fld     st          ; Copy it

fdivr   st,st(3)    ; Divide plus version
fstp    posx        ; Store it
fdivr               ; Divide minus version
fstp    negx        ; Store it
ENDM
```

This macro solves quadratic equations. The arguments `a`, `b`, and `c` must be memory locations containing the values for which the equation is to be solved. The arguments `posx` and `negx` must be memory locations where the result will be stored. This example does no error checking. It will fail for some values, because it will attempt to find the square root of a negative number. You could enhance the macro by using the **FTST** instruction (see Section 19.6.1) to check for a negative value just before the square root is calculated. If `b` squared minus `4ac` is negative, jump to an error handler.

19.6 Controlling Program Flow

The math coprocessors have several instructions that set control flags in the status word. The 8087-family control flags can be used with conditional jumps to direct program flow in the same way that 8086-family flags are used.

Since the coprocessor does not have jump instructions, you must transfer the status word to memory so that the flags can be used by 8086-family instructions.

An easy way to use the status word with conditional jumps is to move its upper byte into the lower byte of the processor flags. For example, use the following statements:

```
fstw    mem16       ; Store status word in memory
mov     ax,mem16    ; Move to AX
sahf                ; Store upper word in flags
```

Figure ?.? shows how the coprocessor control flags line up with the processor flags. **C3** overwrites the zero flag, **C2** overwrites the parity flag, and **C0** overwrites the carry flag. **C1** overwrites an undefined bit, so it cannot be used directly with conditional jumps, although you can use the **TEST** instruction to check **C1** in memory or in a register. The sign and auxiliary-carry flags are also overwritten, so you can't count on them being unchanged after the operation.

```
               15  14 13   12 11   10      9   8
              +---+---+---+---+---+---+---+---+
Status Word   |   |C3 |           |C2 |C1 |C0 |
              +---+---+---+---+---+---+---+---+

                7   6   5   4   3   2   1   0
              +---+---+---+---+---+---+---+---+
Flags         |SF |ZF | ? |AF | ? |PF | ? |CF |
              +---+---+---+---+---+---+---+---+
```

See Section 17.1.1 for more information on using conditional-jump instructions based on flag status.

■ 80287-80387 Processors Only

Starting with the 80287, **FSTSW** and **FNSTSW** can store the status word directly to **AX** instead of going through memory. Use this syntax only if you know your code will never need to run under the 8087. For example, use these statements:

```
fstw    ax          ; Store status word to AX
sahf                ; Store upper word in flags
```

19.6.1 Comparing Operands to Control Program Flow

The 8087-family coprocessors provide several instructions for comparing operands. All these instructions compare the stack top (**ST**) to a source operand, which may either be specified, or it can be implied as **ST(1)**.

The comparison instructions affect the **C3**, **C2**, and **C0** control flags. The **C1** flag is not affected. Table 19.1 shows the flags set for each possible result of a comparison.

Table 19.2

Control-Flag Settings after Compare Instructions

Result	C3	C2	C0
ST > *source*	0	0	0
ST < *source*	0	0	1
ST = *source*	1	0	0
Not comparable	1	1	1

Variations on the compare instruction allow you to pop the stack once or twice, and to compare integers and zero. For each instruction, the stack top is always the implied destination operand. If you do not give an operand, **ST(1)** is the implied source. Some compare instructions allow you to specify the source as a memory or register operand.

The instructions are listed below:

Compare

Compares the stack top to the source. The source and destination are unaffected by the comparison.

Syntax	Description
FCOM	Classical stack form. **ST(1)** is the implied source.
FCOM ST(*num***)**	Register form. Specified register is source.
FCOM *mem*	Memory-real form. Memory operand is interpreted as real number. The operand cannot be a 10-byte real number.
FICOM *mem*	Memory-integer form. Memory operand is interpreted as an integer. The operand cannot be a doubleword integer (except on the 80387).

Compare and Pop

Compares the stack top to the source, then pops the stack. Thus the destination is destroyed by the comparison.

Syntax	Description
FCOMP	Classical stack form. **ST(1)** is the implied source.
FCOMP ST(*num*)	Register form. Specified register is source.
FCOMP *mem*	Memory form. Memory operand is interpreted as real number. The operand cannot be a 10-byte real number.
FICOMP *mem*	Memory form. Memory operand is interpreted as an integer. The operand cannot be a doubleword integer (except on the 80387).

Others

Syntax	Description
FCOMPP	Classical stack form. Compares **ST** to **ST(1)**, then pops the stack twice. Both of the source and destination are destroyed by the comparison.
FTST	Classical stack form. Compares the stack top to zero. The control registers shown in Table 19.1 will be effected as if a 0 in **ST(1)** had been compared to **ST**.

80387 Processor Only

Unordered compare instructions are available with the 80387. The **FUCOM**, **FUCOMP**, and **FUCOMPP** instructions are like **FCOM**, **FCOMP**, and **FCOMPP** except that it will not cause an invalid operation exception when one of the operands is a quiet NAN. Exceptions and NANs are beyond the scope of the manual, so theses instructions are not explained here. See Intel coprocessor reference books for more information.

■ Example

```
            IFDEF   c287
            .287
            ENDIF
            .DATA
right       DD      10.35       ; Sides of a rectangle
left        DD      13.07
diameter    DD      12.93       ; Diameter of a circle
status      DW      ?
            .CODE
; Get area of rectangle
            fld     across      ; Load one side
            fmul    down        ; Multiply by the other

; Get area of circle
            fld1                ; Load one and
            fadd    st,st       ;   double it to get constant 2
            fdivr   diameter    ; Divide diameter to get radius
            fmul    st,st       ; Square radius
            fldpi               ; Load pi
            fmul                ; Multiply it

; Compare area of circle and rectangle
            fcompp              ; Compare and throw both away
            IFNDEF  c287
            fstsw   status      ; Load from coprocessor to memory
            mov     ax,status   ;   to register
            ELSE
            fstw    ax          ;   (for 287+, skip memory)
            ENDIF
            sahf                ;   to flags
            jp      nocomp      ; If parity set, can't compare
            jz      same        ; If zero set, they're the same
            jc      rectangle   ; If carry set, rectangle is bigger
            jmp     circle      ;   else circle is bigger
nocomp:     .                   ; Error handler
            .
same:       .                   ; Both equal
            .
rectangle:  .                   ; Rectangle bigger
            .
circle:     .                   ; Circle bigger
```

Notice how conditional blocks are used to enhance 80287 code. If you define the symbol c287 from the command line using the **/D***symbol* option (see Section 2.4.4), the code would be smaller and faster, but could not be run on an 8087.

19.6.2 Testing Control Flags after Other Instructions

In addition to the compare instructions, the **FXAM** and **FPREM** instructions affect coprocessor control flags.

The **FXAM** instruction sets the value of the control flags based on the type of the number in the stack top (**ST**). This instruction is used to identify and handle special values such as infinity, zero, unnormal numbers, denormal numbers, and NANs (Not a Number). Certain math operations are capable of producing these special-format numbers. A description of them is beyond the scope of this manual. The possible settings of the flags are shown in the *Microsoft Macro Assembler Reference.*

FPREM also sets control flags. Since this instruction must sometimes be repeated to get a correct remainder for large operands, it uses the **C2** flag to indicate whether the remainder returned is partial (**C2** is set) or complete (**C2** is clear). If the bit is set, the operation should be repeated.

FPREM also returns the least significant three bits of the quotient in **C0**, **C3**, and **C1**. These bits are useful for reducing operands of periodic transcendental functions such as sine and cosine to an acceptable range. The technique is not explained here. The possible settings for each flag are shown in the *Microsoft Macro Assembler Reference.*

19.7 Using Transcendental Instructions

The 8087-family coprocessors provide a variety of instructions for doing transcendental calculations, including exponentiation, logarithmic calculations, and some trigonometric functions.

Use of these advanced instructions is beyond the scope of this manual. However, the instructions are listed below for reference. All transcendental instructions have implied operands—either **ST** as a single destination operand, or **ST** as the destination and **ST(1)** as the source.

Instruction	**Description**
F2XM1	Calculates 2^x-1, where x is the value of the stack top. The value x must be between 0 and .5, inclusive. Returning 2^X-1 instead of 2^X allows the instruction to return the value with greater accuracy. The programmer can adjust the result to get 2^X.
FYL2X	Calculates Y times $\log_2 X$, where X is in **ST** and Y is in **ST(1)**. The stack is popped, so both X and Y are destroyed, leaving the result in **ST**. The value of X must be positive.

FYL2XP1	Calculates Y times $\log_2 (X+1)$, where X is in **ST** and Y is in **ST(1)**. The stack is popped, so both X and Y are destroyed, leaving the result in **ST**. The absolute value of X must be between 0 and the square root of 2 divided by 2. This instruction is more accurate than **FYL2X** if YX is very close to 1.
FPTAN	Calculates the tangent of the value in **ST**. The result is a ratio Y/X, with Y replacing the value in **ST** and X pushed onto the stack so that after the instruction, **ST** contains Y and **ST(1)** contains X. The value being calculated must be a positive number less than pi/4. The **FPTAN** instruction can be used to calculate other trigonometric functions, including sine and cosine.
FPATAN	Calculates the arctangent of the ratio Y/X, where X is in **ST** and Y is in **ST(1)**. The stack is popped, so both X and Y are destroyed, leaving the result in **ST**. Both X and Y must be positive numbers less than infinity, and Y must be less than X. The **FPATAN** instruction can be used to calculate other inverse trigonometric functions, including arcsine and arccosine.

■ 80387 Processor Only

Additional trigonometric functions are available on the 80387.

Instruction	Description
FSIN	Calculates the sine of the value in **ST**. The stack-top value is replaced by its sine.
FCOS	Calculates the cosine of the value in **ST**. The stack-top value is replaced by its cosine.
FSINCOS	Calculates the sine and cosine of the value in **ST**. When the instruction is complete, the value in **ST** is the sine of the original stack-top value. The value in **ST(1)** is the cosine of the original stack-top value.

19.8 Controlling the Coprocessor

Additional instructions are available for controlling various aspects of the coprocessor. With the exception os **FINIT**, these instructions are generally used only by systems programmers. They are summarized below, but not fully explained or illustrated. Some instructions have a wait version and a no-wait version. The no-wait versions have N as the second letter.

Syntax	Description
F⟦N⟧INIT	Resets the coprocessor and restores all the default conditions in the control and status words. It is a good idea to use this instruction at the start and end of your program. Placing it at the start makes sure that no register values from previous programs will affect your program. Placing it at the end makes sure that register values from your program will not affect later programs.
F⟦N⟧CLEX	Clears all exception flags and the busy flag of the status word. Also clears the error-status flag on the 80287 and 80387, or the interrupt-request flag on the 8087.
FINCSTP	Adds one to the stack pointer in the status word. Do not use to pop the register stack. No tags or registers are altered.
FDECSTP	Subtracts one from the stack pointer in the status word. No tags or registers are altered.
FREE ST(*num*)	Marks the specifed register as empty.
FNOP	Copies the stack top to itself, thus padding the executable file and taking up processing time without having any effect on registers or memory.

■ 8087 Processors Only

The 8087 has the instructions **FDISI**, **FNDISI**, **FENI**, and **FNENI**. These instructions can be used to enable or disable interrupts. The 80287 and 80387 permit these instructions, but ignores them. Applications programmers will not normally need these instructions. Systems programmers should avoid using them so that their programs will be portable to all coprocessors.

■ 80287-80387 Processors Only

Starting with the 80287, the **FSETPM** (Set Protected Mode) instruction is available. This instruction enables the coprocessor to run in protected mode. The primary difference is that the addresses stored in the instruction and operand pointers have a segment selector instead of an actual segment address. See Section 13.2 for information on segment selectors.

The **.PRIV** directive and either the **.286** or **.386** directive must be given before the **FSETPM** instruction can be used. Protected-mode operating systems normally set protected mode automatically. Therefore, you only need this instruction if you are writing control software. You do not need it to run applications software under control software that supports protected mode.

Chapter 20
Controlling the Processor

The 8086-family processors provide instructions for controlling the processor. A few of these instructions are available on all processors, but most of them are for controlling protected mode operations on the 80286 and 80386.

System control instructions have limited use in applications programming. They are primarily used by systems programmers who write operating systems and other control software. Since systems programming is beyond the scope of this manual, the systems control instructions are summarized, but not explained or illustrated in the next sections.

20.1 Controlling Timing and Alignment

The **NOP** instruction does nothing but take up time and space. It can be used for delays in timing loops, or to pad executable code for alignment.

Normally applications programmers should avoid using the **NOP** instruction in timing loops, since such loops will take different lengths of time on different machines. A better way to control timing is to use the DOS or BIOS time functions, since they are based on the computer's internal clock rather than on the speed of the processor.

MASM automatically inserts **NOP** instructions for padding in two situations:

- You can use the **ALIGN** or **EVEN** directives (see Section 6.4) to align data or code on a given boundary. The assembler automatically inserts **NOP** instructions so that the next code or data will be on the specified boundary. This is usually a better method than inserting **NOP** instruction in your code.
- On the first pass, the assembler assumes that **JMP** instructions are near (16-bit). If the instructions turn out to be short on the second pass, the assembler adjusts by inserting a **NOP** instruction. You can avoid this situation by using the **SHORT** operator, as explained in Section 9.4.1.

20.2 Checking Memory Ranges

■ All Except 8088/8086

Starting with the 80186 processor, the **BOUND** instruction can check to see if a value is within a specified range. This instruction is usually used to check a signed index value to see if it is within the range of an array.

To use it for this purpose, the starting and ending values of the array must be stored as 8-bit values in the low and high bytes of a word memory operand. This operand is given as the source operand. The index value to be checked is given as the destination operand.

80386 Only

For the 80386 processor, the **BOUND** instruction can check a 32-bit index against a 32-bit operand containing 16-bit starting and ending values.

If the index value is out of range, the instruction issues interrupt 5. The **BOUND** instruction only makes sense if the programmer has written an error routine for interrupt 5. See Section 17.5 for more information on interrupts.

■ Example 1

```
            .DATA
bottom      EQU     0
top         EQU     19
array       DB      top+1 DUP (?)       ; Allocate array
bounds      LABEL   WORD                ; Allocate boundaries
bbounds     DB      bottom,top          ;   initialized to bounds
index       DW      ?
            .CODE
            .
            .
            .
            mov     bx,index            ; Load index
            bound   bx,bounds           ; Check to see if it is in range
                                        ;   if out of range, interrupt 5
            mov     dx,array[index]     ; If in range, use it
```

The same operation can be done less efficiently with multiple comparisons.

Example 2 shown an alternative to the code section of Example 1.

■ **Example 2**

```
            mov     bx,index             ; Load index
            cmp     bx,bbounds[0]        ; Is it too low?
            jl      noway                ; Yes? Error
            cmp     bx,bbounds[1]        ; No? Is it too high?
            jg      noway                ; Yes? Error
            jmp     ok                   ; No? OK
noway:      int     05h                  ; Call interrupt if out of range
            .
            .
            .
ok:         mov     dx,array[index]      ; If in range, use it
```

20.3 Controlling the Processor in Real Mode

The **WAIT**, **ESC**, **LOCK**, and **HLT** instructions control different aspects of the processor in real mode.

They can be used to control processes that are handled by external coprocessors. The 8087-family coprocessors are the most commonly used coprocessors with 8086-family processors, but 8086-based machines can use other coprocessors with the proper hardware and control software.

These instructions are summarized below:

Instruction	Description
LOCK	Locks out other processors until a specified instruction is finished. This is a prefix that precedes the instruction. It can be used to make sure that a coprocessor does not interrupt a crucial instruction.
WAIT	Instructs the processor to doing nothing until it receives a signal of an external event from a coprocessor.
ESC	Provides an instruction and possibly a memory operand for use by a coprocessor. It is not required for 8087-family coprocessors, since the assembler automatically handles these details for all coprocessor instructions.

HLT Stops the processor until an interrupt is received. It can be used in place of an endless loop if a program needs to wait for an interrupt.

20.4 Controlling Protected Mode Processes

■ **80286 and 80386 Only**

Protected mode is available starting with the 80286 processors. This mode is generally initiated and controlled by an operating system. Current versions of DOS do not support protected mode.

The instructions that control protected mode can only be used if the **.PRIV** and **.286** or **.386** directives have been used. These instructions are generally needed only for operating systems and other control software.

Note that under protected mode operating systems (including future versions of DOS), applications programmers do not need to use protected-mode instructions or the **.PRIV** directive. Process control is managed through operating system function calls.

Some protected mode instructions use internal registers of the 80286 or 80386 processors. Instructions are provided for loading values from these registers into memory where they can be modified. Other instructions can then be used to store the values back to the special registers.

The protected mode instructions are listed below:

Instruction	Action
LAR	Loads access rights
LSL	Loads segment limit
LGDT	Loads global descriptor table
SGDT	Stores global descriptor table
LIDT	Loads 8-byte interrupt descriptor table
SIDT	Stores 8-byte interrupt descriptor table
LLDT	Loads local descriptor table

SLDT	Stores local descriptor table
LTR	Loads task register
STR	Stores task register
LMSW	Loads machine status word
SMCW	Stores machine status word
ARPL	Adjusts requested privilege level
CLTS	Clear task-switched flag
VERR	Verify read access
VERW	Verify write access

20.5 Controlling the 80386

■ 80386 Only

The 80386 processor can use all the protected mode instructions, but it also allows you to use **MOV** to transfer data between general-purpose registers and special registers.

The following special registers can be accessed with move instructions on the 80386:

Type	Registers
Control	**CR0**, **CR2**, and **CR3**
Debug	**DR0**, **DR1**, **DR2**, **DR3**, **DR6**, and **DR7**
Test	**TR6** and **TR7**

These registers can be moved directly to or from 32-bit registers.

■ Examples

```
        mov     eax,cr0                 ; Load CR0 into EAX
        mov     cr1,ecx                 ; Store ECX in CR1
```

Appendixes

Appendix A
New Features

of the Microsoft Macro Assembler has many significant new features. Some of the most important are the new CodeView debugger, support for the 80386 processor, and an optional simplified system of defining segments. This appendix describes these features and tell where they are documented.

A.1 MASM Enhancements

MASM has the several important enhancements over Version 4.0.

A.1.1 80386 Support

MASM supports the 80386 instruction set and addressing modes. The 80386 processor is a superset of other 8086-family processors. Most new features are simply 32-bit extensions of 16-bit features.

If you understand the features of the 16-bit 8086-family processors, using the 32-bit extensions is not difficult. The new 32-bit registers are used in much the same way as the 16-bit registers. The 80386 registers are explained in Section 13.3.

However, some features of the 80386 processor are significantly different. Throughout the manual the heading **80386 Processor Only** is used to flag sections where 80386 enhancements are described. Areas of particular importance include the **.386** directive for initializing the 80386 (Section 4.6.1), the **USE32** and **USE16** segment types for setting the segment size (Section 5.2.2.2), and indirect addressing modes (Section 14.3.3).

The 80386 processor and the 80387 coprocessor also have the new instructions listed in Table A.1.

Table A.1

80386 and 80387 Instructions

Name	Mnemonic	Reference
Bit Scan Forward	BSF	Section 16.8
Bit Scan Reverse	BFR	Section 16.8
Bit Test	BT	Section 16.7
Bit Test and Complement	BTC	Section 16.7

Bit Test and Reset	BTR	Section 16.7
Bit Test and Set	BTS	Section 16.7
Move with Sign Extend	MOVSX	Section 15.2
Move with Zero Extend	MOVZX	Section 15.2
Set Byte on Condition	SET*condition*	Section 17.2
Double Precision Shift Left	SHLD	Section 16.9.5
Double Precision Shift Right	SHRD	Section 16.9.5
Move to/from Special Registers	MOV	Section 20.5
Sine	FSIN	Section 19.7
Cosine	**FCOS**	Section 19.7
Sine Cosine	**FSINCOS**	Section 19.7
IEEE Partial Remainder	**FPREM1**	Section 19.5
Unordered compare real	**FUCOM**	Section 19.6.6
Unordered compare realand pop	**FUCOMP**	Section 19.6.1
Unordered compare real and pop twice	**FUCOMPP**	Section 19.6.1

A.1.2 Segment Simplification

A new system of defining segments is available in **MASM** naming conventions. If you are willing to accept these conventions, segments can be defined more easily and consistently. However, this feature is optional. You can still use the old system if you need more direct control over segments or if you need to be consistent with existing code. See Section 5.1.

A.1.3 Enhanced Error Handling

Error handling has been enhanced in the following ways:

- Messages have been reworded, enhanced, or reorganized.
- Messages are divided into three levels: severe errors, serious warnings, and advisory warnings. The level of warning can be changed using with the **/W** option. See Section 2.4.13.
- During assembly, messages are output to the standard output device (by default, the screen). They can be redirected to a file or device. In Version 4.0 they were sent to the standard error device. See Section 2.3.

A.1.4 New Options

The following command-line options have been added:

Option	Description
/W⟦**0**|**1**|**2**⟧	Sets the warning level to determine what type of messages will be displayed. The three kinds are severe errors, serious warnings, and advisory warnings. See Section 2.4.13.
/ZI and **/ZD**	Sends debugging information for symbolic debuggers to the object file. The **/ZD** outputs line number information, while the **/ZI** option outputs both line number and type information. See Section 2.4.15.
/H	Displays the MASM command line and options. See Section 2.4.5.
/D*sym*⟦=*val*⟧	Allows definition of a symbol from the command line. This is an enhancement of a current option. See Section 2.4.4.

In addition new directives **.ALPHA** and **.SEQ** directives are added that have the same effect as the **/A** and **/S** options. See Section 5.2.1.

A.1.5 Environment Variables

MASM now supports two environment variables: **MASM** for specifying default options, and **INCLUDE** for specifying the search path for include files. See Section 2.2.

A.1.6 String Equates

String equates have been enhanced to be more reliable. By enclosing the argument to the **EQU** directive in angle brackets, you can ensure that the argument will be evaluated as a string equate rather than as an expression. See Section 11.1.3.

A.1.7 RETF and RETN Instructions

The **RETF** (Return Far) and **RETN** (Return Near) instructions are now available. These instructions enable you to define procedures without the **PROC** and **ENDP** directives. See Section 17.4.2.

A.1.8 Communal Variables

MASM now allows you to declare communal variables. These are uninitialized global data items. Declaring a variable communal is similar to declaring it both public and external. See Section 8.3.

A.1.9 Including Libary Files

The **INCLUDELIB** directive enables you to specify in the assembly source file any libraries that want to linked with your program module.

A.2 Link Enhancements

LINK has several new features. These enhancements are discussed in the **LINK** chapter of the *Microsoft CodeView and Utilities Software Development Tools Guide.* They are summarized below:

- There is now a **LINK** environment variable for specifying default options.
- The **/CODEVIEW** option puts debugging information in executable files for the CodeView debugger.
- The **/INFORMATION** option displays each step of the linking process including parsing command line, pass 1, and so on. The path and name of each module are displayed as the modules are linked.

A.3 The CodeView Debugger

In , of the Macro Assembler package, the CodeView debugger replaces **SYMDEB**. This source-level symbolic debugger is capable of working with code developed with **MASM** or with any of the Microsoft high-level-language compilers.

The CodeView debugger features a window-oriented environment with multiple windows displaying different types of information. Commands can be executed with a mouse, function keys, or command lines. Variables can be watched in a separate window as the program executes.

MASM and **LINK** have been enhanced to support the features of the CodeView debugger.

A.4 SETENV

Since **MASM** and **LINK** now support two environment variables each, users may wish to define environment strings that exceed the default size of the DOS environment. The **SETENV** program is provided as a means of modifying the environment size.

A.5 Other Enhancements

The Macro Assembler package now includes more example programs and an on-line CodeView samples session. CodeView keyboard templates are included.

The Macro Assembler documentation has been reorganized and revised. The instruction sets for all 8086-family processors are documented in both topical and reference format.

A.6 Compatibility with Assemblers and Compilers

If you are upgrading from a previous version of the Microsoft or IBM macro assembler, you may need to make some adjustments before assembling source code developed with previous versions. The potential compatibility problems are listed below:

- Some previous versions of the IBM Macro Assembler wrote segments to object files in alphabetical order. The current version writes segments to object files in the order encountered in the source file. You can use the **/A** option to order segments alphabetically if this segment order is crucial in your previous source code. See Section 5.2.1 for an explanation of segment order.

- Some early versions of the Macro Assembler did not have strict type checking. Later versions had strict type checking that produced errors on source code that would have run under the earlier versions. **MASM** solves this incompatibility by making type errors into warning messages. You can set the warning level so that type warnings will not be displayed, or you can modify the code so that the type is clear. Section 9.5 describes strict type checking and how to modify source code developed without this feature.

The programs in the Microsoft Macro Assembler package are compatible with Microsoft (and most IBM) high-level languages. An exception occurs when the current version of **LINK** is used with IBM COBOL 1.0, IBM FORTRAN 2.0, or IBM Pascal 2.0. If source code developed with these compilers has overlays, you must use the linker provided with the compiler. Do not use the new version of **LINK** provided with the assembler.

Appendix B
Error Messages and Exit Codes

This appendix lists and explains the messages and exit codes that can be generated by **MASM** and **CREF**.

Messages are sent to the standard output device. By default, this device is the screen, but you can redirect the messages to a file or to a device, such as a printer.

B.1 MASM Messages and Exit Codes

The assembler can display several kinds of messages, and it outputs an exit code that varies depending on whether errors were encountered during the assembly.

B.1.1 Assembler Status Messages

After every assembly, **MASM** reports on the symbol space, errors, and warnings. A sample display is shown below:

```
Microsoft (R) Macro Assembler Version 5.00
Copyright (C) Microsoft Corp 1981, 1987.  All rights reserved.


  47904 + 353887 Bytes symbol space free

      0 Warning Errors
      0 Severe  Errors
```

The first line indicates how much near and far symbol space was unused during the assembly. This data may help you determine whether increasing the size of your program will exhaust available memory.

The first number indicates near symbol space. There is 64K total. Most symbols go into near space if there is room for them. When near space is exhausted, symbols go into far space. This causes a significant decrease in assembly speed.

The second number indicates far symbol space. This is equal to the amount of available memory minus near data space minus the size of MASM and its file buffers.

You can use the **/V** option to direct **MASM** to display the addition statistics. The number of source lines, the total number of source- and include-file lines, and the number of symbols are shown. This information appears only if no severe errors are encountered. An example is shown below:

```
742 Source  Lines
799 Total   Lines
 44 Symbols
```

The **/T** option can be used to suppress all output to the screen after assembly.

B.1.2 Numbered Assembler Messages

The assembler displays messages on the screen whenever it encounters an error while processing a source file. It displays a warning message whenever it encounters an instance of questionable syntax. Messages that can be associated with a particular line of code are numbered. General errors that are related to the entire assembly rather than a particular line are unnumbered (see Section B1.3).

Numbered error messages are displayed in the following format:

sourcefile(*line*) **:** *code***:** *message*

The *sourcefile* is the name of the source file where the error occurred. If the error occurred in a macro in an include file, the *sourcefile* will be the file where the macro was called and expanded, not the file where it was defined.

The *line* indicates the point in the source file where **MASM** was no longer able to assemble.

The *code* is an identifying code in the format used by all Microsoft language programs. It starts with the word "error" or "warning" followed by a five-character code. The first character is a letter indicating the program or language. Assembler messages start with A. The first digit is the warning level. The next three digits are the error number. For example, severe error 36 is shown as `A0036`.

The *message* is a descriptive line describing the error.

MASM messages are listed in numerical order in this section with a short explanation for each.

Note

Some numbers in sequence are not assigned messages because errors that could be generated in previous versions of **MASM** have been removed or reorganized in this version.

Code	Message
0	`Block nesting error` Nested procedures, segments, structures, macros, or repeat blocks are not properly terminated. This error may indicate that you closed an outer level of nesting with inner levels still open.
1	`Extra characters on line` This occurs when sufficient information to define a statement has been received on a line, but superfluous characters beyond the statement are received.
2	`Internal error - Register already defined` Note the conditions when the error occurs and contact Microsoft Corporation using the Product Assistance Report at the end of this manual.
3	`Unknown type specifier` **MASM** does not recognize the type specifier used to give the size of a label or external declaration. For example, **BYTE** or **NEAR** might be misspelled.
4	`Redefinition of symbol` If a symbol is defined in two places with different types, this error occurs during Pass 1 on the second declaration of the symbol.

5 `Symbol is multi-defined`

If a symbol is defined in two places, this error occurs during Pass 2 on each declaration of the symbol.

6 `Phase error between passes`

An ambiguous instruction or directive caused the relative address of a label to be changed between Pass 1 and Pass 2. You can use the **/D** option to produce a Pass 1 listing to aid in resolving phase errors between passes. See Section 2.5.7.

7 `Already had ELSE clause`

More than one **ELSE** clause was used within a conditional assembly block. Each nested **ELSE** must have its own **IF** directive and **ENDIF**.

8 `Not in conditional block`

An **ENDIF** or **ELSE** is specified without a corresponding **IF** directive.

9 `Symbol not defined`

A symbol was used without being defined. This error is produced for forward references on the first pass, and will be ignored if the references are resolved on the second pass. See Section 2.5.7.

10 `Syntax error`

A statement does not match any recognizable assembler syntax. **MASM** tries to be specific, so this error will only occur if the statement bears no resemblence to any legal statement.

11 `Type illegal in context`

The type specified is of an unacceptable size. For example, a procedure was defined (using **PROC**) as having **BYTE** type, instead of **NEAR** or **FAR** type.

12 `Group name must be unique`

A name assigned as a group name was already defined as another type of symbol.

13 `Must be declared in pass 1:` *symbol*

An item was referenced before it was defined in Pass 1. For example, IF DEBUG is illegal if the symbol DEBUG was not previously defined.

14 `Illegal public declaration`

A symbol was declared **PUBLIC** illegally. For example, a text equate cannot be declared public. See Section 8.1.

15 `Symbol already different kind:` *symbol*

A symbol was redefined to a different kind of symbol. For example, a segment name was reused as a variable name, or a structure name was reused as an equate name.

16 `Symbol is reserved word`

An assembler key word was used as a symbol. This is a warning, not an error, and can be ignored if you wish. However, the key word will no longer be available for its original purpose. For example, if you name a macro add, the **ADD** instruction will no longer be available.

17 `Forward reference is illegal`

A symbol was referenced before it was defined in Pass 1. For example, the following lines produce an error:

```
        DB      count DUP(?)
count   EQU     10
```

The statements would be legal if the lines were reversed.

18 `Must be register:` *operand*

A register was expected as an operand, but a symbol or constant was supplied.

20 `Must be segment or group`

A segment or group name was expected, but some other kind of operand was given. For example, the **ASSUME** directive requires that the symbol assigned to a segment register be a segment name, a group name, a **SEG** expression, or a text equate representing a segment or group name. Thus the following statement is accepted:

```
                ASSUME   ds:SEG variable   ; Legal
```

However, if the same statement is assigned to an equate, it is not accepted, as shown below:

```
segvar          EQU      SEG variable
                ASSUME   ds:segvar          ; Illegal
```

22 `Must be NEAR or FAR`

An operand was expected to be a distance specifier, such as **NEAR**, **FAR**, or **PROC**, but some other kind of operand was received.

23 `Already defined locally`

A symbol that had already been defined within the current module was declared **EXTRN**.

24 `Segment parameters are changed`

A segment declaration with the same name as a previous segment declaration was given with arguments that did not match the previous declaration. See Section 5.2.

25 `Not proper align/combine type`

SEGMENT parameters are incorrect. Check the align and combine types to make sure you have entered valid types from among those discussed in Section 5.2.

26 `Reference to multi-defined symbol`

The instruction references a symbol that has been defined in more than one place.

27 `Operand was expected`

The assembler expected an operand, but received an operator.

28 `Operator was expected`

The assembler expected an operator, but received an operand.

29 `Division by 0 or overflow`

An expression results in division by 0 or in a number larger than can be represented.

30 `Shift count is negative`

A expression using the **SHR** or **SHL** operator evaluated to a negative shift count.

31 `Operand types must match`

An instruction received operands of different sizes. For example, the warning is generated by the following code:

```
                .DATA
string          DB          "This is a test"
                .CODE
                mov         ax,string[4]
```

Since this is a warning rather than an error, **MASM** will attempt to generate code based on its best guess of the intended result. If one of the operands is a register, the register size will override the size of the other operand. In the example, the word size of **AX** would override the byte size of `string[4]`. You can avoid this warning and make your code less ambiguous by specifying the operand size with the **PTR** operator. For example:

```
                move        ax,WORD PTR string[4]
```

32 `Illegal use of external`

An external variable was used incorrectly. See Section 8.2 for information about correct definition and use of external symbols.

34 `Must be record or field name`

An operand was expected to be a record name or field name, but another kind of operand was received.

35 `Operand must have size`

An operand was expected to have a specified size, but no size was supplied. For example, the following statement is illegal:

```
inc     [bx]
```

Often this error can be remedied by using the **PTR** operator to specify a size type, as shown below:

```
inc     BYTE PTR [bx]
```

36 `Must be var, label or constant`

An operand was expected to be a variable, label, or constant, but some other type of operand was received.

38 `Left operand must have segment`

The left operand of a segment-override expression must be a segment register, group, or segment name. For example, if mem1 and mem2 are variables, the following statement is illegal:

```
mov     dx,mem1:mem2
```

39 `One operand must be constant`

The addition operator was used incorrectly. For example, two memory operands cannot be added in an expression. See Section 9.2.1.1.

40 `Operands must in same segment, or one must be constant`

The subtraction operator was used incorrectly. For example, a memory operand in the code segment cannot be subtracted from a memory operand in the data segment. See Section 9.2.2.1.

42 `Constant was expected`

A constant operand was expected, but an operand or expression that does not evaluate to a constant was supplied.

43 `Operand must have segment`

The **SEG** operator was used incorrectly. For example, a constant operand cannot have a segment. See Section 9.2.4.5 for a description of valid uses of the **SEG** operator.

44 `Must be associated with data`

A code-related item was used where a data-related item was expected.

45 `Must be associated with code`

A data-related item was used where a code-related item was expected.

46 `Already have base register`

More than one base register was used in an operand. For example:

```
mov     ax,[bx+bp]
```

47 `Already have index register`

More than one index register was used in an operand. For example:

```
mov     ax,[si+di]
```

48 `Must be index or base register`

An indirect memory operand requires a base or index register, but some other register was specified. For example:

```
mov     ax,[bx+ax]
```

Only **BP**, **BX**, **DI**, and **SI** may be used in indirect operands (except with 32-bit registers on the 80386).

49 `Illegal use of register`

A register was used in an illegal context. For example, the following statement is illegal:

```
mov     ax,cs:[si]
```

50 `Value is out of range`

A value is too large. For example,

```
mov     al,5000
```

is illegal; you must use a byte value for a byte register.

51 `Operand not in current CS ASSUME segment`

An operand represents a code address that is not in the code segment currently assigned with the **ASSUME** statement. This usually indicates a call or jump to a label outside the current code segment.

52 `Improper operand type`

An illegal operand is given for a particular context. For example

```
mov     mem1,mem2
```

is illegal if both operands are memory operands.

53 `Conditional jump out of range by` *number* `bytes`

A conditional jump is not within the required range of 128 bytes backward or 127 bytes forward from the start of the instruction following the jump instruction. You can usually correct the problem by reversing the condition of the conditional jump and using an unconditional jump (**JMP**) to the out-of-range label, as described in Section 9.4.1.

55 `Illegal register value`

56 `No immediate mode`

An immediate operand was supplied to an instruction that cannot use immediate data. For example, the following statement is illegal:

```
mov     ds,DGROUP
```

You must move the segment address into a general register and then move it from that register to **DS**.

57 `Illegal size for item`

The size of an operand is illegal with the specified instruction. For example, you cannot use a shift or rotate instruction with a doubleword (except on the 80386). Since this is a warning rather than an error, **MASM** does assemble code for the instruction, making a reasonable guess at your intention. For example, if the statement

```
inc     mem32
```

is given where mem32 is a doubleword memory operand, **MASM** actually only increments the low-order word of the operand, since a word is the largest operand that can be incremented (except on the 80386). This error may occur if you try to assemble source code written for assemblers that have less strict type checking than the Microsoft Macro Assembler (such as early versions of the IBM Macro Assembler). Usually you can solve the problem by specifying the size of the item with the **PTR** operator. See Section 9.5.

58 `Byte register is illegal`

A byte register was used in a context where a word register is required. For example, `push al` is illegal; use `push ax`.

59 `Illegal use of CS register`

The **CS** register was used in an illegal context, such as those listed below:

```
pop     cs
mov     cs,ax
```

60 `Must be AX or AL`

A register other than **AL**, **AX**, or **EAX** was supplied in a context where only the accumulator register is acceptable. For example, the **IN** instruction requires the accumulator register as its left (destination) operand.

61 `Improper use of segment register`

A segment register was used in a context where it is illegal. For example, `inc cs` is illegal.

62 `Missing or unreachable code segment`

A jump was attempted to a label in a segment that **MASM** does not recognize as a code segment. This usually indicates that there is no **ASSUME** statement associating the **CS** register with a segment.

63 `Operand combination illegal`

Two operands were used with an instruction that does not allow the specified combination of operands. For example, the following operand combination is illegal:

```
xchg    mem1,mem2
```

64 `Near JMP/CALL to different code segment`

A near jump or call instruction attempted to access an address in a code segment other than the one used in the currently active **ASSUME**. To correct the error, use a far call or jump, or use an **ASSUME** statement to change the code segment currently referenced by **CS**. See Section 5.4.

65 `Label cannot have segment override`

A segment override was used incorrectly. See Section 9.2.3 for examples of valid uses of the segment override operator.

66 `Must have instruction after prefix`

A repeat prefix such as **REP**, **REPE**, **REPNE**, **REPZ**, or **REPNZ** was given without specifying the instruction to repeat.

67 `Cannot override ES segment`

A segment override was used on the destination of a string instruction. Although the default **DS**:**SI** register pair for the source can have a segment override, the destination must always be in the **ES**:**DI** register pair. The **ES** segment cannot be overridden. For example, the following statement is illegal:

```
rep     stos ds:destin     ; Can't override ES
```

68 `Cannot address with segment register`

A statement tried to access a memory operand, but no **ASSUME** directive had been used to specify a segment for the operand. See Section 5.4.

69 `Must be in segment block`

A directive (such as **EVEN**) that is expected to be in a segment is used outside a segment.

70 `Cannot use EVEN or ALIGN in byte-aligned segment`

The **EVEN** or **ALIGN** directive was used in a segment that is byte aligned. See Section 6.4.

71 `Forward reference needs override or FAR`

A far label is used in a call or jump to a label that has not been declared far.

72 `Illegal value for DUP count`

The count value specified for a **DUP** operator did not evaluate to a constant integer greater than zero.

73 `Symbol is already external`

A symbol that had already been defined as external was later defined locally. See Section 8.2.

74 `DUP is too large for linker`

DUP operators were nested to more than 17 levels.

75 `Illegal use of undefined operand (?)`

The undefined operand (**?**) was used incorrectly. For example, the following statements are illegal:

```
stuff    DB     5 DUP (?+5) ; Can't use in expression
         mov    ax,?        ; Can't use in code
```

Valid uses of the undefined operand are explained in Section 6.2.2.

76 `Too many values for structure or record initialization`

Too many initial values were given when declaring a record or structure variable. The number of values in the declaration must match the number in the definition. For example, a structure `test` defined with four fields could be declared as shown below:

```
stest        test    <4,,'c',0>
```

The declaration must have four or fewer fields.

77 `Brackets required around initialized list`

A structure variable was defined without angle brackets around the initial values in the list. For example, the following definition is illegal:

```
stest       test    4,,'c'O
```

The following definitions are correct:

```
stest    test   <4,,'c',0> ; Three initial values, one blank
ttest    test   <>         ; No initial values
```

78 `Directive illegal in STRUC`

All statements within structure definitions must either be one of three kinds of statements: data definitions using define directives such as **DB** or **DW**; comments preceded by a semicolon; or conditional assembly directives.

79 `Override with DUP is illegal`

The **DUP** operator was used in a structure initialization list. For example, the following example is illegal because of the **DUP** operator:

```
stest       test    <3,4 DUP (3),5>
```

80 `Field cannot be overridden`

An item in a structure initialization list attempted to override a structure field that could not be overridden. For example, if a field is initialized in the structure definition with the **DUP** operator, it cannot be overridden in a declaration. See the note in Section 7.1.2.

81 `Override is of wrong type`

An item in a structure initialization list attempted to override a structure field that was initialized to a different size. For example, a string constant of more than two characters cannot override a field that does not have byte size.

83 `Circular chain of EQU aliases`

An alias **EQU** eventually points to itself. For example, the following lines are illegal:

```
a            EQU    b
b            EQU    a
```

84 `Cannot emulate coprocessor opcode`

Either a coprocessor instruction, or operands used with such an instruction, produce an opcode that the coprocessor emulator cannot support. Since the emulator library is not supplied with the Microsoft Macro Assembler, this error should not occur unless you are linking assembler routines with code from a high-level language compiler that does use the emulator.

85 `End of file, no END directive`

The source code was not terminated by an **END** statement. This error can also occur as the result of segment nesting errors.

86 `Data emitted with no segment`

A statement that generates code was used outside all segment blocks. For example, all instructions and data declarations must be in segments. Directives that specify assembler behavior but do not generate code or data can be outside segments.

87 `Forced error - pass1`

An error was forced with the **.ERR1** directive.

88 `Forced error - pass2`

An error was forced with the **.ERR2** directive.

89 `Forced error`

An error was forced with the **.ERR** directive.

90 `Forced error - expression equals 0`

An error was forced with the **.ERRE** directive.

91 `Forced error - expression not equal 0`

An error was forced with the **.ERRNZ** directive.

92 `Forced error - symbol not defined`

An error was forced with the **.ERRNDEF** directive.

93 `Forced error - symbol defined`

An error was forced with the **.ERRDEF** directive.

94 `Forced error - string blank`

An error was forced with the **.ERRB** directive.

95 `Forced error - string not blank`

An error was forced with the **.ERRNB** directive.

96 `Forced error - strings identical`

An error was forced with the **.ERRIDN** directive.

97 `Forced error - strings different`

An error was forced with the **.ERRDIF** directive.

98 `Override value is wrong length`

The override value for a structure field is too large to fit in the field. An example is shown below:

```
x           STRUC
x1          DB      "A"
x           ENDS

y           x       <"AB">
```

The override value is a string consisting of 2 bytes, while the structure declaration only provided room for 1 byte.

99 `Line to long expanding` *symbol*

An equate name defined with an **EQU** or equal-sign (=) directive is so long that expanding it causes the assembler's internal buffers to overflow. This message may indicate a recursive text macro.

100 `Impure memory reference`

The code contains an attempt to store data into the code segment when the **.PRIV** directive is used with the **.286** or **.386** directive and the **/P** option is in effect. An example of storing code to the code segment is shown below:

```
               .CODE
c_word         DW       ?
               .
               .
               .
               mov      cs:c_word,data
```

The **/P** option checks for such statements, which are acceptable in nonprotected mode, but can cause problems in protected mode.

101 `Missing data; zero assumed`

An operand is missing from a statement. For example:

```
mov     ax,
```

Since this is a warning, **MASM** attempts to assemble the code by assuming that a 0 was intended. Thus

```
mov     ax,0
```

is assembled.

102 `Segment near (or at) 64K limit`

A bug in the 80286 processor causes jump errors when a code segment approaches within a few bytes of the 64K limit. This error warns about code that may fail because of the bug. The error can only be generated when the **.286** directive is given.

103 `Cannot change processor after first segment`

A processor directive was given after the first segment definition. Only one processor directive and one coprocessor directive are allowed per source file, and they must be specified before the first segment definition. If no directive is used, the defaults (**.8086** and no coprocessor directive) are assumed.

104 `Operand size does not match segement size`

A 32-bit operand was used in a 16-bit segment, or vice versa. This warning can only occur with the 80386. For example, the following statement is questionable practice in a 32-bit segment:

```
mov   ax,OFFSET nearlabel ; Load near (32-bit) label
```

The following statement is questionable practice in a 16-bit segment:

```
mov   eax,OFFSET farlabel ; Load far (48-bit) label
```

This is a warning that you can ignore if you are certain you know what you are doing.

105 `Address size does not match segment size`

A 32-bit address was used in a 16-bit segment, or vice versa. This warning can only occur with the 80386. For example, the following statement is questionable practice in a 32-bit segment:

```
mov   eax,[si]   ; Load value pointed to by 16-bit pointer
```

The following statement is questionable practice in a 16-bit segment:

```
mov   ax,[esi]   ; Load value pointed to by 32-bit pointer
```

This is a warning that you can ignore if you are certain you know what you are doing.

106 `Jump shortened. NOP inserted`

A **JMP** instruction was used to jump to a short label (one 128 or fewer bytes before the end of the **JMP** instruction, or 127 or fewer bytes beyond the instruction). By default the assembler assumes that jumps are near (greater than short, but still in one segment). If a short jump is encountered, **MASM** pads the object file with the **NOP** instruction. You can make your code slightly more efficient by using the **SHORT** operator to specify that a jump is short rather than near. This elimates unnecessary **NOP** instructions. For example using the **SHORT** operator in the following example saves one byte of code:

```
jmp     SHORT there
```

```
.
there:    .                ; Less than 127 bytes
```

Using the **SHORT** operator with forward references to code labels is explained in Section 9.4.1.

107 `Align must be power of 2`

A number that is not a power of 2 was used with the **ALIGN** directive. See Section 6.4.

108 `Expected` *element*

An element such as a punctuation mark or operator was omitted. For example, if you omit the comma between source and destination operands, the message `Expected comma` will be generated.

109 `Line too long`

A source line is longer than 128 characters, the maximum allowed by **MASM**.

110 `Illegal digit in number`

A constant number contained a digit that is not allowed in the current radix.

111 `Empty string not allowed`

A statement uses an empty string. For example, the following definition is illegal:

```
null        DB        ""
```

In many languages an empty string represents ASCII character 0. In assembler, you must give the value 0, as shown below:

```
null        DB        0
```

112 `Missing operand`

The instruction or directive requires more operands than were provided.

113 `Open parenthesis or bracket`

Only one parenthesis or bracket was given in a statement that requires opening and closing parentheses or brackets.

114 `Directive must be in macro`

A directive that is expected only in macro definitions was used outside a macro.

115 `Unexpected end of line`

A line end before a complete statement was formed. **MASM** expects more information, but can't identify what information is missing.

B.1.3 Unnumbered Error Messages

Unnumbered messages appear when an error occurs that cannot be associated with a particular line of code. Generally these errors indicate problems with the command line, memory allocation, or file access. **MASM** may generate the following unnumbered error messages:

- **File-Access Errors**

Any of the following errors may occur when **MASM** tries to access a file for processing. They usually indicate insufficient disk space, a corrupted file, or some other file error.

`Include file` *filename* `not found`

`End of file encountered on input file`

`Write error on object file`

`Write error on listing file`

`Write error on cross-reference file`

`Unable to open cref file` *filename*

`Unable to open input file` *filename*

`Unable to access input file` *filename*

`Unable to open listing file` *filename*

`Unable to open object file` *filename*

`Read error on standard input`

■ Command-Line Errors

Any of the following errors may occur if you give an invalid command line when starting **MASM**.

`Extra file name ignored`

`Line invalid, start again`

`Error defining symbol` *name* `from command line`

`Buffer size expected after B option`

`Path expected after I option`

`Unknown case option:` *option*

`Unknown option:` **option**

■ Miscellaneous Errors

The following errors indicate a problem with memory allocation or some other assembler problem that is not related to a specific source line.

`Out of memory`

All available memory has been used, either because the source file is too long, or because there are too many symbols defined in the symbol table. There are several things you can do to resolve this problem. First, try assembling with only an object file. If this works, you can reassemble specifying a null object file to get a listing or cross-reference file. You can also rewrite the source file to take up less symbol space. Techniques for reducing symbol space include: minimizing use of macros, structures, and the **EQU** and equal-sign (=) directives;

using short symbol names; using tab characters in macros rather than series of spaces; using macro comments (;;) rather than normal comments (;); and purging macro definitions after the last use.

`Open segments`

A segment was defined, but never terminated with an **ENDS** directive. This error will not occur with simplified segment directives.

`Open procedures`

A **PROC** directive was given without a corresponding **ENDP** directive.

`Number of open conditionals: <number>`

Conditional-assembly directives (starting with **IF**) were given without corresponding **ENDIF** directives.

`Internal assembler error`

Note the conditions when the error occurs and contact Microsoft Corporation using the Product Assistance Report at the end of this manual.

`Internal error - Problem with expression analyzer`

This problem may indicate an expression that **MASM** does not understand. Note the conditions when the error occurs and contact Microsoft Corporation using the Product Assistance Report at the end of this manual.

`Internal unknown error`

This error may indicate that the internal error table has been corrupted and **MASM** can't figure out what the error is. Note the conditions when the error occurs and contact Microsoft Corporation using the Product Assistance Report at the end of this manual.

`Internal assembler error`

Note the conditions when the error occurs and contact Microsoft Corporation using the Product Assistance Report at the end of this manual.

B.1.4 MASM Exit Codes

The assembler returns one of the following codes after an assembly. The codes can be tested by a make file or batch file.

Code	Meaning
0	No error
1	Argument error
2	Unable to open input file
3	Unable to open listing file
4	Unable to open object file
5	Unable to open cross-reference file
6	Unable to open include file
7	Assembly error
8	Memory-allocation error
10	Error defining symbol from command line (**/D**)
11	User interrupted

Note that if the exit code is 7, **MASM** automatically deletes the invalid object file.

B.2 CREF Error Messages and Exit Codes

The Microsoft Cross-Reference Utility, **CREF**, terminates operation and displays one of the following messages when it encounters an error:

`Can't open cross-reference file for reading`

The **.CRF** file is not found. Make sure the file is on the specified disk and that the name is spelled correctly in the command line.

`Can't open listing file for writing`

May indicate that the disk is full or write protected, that a file with the specified name already exists, or the specified device is not available.

`CREF has no options`

You specified an option in the command line with the slash (/) or dash (-) character, but **CREF** has no options.

`Extra file name ignored`

You specified more than two files in the file name. **CREF** will create the reference file using only the first two files given.

`Line invalid, start again`

No **.CRF** file was provided in the command line or at the prompt. **CREF** will display this message followed by a prompt asking for a **.CRF** file.

`Out of heap space`

CREF cannot find enough memory to process the files. Try again with no resident programs or shells, or add more memory.

`Premature EOF`

You specified a file that is not a valid **.CRF** file, or the file is damaged.

`Read error on standard input`

This error only occurs if the program receives a CONTROL-Z from the keyboard or from a redirected file.

CREF only returns two exit codes: 0 if the program is successful, or 1 if an error occurs.

Index