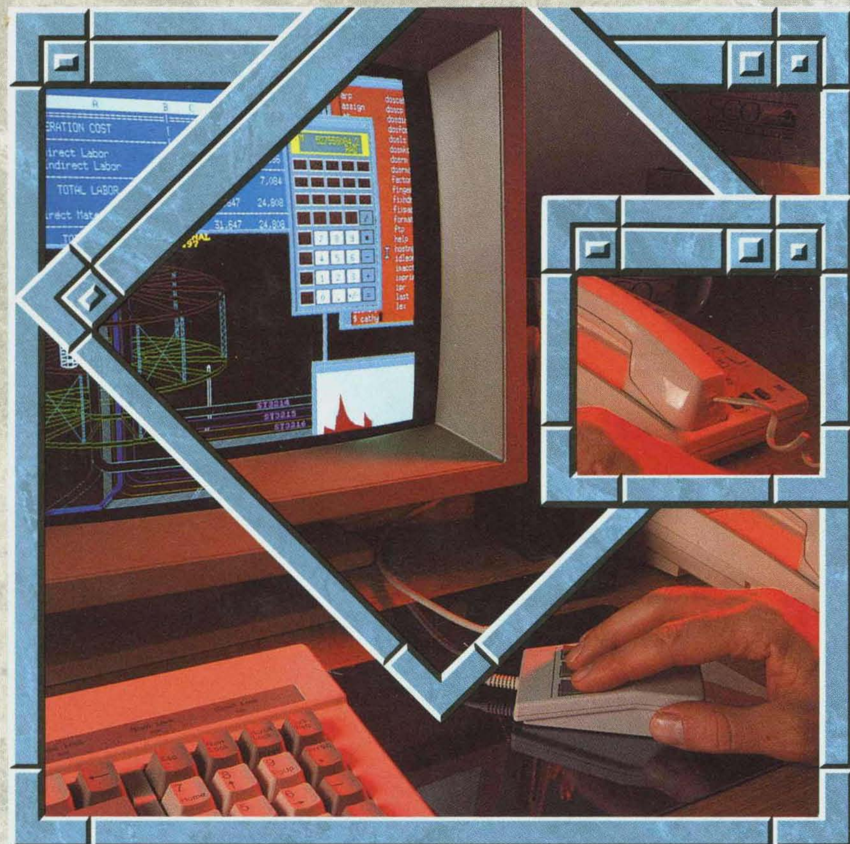
OPEN DESKTOP™

Open Desktop® Development System

CodeView® MASM

OPEN DESKTOP™



The Complete Graphical Operating System

SCO UNIX® System V/386

Development System

The CodeView Debugger

and

The Macro Assembler User's Guide

SCO Document Number: 6-26-89-6.0/3.2.0

SCO UNIX® System V/386

Development System

The CodeView Debugger

The Santa Cruz Operation, Inc.

SCO Document Number: 6-26-89-6.0/3.2.0

Contents

12 Using CodeView System-Control Commands

Chapter 1

Introduction

Introduction

Welcome to the CodeView® debugger. This is an executable program that helps you debug software written with the C and Macro Assembler languages.

The CodeView debugger is a window-oriented tool that enables you to track down logical errors in programs; it allows you to analyze a program as the program is actually running. The CodeView debugger displays source code or assembly code, indicates which line is about to be executed, dynamically watches the values of variables (local or global), switches screens to display program output, and performs many other related functions. The debugger can be easily learned and used, by assembly and high-level language programmers alike.

To use CodeView, you first create an executable file from compiled object files. (When a program is made into an executable file, it is in the form that can be loaded and executed by the system.) This executable file must be compiled and linked with the correct options so that it contains the line-number information and a symbol table needed by CodeView. You can use the C compiler, or **cc**, which calls the linking program, **ld**, The correct options for compiling and linking for use with CodeView are described in Chapter 2, "Getting Started."

1

About this Manual

This manual explains the use of the CodeView debugger. Commands, display, and interface of the debugger are presented here.

The manual is comprised of the following chapters:

- Chapter 2, ''Getting Started,'' explains how to create a C or assembly program that can be run with the CodeView debugger; it also explains how to start the debugger and select various command-line options.

- Chapter 3, ''The CodeView Display,'' discusses the CodeView display screen and interface, including function keys and keyboard commands.

- Chapter 4, ''Using Dialog Commands,'' presents the general form of CodeView commands.

- Chapter 5, ''CodeView Expressions,'' describes how to build complex expressions for use in commands.

- Chapter 6, ''Executing Code,'' explains the CodeView commands that execute code from within a program.

- Chapter 7, ''Examining Data and Expressions,'' discusses several data-evaluation commands.

- Chapter 8, ''Managing Breakpoints,'' explains how to use breakpoints to suspend execution.

- Chapter 9, ''Managing Watch Statements,'' describes the use of watch statement commands to set, delete, and list watch statements.

- Chapter 10, ''Examining Code,'' discusses several commands that let you examine program code or data related to code.

- Chapter 11, ''Modifying Code or Data,'' explains how to alter code temporarily for testing in the CodeView debugger.

- Chapter 12, ''Using CodeView System-Control Commands,'' discusses commands that control the operation of the CodeView debugger.

Chapter 2

Getting Started

Introduction

Getting started with the CodeView debugger requires several simple steps. First you must prepare a special-format executable file for the program that you wish to debug; then you can invoke the debugger. You may also wish to specify options that affect the debugger's operation.

This chapter describes how to produce executable files in the CodeView format using C or assembly language, and how to load a program into the CodeView debugger. This chapter lists restrictions and programming considerations with regard to the debugger, which you may want to consult before compiling or assembling. Finally, this chapter describes how to use the debugger with the Macro Assembler.

Restrictions

You cannot use the CodeView debugger to debug source code in include files. This restriction applies generally to the use of the CodeView debugger, regardless of the language being used.

Preparing Programs for the CodeView Debugger

You must compile and link with the correct options, in order to use a program with the CodeView debugger. These options direct the compiler and the linker to produce an executable file, which contains line-number information and a symbol table, in addition to the executable code.

Note

For the sake of brevity, this section and its three subsections use the term ''compiling'' to refer to the process of producing object modules. However, almost everything said about compiling in this section applies equally well to assembling. Exceptions are noted in the section ''Preparing Assembly Programs'' in this chapter.

Not all compiler and linker versions support CodeView options. Consult the specific language documentation for information about compiler versions. If you try to debug an executable file that was not compiled and linked with CodeView options, or if you use a compiler that does not support these options, then you are only able to use the debugger in assembly mode. This means that the CodeView debugger does not display source code or understand source-level symbols, such as symbols for functions and variables.

The two CodeView basic display modes are source mode, in which the program is displayed as source lines, and assembly mode, in which the program is displayed as assembly-language instructions. These two modes can be combined in mixed mode, in which the program is displayed with both source lines and assembly-language instructions.

Programming Considerations

Any source code that is legal in C, or Macro Assembler can be compiled or assembled to create an executable file, and then debugged with the CodeView debugger. However, some programming practices make debugging more difficult.

The C and Macro Assembly languages permit you to put code in separate include files, and to read the files into your source file by using an include directive. However, you cannot use the CodeView debugger to debug source code in include files. The preferred method of developing programs is to create separate object modules, and then link the object modules with your program. The CodeView debugger supports the debugging of separate object modules in the same session.

Also, the CodeView debugger is more effective and easier to use if you put each source statement on a separate line. A number of languages permit you to place more than one statement on a single line of the source file. This practice does not prevent the CodeView debugger from functioning. However, the debugger must treat the line as a single unit; it cannot break the line down into separate statements. Therefore, if you have three statements on the same line, you cannot put a breakpoint or freeze execution on the individual statements. The best you are able to do is freeze execution at the beginning of the three statements, or at the beginning of the next line.

The C and Macro Assembly languages support a type of macro expansion. However, the CodeView debugger does not help you debug macros in source mode. You need to expand the macros yourself before debugging them; otherwise, the debugger treats them as simple statements or instructions.

CodeView Compile Options

When you compile a source file for a program you want to debug, you must specify the **-Zi** option on the command line. The **-Zi** option instructs the compiler to include line-number and symbolic information in the object file. You can also use **-g**, which is synonymous with **-Zi**.

If you do not need complete symbolic information in some modules, you can compile those modules with the **-Zd** option instead of **-Zi**. The **-Zd** option writes less symbolic information to the object file, so using this option saves disk space and memory. For example, if you are working on a program made up of five modules, but only need to debug one module, you can compile that module with the **-Zi** option and the other modules with the **-Zd** option. You are able to examine global variables and see source lines in modules compiled with the **-Zd** option, but local variables are unavailable.

In addition, if you are working with a high-level language, you probably want to use the **-Od** option, which turns off optimization. Optimized code may be rearranged for greater efficiency and, as a result, the instructions in your program may not correspond closely to the source lines. After debugging, you can compile a final version of the program with the optimization level you prefer.

Note

The **-Od** option has no effect when used with the Macro Assembler.

You cannot debug a program until you compile it successfully. The CodeView debugger cannot help you correct syntax or compiler errors. Once you successfully compile your program, you can then use the debugger to locate logical errors in the program.

Compiling examples are given in the sections below on compiling and linking with specific languages.

CodeView Link Options

If you use **ld** separately to link an object file or files for debugging, you should specify the **-g** option. This option instructs the linker to incorporate addresses for symbols and source lines into the executable file.

Note that if you use a driver program that automatically invokes the linker (such as **cc** with C), then the linker is automatically invoked with the **-g** option whenever you specify **-Zi** on the command line.

Although executable files prepared with the **-g** option can be executed from the command line like any other executable files, they are larger because of the extra symbolic information in them. To minimize program size, you may want to use the **strip** command or recompile and link your final version without the **-Zi** option when you finish debugging a program. See the *Programmer's Reference* for information about the **strip** command.

Linking examples are given in the sections below on compiling and linking C and assembly language programs.

Preparing C Programs

In order to use the CodeView debugger with a program written in C, you need to compile it with the C Compiler. Early versions of the compiler do not support the CodeView compile options. Please see the *Development System Release Notes* for more information.

2

Writing C Source Code

The C language supports the use of include files, through the use of the **#include** directive. However, you cannot debug source code put into include files. Therefore, you should reserve the use of include files for **#define** macros and structure definitions.

The C language permits you to put more than one statement on a line. This practice makes it difficult for you to debug such lines of code. For example, the following code is legal in C:

```
code = buffer[count]; if (code == '\n') ++lines;
```

This code is made up of three separate source statements. When placed on the same line, the individual statements cannot be accessed during debugging. You could not, for example, stop program execution at *++lines;*. The same code would be easier to debug in the following form:

```
code = buffer[count];
if (code == '\n')
        ++lines;
```

This makes code easier to read and corresponds with what is generally considered good programming practice.

You cannot easily debug macros with the CodeView debugger. The debugger cannot break down the macro for you. Therefore, if you have complex macros with potential side effects, you may need to write them first as regular source statements.

Compiling and Linking C Programs

The **-Zi**, **-Zd**, and **-Od** options are all supported by the C Compiler. (For a description of these options, see the section "CodeView Compile Options.") The options are accepted by the **cc** driver.

The CodeView debugger supports mixed-language programming. For an example of how to link a C module with modules from other languages, see the section ''Preparing Assembly Programs'' in this chapter.

Examples

```
cc -Zi -Od -o example example.c

cc -c -Zi -Od example.c
cc -g -o example example.o

cc -Zi -Od -c mod1.c
cc -Zd -Od -c mod2.c
cc -Zi mod1.o mod2.o
```

In the first example, **cc** is used to compile and link the source file **example.c** The **cc** command creates an object file in the CodeView format, **example.o**, and then automatically invokes the linker with the **-g** option. The second example demonstrates how to compile and link the source file, **example.c**, by using the **-c** option with **cc**. Since **cc -c** does not invoke the linker, you must enter **cc** a second time to link the object file. These examples result in an executable file, **example**, which has the line-number information, symbol table, and unoptimized code required by the CodeView debugger.

In the third example, the source module **mod1.c** is compiled to produce an object file with full symbolic and line information, while **mod2.c** is compiled to produce an object file with limited information. Then, **cc** is used again to link the resulting object files. (This time, **cc** does not recompile, because the arguments have a **.o** extension.) Typing **-Zi** on the command line causes the linker to be invoked with the **-g** option. The result is an executable file, called **a.out**, in which one of the modules, **mod2.c** is harder to debug. It contains less symbolic information, such as the names of local variables. However, the executable file takes up substantially less space on disk than it would if both modules were compiled with full symbolic information.

Preparing Assembly Programs

In order to use all the features of the CodeView debugger with assembly programs, you need to assemble with Macro Assembler. (The section "Working with Older Versions of the Assembler" in this chapter discusses how to use earlier versions the Macro Assembler with the debugger.)

2

Writing Assembler Source Code

If you have Version 2.3 or later of the Macro Assembler, then you can use the simplified segment directives. Use of these directives ensures that segments are declared in the correct way for use with the CodeView debugger. (These directives also aid mixed-language programming.) If you do not use these directives, then you need to make sure that the class name for the code segment is **CODE**.

You cannot trace through macros while in source mode. Macros are treated as single instructions unless you are in assembly or mixed mode, so you do not see comments or directives within macros. Therefore, you may want to debug code before putting it into a macro.

The Macro Assembler also supports include files, but you cannot debug code in an include file. You are better off reserving include files for macro and structure definitions.

Because the assembler does not have its own expression evaluator, you have to use the the C-expression, evaluator. C is the closest to assembly language. To make sure that the expression evaluator recognizes your symbols and labels, you should observe the following guidelines when you write assembly modules:

- The assembler has no explicit way to declare real numbers. However, it passes the correct symbolic information for reals and integers if you initialize each real number with a decimal point and each integer without a decimal point. (The default type is integer.) For example, the following statements correctly initialize *REALSUM* as a real number and *COUNTER* as an integer:

```
REALSUM     DD     0.0
COUNTER     DD     0
```

You must initialize real number data in data definitions. If you use *?*, then the assembler considers the variable an integer when it generates symbolic information. The CodeView debugger, in turn, does not properly evaluate the value of the variable.

- Avoid the use of special characters in symbol names.
- Assemble with **-Mx** or **-Ml** to avoid conflicts due to case when you do mixed-language programming. By default, the assembler converts all symbols to uppercase when it generates object code. C, however, does not do this conversion. Therefore, the CodeView debugger does not recognize that *var* in a C program and *var* in an assembly program are the same variable, unless you leave Case Sense off when using the debugger.

Assembling and Linking

The assembler supports the **-Zi** and **-Zd** assemble-time options. The **-Od** option does not apply, and so is not supported.

If you link your assembly program with a module written in C (which is case sensitive), you probably need to assemble with **-Mx** or **-Ml**.

After assembling, link with the **-g** option to produce an executable file in the CodeView format.

Examples

```
masm -Zi example.asm
cc -g example.o

masm -Zi mod1.asm
masm -Zd mod2.asm
cc -g mod1.o mod2.o
```

The first example assembles the source file **example.asm** and produces the object file **example.o** which is in the CodeView format. The linker is then invoked by entering **cc** with the **-g** option and produces an executable file, called **a.out**, containing the symbol table and line-number information required by the debugger.

The second example produces the object file **mod1.o** which contains symbol and line-number information, and the object file **mod2.o** which contains line-number information but no symbol table. The object files are then linked. The result is an executable file, called **a.out**, in which the second module is harder to debug. The second module contains less symbolic information, such as the names of local variables. This executable file, however, is smaller than it would be if both modules were assembled with the **-Zi** option.

Starting the CodeView Debugger

Before starting the debugger, make sure all the files it requires are available in the proper places. The following files are recommended for source-level debugging:

File	Location
/usr/bin/cv	The CodeView program file is located in the */usr/bin* directory.
/usr/lib/cv.hlp	The CodeView help file is located in the directory */usr/lib*. If the CodeView debugger cannot find the help file, you can still use the debugger, but you see an error message if you use one of the help commands.
program	The executable file for the program that you wish to debug must be in the current directory or in a directory that you specify by including its pathname when you type the CodeView command line. The CodeView debugger displays an error message and does not start unless the executable file is found.

source.ext (extension depends on language)
Normally, source files should be in the current directory. However, if you specify a file specification for the source file during compilation, that specification becomes part of the symbolic information stored in the executable file. For example, if you compiled with the command line argument *demo.ext*, the CodeView debugger expects the source file to be in the current directory. However, if you compiled with the command line argument with the pathname */source/demo.ext*, then the debugger expects the source file to be in directory */source*. If the debugger cannot find the source file in the directory specified in the executable file (usually the current directory), the program prompts you for a

new directory. You can either enter a new directory, or you can press the <Return> key to indicate that you do not want a source file to be used for this module. If no source file is specified, you must debug in assembly mode.

If the appropriate files are in the correct directories, you can enter the CodeView command line at the command prompt. The command line has the following form:

cv [*options*] *executablefile* [*arguments*]

The *options* are one or more of the options described in the section "Using CodeView Options" in this chapter. The *executablefile* is the name of an executable file to be loaded by the debugger. If you try to load a nonexecutable file, the following message appears:

```
Not an executable file
```

The optional *arguments* are parameters passed to the *executablefile*. If the program you are debugging does not accept command-line arguments, you do not need to pass any arguments.

If the file is not in the CodeView format, the debugger starts in assembly mode and displays the following message:

```
No symbolic information
```

You must specify an executable file when you start the CodeView debugger. If you omit the executable file, the debugger displays a message showing the correct command-line format.

When you give the debugger a valid command line, the executable program and the source file are loaded, the address data are processed, and the CodeView display appears. The initial display is in window mode.

For example, if you wanted to debug the program **benchmrk**, you could start the debugger with the following command line:

```
cv benchmrk
```

If you give this command line, window mode is selected automatically. The display looks like the following screen example:

```
File View Search  Run  Watch    Options    Language    Calls    Help  | F8=Trace  F5=Go
----------------------------| stats.for |------------------------------------------------

1:          /************************************************************************
2:             stats.c
3:
4:                   Calculates simple statistics (minimum, maximum, mean, median,
5:                   variance, and standard deviation) of up to 50 values.
6:
7:
8:
9:
10:         ************************************************************************/
11:
12:
13:
14:              int dat[50], file, n, i;
15:              file=open("datafile", O_RDONLY);
16:
17:              n=0;
18:              for (i=0; i<50; i++)
-----------------------------------------------------------------------------------------
Microsoft (R) CodeView (R)  Version 2.0
(C) Copyright Microsoft Corp. 1986, 1987.  All rights reserved.
Portions (C) Copyright The Santa Cruz Operation, Inc. 1989
>_
```

If sequential mode is selected, the following lines appear:

```
Microsoft (R) CodeView (R)  Version 2.0
(C) Copyright Microsoft Corp. 1986, 1987. All rights reserved.
Portions (C) Copyright The Santa Cruz Operation, Inc. 1989

>
```

You can use CodeView options, as described in the section ''Using CodeView Options'' in this chapter, to override the default start-up mode.

If your program is written in a high-level language, the CodeView debugger is now at the beginning of the start-up code that precedes your program. In source mode, you can enter an execution command (such as Trace or Program Step) to execute automatically through the start-up code to the beginning of your program. At this point, you are ready to start debugging your program, as described in Chapters 4-12.

Using CodeView Options

2

You can change the start-up behavior of the debugger by specifying options in the command line.

An option is a sequence of characters preceded by a dash (-). Unlike compiler command-line options, CodeView command-line options are not case sensitive.

A file whose name begins with a dash must be renamed before you use it with the CodeView debugger, so that the debugger does not interpret the dash as an option designator. You can use more than one option in a command line, but each option must have its own dash, and spaces must separate each option from other elements of the line. The following list suggests some situations in which you might want to use an option. If more than one condition applies, you can use more than one option (in any order). If none of the conditions applies, you need not use any options.

Condition	Option
You have a two-color monitor, a color graphics adapter, and an IBM or IBM-compatible computer.	**-B**
You want the CodeView debugger to automatically execute a series of commands when it starts up.	**-C***commands*
You wish to debug in sequential mode (for example, with redirection).	**-T**

The CodeView options are described in more detail in the following sections.

Starting with a Black-and-White Display

Option

-B

The **-B** option forces the CodeView debugger to display in two colors even if you have a color adapter (CGA, EGA, or compatible). By default, the debugger checks on start-up to see what kind of display adapter is attached to your computer. If the debugger detects an MA, it displays in two colors. If it detects a color adapter, it displays in multiple colors.

If you use a two-color monitor with a CGA or EGA, you may want to disable color. Monitors that display in only two colors (usually green and black, or amber and black) often attempt to show colors with different cross-hatching patterns, or in gray-scale shades of the display color. In either case, you may find the display easier to read if you use the **-B** option to force black-and-white display. Most two-color monitors still have four color distinctions: background (black), normal text, high-intensity text, and reverse-video text.

Example

```
cv -B calc calc.dat
```

The example above starts the CodeView debugger in black-and-white mode. This is the only mode available if you have an MA. The display is usually easier to read in this mode if you have a CGA and a two-color monitor.

Specifying Start-Up Commands

Option

-Ccommands

The **-C** option allows you to specify one or more *commands* that is executed automatically upon start-up. You can use these options to invoke the debugger from a shell script file or **make** file. Each command is separated from the previous command by a semicolon.

If one or more of your start-up commands have arguments that require spaces between them, you should enclose the entire option in double quotation marks. Otherwise, the debugger interprets each argument as a separate CodeView command-line argument rather than as a debugging-command argument.

Note

Any start-up option that uses the less-than (<) or greater-than (>) symbol must be enclosed in single or double quotation marks even if it does not require spaces. This ensures that the redirection command are interpreted by the CodeView debugger rather than by the shell.

Examples

```
cv -CGmain calc calc.dat
```

The example above loads the CodeView debugger with **calc** as the executable file and **calc.dat** as the argument. Upon start-up, the debugger executes the high-level-language start-up code with the command **Gmain**. Since no space is required between the CodeView command (**G**) and its argument (**main**), the option is not enclosed in double quotation marks.

```
cv "-C;S&;G INTEGRAL;DS ARRAYX L 20" calc calc.dat
```

The example above loads the same file with the same argument as the first example, but the command list is more extensive. The debugger starts in mixed source/assembly mode (**S&**). It executes to the routine **INTEGRAL** (**G INTEGRAL**), and then dumps 20 short real numbers, starting at the address of the variable **ARRAYX** (**DS ARRAYX L 20**). Since several of the commands use spaces, the entire option is enclosed in double quotation marks.

```
cv "-C<input.fil" calc calc.dat
```

The example above loads the same file and argument as the first example, but the start-up command directs the debugger to accept input from the file **input.fil** rather than from the keyboard. Although the option does not include any spaces, it must be enclosed in double quotation marks so that the less-than symbol is read by the CodeView debugger rather than by the shell.

Enabling Sequential Mode

Options

-T

The CodeView debugger can operate in window mode or in sequential mode. Window mode displays up to four windows, enabling you to see different aspects of the debugging-session program simultaneously.
M003 You can also use a mouse in window mode. Window mode

requires a console. Sequential mode works with any computer and is useful with redirection commands. Debugging information is displayed sequentially on the screen.

The behavior of each mode is discussed in detail in Chapter 3, "The CodeView Display."

Note

Although window mode is more convenient, any debugging operation that can be done in window mode can also be done in sequential mode.

Examples

```
cv -T sieve
```

The example above starts the debugger in sequential mode. You might want to use this option if you have a specific reason for using sequential mode. For instance, sequential mode usually works better if you are redirecting your debugging output to a remote terminal.

Working with Older Versions of the Assembler

You can run the CodeView debugger with files developed using prior versions of the Macro Assembler. Since older versions do not write line numbers to object files, some of the CodeView debugger's features are unavailable when you debug programs developed with the older assemblers. The following considerations apply, in addition to the considerations mentioned in the section "Preparing Assembly Programs" in this chapter.

The procedure for assembling and debugging executable files by using older versions of the assembler is summarized below.

1. In your source file, declare public any symbols, such as labels and variables, that you want to reference in the debugger. If the file is small, you may want to declare all symbols public.

2. As mentioned earlier, make sure that the code segment has class name *CODE*.

3. Assemble as usual. No special options are required, and all assembly options are allowed.

4. Use **ld**. Refer to the *Development System Release Notes* for information about which version of **ld** to use. Use the **-g** option when linking.

5. Debug in assembly mode (this is the start-up default if the debugger fails to find line-number information). You cannot use source mode for debugging, but you can load the source file into the display window and view it in source mode. Any labels or variables that you declared public in the source file can be displayed and referenced by name instead of by address. However, they cannot be used in expressions because type information is not written to the object file.

Chapter 3

The CodeView Display

Introduction

The CodeView screen display can appear in two different modes—window and sequential. Either mode provides a useful debugging environment, but the window mode is the more powerful and convenient of the two. The CodeView debugger accepts either window commands or dialog commands. Dialog commands are entered as command lines following the CodeView prompt (>) in sequential mode. They are discussed in Chapter 4, ''Using Dialog Commands.''

You probably want to use window mode. In window mode, the pull-down menus and function keys offer fast access to the most common commands. Different aspects of the program and debugging environment can be seen in different windows simultaneously. Window mode is described in the section ''Using Window Mode'' in this chapter.

Sequential mode is sometimes useful when redirecting command input or output. Sequential mode is described in the section ''Using Sequential Mode'' in this chapter.

Using Window Mode

The elements of the CodeView display marked in the figure on the next page include the following:

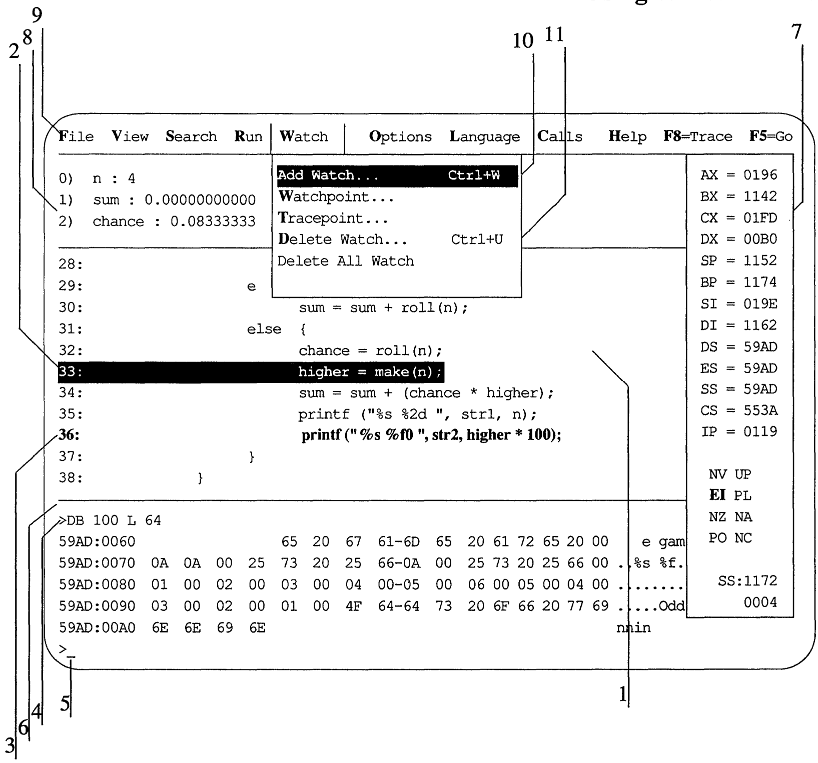
1. The display window shows the program being debugged. It can contain source code (as in the example), assembly-language instructions, or any specified text file.

2. The current location line (the next line the program will execute) is displayed in reverse video or in a different color. This line may not always be visible, because you can scroll to earlier or later parts of the program.

3. Lines containing previously set breakpoints are shown in high-intensity text.

4. The dialog window is where you enter dialog commands. These are the commands with optional arguments that you can enter at the CodeView prompt (>). You can scroll up or down in this window to view previous dialog commands and command output.

5. The cursor is a thin, blinking line that shows the location at which you can enter commands from the keyboard. You can move the cursor up and down, and place it in either the dialog or display window.



6. The display/dialog separator line divides the dialog window from the display window.

7. The register window shows the current status of processor registers and flags. This is an optional window that can be opened or closed with one keystroke. The register window also displays the effective address at the bottom of the window; the effective address shows the actual location of an operand in physical memory. It is useful when debugging in assembly mode.

8. The optional watch window shows the current status of specified variables or expressions. It appears whenever you create watch statements.

9. The menu bar shows titles of menus and commands that you can activate with the keyboard. Trace and Go represent commands; the other titles are all menus.

10. Menus can be opened by specifying the appropriate title on the menu bar. On the sample screen, the Watch menu has been opened.

11. The menu "highlight" is a reverse-video or colored strip indicating the current selection in a menu. You can move the highlight up or down to change the current selection.

12. The scroll bar (not shown) is the vertical bar on the right side of the screen. This bar graphically represents the ratio of read to unread portions as you scroll through the file.

13. Dialog boxes (not shown) appear in the center of the screen when you choose a menu selection that requires a response. The box prompts you for a response and disappears when you enter your answer.

14. Message boxes (not shown) appear in the center of the screen to display errors or other messages.

The screen elements are described in more detail in the rest of this chapter.

Executing Window Commands

The most common CodeView debugging commands, and all the commands for managing the CodeView display, are available with window commands. Window commands are one-keystroke commands that can be entered with function keys, <CTL> key combinations, <ALT> key combinations, or the direction keys on the numeric keypad. The window commands available from the keyboard are described by category in the following sections.

Moving the Cursor with Keyboard Commands

The following keys move the cursor or scroll text in the display or dialog window.

Key	Function
F6	Moves the cursor between the display and dialog windows. If the cursor is in the dialog window when you press **F6**, it moves to its previous position in the display window. If the cursor is in the display window, it moves to its previous position in the dialog window.

<CTL>g	Makes the size of the dialog or display window grow. This works for whichever window the cursor is in. If the cursor is in the display window, then the display/dialog separator line moves down one line. If the cursor is in the dialog window, then the separator line moves up one line.
<CTL>t	Makes the size of the dialog or display window smaller. This works for whichever window the cursor is in. If the cursor is in the display window, then the display/dialog separator line moves up one line. If the cursor is in the dialog window, then the separator line moves down one line.
UP ARROW	Moves the cursor up one line in either the display or dialog window.
DOWN ARROW	Moves the cursor down one line in either the display or dialog window.
<PgUp>	Scrolls up one page. If the cursor is in the display window, the source lines or assembly-language instructions scroll up. If the cursor is in the dialog window, the buffer of commands entered during the session scrolls up. The cursor remains at its current position in the window. The length of a page is the current number of lines in the window.
<PgDn>	Scrolls down one page. If the cursor is in the display window, the source lines or assembly-language instructions scroll down. If the cursor is in the dialog window, the buffer of commands entered during the session scrolls down. The cursor remains at its current position in the window. The length of a page is the current number of lines in the window.
<HOME>	Scrolls to the top of the file or command buffer. If the cursor is in the display window, the text scrolls to the start of the source file or program

instructions. If the cursor is in the dialog window, the commands scroll to the top of the command buffer. The top of the command buffer may be blank if you have not yet entered enough commands to fill the buffer. The cursor remains at its current position in the window.

<END>	Scrolls to the bottom of the file or command buffer.
	If the cursor is in the display window, the text scrolls to the end of the source file or program instructions. If the cursor is in the dialog window, the commands scroll to the bottom of the command buffer, and the cursor moves to the CodeView prompt (>) at the end of the buffer.

Changing the Screen

The following keys change the screen or switch to a different screen.

Key	Function
F1	Displays initial on-line help screen.
	The help system is discussed in the section ''Using the Help System.'' You can also take advantage of the help system by using the Help menu, as mentioned in the section ''Using Menu Selections'' in this chapter.
F2	Toggles the register window.
	The window disappears if present, or appears if absent. You can also toggle the register window with the Register selection from the View menu, as described in ''Using Menu Selections.''
F3	Switches between source, mixed, and assembly modes.
	Source mode shows source code in the display window, whereas assembly mode shows assembly-language instructions. Mixed mode shows both. You can also change modes with the Source, Mixed, and Assembly selections from the View menu, as described in ''Using Menu Selections.''

F4 — Switches to the output screen.

The output screen shows the output, if any, from your program. Press any key to return to the CodeView screen.

Controlling Program Execution

The following keys set and clear breakpoints, trace through your program, or execute to a breakpoint.

Key	Function
F5	Executes to the next breakpoint or to the end of the program if no breakpoint is encountered. This keyboard command corresponds to the Go dialog command when it is given without a destination breakpoint argument.
F7	Sets a temporary breakpoint on the line with the cursor, and executes to that line (or to a previously set breakpoint or the end of the program if either is encountered before the temporary breakpoint). In source mode, if the line does not correspond to code (for example, data declaration or comment lines), the CodeView debugger sounds a warning and ignores the command. This window command corresponds to the Go dialog command when it is given with a destination breakpoint.
F8	Executes a Trace command. The CodeView debugger executes the next source line in source mode or the next instruction in assembly mode. If the source line or instruction contains a call to a routine or interrupt, the debugger starts tracing through the call (enters the call and is ready to execute the first source line or instruction). This command will not trace into function calls.
F9	Sets or clears a breakpoint on the line with the cursor. If the line does not currently have a breakpoint, one is set on that line. If the line already has a breakpoint, the breakpoint is cleared. If the cursor is in the dialog

window, the CodeView debugger sounds a warning and ignores the command. This window command corresponds to the Breakpoint Set and Breakpoint Clear dialog commands.

F10 Executes the Program Step command.

The CodeView debugger executes the next source line in source mode, or the next instruction in assembly mode. If the source line or instruction contains a call to a routine or interrupt, the debugger steps over the entire call (executes it to the return) and is ready to execute the line or instruction after the call.

3

Note

You can usually interrupt program execution by pressing either <CTL><BREAK> or <DEL>. These key combinations can be used to exit endless loops or to interrupt loops that are slowed by the Watchpoint or Tracepoint commands (see Chapter 9, ''Managing Watch Statements''). The <CTL><BREAK> or <DEL> keystrokes may not work if your program has a special use for one or both of these key combinations.

Selecting from Menus with the Keyboard

This section discusses how to make selections from menus with the keyboard. The effects of the selections are in the section ''Using Menu Selections.''

The menu bar at the top of the screen has eleven titles: File, View, Search, Run, Watch, Options, Language, Calls, Help, Trace, and Go. The first nine titles are menus, and the last two are commands.

The four steps for opening a menu and making a selection are:

1. To open a menu, press the <ALT> key and the mnemonic (the first letter) of the menu title. This can be accomplished by holding down the <ALT> key and then pressing the letter. For example, press <ALT>**s**. to open the Search menu. The menu title is highlighted, and a menu box listing the selections pops up below the title.

You can type either an uppercase or lowercase letter to open any of the menus.

2. There are two ways to make a selection from an open menu:

    a. Press the DOWN ARROW key on the numeric keypad to move down the menu. The highlight follows your movement. When the item you want is highlighted, press the <RETURN> key to execute the command. For example, press the DOWN ARROW once to select Find from the Search menu.

       You can also press the UP ARROW key to move up the menu. If you move off the top or bottom of the menu, the highlight wraps around to the other end of the menu.

    b. Press the key corresponding to the menu-selection mnemonic. The mnemonic is simply a single letter that represents the selection. In color displays, this letter is in red; in black-and-white displays, this letter is in bold. In most cases, but not all, the letter is simply the first letter of the name of the selection. You can type either an uppercase or lowercase letter for the same selection.

3. After a selection is made from the menu, one of three things happens:

    a. For most menu selections, the choice is executed immediately.

    b. The items on the View, Options, and Language menus have small double arrows next to them if the option is on, or no arrows if the option is off. Choosing the item toggles the option. The status of the arrows is reversed the next time an option is chosen.

    c. Some items require a response. In this case, there is another step in the menu-selection process.

4. If the item you select requires a response, a dialog box opens when you select a menu item. Type your response to the prompt in the box and press the <RETURN> key. For example, the Find dialog box asks you to enter a regular expression.

   If your response is valid, the command is executed. If you enter an invalid response, a message box appears, telling you the problem and asking you to press a key. Press any key to make the message box disappear.

At any point during the process of selecting a menu item, you can press the <ESC> key to cancel the menu. While a menu is open, you can press the LEFT ARROW or RIGHT ARROW key to move from one menu to an adjacent menu, or to one of the command titles on the menu bar. Pressing <RETURN> without entering any characters in response to a message box also cancels the menu.

Using Menu Selections

This section describes the selections on each of the CodeView menus. These selections can be made with the keyboard, as described in the section "Executing Window Commands."

Note that although the Trace and Go commands appear on the menu bar, they are not menus.

The File Menu

The File menu includes selections for working on the current source or program file. The File menu selections are explained below.

Selection	Action
Open...	Opens a new file. When you make this selection, a dialog box appears asking for the name of the new file you want to open. Type the name of a source file, an include file, or any other text file. The text of the new file replaces the current contents of the display window (if you are in assembly mode, the CodeView debugger switches to source mode). When you finish viewing the file, you can reopen the original file. The last location and breakpoints are still marked when you return. You may not need to open a new file to see source files for a different module of your program. The CodeView debugger automatically switches to the source file of a module when program execution enters that module. Although switching source files is never necessary, it may be desirable if you want to set breakpoints or execute to a line in a module not currently being executed.

Note

If the debugger cannot find the source file when it switches modules, a dialog box appears asking for a file specification for the source file. You can either enter a new file specification if the file is in another directory, or press the <RETURN> key if no source file exists. If you press the <RETURN> key, the module can only be debugged in assembly mode.

Shell — Exits to a shell. This brings up the standard screen, where you can execute operating system commands or executable files. To return to the CodeView debugger, type **exit** at the operating system command prompt. The CodeView screen reappears with the same status it had when you left it.

Exit — Terminates the debugger and returns to the system.

The View Menu

The View menu includes selections for switching between source and assembly modes, and for switching between the debugging screen and the output screen. The corresponding function keys for menu selection are shown on the right side of the menu where appropriate. The View menu selections are explained below.

Note

The terms ''source mode'' and ''assembly mode'' apply to Macro Assembler programs as well as to high-level-language programs. Source mode used with assembler programs shows the source code as originally written, including comments and directives. Assembly mode displays unassembled machine code, without symbolic information.

The use of one mode or another affects Trace and Program Step commands, as explained in Chapter 6, ''Executing Code.''

At all times only one of the following selections has a small double arrow to the left of the name: Source, Mixed, and Assembly. This arrow indicates which of the three display modes is in use. If you select a mode when you are already in that mode, the selection has no effect. The Registers selection may or may not have a double arrow to the left, depending on whether or not the register window is being displayed.

Selection	**Action**
Source	Changes to source mode (showing source lines only).
Mixed	Changes to mixed mode (showing both unassembled machine code and source lines).
Assembly	Changes to assembly mode (showing only unassembled machine code).
Registers	Selecting this option toggles the register window on and off. You can also turn the register on and off by pressing the **F2** key.
Output	Selecting this option displays the output screen. The entire CodeView display temporarily disappears, but come back as soon as you press any key. The Output command can also be selected with the **F4** key.

The Search Menu

The Search menu includes selections for searching through text files for text strings and for searching executable code for labels. The Search menu selections are explained below.

Selection	Action
Find...	Searches the current source file or other text file for a specified regular expression. (This selection can also be made without pulling down a menu, simply by pressing <CTL>**f**.

When you make this selection, a dialog box opens, asking you to enter a regular expression. Type the expression you want to search for and press the <RETURN> key. The CodeView debugger starts at the current or most recent cursor position in the display window and searches for the expression.

If your entry is found, the cursor moves to the first source line containing the expression. If you are in assembly mode, the debugger automatically switches to source mode when the expression is found. If the entry is not found, a message box opens, telling you the problem and asking you to press a key to continue.

Regular expressions are a method of specifying variable text strings. This method is similar to the standard method of using wild cards in file names.

You can use the Search selections without understanding regular expressions. Since text strings are the simplest form of regular expressions, you can simply enter a string of characters as the expression you want to find. For example, you could enter **count** if you wanted to search for the word ‘‘count.’’

The following characters have a special meaning in regular expressions: backslash (\), asterisk (*), left bracket ([), period (.), dollar sign ($), and caret (^). In order to find strings containing these characters, you must precede the characters with a backslash; this cancels their special meanings.

For example,
with C, you would use ***ptr** to find ***ptr**.

The Case Sense selection from the Options menu has no effect on searching for regular expressions.

Next

Searches for the next match of the current regular expression.

This selection is meaningful only after you have used the Search command to specify the current regular expression. If the CodeView debugger searches to the end of the file without finding another match for the expression, it wraps around and starts searching at the beginning of the file.

Previous

Searches for the previous match of the current regular expression.

This selection is meaningful only after you have used the Search command to specify the current regular expression. If the debugger searches to the beginning of the file without finding another match for the expression, it wraps around and starts searching at the end of the file.

Label...

Searches the executable code for an assembly-language label.

If the label is found, the cursor moves to the instruction containing the label. If you start the search in source mode, the debugger switches to assembly mode to show a label in a library routine or an assembly-language module.

The Run Menu

The Run menu includes selections for running your program. The Run menu selections are explained below.

Selection	**Action**
Start	Starts the program from the beginning and runs it. Any previously set breakpoints or watch statements are still in effect. The CodeView debugger runs your program from the beginning to the first

breakpoint, or to the end of the program if no breakpoint is encountered. This has the same effect as selecting Restart (see the next selection), then entering the Go command.

Restart
: Restarts the current program, but does not begin executing it.

  You can debug the program again from the beginning. Any previously set breakpoints or watch statements are still in effect.

Execute
: Executes in slow motion from the current instruction.

  This is the same as the Execute dialog command (**e**). To stop execution, press any key.

Clear Breakpoints
: Clears all breakpoints.

  This selection may be convenient after selecting Restart if you don't want to use previously set breakpoints. Note that watch statements are not cleared by this command.

Note

Although Start Restart retain breakpoints, along with pass count and arguments (see Chapter 6, "Executing Code,"), any instructions entered with the Assemble command will be overwritten by the original program.

The Watch Menu

The Watch menu includes selections for managing the watch window. Selections on this menu are also available with dialog commands. The Watch menu selections are explained below.

Selection	Action
Add Watch...	Adds a watch-expression statement to the watch window. (This selection can also be made directly, by pressing <CTL>**w**.) A dialog window opens, asking for the source-level expression (which may be simply a variable) whose value you want to see displayed in the watch window. Type the expression and press the <RETURN> key The statement appears in the watch window in normal text. You cannot specify a memory range to be displayed with the Add Watch selection as with the Watch dialog command. You can specify the format in which the value is displayed. Type the expression, followed by a comma and a CodeView format specifier. If you do not give a format specifier, the CodeView debugger displays the value in a default format. See Chapter 8, "Examining Data and Expressions," for more information about format specifiers and the default format. See the section "Setting Watch-Expression and Watch-Memory Statements" in Chapter 9 for more information about the Watch command.
Watchpoint...	Adds a watchpoint statement to the window. A dialog window opens, asking for the source-level expression whose value you want to test. The watchpoint statement appears in the watch window in high-intensity text when you enter the expression. A watchpoint is a conditional breakpoint that causes execution to stop when the expression becomes nonzero (true). See the section "Setting Watchpoints" in Chapter 9 for more information.
Tracepoint...	Adds a tracepoint statement to the watch window. A dialog window opens, asking for the source-level expression or memory range whose value you want to test. The tracepoint statement appears in the watch window in high-intensity text when you enter the expression. A tracepoint is a conditional breakpoint that causes execution to stop

when the value of a given expression changes. You cannot specify a memory range to be tested with the Tracepoint selection as you can with the Tracepoint dialog command.

When setting a tracepoint expression, you can specify the format in which the value is displayed. After the expression type a comma and a format specifier. If you do not give a format specifier, the CodeView debugger displays the value in a default format. See Chapter 7, ''Examining Data and Expressions,'' for more information about format specifiers and default. See the section ''Setting Tracepoints'' in Chapter 9 for more information about tracepoints.

Delete Watch... Deletes a statement from the watch window. (This selection can also be made directly, by pressing <CTL>**u**).

A dialog window opens, showing the current watch statements. If you are using a mouse, move the pointer to the statement you want to delete and click either button. If you are using the keyboard, press the UP ARROW or DOWN ARROW key to move the highlight to the statement you want to delete, then press the <RETURN> key.

Delete All Watch

Deletes all statements in the watch window.

All watch, watchpoint, and tracepoint statements are deleted, the watch window disappears, and the display window is redrawn to take advantage of the freed space on screen.

The Options Menu

The Options menu allows you to set options that affect various aspects of the behavior of the CodeView debugger. The Options menu selections are explained below. Selections on the Options menu have small double arrows to the left of the selection name when the option is on. The status of the option (and the presence of the double arrows) is reversed each time you select the option. By default, the Save Output and Bytes Coded options are on when you start the CodeView debugger. Depending on which language your main program is in, the debugger automatically turns Case Sense on (if your program is in C) or off (if your program is in another language) when you start debugging.

The selections from the Options menu are discussed below.

Selection	Action
Save Output	When this option is on, which is the default setting, the output from your debugged program is saved. When it is off, any program output is not saved.
Bytes Coded	When on (the default), the instructions, instruction addresses, and the bytes for each instruction are shown; when off, only the instructions are shown.

3

This option affects only assembly mode. The following display shows the appearance of sample code when the option is off:

```
27:         name = gets(namebuf);
   LEA        AX,Word Ptr [namebuf]
   PUSH       AX
   CALL       _gets (03E1)
   ADD        SP,02
   MOV        Word Ptr [name],AX
```

The following display shows the appearance of the same code when the option is on:

```
27:           name = gets(namebuf);
32AF:003E 8D46DE  LEA   AX,Word Ptr [namebuf]
32AF:0041 50      PUSH  AX
32AF:0042 E89C03  CALL  _gets (03E1)
32AF:0045 83C402  ADD   SP,02
32AF:0048 8946DA  MOV   Word Ptr [name],AX
```

Selection	Action
Case Sense	When the selection is turned on, the CodeView debugger assumes that symbol names are case sensitive (each lowercase letter is different from the corresponding uppercase letter); when off, symbol names are not case sensitive.

This option is on by default for C programs, and off by default for assembly programs. You probably want to leave the option in its default setting.

The Language Menu

The Language menu allows you either to select the expression evaluator, or to instruct the CodeView debugger to select it for you automatically. The Language menu selections are explained below.

As with the Options menu, the selection that is on is marked by double arrows. Unlike the Options menu, however, exactly one item (and no more) on the Language menu is selected at any given time.

The Auto selection causes the debugger to select automatically the expression evaluator each time a new source file is loaded. The debugger examines the extension of the source file in order to determine which expression evaluator to select. The Auto selection uses the C expression evaluator if the current source file does not have a **.bas, .f, .for,** or **.pas** extension.

If you change to a source module with an **.asm** extension, then Auto causes the debugger to select the C expression evaluator, but not all of the C defaults are used; system radix is hexadecimal, case sensitivity is turned off, and the register window is displayed.

When a language expression evaluator is selected, the debugger uses that evaluator, regardless of what kind of program is being debugged.

The Calls Menu

The Calls menu is different from other menus in that its contents and size change, depending on the status of your program. The Calls menu is explained below.

The mnemonic for each item in the Calls menu is a number. Type the number displayed immediately to the left of a routine in order to select it. You can also use the UP ARROW or DOWN ARROW key to move to your selection, and then press the <RETURN> key.

The effect of making a selection from the Calls menu is to view a routine. The cursor goes to the line at which the selected routine was last executing. For example, selecting **main** causes CodeView to display **main**, at the point at which **main** made a call to **calc** (the function immediately above it). Note that selecting a routine from the Calls menu does not (by itself) affect program execution. It simply provides a convenient way to view previously called routines.

The Calls menu shows the current routine and the trail of routines from which it was called. The current routine is always at the top. The routine from which the current routine was called is directly below. Other active routines are shown in the reverse order in which they were called. With C programs, the bottom routine should always be **main**. (The only time when **main** will not be the bottom routine is when you are tracing through the standard library's start-up or termination routines.)

The current value of each argument, if any, is shown in parentheses following the routine. The menu expands to accommodate the arguments of the widest routine. Arguments are shown in the current radix (the default is decimal). If there are more active routines than fit on the screen, or if the routine arguments are too wide, the display expands to both the left and right. The Stack Trace dialog command (**K**) also shows all the routines and arguments.

The Help Menu

The Help menu lists the major topics in the help system. For help, open the Help menu and then select the topic that you want to view.

Each topic may have any number of subtopics. You must go to the major topic first. Information on how to move around within the help system is provided in the next section.

The bottom selection on the Help menu is the About command. When you make this selection, the debugger displays a small box at the center of the screen that gives the name of the product and the version number.

Using the Help System

The CodeView on-line help system uses tree-structured menus to give you quick access to help screens on a variety of subjects. The system uses a combination of menu access and sequentially linked screens, as explained below.

The help file is called *cv.hlp* and is located in the */usr/lib* directory. If this file is not found, the CodeView debugger still operates, but you cannot use the help system. An error message appears if you try to use a help command.

When you request help, either by pressing the **F1** key, by using the **H** dialog command, or by selecting the Help menu, the first help screen appears. You can select **N** for Next and **P** for Previous to page through the screens. The screens are arranged in a circular fashion, so that selecting

Next on the last screen get you to the first screen. Select **C** for Cancel to return to the CodeView screen. Pressing the <PgDn>, <PgUp>, and <ESC> keys achieves the same results as selecting Next (**N**), Previous (**P**), and Cancel (**C**).

You can enter the help system at a particular topic by selecting the topic from the Help menu. Once into the system, use Next (**N**) and Previous (**P**) to page to other screens.

Using Sequential Mode

Sequential mode is useful when you are using redirected CodeView input and output. In sequential mode, the CodeView debugger's input and output always move down the screen from the current location. When the screen is full, the old output scrolls off the top of the screen to make room for new output appearing at the bottom. You can never return to examine previous commands once they scroll off, but in many cases, you can reenter the command to put the same information on the screen again.

3

Most window commands cannot be used in sequential mode. However, the following function keys, which are used as commands in window mode, are also available in sequential mode.

Command	Action
F1	Displays a command-syntax summary.
F2	Displays the registers. This is equivalent to the Register (**R**) dialog command.
F3	Toggles between source, mixed, and assembly modes. Pressing this key rotates the mode between source, mixed, and assembly. You can achieve the same effect by using the Set Assembly (**S-**), Set Mixed (**S&**), and Set Source(S+) dialog commands.
F4	Switches to the output screen, which shows the output of your program. Press any key to return to the CodeView debugging screen. This is equivalent to the Screen Exchange (\\) dialog command.
F5	Executes from the current instruction until a breakpoint or the end of the program is encountered. This is equivalent to the Go dialog command (**G**) with no argument.
F8	Executes the next source line in source mode, or the next instruction in assembly mode.

If the source line or instruction contains a function, procedure, or interrupt call, the CodeView debugger executes the first source line or instruction of the call and is ready to execute the next source line or instruction within the call. This is equivalent to the Trace (**T**) dialog command.

F9 Sets or clears a breakpoint at the current program location.

If the current program location has no breakpoint, one is set. If the current location has a breakpoint, it is removed. This is equivalent to the Breakpoint Set (**BP**) dialog command with no argument.

F10 Executes the next source line in source mode, or the next instruction in assembly mode.

If the source line or instruction contains a function, procedure, or interrupt call, the call is executed to the end, and the CodeView debugger is ready to execute the line or instruction after the call. This is equivalent to the Program Step (**P**) dialog command.

The CodeView Watch (**W**), Watchpoint (**WP**), and Tracepoint (**TP**) commands work in sequential mode, but since there is no watch window, the watch statements are not shown. You must use the Watch List command (**W**) to examine watch statements and watch values. See Chapter 9, "Managing Watch Statements," for information on Watch Statement commands.

All the CodeView commands that affect program operation (such as Trace, Go, and Breakpoint Set) are available in sequential mode. Any debugging operation done in window mode can also be done in sequential mode.

Chapter 4

Using Dialog Commands

Introduction

CodeView dialog commands can be used in sequential mode or from the dialog window. In sequential mode, they are the primary method of entering commands. In window mode, dialog commands are used to enter commands that require arguments or that do not have corresponding window commands.

Many window commands have duplicate dialog commands. Generally, the window version of a command is more convenient, but the dialog version is more powerful. For example, to set a breakpoint on a source line in window mode, put the cursor on the source line and press **F9**. The dialog version of the Breakpoint command (**BP**) requires more keystrokes, but it allows you to specify an address, a pass count, and a string of commands to be taken whenever the breakpoint is encountered.

The rest of this chapter explains how to enter dialog commands.

Entering Commands and Arguments

Dialog commands are entered at the CodeView prompt (>). Type the command and arguments, and then press the <RETURN> key.

In window mode, you can enter commands whether or not the cursor is at the CodeView prompt. If the cursor is in the display window, the text you type appears after the prompt in the dialog window, even though the cursor remains in the display window.

Using Special Keys

When entering dialog commands or viewing output from commands, you can use the following special keys:

Key	Action
<DEL>	Stops the current output or cancels the current command line. For example, if you are watching a long display from a Dump command, you can press <DEL> to interrupt the output and return to the CodeView prompt. If you make a mistake while entering a command, you can press <DEL> to cancel the command without executing it. A new prompt appears, and you can reenter the command.
<CTL>s	Pauses during output of a command. You can press any key to continue output. For example, if you are watching a long display from a Dump command, you can press <CTL>s when a part of the display appears that you want to examine more closely. Then press any key when you are ready for the output to continue scrolling.
<BKSP>	Deletes the previous character on the command line and moves the cursor back one space. For example, if you make an error while typing a command, you can use the <BKSP> key to delete the characters back to the error-then retype the rest of the command.

Using the Command Buffer

In window mode, the CodeView debugger has a command buffer where the last 2-4 screens of commands and command output are stored. The command buffer is not available in sequential mode.

When the cursor is in the dialog window, you can scroll up or down to view the commands you have entered earlier in the session. The commands for moving the cursor and scrolling through the buffer are explained in Chapter 3, ''The CodeView Display.''

Scrolling through the buffer is particularly useful for viewing the output from commands, such as Dump or Examine Symbols, whose output may scroll off the top of the dialog window.

If you have scrolled through the dialog buffer to look at previous commands and output, you can still enter new commands. When you type a command, it appears to be overwriting the previous line where the cursor is located, but when you press the <RETURN> key, the new command is entered at the end of the buffer. For example, if you enter a command while the cursor is at the start of the buffer and then scroll to the end of the buffer, you see the command you just entered. If you scroll back to the point where you entered the command, you see the original characters rather than the characters you typed over the originals.

When you start the debugger, the buffer is empty except for the copyright message. As you enter commands during the session, the buffer is gradually filled from the bottom to the top. If you have not filled the entire buffer and you press the <HOME> key to go to the top of the buffer, you do not see the first commands of the session. Instead you see blank lines, since there is nothing at the top of the buffer.

Format for CodeView Commands and Arguments

The general format for CodeView commands is shown below:

"<*command*> [<*arguments*>] [;<*command2*>]"

The *command* is a one-, two-, or three-character command name, and *arguments* are expressions that represent values or addresses to be used by the command. The *command* is not case sensitive; any combination of uppercase and lowercase letters can be used. However, *arguments* consisting of source-level expressions may or may not be case sensitive. (Case sensitivity can be affected by the language selected for expression evaluation, in the Options menu.) Usually, the first *argument* can be placed immediately after *command* with no space separating the two fields.

The number of arguments required or allowed with each command varies. If a command takes two or more arguments, you must separate the arguments with spaces. A semicolon (;) can be used as a command separator if you want to specify more than one command on a line.

Examples

```
>DB 100 200    ;* Example 1

>U Label1      ;* Example 2, C variable as argument

>U sum; DB     ;* Example 3, multiple commands
```

In Example 1, **DB** is the first command (for the Dump Bytes command). The arguments to the command are **100** and **200** . The second command on this line is the Comment command (*). A semicolon is used to separate the two commands. The Comment command is used throughout the rest of the manual to number examples.

In Example 2, **U** is the first command (for the Unassemble command), and the C language variable **Label1** is a command argument.

Example 3 consists of three commands, separated by semicolons. The first is the Unassemble command (**U**) with the C variable **sum** as an argument. The second is the Dump Bytes command (**DB**) with no arguments. The third is the Comment command (*).

Chapter 5

CodeView Expressions

Introduction

CodeView command arguments are expressions that can include symbols, constant numbers, operators, and registers. Arguments can be simple machine-level expressions that directly specify an address or range in memory, or they can be source-level expressions that correspond to operators and symbols used in C or the Macro Assembler. CodeView has an expression evaluator for C that computes the value of source-level expressions.

Each of the expression evaluators has a different set of operators and rules of precedence. However, the basic syntax for registers, addresses, and line numbers is the same regardless of the language. You can always change the expression evaluator. If you specify a language other than the one used in the source file, then the expression evaluator still recognizes your program symbols, if possible.

If the Auto option is on, then the debugger examines the file extension of each new source file you trace through. Both C and assembly modules cause the debugger to select C as the expression evaluator.

This chapter deals first with the expressions specific to each language. Line-number expressions are presented next; they work the same way regardless of the language. Then, register and address expressions are presented; generally, these do not have to be mastered unless you are doing assembly-level debugging. Finally, the chapter describes how to switch the expression evaluator.

Note

When you use a variable in an expression where that variable is not defined, the CodeView debugger displays the message `UNKNOWN SYMBOL`. For example, the message appears if you reference a local variable outside the function where the variable is defined.

C Expressions

The C expression evaluator uses a subset of the most commonly used C operators. It also supports the colon operator (**:**), which is described in the section "Addresses" in this chapter, and the three memory operators (**BY**, **WO**, and **DW**), which are discussed in the section "Memory Operators" in this chapter. The memory operators are primarily useful for debugging assembly source code. The CodeView C-expression operators are listed in Table 5.1 in order of precedence.

5

Table 5.1

CodeView C-Expression Operators

<table>
<tr><th>Precedence</th><th>Operators</th></tr>
<tr><td>(Highest)</td><td></td></tr>
<tr><td>1</td><td>() [] -> .</td></tr>
<tr><td>2</td><td>! ~ – (type) ++ –– * & sizeof</td></tr>
<tr><td>3</td><td>* / % :</td></tr>
<tr><td>4</td><td>+ –</td></tr>
<tr><td>5</td><td>< > <= >=</td></tr>
<tr><td>6</td><td>== !=</td></tr>
<tr><td>7</td><td>&&</td></tr>
<tr><td>8</td><td>||</td></tr>
<tr><td>9</td><td>= += –= *= /= %=</td></tr>
<tr><td>10</td><td>BY WO DW</td></tr>
<tr><td>(Lowest)</td><td></td></tr>
</table>

The minus sign with precedence 2 is the *unary minus* indicating the sign of a number; the minus sign with precedence 4 is a *binary minus* indicating subtraction. The asterisk with precedence 2 is the pointer operator; the asterisk with precedence 3 is the multiplication operator. The ampersand with precedence 2 is the address-of operator. The ampersand as a bitwise AND operator is not supported by the CodeView debugger.

See the *C Language Referece* for a description of how C operators can be combined with identifiers and constants to form expressions. With the C-expression evaluator, the period (.) has its normal use as a member selection operator, but it also has an extended use as a specifier of local variables in parent functions. The syntax is shown below:

<function>.<variable>

The *function* must be a high-level-language function, and the *variable* must be a local variable within the specified function. The *variable* cannot be a register variable. For example, you can use the expression **main.argc** to refer to the local variable *argc* when you are in a function that has been called by **main**.

The **type** operator (used in type casting) can be any of the predefined C types. The CodeView debugger limits casts of pointer types to one level of indirection. For example, **(char *)sym** is accepted, but **(char **)sym** is not.

When a C expression is used as an argument with a command that takes multiple arguments, the expression should not have any internal spaces. For example, **count+6** is allowed, but **count + 6** may be interpreted as three separate arguments. Some commands (such as the Display Expression command) do permit spaces in expressions.

C Symbols

5

Syntax

<name>

A symbol is a name that represents a register, a segment address, an offset address, or a full 32-bit address. At the C source level, a symbol is a variable name or the name of a function. Symbols (also called identifiers) follow the naming rules of the C compiler. Note that although CodeView command letters are not case sensitive, symbols given as arguments are case sensitive (unless you have turned off case sensitivity with the Case Sense selection from the Options menu).

In assembly language output or input from the Examine Symbols command, the CodeView debugger displays some symbol names in the object-code format produced by the C Compiler. This format includes a leading underscore. For example, the function **main** is displayed as **_main**. Only global labels (such as procedure names) are shown in this format. You do not need to include the underscore when specifying such a symbol in CodeView commands. Labels within library routines are sometimes displayed with a double underscore (**__chkstk**). You must use two leading underscores when accessing these labels with CodeView commands.

C Constants

Syntax

<digits>	Default radix
0*<digits>*	Octal radix
0x*<digits>*	Hexadecimal radix
0n*<digits>*	Decimal radix

Numbers used in CodeView commands represent integer constants. They are made up of octal, hexadecimal, or decimal digits, and are entered in the current input radix. The C-language format for entering numbers of different radixes can be used to override the current input radix.

The default radix for the C expression evaluator is decimal. However, you can use the Radix command (**N**) to specify an octal or hexadecimal radix, as explained in ''Radix Command'' in Chapter 12.

If the current radix is 16 (hexadecimal) or 8 (octal), you can enter decimal numbers in the special CodeView format **0n***digits*. For example, enter 21 decimal as **0n21**.

With radix 16, it is possible to enter a value or argument that could be interpreted either as a symbol or as a hexadecimal number. The Code-View debugger resolves the ambiguity by searching first for a symbol (identifier) with that name. If no symbol is found, the debugger interprets the value as a hexadecimal number. If you want to enter a number that overrides an existing symbol, use the hexadecimal format (**0x***digits*).

For example, if you enter **abc** as an argument when the program contains a variable or function named *abc*, the CodeView debugger interprets the argument as the symbol. If you want to enter **abc** as a number, enter it as **0xabc**.

Table 5.2 shows how a sample number (63 decimal) would be represented in each radix.

Table 5.2

C Radix Examples

Input Radix	Octal	Decimal	Hexadecimal
8	77	0n63	0x3F
10	077	63	0x3F
16	077	0n63	3F

C Strings

Syntax

"*<null-terminated-string>*"

5

Strings can be specified as expressions in the C format. You can use C escape characters within strings. For example, double quotation marks within a string are specified with the escape character backslash double quotation mark (\").

Example

```
>EA message "This \"string\" is okay."
```

The example uses the Enter ASCII command (**EA**) to enter the given string into memory starting at the address of the variable *message*.

Assembly Expressions

The **-Zi** Macro Assembler option provides variable size information for the CodeView debugger. This makes for correct evaluation of expressions derived from assembly code (except with arrays, which are discussed later in this section). If you have an early version of the Macro Assembler, you need to use C type casts to get correct evaluation. See the *Release Notes* for more information about Macro Assembler versions.

When a program assembles or when the Auto switch is on, source files with an **.asm** extension cause CodeView to select the C-expression evaluator. However, the following options are set differently from the C default options:

- System radix is hexadecimal (not decimal).
- Register window is on.
- Case Sense is off.

5

The C-expression evaluator supports the memory operators described in the section "Memory Operators" in this chapter, and generally is the appropriate expression evaluator to debug assembly with, because of its flexibility.

However, you cannot always use the C-expression evaluator to specify an expression exactly as it would appear in assembly code. The list below describes the principal differences between assembler syntax and syntax used with the C-expression evaluator.

Note

The examples below present *expressions*, not CodeView commands. You can see the results of these expressions by using them as operands for the Display Expression command (**?**), described in Chapter 7, "Examining Data and Expressions."

In the following list, examples of assembly source code are shown in the left-hand column. Corresponding CodeView expressions (with the C-expression evaluator) are shown in the right-hand column.

1. Register indirection.

   The C-expression evaluator does not extend the use of brackets to registers. To refer to the byte, word, or double word pointed to by a register, use the **BY**, **WO**, or **DW** operator.

```
BYTE PTR [bx]                  BY bx
WORD PTR [bp]                  WO bp
DWORD PTR [bp]                 DW bp
```

2. Register indirection with displacement.

   To perform based, indexed, or based-index indirection with a displacement, use the **BY**, **WO**, or **DW** operator along with addition in a complex expression:

```
BYTE PTR [di+6]                BY di+6
BYTE PTR [si][bp+6]            BY si+bp+6
WORD PTR [bx][si]              WO bx+si
```

3. Taking the address of a variable.

   Use the ampersand (**&**) to get the address of a variable with the C-expression evaluator.

```
OFFSET var                     &var
```

4. The **PTR** operator.

   With the CodeView debugger, C type casts perform the same function as the assembler **PTR** operator.

```
BYTE PTR var                    (char) var
WORD PTR var                    (int) var
DWORD PTR var                   (long) var
```

5. Accessing array elements.

   Accessing arrays declared in assembly code is problematic, because the Macro Assembler emits no type information to indicate which variables are arrays. Therefore the CodeView debugger treats an array name like any other variable.

   In C, an array name is equated with the address of the first element. Therefore, if you prefix an array with the address operator (&), the C-expression evaluator gives correct results for array operations.

```
string[12]              (&string)[12]
warray[bx+di]           (&warray)(bx+di)/2
darray[4]               (&darray)[1]
```

In the second and third examples above, notice that the indexes used in the assembly source-code expressions differ from the indexes used in the CodeView expressions. This difference is necessary because C arrays are automatically scaled according to the size of elements. In assembly, the program must do the scaling.

5

Line Numbers

Line numbers are useful for source-level debugging. They correspond to the lines in Macro Assembler source-code files In source mode, you see a program displayed with each line numbered sequentially. The CodeView debugger allows you to use these same numbers to access parts of a program.

Syntax

.[<*filename*>:]<*linenumber*>

The address corresponding to a source-line number can be specified as *linenumber* prefixed with a period (.). The CodeView debugger assumes that the source line is in the current source file, unless you specify the optional *filename* followed by a colon and the line number.

The CodeView debugger displays an error message if *filename* does not exist, or if no source line exists for the specified number.

Examples

```
>V .100
```

The example above uses the View command (**V**) to display code starting at the source line *100*. Since no file is indicated, the current source file is assumed.

```
>V .DEMO.C:301
```

The example above uses **V** to display source code starting at line **301** of **demo.c,** respectively.

Registers and Addresses

This section presents alternative ways to refer to objects in memory, including values stored in the processor's registers. Addresses are basic to each of the expression evaluators. A data symbol represents an address in a data segment; a procedure name represents an address in a code segment. All of the syntax in this section can be considered as an extension to the C-expression evaluator.

Registers

Syntax

[@]*<register>*

You can specify a register name if you want to use the current value stored in the register. Registers are rarely needed in source-level debugging, but they are used frequently for assembly-language debugging.

When you specify an identifier, the CodeView debugger first checks the symbol table for a symbol with that name. If the debugger does not find a symbol, it checks to see if the identifier is a valid register name. If you want the identifier to be considered a register, regardless of any name in the symbol table, use the ''at'' sign (**@**) as a prefix to the register name. For example, if your program has a symbol called **AX,** you could specify **@AX** to refer to the **AX** register. You can avoid this problem entirely by making sure that identifier names in your program do not conflict with register names.

The register names known to the CodeView debugger are shown in the following table.

Table 5.3

Registers

Type	Names			
8-bit high byte	**AH**	**BH**	**CH**	**DH**
8-bit low byte	**AL**	**BL**	**CL**	**DL**
16-bit general purpose	**AX**	**BX**	**CX**	**DX**
16-bit segment	**CS**	**DS**	**SS**	**ES**
16-bit pointer	**SP**	**BP**	**IP**	
16-bit index	**SI**	**DI**		
32-bit general purpose	**EAX**	**EBX**	**ECX**	**EDX**
32-bit pointer	**ESP**	**EBP**		
32-bit index	**ESI**	**EDI**		

Addresses

Syntax

[<*segment*>:]<*offset*>

Addresses can be specified in the CodeView debugger through the use of the colon operator as a *segment:offset* connector. Both the *segment* and the *offset* are made up of expressions.

A full address has a *segment* and an *offset*, separated by a colon. A partial address has just an *offset*; a default segment is assumed. The default segment varies, depending on the command with which the address is used. Commands that refer to data (Dump, Enter, Watch, and Tracepoint) use the contents of the **DS** register. Commands that refer to code (Assemble, Breakpoint Set, Go, Unassemble, and View) use the contents of the **CS** register.

Examples

```
>DB 100
```

In the example above, the Dump Bytes command (**DB**) is used to dump memory starting at offset address **100**. Since no segment is given, the data segment (the default for Dump commands) is assumed. In C, a variable might be denoted as *array[count]*.

```
>DB label+10
```

In the example above, the Dump Bytes command is used to dump memory starting at a point 10 bytes beyond the symbol **label**.

```
>DB ES:200
```

In the example above, the Dump Bytes command is used to dump memory at the address having the segment value stored in **ES** and the offset address **200**.

Address Ranges

Syntax

<startaddress> *<endaddress>*
<startaddress> L *<count>*

A range is a pair of memory addresses that bound a sequence of contiguous memory locations.

You can specify a range in two ways. One way is to give the start and end points. In this case the range covers *startaddress* to *endaddress*, inclusively. If a command takes a range, but you do not supply a second address, the CodeView debugger usually assumes the default range. Each command has its own default range. (The most common default range is 128 bytes.)

You can also specify a range by giving its starting point and the number of objects you want included in the range. This type of range is called an object range. In specifying an object range, *startaddress* is the address of the first object in the list, **L** indicates that this is an object range rather than an ordinary range, and *count* specifies the number of objects in the range.

The size of the objects is the size taken by the command. For example, the Dump Bytes command (**DB**) has byte objects, the Dump Words command (**DW**) has words, the Unassemble command (**U**) has instructions, and so on.

Examples

```
>DB buffer
```

The example above dumps a range of memory starting at the symbol **buffer**. Since the end of the range is not given, the default size (128 bytes for the Dump Bytes command) is assumed.

5

```
>DB buffer buffer+20
```

The example above dumps a range of memory starting at **buffer** and ending at **buffer+20** (the point 20 bytes beyond **buffer**).

```
>DB buffer L 20
```

The example above uses an object range to dump the same range as in the previous example. The **L** indicates that the range is an object range, and **20** is the number of objects in the range. Each object has a size of 1 byte, since that is the command size.

```
>U funcname-30 funcname
```

The example above uses the Unassemble command (**U**) to list the assembly-language statements starting 30 instructions before *funcname* and continuing to *funcname*.

Memory Operators

Memory operators return the content of specific locations in memory. They are unary operators that work in the same way regardless of the language selected, and return the result of a direct memory operation. They are chiefly of interest to programmers who debug in assembly mode, and are not necessary for high-level debugging.

All of the operators listed in this section are part of the CodeView C-expression evaluator and should not be confused with CodeView commands. As operators, they can only build expressions, which in turn are used as arguments in commands.

Note

The memory operators discussed in this section are only available with the C-expression evaluator, and have lowest precedence of any C operators.

Accessing Bytes (BY)

You can access the byte at an address by using the **BY** operator. This operator is useful for simulating the **BYTE PTR** operation of the Macro Assembler. It is particularly useful for watching the byte pointed to by a particular register.

Note

The examples that follow in the section ''Memory Operators'' make use of the Display Expression (?) Command, which is described in ''Display Expression Command'' in Chapter 7. The **x** format specifier causes the debugger to produce output in hexadecimal.

Memory Operators

Syntax

BY *<address>*

The result is a short integer that contains the value of the first byte stored at *address*.

Examples

```
>? BY sum
101
```

The example above returns the first byte at the address of **sum**.

```
>? BY bp+6
42
```

This example returns the byte pointed to by the **BP** register, with a displacement of 6.

Accessing Words (WO)

You can access the word at an address by using the **WO** operator. This operator is useful for simulating the **WORD PTR** operation of the assembler. It is particularly useful for watching the word pointed to by a particular register, such as the stack pointer.

Syntax

WO *<address>*

The result is a short integer that contains the value of the first two bytes stored at *address*.

Examples

```
>? WO sum
>13120
```

The example above returns the first word at the address of **sum**.

```
>? WO sp,x
>2F38
```

This example returns the word pointed to by the stack pointer; the word therefore represents the last word pushed (the ''top'' of the stack).

Accessing Double Words (DW)

You can access the word at an address by using the **DW** operator. This operator is useful for simulating the **DWORD PTR** operation of the Macro Assembler. It is particularly useful for watching the word pointed to by a particular register.

Syntax

DW *<address>*

The result is a long integer that contains the value of the first four bytes stored at *address*.

Note

Be careful not to confuse the **DW** operator with the **DW** command. The operator is only useful for building expressions; it occurs within a CodeView command line, but never at the beginning. The second use of **DW** mentioned above, the Dump Words Command, occurs only at the beginning of a CodeView command line. It displays an entire range of memory (in words, not double words) rather than returning a single result.

Examples

```
>? DW sum
>132120365
```

The example above returns the first double word at the address of **sum**.

```
>? DW si,x
>3F880000
```

This example returns the double word pointed to by the **SI** register.

Chapter 6

Executing Code

Introduction

Several commands execute code within a program. Among the differences between the commands is the size of step executed by each. The commands and their step sizes are listed below.

Command	Action
Trace (**T**)	Executes the current source line in source mode, or the current instruction in assembly mode; traces into routines, procedures, or interrupts
Program Step (**P**)	Executes the current source line in source mode, or the current instruction in assembly mode; steps over routines, procedures, or interrupts
Go (**G**)	Executes the current program
Execute (**E**)	Executes the current program in slow motion
Restart (**L**)	Restarts the current program

6

In window mode, the screen is updated to reflect changes that occur when you execute a Trace, Program Step, or Go command. The highlight marking the current location is moved to the new instruction in the display window. When appropriate, values are changed in the register and watch windows.

In sequential mode, the current source line or instruction is displayed after each Trace, Program Step, or Go command. The format of the display depends on the display mode. The three display modes available in sequential mode (source, assembly, and mixed) are discussed in Chapter 10, ''Examining Code.''

If the display mode is source (**S+**) in sequential mode, the current source line is shown. If the display mode is assembly (**S-**), the status of the registers and the flags and the new instruction are shown in the format of the Register command (see Chapter 7, ''Examining Data and Expressions''). If the display mode is mixed (**S&**), then the registers, the new source line, and the new instruction are all shown.

The commands that execute code are explained in the following sections.

Note

> If you are executing a section of code with the Go or Program Step command, you can usually interrupt program execution by pressing <CTL><BREAK> or <DEL>. This can terminate endless loops, or it can interrupt loops that are delayed by the Watchpoint or Tracepoint command (see Chapter 9, ''Managing Watch Statements'').

Trace Command

The Trace command executes the current source line in source mode, or the current instruction in assembly mode. The current source line or instruction is the one pointed to by the **CS** and **IP** registers. In window mode, the current instruction is shown in reverse video or in a contrasting color.

In source mode, if the current source line contains a call, the CodeView debugger executes the first source line of the called routine. In this mode, the CodeView debugger only traces into functions and routines that have source code. For example, if the current line contains a call to an intrinsic function or a standard C library function, the debugger simply executes the function if you are in source mode, since the source code for standard libraries is not available.

If you are in assembly or mixed mode, the debugger traces into the function. In this mode, if the current instruction is **CALL**, **INT** or **REP**, the debugger executes the first instruction of the procedure, interrupt, or repeated string sequence.

Note

When you debug Macro Assembler programs in source mode, the paragraph above still applies. The debugger does not trace into an **INT** or **REP** sequence when you are in source mode.

Use the Trace command if you want to trace into calls. To execute calls without tracing into them, you should use the Program Step command (**P**) instead. Both commands execute system calls without tracing into them. There is no direct way to trace into system calls.

Keyboard

To execute the Trace command with a keyboard command, press the **F8** key. This works in both window and sequential modes.

Dialog

To execute the Trace command using a dialog command, enter a command line with the following syntax:

T [<*count*>]

If the optional *count* is specified, the command executes *count* times before stopping.

Example

The following example shows the Trace command in sequential mode. (In window mode, there would be no output from the commands, but the display would be updated to show changes caused by the command.)

```
>S+    ;* FORTRAN example
source
>.
9:            CALL INPUT (DATA,N,INPFMT)
>T 3
34:           OPEN (1,FILE='EXAMPLE.DAT',STATUS='OLD')
35:           DO 100 I=1,N
36:           READ (1,'(BN,I10)',END=999) DATA(I)

>
```

6

The FORTRAN example above sets the display mode to source, and then uses the Source Line command to display the current source line. (See Chapter 10, "Examining Code," for a further explanation of the Set Source and Source Line commands.) Note that the current source line calls the subroutine **INPUT**. The Trace command is then used to execute the next three source lines. These lines are the first three lines of the subroutine **INPUT**.

Debugging C and BASIC source code is very similar. If you execute the Trace command when the current source line contains a C system call or a BASIC subprogram call, then the debugger executes the first line of the called routine.

```
>S-
assembly
>T
AX=0058  BX=3050  CX=000B  DX=3FB0  SP=304C  BP=3056  SI=00CC  DI=40E0
DS=49B7  ES=49B7  SS=49B7  CS=3FB0  IP=0013  NV UP EI PL NZ AC PO NC
3FB0:0013 50            PUSH      AX
>
```

The example above sets the display mode to assembly and traces the current instruction. This example and the next example are the same as the examples of the Program Step command in the section ''Program Step Command'' in this chapter. The Trace and Program Step commands behave differently only when the current instruction is a **CALL**, **INT**, or **REP** instruction.

```
>S&
mixed
>T
AX=0000  BX=319C  CX=0028  DX=0000  SP=304C  BP=3056  SI=00CC  DI=40E0
DS=49B7  ES=49B7  SS=49B7  CS=3FB0  IP=003C  NV UP EI PL NZ NA PO NC
8:             IF (N.LT.1 .OR. N.GT.1000) GO TO 100
3FB0:003C 833ECE2101    CMP      Word Ptr [21CE],+01        DS:21CE=0028
>
```

The example above sets the display mode to mixed and traces the current instruction.

Program Step Command

The Program Step command executes the current source line in source mode, or the current instruction in assembly mode. The current source line or instruction is the one pointed to by the **CS** and **IP** registers. In window mode, the current instruction is shown in reverse video or in a contrasting color.

In source mode, if the current source line contains a call, the CodeView debugger executes the entire routine and is ready to execute the line after the call. In assembly mode, if the current instruction is **CALL**, **INT**, or **REP**, the debugger executes the entire procedure, interrupt, or repeated string sequence. Use the Program Step command if you want to execute over routine, function, procedure, and interrupt calls. If you want to trace into calls, you should use the Trace command (**T**) instead. Both commands execute system calls without tracing into them. There is no direct way to trace into system calls.

Keyboard

To execute the Program Step command with a keyboard command, press the **F10** key. This works in both window and sequential modes.

6

Dialog

To execute the Program Step command with a dialog command, enter a command line with the following syntax:

P [<*count*>]

If the optional *count* is specified, the command executes *count* times before stopping.

Example

This example shows the Program Step command in sequential mode. In window mode, there would be no output from the commands, but the display would be updated to show changes.

```
>S+     ;* FORTRAN/BASIC example
source
>.
9:          CALL INPUT (DATA,N,INPFMT)
>P 3
10:         CALL BUBBLE (DATA,N)
11:         CALL STATS (DATA,N)
12:         END
>
```

The example above (in FORTRAN or BASIC) sets the display mode to source, and then uses the Source Line command to display the current source line. (See Chapter 10, "Examining Code," for a further explanation of the Set Source and Source Line commands.) Notice that the current source line calls the subprogram **INPUT**. The Program Step command is then used to execute the next three source lines. The first program step executes the entire subprogram **INPUT**. The next two steps execute the subprograms **BUBBLE** and **STATS**, also in their entirety.

The same program, written in C, would behave exactly the same way with the Program Step command. The Program Step command does not trace into a C system call.

```
>S-
assembly
>P
AX=0058  BX=3050  CX=000B  DX=3FB0  SP=304C  BP=3056  SI=00CC  DI=40E0
DS=49B7  ES=49B7  SS=49B7  CS=3FB0  IP=0013  NV UP EI PL NZ AC PO NC
3FB0:0013 50            PUSH    AX
>
```

The example above sets the display mode to assembly and steps through the current instruction. This example and the next example are the same as the examples of the Trace command in the section "Trace Command" in this chapter. The Trace and Program Step commands behave differently only when the current instruction is a **CALL, INT**, or **REP** instruction.

Program Step Command

```
>S&
mixed
>P
AX=0000  BX=319C  CX=0028  DX=0000  SP=304C  BP=3056  SI=00CC  DI=40E0
DS=49B7  ES=49B7  SS=49B7  CS=3FB0  IP=003C  NV UP EI PL NZ NA PO NC
8:              IF (N.LT.1 .OR. N.GT.1000) GO TO 100
3FB0:003C 833ECE2101    CMP     Word Ptr [21CE],+01        DS:21CE=0028
>
```

The example above sets the display mode to mixed and steps through the current instruction.

Go Command

The Go command starts execution at the current address. There are two variations of the Go command, Go and Goto. The Go variation simply starts execution and continues to the end of the program or until a breakpoint set earlier with the Breakpoint Set (**BP**), Watchpoint (**WP**), or Tracepoint (**TP**) command is encountered. The other variation is a Goto command, in which a destination is given with the command.

If a destination address is given but never encountered (for example, if the destination is on a program branch that is never taken), the CodeView debugger executes to the end of the program.

If you enter the Go command and the debugger does not encounter a breakpoint, the entire program is executed and the following message is displayed:

```
Program terminated normally (number)
```

The *number* in parentheses is the value returned by the program (sometimes called the exit or ''errorlevel'' code).

Keyboard

To use a keyboard command to execute the Go command with no destination, press the **F5** key. This works in both window and sequential modes.

To execute the Goto variation of the Go command, move the cursor to the source line or instruction you wish to go to. If the cursor is in the dialog window, first press the **F6** key to move the cursor to the display window. When the cursor is at the appropriate line in the display window, press the **F7** key. The highlight marking the current location moves to the source line or instruction you pointed to (unless a breakpoint is encountered first). The CodeView debugger sounds a warning and take no action if you try to go to a comment line or other source line that does not correspond to code.

If the line you wish to go to is in another module, you can use the Load command from the Files menu to load the source file for the other module. Then move the cursor to the destination line and press the **F7** key.

Go Command

Dialog

To execute the Go command with a dialog command, enter a command line with the following syntax:

G [<*breakaddress*>]

If the command is given with no argument, execution continues until a breakpoint or the end of the program is encountered.

The Goto form of the command can be given by specifying *breakaddress*. The *breakaddress* can be given as a symbol, a line number, or an address in the *segment:offset* format. If the offset address is given without a segment, the address in the **CS** register is used as the default segment. If you give *breakaddress* as a line number, but the corresponding source line is a comment, declaration, or blank line, the following message appears:

```
No code at this line number
```

Examples

The following examples show the Go command in sequential mode. In window mode there would be no output from the commands, but the display would be updated to show changes caused by the command.

```
>G

Program terminated normally (0)
>
```

The example above passes control to the instruction pointed to by the current values of the **CS** and **IP** registers. No breakpoint is encountered, so the CodeView debugger executes to the end of the program, where it prints a termination message and the exit code returned by the program (**0** in the example).

```
>S+    ;* FORTRAN/BASIC example (source mode)
source
>G BUBBLE
17:          A = B + C
>
```

In the example above, the display mode is first set to source (S+). (See Chapter 10, "Examining Code," for information on setting the display mode.) When the Go command is entered, the CodeView debugger starts program execution at the current address and continues until it reaches the start of the subprogram BUBBLE.

```
>S&   ;* C example (mixed mode)
mixed
>G .22
AX=02F4  BX=0002  CX=00A8  DX=0000  SP=3036  BP=3042  SI=0070  DI=40E0
DS=49B7  ES=49B7  SS=49B7  CS=3FB0  IP=0141  NV UP EI PL NZ NA PO NC
22:           x[i] = x[j];
3FB0:0141 8B7608       MOV     SI,Word Ptr [BP+08]         SS:304A=0070
>
```

The example above passes execution control to the program at the current address and executes to the address of source line **22**. If the address with the breakpoint is never encountered (for example, if the program has less than 22 lines, or if the breakpoint is on a program branch that is never taken), the CodeView debugger executes to the end of the program.

Note

Mixed and source mode can be used equally well with all three languages. The examples alternate languages in this chapter simply to be accessible to more users.

```
>S-
assembly
>G #2A8
AX=0049  BX=0049  CX=028F  DX=0000  SP=12F2  BP=12F6  SI=04BA  DI=1344
DS=5DAF  ES=5DAF  SS=5DAF  CS=58BB  IP=02A8  NV UP EI PL NZ NA PE NC
58BB:02A8 98            CBW
>
```

The example above executes to the hexadecimal address **CS:2A8**. Since no segment address is given, the **CS** register is assumed.

Execute Command

The Execute command is similar to the Go command with no arguments, except that it executes in slow motion (several source lines per second). Execution starts at the current address and continues to the end of the program or until a breakpoint, tracepoint, or watchpoint is reached. You can also stop automatic program execution by pressing any key.

Keyboard

To execute code in slow motion with a keyboard command, press <ALT>**r** to open the Run menu, and then press <ALT>**e** to select Execute.

Dialog

To execute code in slow motion with a dialog command, enter a command line with the following syntax:

E

You cannot set a destination for the Execute command as you can for the Go command.

6

In sequential mode, the output from the Execute command depends on the display mode (source, assembly, or mixed). In assembly or mixed mode, the command executes one instruction at a time. The command displays the current status of the registers and the instruction. In mixed mode, it also shows a source line if there is one at the instruction. In source mode, the command executes one source line at a time, displaying the lines as it executes them.

Important

> The Execute command has the same command letter (**E**) as the Enter command. If the command has at least one argument, it is interpreted as Enter; if not, it is interpreted as Execute.

Restart Command

The Restart command restarts the current program. The program is ready to be executed just as if you had restarted the CodeView debugger. Program variables are reinitialized, but any existing breakpoints or watch statements are retained. The pass count for all breakpoints is reset to 1. Any program arguments are also retained, though they can be changed with the dialog version of the command.

The Restart command can only be used to restart the current program. If you wish to load a new program, you must exit and restart the CodeView debugger with the new program name.

Keyboard

To restart the program with a keyboard command, press <ALT>**r** to open the Run menu, and then press either <ALT>**r** to select Restart or <ALT>**s** to select Start. The program is restarted. If the Restart selection is chosen, the program is ready to start executing from the beginning (but not actually running). If the Start selection is chosen, the program starts executing from the beginning and continues until a breakpoint or the end of the program is encountered.

Dialog

To restart the program with a dialog command, enter a command line with the following syntax:

L [<*arguments*>]

When you restart using the dialog version of the command, the program is ready to start executing from the beginning. If you want to restart with new program arguments, you can give optional *arguments*. You cannot specify new arguments with the keyboard version of the command.

Note

The command letter **L** is a mnemonic for Load, but the command should not be confused with the Load selection from the File menu, since that selection only loads a source file without restarting the program.

Examples

```
>L
>
```

The example above restarts the current executable file, retaining the same breakpoints, watchpoints, tracepoints, and command line arguments.

```
>L 6
>
```

The example above restarts the current executable file, but with **6** as the new program argument.

6

Chapter 7

Examining Data and Expressions

Introduction

The CodeView debugger provides several commands for examining different kinds of data, including expressions, symbols, memory, and registers. The data-evaluation commands discussed in this chapter are summarized below.

Command	Action
Display Expression (**?**)	Evaluates and displays the value of symbols or expressions
Examine Symbol (**X?**)	Displays the addresses of symbols
Dump (**D**)	Displays sections of memory containing data (with variations for examining different kinds of data)
Compare Memory (**C**)	Compares two blocks of memory, byte by byte
Search Memory (**S**)	Scans memory for specified byte values
Register (**R**)	Shows the current values of each register and each flag
8087 (**7**)	Shows the current value in the 80387 or 80287 register

Display Expression Command

The Display Expression command displays the value of a CodeView expression.

Each of the expression evaluators (C, FORTRAN, BASIC, and Pascal) accepts a different set of symbols, operators, functions, and constants, as explained in Chapter 5, "Code View Expressions." The resulting expressions can contain the intrinsic functions listed for the FORTRAN- and BASIC-expression evaluators. They may also contain functions that are part of the executable file. The simplest form of expression is a symbol representing a single variable or routine.

Note

> FORTRAN subroutines and BASIC subprograms do not return values as functions do. They can be used in expressions, and in fact may be useful for observing side effects. However, the value returned by the expression is meaningless.

In addition to displaying values, the Display Expression command can also set values as a side effect. For example, with the C-expression evaluator you can increment the variable *n* by using the expression ++*n* with the Display Expression command. With the FORTRAN-expression evaluator you would use *N*=*N*+**1**, and with the BASIC-expression evaluator you would use **LET** *N*=*N*+**1**. After being incremented, the new value is displayed.

7

You can specify the format in which the values of expressions are displayed by the Display Expression command. Type a comma after the expression, followed by a CodeView format specifier. The format specifiers used in the CodeView debugger are a subset of those used by the C **printf** function. They are listed in table 7.1.

Table 7.1
CodeView Format Specifiers

Character	Output Format	Sample Expression	Sample Output
d	Signed decimal integer	`?40000,d`	`40000`
i	Signed decimal integer	`?40000,i`	`40000`
u	Unsigned decimal integer	`?40000,u`	`40000`
o	Unsigned octal integer	`?40000,o`	`116100`
x or X	Hexadecimal integer	`?40000,x`	`9c40`
f	Signed value in floating-point decimal format with six decimal places	`?3./2.,f`	`1.500000`
e or E	Signed value in scientific-notation format with up to six decimal places (trailing zeros and decimal point are truncated)	`?3./2.,e`	`1.500000e+000`
g or G	Signed value with floating-point decimal format (**f**) or scientific-notation format (**g** or **G**), whichever is more compact	`?3./2.,g`	`1.5`
c	Single character	`?65,c`	`A`
s	Characters printed up to the first null character	`?"String",s`	`String`

FORTRAN and BASIC have no unsigned data types. Using an unsigned format specifier has no effect on the output of positive numbers, but causes negative numbers to be output as positive values.

Hexadecimal letters are uppercase if the type is **X** and lowercase if the type is **x**.

The ''E'' is uppercase if the type is **E** or **G**; lowercase if the type is **e** or **g**.

The **s** string format is used only with the C-expression evaluator; it prints characters up to the first null.

If no format specifier is given, single- and double-precision real numbers are displayed as if the format specifier had been given as **g**. (If you are familiar with the C language, you should note that the **n** and **p** format specifiers and the **F** and **H** prefixes are not supported by the CodeView debugger, even though they are supported by the C **printf** function.)

The prefix **h** can be used with the integer format specifiers (**d**, **o**, **u**, **x**, and **X**) to specify a two-byte integer. The prefix **l** can be used with the same types to specify a four-byte integer. For example, the command **?100000,ld** produces the output **100000**. However, the command **?100000,hd** evaluates only the low-order two bytes, producing the output **-31072**.

When calling a FORTRAN subroutine that uses alternate returns, the value of the return labels in the actual parameter list must be 0. For example, the subroutine call **CALL PROCESS (s-1I,*10,J,*20,*30)** must be called from the debugger as **?PROCESS(IARG1,0,IARG2,0,0)**. Using other values as return labels cause the error `Type clash in function argument` or `Unknown symbol`.

Note

Do not use a type specifier when evaluating strings in FORTRAN, BASIC, or Pascal. Simply leave off the type specifier, and the expression evaluator displays the string correctly. The **s** type specifier assumes the C language string format, with which other languages conflict; if you use **s**, then the debugger simply displays characters at the given address until a null is encountered.

Keyboard

The Display Expression command cannot be executed with a keyboard command.

Dialog

To display the value of an expression using a dialog command, enter a command line with the following syntax:

? *<expression>*[,*<format>*]

The *expression* is any valid CodeView expression, and the optional *format* is a CodeView format specifier.

The remainder of this section first gives examples that are relevant to all languages, and then gives examples specific to C, FORTRAN, BASIC and Pascal.

If you are debugging code written with the assembler, you use the C-expression evaluator by default. Consult the section "Assembly Expressions" in Chapter 5 for guidelines on how to use the C-expression evaluator with assembly code.

Examples

```
>? amount
500
>? amount,x
1f4
>? amount,o
764
>
```

The example above displays the value stored in the variable *amount*, an integer. This value is first displayed in the system radix (in this case, decimal), then in hexadecimal, and then in octal.

7

```
>? 92,x
5c
>? 109*(35+2),o
7701
>? 118,c
v
>
```

The example above shows how the CodeView debugger can be used as a calculator. You can convert between radixes, calculate the value of constant expressions, or check ASCII equivalences.

```
>? chance,f
0.083333
>? chance,e
8.333333e-002
>? chance,E
8.333333E-002
```

The example above shows a double-precision real number, **chance**, displayed in three formats. The **f** format always displays six digits of precision. The **e** format uses scientific notation. Note that the **E** format yields essentially the same display as **e** does.

The rest of the examples in this section are specific to particular languages.

C Examples

The following examples assume that a C source file is being debugged, and that it contains the following declarations:

```
char *text = "Here is a string.";
int  amount;
struct {
      char name[20];
      int  id;
      long class;
} student, *pstudent;

int square(int);
```

Assume also that the program has been executed to the point where the above variables have been assigned values, and that the C-expression evaluator is in use.

7

```
>? text, X
13F3
>DA 0x13F3
3D83:13F0   Here is a string.
>? text,s
Here is a string.
>
```

The example above shows how to examine strings. One method is to evaluate the variable that points to the string, and then dump the values at that address (the Dump commands are explained in the section "Dump Commands" in this chapter). A more direct method is to use the **s** type specifier.

```
>? student.id
19643
>? pstudent->id
19643
>
```

The example above illustrates how to display the values of members of a structure. The same syntax applies to unions.

```
>? amount
500
>? ++amount
501
>? amount=600
600
>
```

The example above shows how the Display Expression command can be used with the C-expression evaluator to change the values of variables.

```
>? square(9)
81
>
```

The example above shows how functions can be evaluated in expressions. The CodeView debugger executes the function **square** with an argument of **9**, and displays the value returned by the function. You can only display function values after you have executed into the function **main**.

Assembly Examples

By default, the C-expression evaluator is used for debugging assembly modules. However, some C expressions are particularly helpful for debugging assembly code. Some typical examples are presented below.

```
>? BY bx
12
>
```

The example above displays the first byte at the location pointed to by **BX**, and is equivalent to the assembly expression **BYTE PTR [bx]**.

```
>? WO bp+8
9359
>
```

The example above displays the first word at the location pointed to by **[bp+8]**.

```
>? DW si+12
12555324
>
```

The example above displays the first double word at the location pointed to by **[si+12]**.

```
>? (char) var
5
>? (int) var
1005
>
```

The last two examples use type casts, which are similar to the assembler **PTR** operator. The expression **(char) var** displays the byte at the address of **var**, in signed format. The expression **(int) var** displays the word at the same address, also in signed format. You can alter either of these commands to display results in unsigned format simply by using the **u** format specifier.

```
>? (char) var,u

>? (int) var,u
```

7

Examine Symbols Command

The Examine Symbols command displays the names and addresses of symbols, and the names of modules, defined within a program. You can specify the symbol or group of symbols you want to examine by module, procedure, or symbol name.

Keyboard

The Examine Symbols command cannot be executed with a keyboard command.

Dialog

To view the addresses of symbols with a dialog command, enter a command line in one of the following formats,

X*
X
X? [<*module*>!] [<*routine*>.] [<*symbol*>] [*]

in which *routine* is in a program unit, such as a C function or a BASIC subprogram, capable of having its own local variables.

The syntax combinations are listed in more detail below.

Syntax	Display
X?<module>!<routine>.<symbol>	The specified *symbol* in the specified *routine* in the specified *module*
X?<module>!<routine>.*	All symbols in the specified *routine* in the specified *module*
X?<module>!<symbol>	The specified *symbol* in the specified *module* (symbols within routines are not found)

X?<module>!*	All symbols in the specified *module*
X?<routine>.<symbol>	The specified *symbol* in the specified *routine* (looks for *routine* first in the current module, and then in other modules from first to last)
X?<routine>.*	All symbols in the specified *routine* (looks for *routine* first in the current module, and then in other modules from first to last)
X?<symbol>	Looks for the specified *symbol* in this order: 1. In the current routine 2. In the current module 3. In other modules, from first to last
X?*	All symbols in the current routine
X*	All module names
X	All symbolic names in the program, including all modules and all symbols

---

Note

When you debug an assembly module, you cannot use the *routine* field; you *must* use the *module* field. Therefore, the only versions of this command that work with assembly modules are the following:

X?<*module*>!*
X?<*module*>!<*symbol*>

C Examples

For the following examples, assume that the program being examined is called **pi**, and that it consists of two modules: **pi.c** and **math.c**. The `pi.c` module is a skeleton consisting only of the **main** function, whereas the **math.c** module has several functions. Assume that the current function is **div** within the **math** module.

```
>X*             ;*Example 1
pi
math
/lib/slibc.a(chkstk)
/lib/slibc.a(crt0)
.
.
.
/lib/slibc.a(itoa)
/lib/slibc.a(unlink)
>
```

Example 1 lists the two user-created modules of the program, as well as the library modules used in the program.

```
>X?*            ;*Example 2
          DI          int                b
          [BP-0006]   int                quotient
          SI          int                i
          [BP-0002]   int                remainder
          [BP+0004]   int                divisor
>
```

Example 2 lists the symbols in the current function (**div**). Local variables are shown as being stored either in a register (**b** in register **DI**) or at a memory location specified as an offset from a register (**divisor** at location **[BP+0004]**).

```
>X?pi!*         ;* Example 3
3D37:19B2 int    _scratch0    3D37:0A10  char    _p[]
3D37:2954 int    _scratch1    3D377:19B4 char    _t[]
3D37:2956 int    _scratch2    3D377:19B0 int     _q
3A79:0010 int    _main()      3A79:0010  int     main()
3D37:19B2 int     scratch0
3D37:0A10 char    p[]
3D37:2954 int     scratch1
3D37:19B4 char    t[]
3D37:2956 int     scratch2
3D37:19B0 int     q
>
```

Example 3 shows all the symbols in the **pi.c** module.

7

```
>X?math!div.*          ;*Example 4
3A79:0264 int                 div()
         DI         int                 b
         [BP-0006]  int                 quotient
         SI         int                 i
         [BP-0002]  int                 remainder
         [BP+0004]  int                 divisor
>
```

Example 4 shows the symbols in the **div** function in module **math.c**. You wouldn't need to specify the module if **math.c** were the current module, but you would if the current module were **pi.c**.

Variables local to a function are indented under that function.

```
>X?math!arctan.s ;* Example 5
3A79:00FA int                 arctan()
         [BP+0004]  int                 s
>
```

Example 5 shows one specific variable (*s*) within the **arctan** function.

7

Dump Commands

The CodeView debugger has several commands for dumping data from memory to the screen (or other output device). The Dump commands are listed below.

Command	Command Name
D	Dump (size is the default type)
DB	Dump Bytes
DA	Dump ASCII
DI	Dump Integers
DU	Dump Unsigned Integers
DW	Dump Words
DD	Dump Double Words
DS	Dump Short Reals
DL	Dump Long Reals
DT	Dump 10-Byte Reals

Keyboard

The Dump commands cannot be executed with keyboard commands.

Dialog

To execute a Dump command with a dialog command, enter a command line with the following syntax:

D[<*type*>] [<*address*> | <*range*>]

The *type* is a one-letter specifier that indicates the type of the data to be dumped. The Dump commands expect either a starting *address* or a *range* of memory. If the starting *address* is given, the commands assume a default range (usually determined by the size of the dialog window) starting at *address*. If *range* is given, the commands dump from the start to the end of *range*. The maximum size of *range* is 32K.

If neither *address* nor *range* is given, the commands assume the current dump address as the start of the range and the default size associated with the size of the object as the length of the range. The Dump Real commands have a default range size of one real number. The other Dump commands have a default size determined by the size of the dialog window (if you are in window mode), or a default size of 128 bytes otherwise.

The current dump address is the byte following the last byte specified in the previous Dump command. If no Dump command has been used during the session, the dump address is the start of the data segment (**DS**). For example, if you enter the Dump Words command with no argument as the first command of a session, the CodeView debugger displays the first 64 words (128 bytes) of data declared in the data segment. If you repeat the same command, the debugger displays the next 64 words following the ones dumped by the first command.

Note

> If the value in memory cannot be evaluated as a real number, the Dump commands that display real numbers (Dump Short Reals, Dump Long Reals, or Dump 10-Byte Reals) display a number containing one of the following character sequences: **#NAN**, **#INF**, or **#IND**. NAN (not a number) indicates that the data cannot be evaluated as a real number. INF (infinity) indicates that the data evaluates to infinity. IND (indefinite) indicates that the data evaluates to an indefinite number.

The following sections discuss the variations of the Dump commands in order of the size of data they display.

Dump

Syntax

D [<*address*> | <*range*>]

The Dump command displays the contents of memory at the specified *address* or in the specified *range* of addresses. The command dumps data in the format of the default type. The default type is the last type specified with a Dump, Enter, Watch Memory, or Tracepoint Memory command. If none of these commands has been entered during the session, the default type is bytes.

The Dump command displays one or more lines, depending on the address or range specified. Each line displays the address of the first item displayed. The Dump command must be separated by at least one space from any *address* or *range* value. For example, to dump memory starting at symbol **a**, use the command **D a**, not **Da**. The second syntax would be interpreted as the Dump ASCII command.

Dump Bytes

Syntax

DB [<address> | <range>]

The Dump Bytes command displays the hexadecimal and ASCII values of the bytes at the specified *address* or in the specified *range* of addresses. The command displays one or more lines, depending on the address or range supplied.

Each line displays the address of the first byte in the line, followed by up to 16 hexadecimal byte values. The byte values are immediately followed by the corresponding ASCII values. The hexadecimal values are separated by spaces, except the eighth and ninth values, which are separated by a dash (-). ASCII values are printed without separation. Unprintable ASCII values (less than 32 or greater than 126) are displayed as dots. No more than 16 hexadecimal values are displayed in a line. The command displays values and characters until the end of the *range* or, if no *range* is given, until the first 128 bytes have been displayed.

Example

```
>DB 0 36
3D5E:0000 53 6F 6D 65 20 6C 65 74-74 65 72 73 20 61 6E 64 Some letters and
3D5E:0010 20 6E 75 6D 62 65 72 73-3A 00 10 EA 89 FC FF EF  numbers:.......
3D5E:0020 00 F0 00 CA E4          -                         .....
>
```

The example above displays the byte values from **DS:0** to **DS:36** (36 decimal is equivalent to 24 hexadecimal). The data segment is assumed if no segment is given. ASCII characters are shown on the right.

Dump ASCII

Syntax

DA [*<address>* | *<range>*]

The Dump ASCII command displays the ASCII characters at a specified *address* or in a specified *range* of addresses. The command displays one or more lines of characters, depending on the *address* or *range* specified.

If no ending address is specified, the command dumps either 128 bytes or all bytes preceding the first null byte, whichever comes first. Up to 64 characters per line are displayed. Unprintable characters, such as carriage returns and line feeds, are displayed as dots. ASCII characters less than 32 and greater than 126 in number are unprintable.

Examples

```
>DA 0
3D7C:0000  Some letters and numbers:
>
```

The example above displays the ASCII values of the bytes starting at **DS:0**. Since no ending address is given, values are displayed up to the first null byte.

```
>DA 0 36
3D7C:0000  Some letters and numbers:...........
>
```

In the example above, an ending address is given, so the characters from **DS:0** to **DS:36** (24 hexadecimal) are shown. Unprintable characters are shown as dots.

Dump Integers

Syntax

DI [<*address*> | <*range*>]

The Dump Integers command displays the signed decimal values of the words (two-byte values) starting at *address* or in the specified *range* of addresses. The command displays one or more lines, depending on the address or range specified. Each line displays the address of the first integer in the line, followed by up to eight signed decimal words. The values are separated by spaces. The command displays values until the end of the *range* or until the first 64 two-byte integers have been displayed, whichever comes first.

Note

In this manual an integer is considered a two-byte value, since the CodeView debugger assumes that integer size.

7

Example

```
>DI 0 36
3D5E:0000  28499  25965  27680  29797  25972  29554  24864  25710
3D5E:0010  28192  28021  25954  29554     58  -5616   -887  -4097
3D5E:0020  -4096 -13824   2532
>
```

The example above displays the byte values from **DS:** to **DS:36** (24 hexadecimal). Compare the signed decimal numbers at the end of this dump with the same values shown as unsigned integers in the following section.

Dump Unsigned Integers

Syntax

DU [<*address*> | <*range*>]

The Dump Unsigned Integers command displays the unsigned decimal values of the words (two-byte values) starting at *address* or in the specified *range* of addresses. The command displays one or more lines, depending on the address or range specified. Each line displays the address of the first unsigned integer in the line, followed by up to eight decimal words. The values are separated by spaces. The command displays values until the end of the *range* or until the first 64 unsigned integers have been displayed, whichever comes first.

Example

```
>DU 0 36
3D5E:0000   28499  25965  27680  29797  25972  29554  24864  25710
3D5E:0010   28192  28021  25954  29554     58  59920  64649  61439
3D5E:0020   61440  51712   2532
>
```

The example above displays the byte values from **DS:0** to **DS:36** (24 hexadecimal). Compare the unsigned decimal numbers at the end of this dump with the same values shown as signed integers in the section ''Dump Integers'' in this chapter.

Dump Words

Syntax

DW [<*address*> | <*range*>]

The Dump Words command displays the hexadecimal values of the words (two-byte values) starting at *address* or in the specified *range* of addresses. The command displays one or more lines, depending on the address or range specified. Each line displays the address of the first word in the line, followed by up to eight hexadecimal words. The hexadecimal values are separated by spaces. The command displays values until the end of the *range* or until the first 64 words have been displayed, whichever comes first.

Example

```
>DW 0 36
3D5E:0000   6F53 656D 6C20 7465 6574 7372 6120 646E
3D5E:0010   6E20 6D75 6562 7372 003A EA10 FC89 EFFF
3D5E:0020   F000 CA00 09E4
>
```

The example above displays the word values from **DS:0** to **DS:36** (24 hexadecimal). No more than eight values per line are displayed.

Dump Double Words

Syntax

DD [<*address*]> | <*range*>]

The Dump Double Words command displays the hexadecimal values of the double words (four-byte values) starting at *address* or in the specified *range* of addresses.

The command displays one or more lines, depending on the address or range specified. Each line displays the address of the first double word in the line, followed by up to four hexadecimal double-word values. The words of each double word are separated by a colon. The values are separated by spaces. The command displays values until the end of the *range* or until the first 32 double words have been displayed, whichever comes first.

Example

```
>DD 0 36
3D5E:0000  656D:6F53 7465:6C20 7372:6574 646E:6120
3D5E:0010  6D75:6E20 7372:6562 EA10:003A EFFF:FC89
3D5E:0020  CA00:F000 6F73:09E4
>
```

The example above displays the double-word values from **DS:0** to **DS:36** (24 hexadecimal). No more than four double-word values per line are displayed.

Dump Short Reals

Syntax

DS [<*address*> | <*range*>]

The Dump Short Reals command displays the hexadecimal and decimal values of the short (four-byte) floating-point numbers at *address* or in the specified *range* of addresses.

The command displays one or more lines, depending on the address or range specified. Each line displays the address of the floating-point number in the first column. Next, the hexadecimal values of the bytes in the number are shown, followed by the decimal value of the number. The hexadecimal values are separated by spaces.

7

The decimal value has the following form:

[-]<*digit*>.<*digits*>**E**{+ | -<*exponent*>

If the number is negative, it has a minus sign; positive numbers have no sign. The first digit of the number is followed by a decimal point. Six decimal places are shown following the decimal point. The letter **E** follows the decimal digits, and marks the start of a three-digit signed *exponent*.

The command displays at least one value. If a *range* is specified, all values in the range are displayed.

Example

```
>DS SPI
5E68:0100  DB 0F 49 40  3.141593E+000
>
```

The example above displays the short-real floating-point number at the address of the variable *SPI*. Only one value is displayed per line.

Dump Long Reals

Syntax

DL [<*address*> | <*range*>]

The Dump Long Reals command displays the hexadecimal and decimal values of the long (eight-byte) floating-point numbers at the specified *address* or in the specified *range* of addresses.

The command displays one or more lines, depending on the address or range specified. Each line displays the address of the floating-point number in the first column. Next, the hexadecimal values of the bytes in the number are shown, followed by the decimal value of the number. The hexadecimal values are separated by spaces.

The decimal value has the following form:

[-]<*digit*>.<*digits*>**E**{+ | -}<*exponent*>

If the number is negative, it has a minus sign; positive numbers have no sign. The first digit of the number is followed by a decimal point. Six decimal places are shown following the decimal point. The letter **E** follows the decimal digits, and marks the start of a three-digit signed *exponent*.

The command displays at least one value. If a *range* is specified, all values in the range are displayed.

Example

```
>DL LPI
5E68:0200  11 2D 44 54 FB 21 09 40  3.141593E+000
>
```

The example above displays the long-real floating-point number at the address of the variable *LPI*. Only one value per line is displayed.

Dump 10-Byte Reals

Syntax

DT [<*address*> | <*range*>]

The Dump 10-Byte Reals command displays the hexadecimal and decimal values of the 10-byte floating-point numbers at the specified *address* or in the specified *range* of addresses.

The command displays one or more lines, depending on the address or range specified. Each line displays the address of the floating-point number in the first column. Next, the hexadecimal values of the bytes in the number are shown, followed by the decimal value of the number. The hexadecimal values are separated by spaces.

The decimal value has the following form:

[-]<*digit*>.<*digits*>**E**{+ | -}<*exponent*>

If the number is negative, it has a minus sign; positive numbers have no sign. The first digit of the number is followed by a decimal point. Six decimal places are shown following the decimal point. The letter **E** follows the decimal digits, and marks the start of a three-digit signed *exponent*.

The command displays at least one value. If a *range* is specified, all values in the range are displayed.

Example

```
>DT TPI
5E68:0300  DE 87 68 21 A2 DA 0F C9 00 40  3.141593E+000
>
```

The example above displays the 10-byte floating-point number at the address of the variable *TPI*. Only one number per line is displayed.

Compare Memory Command

The Compare Memory command provides a convenient way for comparing two blocks of memory, specified by absolute addresses. This command is primarily of interest to programmers using assembly mode; however, it can be useful to anyone who wants to compare efficiently two large areas of data, such as arrays.

Keyboard

The Compare Memory command cannot be executed with a keyboard command.

Dialog

To compare two blocks of memory, enter a command line with the following syntax:

C *<range1start> <range1end> <range2start> <address>*

The bytes in the memory locations specified by *range* are compared with the corresponding bytes in the memory locations beginning at *address*. If one or more pairs of corresponding bytes do not match, each pair of mismatched bytes is displayed.

Examples

```
>C 100 01FF 300    ;* hexadecimal radix assumed
39BB:0102  0A 00  39BB:0302
39BB:0108  0A 01  39BB:0308
>
```

The first example (in which hexadecimal is assumed to be the default radix) compares the block of memory from 100 to 1FF with the block of memory from 300 to 3FF. It indicates that the third and ninth bytes differ in the two areas of memory.

```
>C arr1[0] L 100 arr2[0]    ;* C notation used.
>
```

The example compares the 100 bytes starting at the address of **arr1[0]**, with the 100 bytes starting at address of **arr2[0]**. The CodeView debugger produces no output in response, so this indicates that the first 100 bytes of each array are identical.

Note

You can enter the Compare Memory command using any radix you like; however, any output is still in hexadecimal format.

Search Memory Command

The Search Memory command (not to be confused with the Search command discussed in Section 11.6) scans a specified area of memory, looking for specific byte values. It is primarily of interest to programmers using assembly mode, and to users who want to test for the presence of specific values within a range of data.

Keyboard

The Search Memory command cannot be executed with a keyboard command.

Dialog

To search a block of memory, enter the Search Memory command with the following syntax:

S *<range>* *<list>*

The debugger searches the specified *range* of memory locations for the byte values specified in the *list*. If bytes with the specified values are found, then the debugger displays the addresses of each occurrence of bytes in the list.

The *list* can have any number of bytes. Each byte value must be separated by a space or comma, unless the list is an ASCII string. If the list contains more than one byte, then the Search Memory command looks for a series of bytes that precisely match the order and value of bytes in *list*. If found, then the beginning address of each such series is displayed.

Examples

```
>S buffer L 1500 "error"
2BBA:0404
2BBA:05E3
2BBA:0604
>
```

The first example displays the address of each memory location containing the string **error**. The command searches the first 1500 bytes at the address specified by **buffer**. The string was found at the three addresses displayed by the CodeView debugger.

```
>S DS:100 200 0A   ;* hexadecimal radix assumed
3CBA:0132
3CBA:01C2
>
```

The second example displays the address of each memory location that contains the byte value **0A** in the range DS:0100 to DS:0200 (hexadecimal). The value was found at two addresses.

7

Register Command

The Register command has two functions. It displays the contents of the central processing unit (CPU) registers. It can also change the values of the registers. The display features of the Register command are explained here. The modification features of the command are explained in Chapter 11, "Modifying Code or Data."

The flag register display colors are significant; if a flag bit is set, the two-letter code for that condition is displayed as BRIGHT (for monochromatic monitors) or RED (for color monitors). If the flag is clear, the two-letter code for that cleared flag is displayed as NORMAL_INTENSITY (for monochromatic monitors) or CYAN (for color monitors).

Keyboard

To display the registers using a keyboard command in window mode, press the **F2** key. The register window appears on the right side of the screen. If the register window is already on the screen, the same command removes it.

In sequential mode, the **F2** key displays the current status of the registers. (This produces the same effect as entering the Register dialog command with no argument.)

Dialog

7

To display the registers in the dialog window (or sequentially in sequential mode), enter a command line with the following syntax:

R

The current values of all registers and flags are displayed. The instruction at the address pointed to by the current **CS** and **IP** register values is also shown. (The Register command can also be given with arguments, but only when used to modify registers, as explained in Chapter 11, "Modifying Code or Data.")

If the display mode is source (S+) or mixed (**S&**) (see "Set Mode Command" in Chapter 10 for more information), the current source line is also displayed by the Register command. If an operand of the instruction contains memory expressions or immediate data, the CodeView debugger evaluates operands and show the value to the right of the instruction. This value is referred to as the ''effective address,'' and is also displayed at the bottom of the register window. If the **CS** and **IP** registers are currently at a breakpoint location, the register display indicates the breakpoint number.

In sequential mode, the Trace (**T**), Program Step (**P**), and Go (**G**) commands show registers in the same format as the Register command.

Examples

```
>S&
mixed
>R
AX=0005  BX=299E  CX=0000  DX=0000  SP=3800  BP=380E  SI=0070  DI=40D1
DS=5067  ES=5067  SS=5067  CS=4684  IP=014F  NV UP EI PL NZ NA PO NC
35:                  VARIAN = (N*SUMXSQ-SUMX**2)/(N-1)
4684:014F 8B5E06        MOV     BX,Word Ptr [BP+06]     ;BR1  SS:3814=299E
>
```

The example above displays all register and flag values, as well as the instruction at the address pointed to by the **CS** and **IP** registers. Because the mode has been set to mixed (**S&**), the current source line is also shown. The example is from a FORTRAN program, but applies equally well to BASIC and C programs.

```
>S-
assembly
>R
AX=0005  BX=299E  CX=0000  DX=0000  SP=3800  BP=380E  SI=0070  DI=40D1
DS=5067  ES=5067  SS=5067  CS=4684  IP=014F  NV UP EI PL NZ NA PO NC
4684:014F 8B5E06        MOV     BX,Word Ptr [BP+06]     ;BR1  SS:3814=299E
>
```

In the example above, the display mode is set to assembly (**S-**), so no source line is shown. Note the breakpoint number at the right of the last line, indicating that the current address is at Breakpoint 1.

8087 Command

The 8087 command dumps the contents of the 8087 registers. If you do not have an 8087, 80287, or 80387 coprocessor chip on your system, then this command dumps the contents of the pseudoregisters created by the operating system's floating point emulator.

Note

This section does not attempt to explain how the registers of the Intel 8087, 80287, and 80387 processors are organized or how they work. In order to interpret the command output, you must learn about the chip from an Intel reference manual or other book on the subject. Since emulator routines mimic the behavior of the 8087 coprocessor, these references apply to emulator routines as well as to the chips themselves.

Keyboard

The 8087 command cannot be executed with a keyboard command.

Dialog

To display the status of the math co-processor chip (or floating-point emulator routines) with a dialog command, enter a command line with the following syntax:

7

The current status of the chip is displayed when you enter the command. In window mode, the output is to the dialog window.

The following example shows a display for this command.

8087 Example

```
>7
Control 037F  (Projective closure, Round nearest, 64-bit precision)
                     iem=0 pm=1 um=1 om=1 zm=1 dm=1 im=1
Status   6004  cond=1000 top=4 pe=0 ue=0 oe=0 ze=1 de=0 ie=0
Tag      A1FF  instruction=59380  operand=59360  opcode=D9EE
Stack          Exp  Mantissa           Value
ST(3) special 7FFF 8000000000000000 = + Infinity
ST(2) special 7FFF 0101010101010101 = + Not a Number
ST(1) valid   4000 C90FDAA22168C235 = +3.141592265110390E+000
ST(0) zero    0000 0000000000000000 = +0.000000000000000E+000
>
```

In the example above, the first line of the dump shows the current closure method, rounding method, and the precision. The number **037F** is the hexadecimal value in the control register. The rest of the line interprets the bits of the number. The closure method can be either projective (as in the example) or affine. The rounding method can be either rounding to the nearest even number (as in the example), rounding down, rounding up, or using the chop method of rounding (truncating toward zero). The precision may be 64 bits (as in the example), 53 bits, or 24 bits.

The second line of the display indicates whether each exception mask bit is set or cleared. The masks are interrupt-enable mask **(iem)**, precision mask **(pm)**, underflow mask **(um)**, overflow mask **(om)**, zero-divide mask **(zm)**, denormalized-operand mask **(dm)**, and invalid-operation mask **(im)**.

The third line of the display shows the hexadecimal value of the status register (**6004** in the example), and then interprets the bits of the register. The condition code **(cond)** in the example is the binary number **1000**. The top of the stack **(top)** is register 4 (shown in decimal). The other bits shown are precision exception **(pe)**, underflow exception **(ue)**, overflow exception **(oe)**, zero-divide exception **(ze)**, denormalized-operand exception **(de)**, and invalid-operation exception **(ie)**.

The fourth line of the display first shows the hexadecimal value of the tag register (**A1FF** in the example). It then gives the hexadecimal values of the instruction **(59380)**, the operand **(59360)**, and the operation code, or opcode, **(D9EE)**.

The fifth line is a heading for the subsequent lines, which contain the contents of each 8087, 80287, or 80387 stack register. The registers in the example contain four types of numbers that may be held in these registers. Starting from the bottom, register 0 contains zero. Register 1 contains a valid real number. Its exponent (in hexadecimal) is **4000** and its mantissa is **C90FDAA22168C235**. The number is shown in scientific notation in the rightmost column. Register 2 contains a value that cannot be interpreted as a number, and register 3 contains infinity.

Chapter 8

Managing Breakpoints

Introduction

The CodeView debugger enables you to control program execution by setting breakpoints. A breakpoint is an address that stops program execution each time the address is encountered. By setting breakpoints at key addresses in your program, you can "freeze" program execution and examine the status of memory or expressions at that point.

The commands listed below control breakpoints:

Command	Action
Breakpoint Set (**BP**)	Sets a breakpoint and, optionally, a pass count and break commands
Breakpoint Clear (**BC**)	Clears one or more breakpoints
Breakpoint Disable (**BD**)	Disables one or more breakpoints
Breakpoint Enable (**BE**)	Enables one or more breakpoints
Breakpoint List (**BL**)	Lists all breakpoints

In addition to these commands, the Watchpoint (**WP**) and Tracepoint (**TP**) commands can be used to set conditional breakpoints (see Chapter 10, "Examining Code," for information on these two commands).

8

Breakpoint Set Command

The Breakpoint Set command (**BP**) creates a breakpoint at a specified address. Any time a breakpoint is encountered during program execution, the program halts and waits for a new command.

The CodeView debugger allows up to 20 breakpoints (0 through 19). Each new breakpoint is assigned to the next available number. Breakpoints remain in memory until you delete them or until you quit the debugger. They are not canceled when you restart the program. Because breakpoints are not automatically canceled, you are able to set up a complicated series of breakpoints, then execute through the program several times without resetting.

If you try to set a breakpoint at a comment line or other source line that does not correspond to code, the CodeView debugger displays the following message:

```
No code at this line number
```

Keyboard

To set a breakpoint with a keyboard command in window mode, move the cursor to the source line or instruction where you want to set a breakpoint. You may have to press the **F6** key to move the cursor to the display window. When the cursor is on the appropriate source line, press the **F9** key. The line is displayed in high-intensity text, and remains so until you remove or disable the breakpoint.

In sequential mode, the **F9** key can be used to set a breakpoint at the current location. You must use the dialog version of the command to set a breakpoint at any other location.

Dialog

To set a breakpoint using a dialog command, enter a command line with the following syntax:

BP [*<address>* [*<passcount>*] [*<commands>*]]

If no *address* is given, a breakpoint is created on the current source line in source mode, or on the current instruction in assembly mode. You can specify the *address* in the *segment:offset* format or as a source line, a routine name, or a label. If you give an offset address, the code segment is assumed.

The dialog version of the command is more powerful than the mouse or keyboard version in that it allows you to give a *passcount* and a string of *commands*. The *passcount* specifies the first time the breakpoint is to be taken. For example, if the pass count is 5, the breakpoint is ignored the first four times it is encountered, and taken the fifth time. Thereafter, the breakpoint is always taken.

The *commands* are a list of dialog commands enclosed in quotation marks (" ") and separated by semicolons (;). For example, if you specify the commands as **?code;T""**, the CodeView debugger automatically displays the value of the variable *code* and then execute the Trace command each time the breakpoint is encountered. The Trace and Display Expression commands are described in Chapter 6, ''Executing Code,'' and Chapter 7, ''Examining Data and Expressions,'' respectively.

In window mode, a breakpoint entered with a dialog command has exactly the same effect as one created with a window command. The source line or instruction corresponding to the breakpoint location is shown in high-intensity text.

In sequential mode, information about the current instruction is displayed each time you execute to a breakpoint. The register values, the current instruction, and the source line may be shown, depending on the display mode. See Chapter 10, ''Examining Code,'' for more information about display modes.

When a breakpoint address is shown in the assembly-language format, the breakpoint number is shown as a comment to the right of the instruction. This comment appears even if the breakpoint is disabled (but not if it is deleted).

Examples

```
>BP .19 10
>
```

The example above creates a breakpoint at line 19 of the current source file (or if there is no executable statement at line 19, at the first executable statement after line 19). The breakpoint is passed over nine times before being taken on the 10th pass.

8

```
>BP STATS 10 "?COUNTER = COUNTER + 1;G"
>
```

The example above creates a breakpoint at the address of the routine **STATS**. The breakpoint is passed over nine times before being taken on the 10th pass. Each time execution stops for the breakpoint, the quoted commands are executed. The Display Expression command increments *COUNTER*, then the Go command restarts execution. If *COUNTER* is set to 0 when the breakpoint is set, this has the effect of counting the number of times the breakpoint is taken.

```
>S-   ;* FORTRAN example - uses FORTRAN hexadecimal notation
assembly
>BP #0a94
>G
AX=0006 BX=304A CX=000B DX=465D SP=3050 BP=3050 SI=00BB DI=40D1
DS=5064 ES=5064 SS=5064 CS=46A2 IP=0A94 NV UP EI PL NZ NA PE NC
46A2:0A94 7205      JB     __chkstk+13 (0A9B)     ;BR1
>
```

The example above first sets the mode to assembly, and then creates a breakpoint at the hexadecimal (offset) address **#0A94** in the default (**CS**) segment. (The same address would be specified as **0x0A94** with the C-expression evaluator, and as **&H0A9** with the BASIC-expression evaluator.) The Go command (**G**) is then used to execute to the breakpoint. Note that in the output to the Go command, the breakpoint number is shown as an assembly-language comment (**;BR1**) to the right of the current instruction. The Go command displays this output only in sequential mode; in window mode no assembly-language information appears.

8

Breakpoint Clear Command

The Breakpoint Clear command (**BC**) permanently removes one or more previously set breakpoints.

Keyboard

To clear a single breakpoint with a keyboard command, move the cursor to the breakpoint line or instruction you want to clear. Breakpoint lines are shown in high-intensity text. Press the **F9** key. The line is shown in normal text to indicate that the breakpoint has been removed.

To remove all breakpoints using a keyboard command, press <ALT>**r** to open the Run menu, and then press <ALT>**c** to select Clear Breakpoints.

Dialog

To clear breakpoints using a dialog command, enter a command line with the following syntax:

> **BC** *<list>*
> **BC** *

If *list* is specified, the command removes the breakpoints named in the list. The *list* can be any combination of integer values from 0 to 19. You can use the Breakpoint List command (**BL**) if you need to see the numbers for each existing breakpoint. If an asterisk (*) is given as the argument, all breakpoints are removed.

Examples

```
>BC 0 4 8
>
```

The example above removes breakpoints 0, 4, and 8.

```
>BC *
>
```

The example above removes all breakpoints.

Breakpoint Disable Command

The Breakpoint Disable command (**BD**) temporarily disables one or more existing breakpoints. The breakpoints are not deleted. They can be restored at any time using the Breakpoint Enable command (**BE**).

When a breakpoint is disabled in window mode, it is shown in the display window with normal text; when enabled, it is shown in high-intensity text.

Note

All disabled breakpoints are automatically enabled whenever you restart the program being debugged. The program can be restarted with the Start or Restart selection from the Run menu, or with the Restart dialog command (**L**). See Chapter 6, ''Executing Code.''

Keyboard

The Breakpoint Disable command cannot be executed with a keyboard command.

Dialog

To disable breakpoints with a dialog command, enter a command line with the following syntax:

BD <*list*>
BD *

If *list* is specified, the command disables the breakpoints named in the list. The *list* can be any combination of integer values from 0 to 19. Use the Breakpoint List command (**BL**) if you need to see the numbers for each existing breakpoint. If an asterisk (*) is given as the argument, all breakpoints are disabled.

The window commands for setting and clearing breakpoints can also be used to enable or clear disabled breakpoints.

Examples

```
>BD 0 4 8
>
```

The example above disables breakpoints 0, 4, and 8.

```
>BD *
>
```

The example above disables all breakpoints.

Breakpoint Enable Command

The Breakpoint Enable command (**BE**) enables breakpoints that have been temporarily disabled with the Breakpoint Disable command.

Keyboard

To enable a disabled breakpoint using a keyboard command, move the cursor to the source line or instruction of the breakpoint, and then press the **F9** key. The line is displayed in high-intensity text, and remains so until you remove or disable the breakpoint. This is the same as creating a new breakpoint at that location.

Dialog

To enable breakpoints using a dialog command, enter a command line with the following syntax:

> **BE** *<list>*
> **BE** *

If *list* is specified, the command enables the breakpoints named in the list. The *list* can be any combination of integer values from 0 to 19. Use the Breakpoint List command (**BL**) if you need to see the numbers for each existing breakpoint. If an asterisk (*) is given as the argument, all breakpoints are enabled. The CodeView debugger ignores all or part of the command if you try to enable a breakpoint that is not disabled.

Examples

```
>BE 0 4 8
>
```

The example above enables breakpoints 0, 4, and 8.

```
>BE*
>
```

The example above enables all disabled breakpoints.

Breakpoint List Command

The Breakpoint List command (**BL**) lists current information about all breakpoints.

Keyboard

The Breakpoint List command cannot be executed with a keyboard command.

Dialog

To list breakpoints with a dialog command, enter a command line with the following syntax:

BL

The command displays the breakpoint number, the enabled status (**e** for "enabled", **d** for "disabled"), the address, the routine, and the line number. If the breakpoint does not fall on a line number, an offset is shown from the nearest previous line number. The pass count and break commands are shown if they have been set. If no breakpoints are currently defined, nothing is displayed.

Example

```
>BL
0 e 56C4:0105  _ARCTAN:10
1 d 56C4:011E  _ARCTAN:19              (pass = 10) "T;T"
2 e 56C4:00FD  _ARCTAN:9+6
>
```

8

In the example above, breakpoint 0 is enabled at address **56C4:0105**. This address is in routine **ARCTAN** and is at line **10** of the current source file. No pass count or break commands have been set.

Breakpoint 1 is currently disabled, as indicated by the **d** after the breakpoint number. It also has a pass count of 10, meaning that the breakpoint is not taken until the 10th time it is encountered. The command string at the end of the line indicates that each time the breakpoint is taken, the Trace command is automatically executed twice.

The line number for breakpoint 2 has an offset. The address is six bytes beyond the address for line 9 in the current source file. Therefore, the breakpoint was probably set in assembly mode, since it would be difficult to set a breakpoint anywhere except on a source line in source mode.

Chapter 9

Managing Watch Statements

Introduction

Watch Statement commands are among the CodeView debugger's most powerful features. They enable you to set, delete, and list watch statements. Watch statements describe expressions or areas of memory to watch. Some watch statements specify conditional breakpoints, which depend upon the value of the expression or memory area. The Watch Statement commands are summarized below:

Command	Action
Watch (**W**)	Sets an expression or range of memory to be watched
Watchpoint (**WP**)	Sets a conditional breakpoint that is taken when the expression becomes nonzero (true)
Tracepoint (**TP**)	Sets a conditional breakpoint that is taken when a given expression or range of memory changes
Watch Delete (**Y**)	Deletes one or more watch statements
Watch List (**W**)	Lists current watch statements

Watch statements, like breakpoints, remain in memory until you specifically remove them or quit the CodeView debugger. They are not canceled when you restart the program being debugged. Therefore, you can set a complicated series of watch statements once, and then execute through the program several times without resetting.

In window mode, Watch Statement commands can be entered either in the dialog window or with menu selections. Current watch statements are shown in a watch window that appears between the menu bar and the source window.

In sequential mode, the Watch, Tracepoint, and Watchpoint commands can be used, but since there is no watch window, you cannot see the watch statements and their values. You must use the Watch List command to examine the current watch statements.

9

Note

In order to set a watch statement containing a local variable, you must be in the function where the variable is defined. If the current line is not in the function, the CodeView debugger displays the message `UNKNOWN SYMBOL`. When you exit from a function containing a local variable referenced in a watch statement, the value of the statement is displayed as `UNKNOWN SYMBOL`. When you reenter the function, the local variable again has a value. With the C expression evaluators, you can avoid this limitation by using the period operator to specify both the function and the variable. For example, enter **main.x** instead of just **x**.

Setting Watch-Expression and Watch-Memory Statements

The Watch command is used to set a watch statement that specifies an expression (watch-expression statement) or a range of addresses in memory (watch-memory statement). The value or values specified by this watch statement are shown in the watch window. The watch window is updated to show new values each time the value of the watch statement changes during program execution. Since the watch window does not exist in sequential mode, you must use the Watch List command to examine the values of watch statements.

When setting a watch expression, you can specify the format in which the value is displayed. Type the expression followed by a comma and a format specifier. If you do not give a format specifier, the CodeView debugger displays the value in a default format. See ''Display Expression Command'' in Chapter 7 for more information about type specifiers and the default format.

Keyboard

To set a watch-expression statement with a keyboard command, press <ALT>**w** to open the Watch menu, and then type **A** (uppercase or lowercase) to select Add Watch. You can also select the Add Watch command directly by pressing <CTL>**w**. A dialog box appears, asking for the expression to be watched. Type the expression and press the <RETURN> key.

You cannot use the keyboard version of the command to specify a range of memory to be watched, as you can with the dialog version.

Dialog

To set a watch-expression statement or watch-memory statement with a dialog command, enter a command line with the following syntax:

W? *<expression>*[,*<format>*]	Watch expression
W[*<type>*] *<range>*	Watch memory

9

Setting Watch-Expression and Watch-Memory Statements

An *expression* used with the Watch command can be either a simple variable or a complex expression using several variables and operators. The expression should be no longer than the width of the watch window. The characters permitted for *format* correspond to format arguments used in a C **printf** function call. See "Display Expression Command" in Chapter 7 for more information on format arguments.

When watching a memory location, *type* is a one-letter size specifier from the following list:

Specifier	Size
None	Default type
B	Byte
A	ASCII
I	Integer (signed decimal word)
U	Unsigned (unsigned decimal word)
W	Word
D	Double word
S	Short real
L	Long real
T	10-byte real

If no type size is specified, the default type used is the last type used by a Dump, Enter, Watch Memory, or Tracepoint Memory command. If none of these commands has been used during the session, the default type is byte.

The data is displayed in a format similar to that used by the Dump commands (see "Display Expression Command" in Chapter 7 for more information on format arguments). The *range* can be any length, but only one line of data is displayed in the watch window. If you do not specify an ending address for the range, the default range is one object.

Examples

The following three examples display watch statements in the watch window.

```
W? n
```

The example above displays the current value of the variable *n*.

```
W? higher * 100
```

The example above displays the value of the expression **higher * 100**.

```
WL chance
```

The example above displays the double-precision floating-point **chance**, first showing exactly how it is stored in memory. (The command **W? chance** would display the value of **chance** but not any actual bytes of memory.)

These commands, entered while debugging a C program, produce the watch window in the following figure.

```
File View Search Run Watch Options Language Calls Help  | F8=Trace   F5=Go
─────────────────────────────── | dice.C | ───────────────────────────────
0) n  :  4
1) higher * 100  :  33.33333333333333
2) chance  :  5958:115A  55 55 55 55 55 55 B5 3F  +8.333333333333E-002
3) higher > chance  :  1
4) n==7 || n==11 : 0
5) sum  :  0.00000000000000
6) 5958:1172  04 .
──────────────────────────────────────────────────────────────────────────
30:                 sum = sum + roll(n);
31:         else {
32:                 chance = roll(n);
33:                 higher = make(n)
34:                 sum = sum + (chance * higher);
35:                 printf("%s %2d ", strl, n);
──────────────────────────────────────────────────────────────────────────
>W? n
>W? higher * 100
>WL chance
>WP? higher > chance
>WP? n==7 || n==11
>TP? sum
>TPB n
>_
```

9

The first three items in the watch window are simple watch statements. They display values but never cause execution to break.

The next two items are watchpoints; they cause execution to break whenever they evaluate to true (nonzero). The fourth item breaks execution whenever *higher* is greater than *chance*, and the fifth item breaks execution whenever *n* is equal to 7 or 11. Setting watchpoints is described in detail later in this chapter.

The last two items are tracepoints, which cause execution to break whenever any bytes change within a specified area of memory. The sixth item breaks execution whenever the value of *sum* changes; the seventh item breaks execution whenever there is a change in the first byte at the address of *n*. Setting tracepoints is described in detail later in this chapter.

Setting Watchpoints

The Watchpoint command is used to set a conditional breakpoint called a watchpoint. A watchpoint breaks program execution when the expression described by its watch statement becomes true. You can think of watchpoints as "break when" points, since the break occurs when the specified expression becomes true (nonzero).

A watch statement created by the Watchpoint command describes the expression that is watched and compared to 0. The statement remains in memory until you delete it or quit the CodeView debugger. Any valid CodeView expression can be used as the watchpoint expression as long as the expression is not wider than the watch window.

In window mode, watchpoint statements and their values are displayed in high-intensity text in the watch window. In sequential mode, there is no watch window, so the values of watchpoint statements can only be displayed with the Watch List command (see the section "Listing Watchpoints and Tracepoints" for more information).

Although watchpoints can be any valid CodeView expression, the command works best with expressions that use the relational operators (such as < and > for C. Relational expressions always evaluate to false (zero) or true (nonzero). Care must be taken with other kinds of expressions when used as watchpoints, because the watchpoints breaks execution whenever they do not equal precisely zero. For example, your program might use a loop variable **I**, which ranges from 1 to 100. If you entered **I** as a watchpoint, then it would always suspend program execution, since **I** is never equal to 0. However, the relational expression **I>90** (or **I.GT.90**) would not suspend program execution until **I** exceeded 90.

Keyboard

To execute the Watchpoint command with a keyboard command, press <ALT>**w** to open the Watch menu, and then press <ALT>**w** to select Watchpoint. A dialog box appears, asking for the expression to be watched. Type the expression and press the <RETURN> key.

Setting Watchpoints

Dialog

To set a watchpoint using a dialog command, enter a command line with the following syntax:

WP? *<expression>*[,*<format>*]

The *expression* can be any valid CodeView expression (usually a relational expression). You can enter a format specifier, but there is little reason to do so, since the expression value is normally either 1 or 0.

Examples

The following dialog commands display two watch statements (watchpoints) in the watch window:

```
WP? higher > chance        ;* C example
```

The examples above instruct the CodeView debugger to break execution when the variable *higher* is greater than the variable *chance*. After setting this watchpoint, you could use the Go command to execute until the condition becomes true.

```
WP? n==7 || n==11          ;* C example
```

The example above instructs the CodeView debugger to break execution when the variable *n* is equal to 7 or 11.

Note

C displays a numerical result in response to a Boolean expression (0 being equivalent to false, nonzero to true).

9

Note

Setting watchpoints significantly slows execution of the program being debugged. The CodeView debugger checks if the expression is true each time a source line is executed in source mode, or each time an instruction is executed in assembly mode. Be careful when setting watchpoints near large or nested loops. A loop that executes almost instantly when run normally can take many minutes if executed from within the debugger with several watchpoints set.

Tracepoints do not slow CodeView execution as much as watchpoints, so you should use tracepoints when possible. For example, although you can set a watchpoint on a Boolean variable (*WP? moving*), a on the same variable (*TP? moving*) has essentially the same effect and does not slow execution as much.

If you enter a seemingly endless loop, press <DEL> to exit. You soon learn the size of loop you can safely execute when watchpoints are set.

Setting Tracepoints

The Tracepoint command is used to set a conditional breakpoint called a tracepoint. A tracepoint breaks program execution when the value of a specified expression or range of memory changes.

The watch statement created by the Tracepoint command describes the expression or memory range to be watched and tested for change. The statement remains in memory until you delete it or quit the CodeView debugger.

In window mode, tracepoint statements and their values are shown in high-intensity text in the watch window. In sequential mode, there is no watch window, so the values of tracepoint statements can only be displayed with the Watch List command (see the section "Listing Watchpoints and Tracepoints" in this chapter for more information).

An expression used with the Tracepoint command must evaluate to an "lvalue." In other words, the expression must refer to an area of memory rather than a constant. Furthermore, the area of memory must be not more than 128 bytes in size. For example, **i==10** would be invalid because it is either 1 (true) or 0 (false) rather than a value stored in memory. The expression **sym1+sym2** is invalid because it is the calculated sum of the value of two memory locations. The expression **buffer** would be invalid if **buffer** is an array of 130 bytes, but valid if the array is 120 bytes. Note that if **buffer** is declared as an array of 64 bytes, then the Tracepoint command given with the expression **buffer** checks all 64 bytes of the array. The same command given with the C expression **buffer[32]**, means that only one byte (the 33rd) is checked.

Note

Register variables are not considered lvalues. Therefore, if *i* is declared as **register int i**, the command **TP? i** is invalid. However, you can still check for changes in the value of *i*. Use the Examine Symbols command to learn which register contains the value of *i*. Then learn the value of *i*. Finally, set up a watchpoint to test the value. For example, use the following sequence of commands:

```
>X? i
3A79:0264 int             div()
            SI        int              i
>?i
10
>WP? @SI!=10
>
```

When setting a tracepoint expression, you can specify the format in which the value is displayed. Type the expression followed by a comma and a type specifier. If you do not give a type specifier, the CodeView debugger displays the value in a default format. See ''Display Expression Command'' in Chapter 7 for more information about type specifiers and the default format.

Keyboard

To set a tracepoint-expression statement with a keyboard command, press <ALT>**w** to open the Watch menu, and then press <ALT>**t** to select Trace point. A dialog box appears, asking for the expression to be watched. Type the expression and press the <RETURN> key.

You cannot use the keyboard version of the command to specify a range of memory to be watched, as you can with the dialog version.

Setting Tracepoints

Dialog

To set a tracepoint with a dialog command, enter a command line with one of the following forms of syntax:

TP? *<expression>*,[*<format>*]	@Tracepoint expression
TP[*<type>*] *<range>*	@Tracepoint memory

An *expression* used with the Tracepoint command can be either a simple variable or a complex expression using several variables and operators. The expression should not be longer than the width of the watch window. You can specify *format* using a C **printf** type specifier if you do not want the value to be displayed in the default format (decimal for integers or floating point for real numbers). See ''Display Expression Command'' in Chapter 7 for more information on format arguments.

In the memory-tracepoint form, *range* must be a valid address range and *type* must be a one-letter memory-size specifier. If you specify only the start of the range, the CodeView debugger displays one object as the default.

Although no more than one line of data is displayed in the watch window, the range to be checked for change can be any size up to 128 bytes. The data is displayed in the format used by the Dump commands (see ''Display Expression Command,'' in Chapter 7 for more information on format arguments). The valid memory-size specifiers are listed below:

Specifier	Size
None	Default type
B	Byte
A	ASCII
I	Integer (signed decimal word)
U	Unsigned (unsigned decimal word)
W	Word
D	Double word
S	Short real

L Long real

T 10-byte real

The default type used if no type size is specified is the last type used by a Dump, Enter, Watch Memory, or Tracepoint Memory command. If none of these commands has been used during the session, the default type is byte.

Examples

The two dialog commands below display watch statements (tracepoints) in the watch window.

```
TP? sum
```

The example above instructs the CodeView debugger to suspend program execution whenever the value of the variable *sum* changes.

```
TPB n
```

The example above instructs the CodeView debugger to suspend program execution whenever the first byte at the address of *n* changes; the address of this byte and its contents are displayed. The value of *n* may change because of a change in the *second* byte at the address of *n*; but that change (by itself) would have no effect on this tracepoint.

Note

Setting tracepoints significantly slows execution of the program being debugged. The CodeView debugger has to check to see if the expression or memory range has changed each time a source line is executed in source mode or each time an instruction is executed in assembly mode. However, tracepoints do not slow execution as much as do watchpoints.

Be careful when setting tracepoints near large or nested loops. A loop that executes almost instantly when run from the operating system can take many minutes if executed from within the debugger with several tracepoints set. If you enter a seemingly endless loop, press <DEL> to exit. Often you can tell how far you went in the loop by the value of the tracepoint when you exited.

Deleting Watch Statements

The Watch Delete command enables you to delete watch statements that were set previously with the Watch, Watchpoint, or Tracepoint command.

When you delete a watch statement in window mode, the statement disappears and the watch window closes around it. For example, if there are three watch statements in the window and you delete statement 1, the window is redrawn with one less line. Statement 0 remains unchanged, but statement 2 becomes statement 1. If there is only one statement, the window disappears.

Keyboard

To execute the Delete Watch command with a keyboard command, press <ALT>**w** to open the Watch menu, and then type **D** (uppercase or lowercase) to select Delete Watch. You can also select the Delete Watch command directly by pressing <CTL>**u**. A dialog box appears, containing all the watch statements. Use the UP and DOWN arrow keys to move the cursor to the statement you want to delete, and then press the <RETURN> key. The dialog box disappears, and the watch window is redrawn without the watch statement.

You can also delete all the statements in the watch window at once, simply by selecting the Delete All selection. Do this by pressing **L** (uppper-case or lowercase) after the Watch menu is open.

Dialog

To delete watch statements with a dialog command, enter a command line with the following syntax:

Y *<number>*

When you set a watch statement, it is automatically assigned a number (starting with 0). In window mode, the number appears to the left of the watch statement in the watch window. In sequential mode, you can use the Watch List (**W**) command to view the numbers of current watch statements.

You can delete existing watch statements by specifying the *number* of the statement you want to delete with the Delete Watch command. (The **Y** is a mnemonic for "yank.")

Deleting Watch Statements

You can use the asterisk (*) to represent all watch statements.

Examples

```
>Y 2
>
```

The command above deletes watch statement 2.

```
>Y *
>
```

The command above deletes all watch statements and closes the watch window.

9

Listing Watchpoints and Tracepoints

The Watch List command lists all previously set watchpoints and with their assigned numbers and their current values.

This command is the only way to examine current watch statements in sequential mode. The command has little use in window mode, since watch statements are already visible in the watch window.

Keyboard

The Watch List command cannot be executed with a keyboard command.

Dialog

To list watch statements with a dialog command, enter a command line with the following syntax:

W

The display is the same as the display that appears in the watch window in window mode.

Example

```
>W
0) code,c  :  I
1) (float)letters/words,f  :  4.777778
2) 3F65:0B20  20 20 43 4F 55 4E 54 COUNT
3) lines==11 :  0
>
```

9

Note

The command letter for the Watch List command is the same as the command letter for the memory version of the Watch command when no memory size is given. The difference between the commands is that the Watch List command never takes an argument. The Watch command always requires at least one argument.

Assembly Examples

By default, assembly source modules are debugged with the C-expression evaluator. Therefore, refer to the C examples for appropriate syntax for entering watch expressions.

In addition, however, certain C expressions tend to be more useful for debugging assembly modules. The following examples show some typical cases used with watch and tracepoint commands.

Examples

```
>WW sp L 8
>WW bp L 8
>W? wo bp+4,d
>W? by bp-2,d
>TPW arr L 5
>
```

The first two examples watch a range of memory. The watch command **WW sp L 8** is particularly useful because it causes the debugger to watch the stack dynamically; the debugger continually displays the first eight words on the top of the stack as items are pushed and popped. The expression **WW bp L 8** is similar; it causes the debugger to watch the first eight words in memory pointed to by **BP** (the framepointer).

The third example, **W? wo bp+4,d,** is useful if you are using the stack to pass parameters. In this case, the position on the stack four bytes above **BP** holds one of three integer parameters. The **WO** operator returns the same value as the assembler expression **WORD PTR [bp+4]**; the result is displayed in decimal.

You must use the expression **bp+4** in order to watch this parameter; you cannot specify a parameter by name. The assembler does not emit symbolic information for parameters. The fourth command, **W? by bp-2,d**, is similar to the third, but instead of watching a parameter, this command watches a local variable. The operator **BY** returns the same value as the assembler expression **BYTE PTR [bp-2]**.

The final example sets a tracepoint on a range of memory, which corresponds to the first five words of the array *arr*. Range arguments for tracepoint and watch expressions are particularly useful for large data structures, such as arrays. The five examples above produce the following screen, when entered in a CodeView debugging session:

```
File View Search Run Watch Options Language Calls Help  | F8=Trace   F5=Go
 ----------------------------- | test.ASM | ------------------------------
0)  sp L 8 : 531C:09A2  0044 09B4 0037 0005 000F 001B 000F 0005 |AX = 001B
1)  bp L 8 : 531C:09A4  09B4 0037 0005 000F 001B 000F 0005 001B |BX = 09A2
2)  wo bp+4,d : 5                                               |CX = 0044
3)  by bp-2,d : 60                                              |DX = 00B0
4)  531F:0006 01 00 02 00 03.....                               |SP = 09A2
---------------------------------------------------------------- BP = 09A4
70:  ;  First parameter largest                                 |SI = 0098
71:  ;                                                          |DI = 0A8C
72:         mov     BYTE PTR [bp-2],1 ; Load indicator value    |DS = 531C
73:                                   ; of 1 into local variabl |ES = 531C
74:         jmp     SHORT finished    ; and finish up           |SS = 531C
75:  next_test:                                                 |CS = 52D7
76:         mov     ax,[bp+8]       ; Load 3rd parm into ax     |IP = 005D
77:         cmp     [bp+6],ax       ; If 2nd parm <= 3rd parm   |
78:         jle     last_test       ;    go to last test        |  NV UP
79:  ;                                                          |  EI NG
----------------------------------------------------------------|  NZ AC
>WW sp L 8                                                      |  PE CY
>WW bp L 8                                                      |
>W? wo bp+4,d                                                   |  SS:09AA
>W? by bp-2,d                                                   |     000F
>TPB arr L 5                                                    |
>_
```

Chapter 10

Examining Code

Introduction

Several CodeView commands allow you to examine program code or data related to code. The following commands are discussed in this chapter:

Command	Action
Set Mode (**S**)	Sets format for code displays
Unassemble (**U**)	Displays assembly instructions
View (**V**)	Displays source lines
Current Location (**.**)	Displays the current location line
Stack Trace (**K**)	Displays routines or procedures

Set Mode Command

The Set Mode command sets the mode in which code is displayed. The two basic display modes are source mode, in which the program is displayed as source lines, and assembly mode, in which the program is displayed as assembly-language instructions. These two modes can be combined in mixed mode, in which the program is displayed with both source lines and assembly-language instructions.

In sequential mode, there are three display modes: source, assembly, and mixed. These modes affect the output of commands that display code (Register, Trace, Program Step, Go, Execute, and Unassemble).

In window mode, these same display modes are available, but affect what kind of code appears in the display window.

Source and mixed modes are only available if the executable file contains symbols in the CodeView format. Programs that do not contain symbolic information are displayed in assembly mode.

Keyboard

To change the display mode with a keyboard command, press the **F3** key. This rotates the mode to the next setting; you may need to press **F3** twice to get the desired mode. This command works in either window or sequential mode. In sequential mode, the word *source*, *mixed*, or *assembly* is displayed to indicate the new mode.

Dialog

To set the display mode from the dialog window, enter a command line with the following syntax:

S[+ | - | &]

If the plus sign is specified (S+), source mode is selected, and the word **source** is displayed.

If the minus sign is specified (S-), assembly mode is selected, and the word **assembly** is displayed. In window mode, the display includes any assembly options, except the Mixed Source option, previously toggled on from the Options menu. The Mixed Source option is always turned off by the S- command.

If the ampersand is specified (**S&**), mixed mode is selected, and the word **mixed** is displayed. In window mode, the display includes any assembly options previously toggled on from the Options menu. In addition, the Mixed Source option is turned on by the **S&** command.

If no argument is specified (**S**), the current mode (*source*, *assembly*, or *mixed*) is displayed.

The Unassemble command in sequential mode is an exception in that it displays mixed source and assembly with both the source (**S+**) and mixed (**S&**) modes. When you enter the dialog version of the Set Mode command, the CodeView debugger outputs the name of the new display mode: *source*, *assembly*, or *mixed*.

Examples

```
>S+
source
>S-
assembly
>S&
mixed
>
```

The examples above show the source mode being changed to *source*, *assembly*, and *mixed*. In window mode, the commands change the format of the display window. In sequential mode, the commands change the output from the commands that display code (Register, Trace, Program Step, Go, Execute, and Unassemble). See the sections on individual commands for examples of how they are affected by the display mode.

Unassemble Command

The Unassemble command displays the assembly-language instructions of the program being debugged. It is most useful in sequential mode, where it is the only method of examining a sequence of assembly-language instructions. In window mode it can be used to display a specific portion of assembly-language code in the display window.

Note

Occasionally, code similar to the following is displayed:

```
FE30    ???    Byte Ptr   [BX + SI]
```

If you attempt to unassemble data, then the CodeView debugger may display meaningless instructions.

Keyboard

The Unassemble command has no direct keyboard equivalent, but you can view unassembled code at any time by changing the mode to assembly or mixed (see the section ''Set Mode Command'' in this chapter for more information).

Dialog

To display unassembled code using a dialog command, enter a command line with the following syntax:

U [<*address*> | <*range*>]

The effect of the command varies depending on whether you are in sequential or window mode.

In sequential mode, if you do not specify *address* or *range*, the disassembled code begins at the current unassemble address and shows the next eight lines of instructions. The unassemble address is the address of the instruction after the last instruction displayed by the previous Unassemble command. If the Unassemble command has not been used during the session, the unassemble address is the current instruction.

If you specify an *address*, the disassembly starts at that address and shows the next eight lines of instructions. If you specify a *range*, the instructions within the range are displayed.

The sequential mode format of the display depends on the current display mode (see "Set Mode Command" for more information). If the mode is source (**S+**) or mixed (**S&**), the CodeView debugger displays source lines mixed with unassembled instructions. One source line is shown for each corresponding group of assembly-language instructions. If the display mode is assembly, only assembly-language instructions are shown.

In window mode, the Unassemble command changes the mode of the display window to assembly. The display format reflects any options previously set from the Options menu. There is no output to the dialog window. If *address* is given, the instructions in the display window begin at the specified address. If *range* is given, only the starting address is used. If no argument is given, the debugger scrolls down and displays the next screen of assembly-language instructions.

Note

The 80286 protected-mode mnemonics (also available with the 80386) cannot be displayed with the Unassemble command.

Examples

```
>S&
mixed
>U 0x11
49D0:0011 35068E      XOR  AX,__sqrtjmptab+8cd4  (8E06)
49D0:0014 189A230     SBB  Byte Ptr [BP+SI+0023],BL
49D0:0018 FC          CLD
49D0:0019 49          DEC  CX
49D0:001A CD351ED418  INT  35 ;FSTP  DWord Ptr [__fpinit+ee  (18D4)]
49D0:001F CD3D        INT  3D ;FWAIT
7:     A = 0.0
49D0:0021 CD35EE      INT  35 ;FLDZ
```

The sequential mode example above sets the mode to mixed and unassembles eight lines of machine code, plus whatever source lines are encountered within those lines. The display would be the same if the mode were source.

The example demonstrates sequential mode.

```
>S-
assembly
>U 0x11
49D0:0011 35068E      XOR  AX,__sqrtjmptab+8cd4 (8E06)
49D0:0014 189A2300    SBB  Byte Ptr [BP+SI+0023],BL
49D0:0018 FC          CLD
49D0:0019 49          DEC  CX
49D0:001A CD351ED418  INT  35 ;FSTP  DWord Ptr [__fpinit+ee (18D4)]
49D0:001F CD3D        INT  3D ;FWAIT
49D0:0021 CD35EE      INT  35 ;FLDZ
>
```

The sequential mode example above sets the mode to assembly and repeats the same command.

View Command

The View command displays the lines of a text file (usually a source module or include file). It is most useful in sequential mode, where it is the only method of examining a sequence of source lines. In window mode, the View command can be used to page through the source file or to load a new source file.

Keyboard

To load a new source file with a keyboard command, press <ALT>**f** to open the File menu, then press L to select Load. A dialog box appears, asking for the name of the file you wish to load. Type the name of the file, and press the <RETURN> key. The new file appears in the display window.

The paging capabilities of the View command have no direct keyboard equivalent, but you can move about in the source file by first putting the cursor in the display window with the **F6** key, then pressing the <PgUp>, <PgDn>, <HOME>, <END>, UP ARROW, and DOWN ARROW keys. See ''Controlling Program Execution with Keyboard Commands'' in Chapter 3 for more information.

Dialog

To display source lines using a dialog command, enter a command line with the following syntax:

V [<*expression*>]

Since addresses for the View command are often specified as a line number (with an optional source file), a more specific syntax for the command would be as follows:

V [.[<*filename*>:]<*linenumber*>]

The effect of the command varies, depending on whether you are in sequential or window mode.

In sequential mode, the View command displays eight source lines. The starting source line is one of the following:

- The current source line if no argument is given.
- The specified *linenumber*. If *filename* is given, the specified file is loaded, and the *linenumber* refers to lines in it.
- The address that *expression* evaluates to. For example, *expression* could be a procedure name or an address in the *segment:offset* format. The code segment is assumed if no segment is given.

In sequential mode, the View command is not affected by the current display mode (source, assembly, or mixed); source lines are displayed regardless of the mode.

In window mode, if you enter the View command while the display mode is assembly, the CodeView debugger automatically switches back to source mode. If you give *linenumber* or *expression*, the display window are redrawn so that the source line corresponding to the given *address* appears at the top of the source window. If you specify a *filename* with a *linenumber*, the specified file is loaded.

If you enter the View command with no arguments, the display scrolls down one line short of a page; that is, the source line that was at the bottom of the window is at the top.

Note

The View command with no argument is similar to pressing the <PgDn> key. The difference is that pressing the <PgDn> key enables you to scroll down one more line.

Examples

```
>V .math.c:30    ;* Example 1, C source code
30:                 register int j;
31:
32:                 for (j = q; j >= 0; j--)
33:                         if (t[j] + p[j] > 9) {
34:                                 p[j] += t[j] - 10;
35:                                 p[j-1] += 1;
36:                         } else
37:                                 p[j] += t[j];
>
```

Example 1 loads the source file *math.c* and displays eight source lines starting at line **30**.

Current Location Command

The Current Location command displays the source line or assembly-language instruction corresponding to the current program location.

Keyboard

The Current Location command cannot be executed with a keyboard command.

Dialog

To display the current location line using a dialog command, enter a command line with the following syntax (a period only):

In sequential mode, the command displays the current source line. The line is displayed regardless of whether the current debugging mode is source or assembly. If the program being debugged has no symbolic information, the command is ignored.

In window mode, the command puts the current program location (marked with reverse video or a contrasting color) in the center of the display window. The display mode (source or assembly) is not affected. This command is useful if you have scrolled through the source code or assembly-language instructions so that the current location line is no longer visible.

For example, if you are in window mode and have executed the program being debugged to somewhere near the start of the program, but you have scrolled the display to a point near the end, the Current Location command returns the display to the current program location.

Example

```
>.
MINDAT = 1.0E6
>
```

The example above illustrates how to display the current source line in sequential mode. The same command in window mode would not produce any output, but it could change the text that is shown in the display window.

Stack Trace Command

The Stack Trace command allows you to display routines that have been called during program execution (see note below). The first line of the display shows the name of the current routine. The succeeding lines (if any) list any other routines that were called to reach the current address. The dialog version of the Stack Trace command also displays the source lines where each routine was called.

For each routine, the values of any arguments are shown in parentheses after the routine name. Values are shown in the current radix (the default is decimal).

The term ''stack trace'' is used because, as each routine is called, its address and arguments are stored on (pushed onto) the program stack. Therefore, tracing through the stack shows the currently active routines. With C programs, the **main** routine is always near the bottom of the stack. Only routines called by the main program are displayed.

Note

> This discussion uses the term ''routines,'' which is a general term for functions, subroutines, procedures, subprograms, and function procedures. Each of which uses the stack to transfer control to an independent program unit. In assembly mode, the term ''procedure'' may be more accurate.
>
> If you are using the CodeView debugger to debug assembly-language programs, the Stack Trace command works only if you call procedures with the calling convention appropriate to the procedure's language.

Keyboard

To view a stack trace with a keyboard command, press <ALT>**c** to open the Calls menu. The menu shows the current routine at the top, and other routines below it in the reverse order in which they were called; for example, the first routine called is at the bottom. The values of any routine arguments are shown in parentheses following the routine.

If you want to view one of the routines that was previously called, select the routine by moving the cursor with the arrow keys and then pressing <RETURN> , or by typing the number or letter to the left of the routine. The effect of selecting a routine in the Calls menu is to cause the debugger to display that routine. The cursor is on the last statement that was executed in the routine.

Dialog

To display a stack trace with a dialog command, enter a command line with the following syntax:

K

The output from the Stack Trace dialog command lists the routines in the reverse order in which they were called. The arguments to each routine are shown in parentheses. Finally, the line number from which the routine was called is shown.

You can enter the line number as an argument to the View or Unassemble command if you want to view code at the point where the routine was called.

In window mode, the output from the Stack Trace dialog command appears in the dialog window.

C Example

```
>K
analyze(67,0), line 94
countwords(0,512), line 73
main(2,5098)
>
```

The example above shows the routines on the stack in the reverse order in which they were called. Since **analyze** is on the top, it has been called most recently; in other words, it is the current routine.

Each routine is shown with the arguments it was passed, along with the last source line that it had been executing. Note that **main** is shown with the command line arguments *argc* (which is equal to 2) and *argv* (which is a pointer equal to 5098 decimal). Since the language is C, **main** is always on the bottom of the stack.

Chapter 11

Modifying Code or Data

Introduction

The CodeView debugger provides the following commands for modifying code or data in memory:

Command	Action
Assemble (**A**)	Modifies code
Enter (**E**)	Modifies memory, usually data
Register (**R**)	Modifies registers and flags
Fill Memory (**F**)	Fills a block of memory
Move Memory (**M**)	Copies one block of memory to another

These commands change code temporarily. You can use the alterations for testing in the CodeView debugger, but you cannot save them or permanently change the program. To make permanent changes, you must modify the source code and recompile.

Assemble Command

The Assemble command assembles 8086-family (8086, 8087, 8088, 80186, 80287, and 80286 unprotected) instruction mnemonics and places the resulting instruction code into memory at a specified address. The only 8086-family mnemonics that cannot be assembled are 80286 protected-mode mnemonics. In addition, the debugger also assembles 80386 instructions.

Note

The effects of the Assemble command are temporary. Any instructions that you assemble are lost as soon as you exit the program.

The instructions you assemble are also lost when you restart the program with the Start or Restart command, because the original code is reloaded on top of memory you may have altered.

To test the results of an Assemble command, you may need to manipulate the **IP** register (and possibly the **CS** register) to the starting address of the instructions you have assembled. If you do this, you must use the Current Line command (.) to reset the debugger's internal variables so that it traces properly.

Keyboard

The Assemble command cannot be executed with a keyboard command.

Dialog

To assemble code using a dialog command, enter a command line with the following syntax:

A [<*address*>]

If *address* is specified, the assembly starts at that address; otherwise the current assembly address is assumed.

The assembly address is normally the current address (the address pointed to by the **CS** and **IP** registers). However, when you use the Assemble command, the assembly address is set to the address immediately following the last assembled instruction. When you enter any command that executes code (Trace, Program Step, Go, or Execute), the assembly address is reset to the current address.

When you type the Assemble command, the assembly address is displayed. The CodeView debugger then waits for you to enter a new instruction in the standard 8086-family instruction-mnemonic form. You can enter instructions in uppercase, lowercase, or both.

To assemble a new instruction, type the desired mnemonic and press the <RETURN> key. The CodeView debugger assembles the instruction into memory and displays the next available address. Continue entering new instructions until you have assembled all the instructions you want. To conclude assembly and return to the CodeView prompt, press the <RETURN> key only.

If an instruction you enter contains a syntax error, the debugger displays the message `^ Syntax error`, redisplays the current assembly address, and waits for you to enter a correct instruction. The caret symbol in the message points to the first character the CodeView debugger could not interpret.

The following eight principles govern entry of instruction mnemonics:

1. The far-return mnemonic is **RETF**.

2. String mnemonics must explicitly state the string size. For example, **MOVSW** must be used to move word strings, and **MOVSB** must be used to move byte strings.

3. The CodeView debugger automatically assembles short, near, or far jumps and calls, depending on byte displacement to the destination address. These may be overridden with the **NEAR** or **FAR** prefix, as shown in the following examples:

   ```
   JMP    0x502
   JMP    NEAR 0x505
   JMP    FAR  0x50A
   ```

   The **NEAR** prefix can be abbreviated to **NE**, but the **FAR** prefix cannot be abbreviated. The examples above use the C notation for hexadecimal numbers.

4. The CodeView debugger cannot determine whether some operands refer to a word memory location or to a byte memory location. In

1

these cases, the data type must be explicitly stated with the prefix **WORD PTR** or **BYTE PTR**. Acceptable abbreviations are **WO** and **BY**. Examples are shown below:

```
MOV     WORD PTR [BP],1
MOV     BYTE PTR [SI-1],symbol
MOV     WO PTR [BP],1
MOV     BY PTR [SI-1],symbol
```

5. The CodeView debugger cannot determine whether an operand refers to a memory location or to an immediate operand. The debugger uses the convention that operands enclosed in square brackets refer to memory. Two examples are shown below:

```
MOV     AX,0x21
MOV     AX,[0x21]
```

   The first statement moves **21** hexadecimal into **AX**. The second statement moves the data at offset **0x21** hexadecimal into **AX**.

6. The CodeView debugger supports all forms of indirect register instructions, as shown in the following examples:

```
ADD     BX,[BP+2].[SI-1]
POP     [BP+DI]
PUSH    [SI]
```

7. All instruction-name synonyms are supported. If you assemble instructions and then examine them with the Unassemble command (**U**), the CodeView debugger may show synonymous instructions, rather than the ones you assembled, as shown in the following examples:

```
LOOPZ   0x100
LOOPE   0x100
JA      0x200
JNBE    0x200
```

8. Do not assemble and execute 8087 or 80287 instructions if your system is not equipped with one of these math coprocessor chips. If you try to execute the **WAIT** instruction without the appropriate chip, for example, your system will crash.

Example

```
>U 0x40 L 1
39B0:0040 89C3              MOV           BX,AX
>A 0x40
39B0:0040 MOV    CX,AX
39B0:0042
>U 0x40 L 1
39B0:0040 89C1              MOV           CX,AX
>
```

The Unassemble command (**U**) is used to show the instruction before and after the assembly.

You can modify a portion of code for testing, as in the example, but you cannot save the modified program. You must modify your source code and recompile.

11

Enter Commands

The CodeView debugger has several commands for entering data to memory. You can use these commands to modify either code or data, though code can usually be modified more easily with the Assemble command (A). The Enter commands are listed below:

Command	Command Name
E	Enter (size is the default type)
EB	Enter Bytes
EA	Enter ASCII
EI	Enter Integers
EU	Enter Unsigned Integers
EW	Enter Words
ED	Enter Double Words
ES	Enter Short Reals
EL	Enter Long Reals
ET	Enter 10-Byte Reals

Keyboard

The Enter commands cannot be executed with keyboard commands.

Dialog

To enter data (or code) to memory with a dialog command, enter a command line with the following syntax:

E [<*type*>] <*address*> [<*list*>]

The *type* is a one-letter specifier that indicates the type of the data to be entered. The *address* indicates where the data is entered. If no segment is given in the address, the data segment (**DS**) is assumed.

The *list* can consist of one or more expressions that evaluate to data of the size specified by *type* (the expressions in the list are separated by spaces). This data is entered to memory at *address*. If one of the values in the list is invalid, an error message is displayed. The values preceding the error are entered; values at and following the error are not entered.

The expressions in the list are evaluated in the current radix, regardless of the size and type of data being entered. For example, if the radix is 10 and you give the value 10 in a list with the Enter Words command, the decimal value 10 is entered even though word values are normally entered in hexadecimal. This means that the Enter Words, Enter Integers, and Enter Unsigned Integers commands are identical when used with the list method, since two-byte data are being entered for each command.

If *list* is not given, the CodeView debugger prompts for values to be entered to memory. Values entered in response to prompts are accepted in hexadecimal for the Enter Bytes, Enter ASCII, Enter Words, and Enter Double Words commands. The Enter Integers command accepts signed decimal integers, while the Enter Unsigned Integers command accepts unsigned decimal integers. The Enter Short Reals, Enter Long Reals, and Enter 10-Byte Reals commands accept decimal floating-point values.

With the prompting method of data entry, the CodeView debugger prompts for a new value at *address* by displaying the address and its current value. As explained below, you can then replace the value, skip to the next value, return to a previous value, or exit the command.

- To replace the value, type the new value after the current value.
- To skip to the next value, press the <SPACE> bar . Once you have skipped to the next value, you can change its value or skip to the following value. If you pass the end of the display, the CodeView debugger displays a new address to start a new display line.
- To return to the preceding value, type a backslash (\). When you return to the preceding value, the debugger starts a new display line with the address and value.
- To stop entering values and return to the CodeView prompt, press the <RETURN> key. You can exit the command at any time.

Examples

```
>EW PLACE 16 32
```

The example above shows how to enter two word-sized values at the address **PLACE**.

```
>EW PLACE

3DA5:0B20   00F3._
```

The example above illustrates the prompting method of entering data. When you supply the address where you want to enter data but supply no data to be entered there, the CodeView debugger displays the current value of the address and waits for you to enter a new value. The underscore in this example and the examples below represents the CodeView cursor. You change the value **F3** to the new value 16 (10 hexadecimal) by typing **10** (without pressing the <RETURN> key yet). The value must be typed in hexadecimal for the Enter Words command, as shown below:

```
>EW PLACE

3DA5:0B20   00F3.10_
```

You can then skip to the next value by pressing the <SPACE> key. The CodeView debugger responds by displaying the next value, as shown below:

```
>EW PLACE

3DA5:0B20   00F3.10   4F20._
```

You can then type another hexadecimal value, such as **30**:

```
>EW PLACE

3DA5:0B20   00F3.10   4F20.30_
```

To move to the next value, press the <SPACE> key.

```
>EW PLACE

3DA5:0B20   00F3.10   4F20.30   3DC1._
```

Assume that you realize that the last value entered, **30**, is incorrect. You really wanted to enter **20**. You could return to the previous value by typing a backslash. The CodeView debugger starts a new line, starting with the previous value. Note that the backslash is not echoed on the screen:

```
>EW PLACE

3DA5:0B20   00F3.10   4F20.30   3DC1.
3DA5:0B22   0030._
```

Type the correct value, **20**:

```
>EW PLACE

3DA5:0B20   00F3.10   4F20.30   3DC1.
3DA5:0B22   0030.20_
```

If this is the last value you want to enter, press the <RETURN> key to stop. The CodeView prompt reappears, as shown below:

```
>EW PLACE

3DA5:0B20   00F3.10   4F20.30   3DC1.
3DA5:0B22   0030.20
>_
```

Enter Command

Syntax

E *<address>* [*<list>*]

The Enter command enters one or more values into memory at the specified *address*. The data are entered in the format of the default type, which is the last type specified with a Dump, Enter, Watch Memory, or Tracepoint Memory command. If none of these commands has been entered during the session, the default type is bytes.

Use this command with caution when entering values in the list format; values are truncated if you enter a word-sized value when the default type is actually bytes. If you are not sure of the current default type, specify the size in the command.

1

Note

The Execute command and the Enter command have the same command letter (**E**). The difference is that the Execute command never takes an argument; the Enter command always requires at least one argument.

Enter Bytes Command

Syntax

EB *<address>* [*<list>*]

The Enter Bytes command enters one or more byte values into memory at *address*. The optional *list* can be entered as a list of expressions separated by spaces. The expressions are evaluated and entered in the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in hexadecimal.

The Enter Bytes command can also be used to enter strings, as described in the section ''Enter ASCII Command'' in this chapter.

Examples

```
>EB 256 10 20 30
>
```

If the current radix is 10, the above example replaces the three bytes at DS:256, DS:257, and DS:258 with the decimal values **10**, **20**, and **30**. (These three bytes correspond to the hexadecimal addresses DS:0100, DS:0101, and DS:0102.)

```
>EB 256

3DA5:0100  130F.A
>
```

The example above replaces the byte at DS:256 (DS:0100 hexadecimal) with 10 (0A hexadecimal).

Enter ASCII Command

Syntax

EA *<address>* [*<list>*]

The Enter ASCII command works in the same way as the Enter Bytes command (**EB**) described in the section ''Enter Bytes Command'' in this chapter. The *list* version of this command can be used to enter a string expression.

Example

```
>EA message "File cannot be found"
>
```

In the example above, the string `File cannot be found` is entered starting at the symbolic address **message**. (Note that the double quotation marks are CodeView string delimiters.)

You can also use the Enter Bytes command to enter a string expression, or you can enter nonstring values using the Enter ASCII command.

Enter Integers Command

Syntax

EI *<address>* [*<list>*]

The Enter Integers command enters one or more word values into memory at *address* using the signed-integers format. With the CodeView debugger, a signed integer can be any decimal integer between -32,768 and 32,767.

The optional *list* can be entered as a list of expressions separated by spaces. The expressions are entered and evaluated in the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in decimal.

Examples

```
>EI 256 -10 10 -20
>
```

If the current radix is 10, the example above replaces the three integers at DS:256, DS:258, and DS:260 with the decimal values **-10**, **10**, and **-20**. (The three addresses correspond to the three hexadecimal addresses DS:0100, DS:0102, and DS:0104.)

```
>EI 256

3DA5:0100  130F.-10
>
```

The example above replaces the integer at DS:256 (hexadecimal address DS:0100) with **-10**.

Enter Unsigned Integers Command

Syntax

EU *<address>* [*<list>*]

The Enter Unsigned Integers command enters one or more word values into memory at *address* using the unsigned-integers format. With the CodeView debugger, an unsigned integer can be any decimal integer between 0 and 65,535. The optional *list* can be entered as a list of expressions separated by spaces. The expressions are entered and evaluated in the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in decimal.

11

Examples

```
>EU 256 10 20 30
>
```

If the current radix is 10, the example above replaces the three unsigned integers at DS:256, DS:258, and DS:260 with the decimal values **10**, **20**, and **30**. (These addresses correspond to the hexadecimal addresses DS:0100, DS:0102, and DS:0104.)

```
>EU 256

3DA5:0100  130F.10
>
```

The example above replaces the integer at DS:256 (DS:0100 hexadecimal) with **10**.

Enter Words Command

Syntax

EW *<address>* [*<list>*]

The Enter Words command enters one or more word values into memory at *address*.

The optional *list* can be entered as a list of expressions separated by spaces. The expressions are entered and evaluated in the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in hexadecimal.

Examples

```
>EW 256 10 20 30
>
```

If the current radix is 10, the example above replaces the three words at DS:256, DS:258, and DS:260 with the decimal values `10`, `20`, and `30`. (These addresses correspond to the hexadecimal addresses DS:0100, DS:0102, and DS:0104.)

```
>EW 256

3DA5:0100   130F.A
>
```

The example above replaces the integer at DS:256 (DS:0100 hexadecimal) with 10 (0A hexadecimal).

Enter Double Words Command

Syntax

ED *<address>* [*<list>*]

The Enter Double Words command enters one or more double-word values into memory at *address*. Double words are displayed and entered in the *segment:offset* address format; that is, two words separated by a colon (:). If the colon is omitted and only one word entered, only the offset portion of the address is changed.

The optional *list* can be entered as a list of expressions separated by spaces. The expressions are entered and evaluated in the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in hexadecimal.

Examples

```
>ED 256 8700:12008
>
```

If the current radix is 10, the example above replaces the double words at DS:256 (DS:0100 hexadecimal) with the decimal address `8700:12008` (hexadecimal address 21FC:2EE8).

```
>ED 256

3DA5:0100   21FC:2EE8.2EE9
>
```

The example above replaces the offset portion of the double word at DS:256 (DS:0100 hexadecimal) with **2EE9** hexadecimal. Since the segment portion of the address is not provided, the existing segment (**21FC** hexadecimal) is unchanged.

Enter Short Reals Command

Syntax

ES <*address*> [<*list*>]

The Enter Short Reals command enters one or more short-real values into memory at *address*.

The optional *list* can be entered as a list of real numbers separated by spaces. The numbers must be entered in decimal, regardless of the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in decimal. Short-real numbers can be entered either in floating-point format or in scientific-notation format.

Examples

```
>ES 256 23.479 1/4 -1.65E+4 235
>
```

The example above replaces the four numbers at DS:256, DS:260, DS:264, and DS:268 with the real numbers **23.479**, **0.25**, **-1650.0**, and **235.0**. (These addresses correspond to the hexadecimal addresses DS:0100, DS:0104, DS:0108, and DS:0112.)

```
>ES PI
3DA5:0064  42 79 74 65   7.215589E+022  3.141593
>
```

The example above replaces the number at the symbolic address **PI** with **3.141593**.

11

Enter Long Reals Command

Syntax

EL *<address>* [*<list>*]

The Enter Long Reals command enters one or more long-real values into memory at *address*.

The optional *list* can be entered as a list of real numbers separated by spaces. The numbers must be entered in decimal, regardless of the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in decimal. Long-real numbers can be entered either in floating-point format or in scientific-notation format.

Examples

```
>EL 256 23.479 1/4 -1.65E+4 235
>
```

The example above replaces the four numbers at DS:256, DS:264, DS:272, and DS:280 with the real numbers **23.479**, **0.25**, **-1650.0**, and **235.0** (These addresses correspond to the hexadecimal addresses DS:0100, DS:0108, DS:0110, and DS:0118.)

```
>EL PI
3DA5:0064  42 79 74 65 DC OF 49 40   5.012391E+001  3.141593
>
```

The example above replaces the number at the symbolic address **PI** with **3.141593**.

Enter 10-Byte Reals Command

Syntax

ET *<address* [*list*]>

The Enter 10-Byte Reals command enters one or more 10-byte-real values into memory at *address*.

The optional *list* can be entered as a list of real numbers separated by spaces. The numbers must be entered in decimal, regardless of the current radix. If *list* is not given, the CodeView debugger prompts for new values, which must be entered in decimal. The numbers can be entered either in floating-point format or in scientific-notation format.

Examples

```
>ET 256 23.479 1/4 -1.65E+4 235
>
```

The example above replaces the four numbers at DS:256, DS:266, DS:276, and DS:286 with the real numbers **23.479**, **0.25**, **-1650.0**, and **235.0**. (These addresses correspond to the hexadecimal addresses DS:0100, DS:010A, DS:0114, and DS:011E.)

```
>ET PI
3DA5:0064  42 79 74 65 DC 0F 49 40 7F BD  -3.292601E-193 3.141593
>
```

The example above replaces the number at the symbolic address **PI** with **3.141593**.

Fill Memory Command

The Fill Memory command provides an efficient way of filling up a large or small block of memory, with any values you specify. It is primarily of interest to assembly programmers because the command enters values directly into memory. However, you may find it useful for initializing large data areas such as an array or structure.

You can enter arguments to the Fill Memory command using any radix.

Keyboard

The Fill Memory command cannot be executed with a keyboard command.

Dialog

To fill an area of memory with values you specify, enter the Fill Memory command as follow:

F *<range> <list>*

The Fill Memory command fills the addresses in the specified *range* with the byte values specified in *list*. The values in the list are repeated until the whole range is filled. (Thus, if you specify only one value, the entire range is filled with that same value.) If the *list* has more values than the number of bytes in the *range*, then the command ignores any extra values.

Examples

```
>F 100 L 100 0   ;* hexadecimal radix assumed
>
```

The first example fills 255 (100 hexadecimal) bytes of memory starting at DS:0100 with the value 0. This command might possibly be used to reinitialize the program's data without having to restart the program.

```
>F table L 64 42 79 74 ;* hexadecimal radix assumed
>
```

The second example fills the 100 (64 hexadecimal) bytes starting at **table** with the following hexadecimal byte values: 42, 79, 74. These three values are repeated until all 100 bytes are filled.

1

Move Memory Command

The Move Memory command enables you to copy all the values in one block of memory directly to another block of memory of the same size. This command is of most interest to assembly programmers, but can be used by anyone who wants to do large data transfers efficiently. For example, you can use this command to copy all the values in one array to the elements of another.

Keyboard

The Move Memory command cannot be executed with a keyboard command.

Dialog

To copy the values in one block of memory to another, enter the Move Memory command with the following syntax:

M *<range>* *<address>*

The values in the block of memory specified by *range* are copied to a block of the same size beginning at *address*. All data in *range* are guaranteed to be copied completely over to the destination block, even if the two blocks overlap. However, if they do overlap, some of the original data in *range* is altered.

To prevent loss of data, the Move Memory command copies data starting at the source block's lowest address whenever the source is at a higher address than the destination. If the source is at a lower address, then the Move Memory command copies data beginning at the source block's highest address.

Example

```
>M arr1(1) L arsize arr2(1)   ;* FORTRAN example
>
```

In the example above, the block of memory beginning with the first element of **arr1**, and **arsize** bytes long, is copied directly to a block of the same size beginning at the address of the first element of **arr2**. In C, this command would be entered as **M arr1[0] L arsize arr2[0]**.

11

Register Command

The Register command has two functions: it displays the contents of the central processing unit registers, and it can also change the values of those registers. The modification features of the command are explained in this section. The display features of the Register command are explained in Section 6.7.

Keyboard

The registers cannot be changed with keyboard commands.

Dialog

To change the value of a register with a dialog command, enter a command line with the following syntax:

R [<*registername*>[[=]<*expression*>]]

To modify the value in a register, type the command letter **R** followed by *registername*. The CodeView debugger displays the current value of the register and prompts for a new value. Press the <RETURN> key if you only want to examine the value. If you want to change it, type an expression for the new value and press the <RETURN> key.

As an alternative, you can type both *registername* and *expression* in the same command. You can use the equal sign (=) between *registername* and *expression*, but a space has the same effect.

The register name can be any of the following names: **AX**, **BX**, **CX**, **DX**, **CS**, **DS**, **SS**, **ES**, **SP**, **BP**, **SI**, **DI**, **IP**, or **F** (for flags). If you have a 386-based machine, then the register name can be one of the 32-bit register names shown in Table 5.11.

To change a flag value, supply the register name **F** when you enter the Register command. The command displays the current value of each flag as a two-letter name.

At the end of the list of values, the command displays a dash (-). Enter new values after the dash for the flags you wish to change, then press the <RETURN> key. You can enter flag values in any order. Flags for which new values are not entered remain unchanged. If you do not want to change any flags, simply press the <RETURN> key.

If you enter an illegal flag name, an error message is displayed. The flags preceding the error are changed; flags at and following the error are not changed.

The flag values are shown in Table 11.1.

Table 11.1

Flag-Value Mnemonics

Flag Name	Set	Clear
Overflow	OV	NV
Direction	DN	UP
Interrupt	EI	DI
Sign	NG	PL
Zero	ZR	NZ
Auxiliary carry	AC	NA
Parity	PE	PO
Carry	CY	NC

Examples

```
>R IP 256
>
```

The example above changes the **IP** register to the value **256** (0100 hexadecimal).

```
>R AX
AX OEOO
:_
```

The example above displays the current value of the **AX** register and prompts for a new value (the underscore represents the CodeView cursor). You can now type any 16-bit value after the colon.

```
>R AX
AX 0E00
:256
>_
```

The example above changes the value of **AX** to 256 (in the current radix).

```
>R F UP EI PL
```

The example above shows the command-line method of changing flag values.

```
>R F
NV(OV) UP(DN) EI(DI) PL(NG) NZ(ZR) AC(NA) PE(PO) NC(CY) -OV DI ZR
>R F
OV(NV) UP(DN) DI(EI) PL(NG) ZR(NZ) AC(NA) PE(PO) NC(CY) -
>
```

With the prompting method of changing flag values (shown above), the first mnemonic for each flag is the current value, and the second mnemonic (in parentheses) is the alternate value. You can enter one or more mnemonics at the dash prompt. In the example, the command is given a second time to show the results of the first command.

Chapter 12

Using CodeView System-Control Commands

Introduction

This chapter discusses commands that control the operation of the Code-View debugger. The commands in this category are listed below:

Command	Action
Help (**H**)	Displays help
Quit (**Q**)	Returns to System V
Radix (**N**)	Changes radix
Redraw (**@**)	Redraws screen
Screen Exchange (****)	Switches to output screen
Search (**/**)	Searches for regular expression
Shell Escape (**!**)	Starts new shell
Tab Set (**#**)	Sets tab size
Option (**O**)	Views or sets CodeView options
Redirection and related commands	Control redirection of CodeView output or input

The system-control commands are discussed in the following sections.

Help Command

The CodeView debugger has two help systems: a complete on-line-help system available only in window mode, and a syntax summary available with sequential mode.

Keyboard

If you are in window mode, press the **F1** key to enter the complete on-line-help system. If you are in sequential mode, a syntax-summary screen appears when you press **F1** .

Dialog

If you are in window mode, you can view the complete on-line-help system with the following command:

H

If you are in sequential mode, this command displays a screen containing all CodeView dialog commands with the syntax for each. This screen is the only help available in sequential mode.

Quit Command

The Quit command terminates the CodeView debugger and returns control to the operating system.

Keyboard

To quit the CodeView debugger with a keyboard command, press <ALT>**f** to open the File menu, and then press **X** to select Exit. The CodeView screen is replaced by the operating system screen, with the cursor at the operating system prompt.

Dialog

To quit the CodeView debugger with a dialog command, enter a command line with the following syntax:

Q

When the command is entered, the CodeView screen is replaced by the standard screen, with the cursor at the shell prompt.

Radix Command

The Radix command changes the current radix for entering arguments and displaying the value of expressions. The default radix when you start the CodeView debugger is 10 (decimal). Radixes 8 (octal) and 16 (hexadecimal) can also be set. Binary and other radixes are not allowed.

The following seven conditions are exceptions; they are not affected by the Radix command:

1. The radix for entering a new radix is always decimal.

2. Format specifiers given with the Display Expression command or any of the Watch Statement commands override the current radix.

3. Addresses output by the Assemble, Dump, Enter, Examine Symbol, and Unassemble commands are always shown in hexadecimal.

4. In assembly mode, all values are shown in hexadecimal.

5. The display radix for Dump, Watch Memory, and Tracepoint Memory commands is always hexadecimal if the size is bytes, words, or double words, and always decimal if the size is integers, unsigned integers, short reals, long reals, or 10-byte reals.

6. The input radix for the Enter commands with the prompting method is always hexadecimal if the size is bytes, words, or double words, and always decimal if the size is integers, unsigned integers, short reals, long reals, or 10-byte reals. The current radix is used for all values given as part of a list, except real numbers, which must be entered in decimal.

7. The register display is always in hexadecimal.

Keyboard

You cannot change the input radix with a keyboard command.

Dialog

To change the input radix with a dialog command, enter a command line with the following syntax:

N[<*radixnumber*>]

The *radixnumber* can be 8 (octal), 10 (decimal), or 16 (hexadecimal). The default radix when you start the CodeView debugger is 10 (decimal), unless your main program is written with the Macro Assembler, in which case the default radix is 16 (hexadecimal). If you give the Radix command with no argument, the debugger displays the current radix.

Examples

```
>N10
>N
10
>? prime
107
>

>N8  ;
>? prime
0153
>

>N16 ;
>? prime
0x006b
>
```

The example aboves show how 107 decimal, stored in the variable *prime*, would be displayed with different radixes.

```
>N8
>? 34,i
28
>N10
>? 28,i
28
>N16
>? 1C,i
28
>
```

In the example above, the same number is entered in different radixes, but the **i** format specifier is used to display the result as a decimal integer in all three cases. See Chapter 7, ''Examining Data and Expressions,'' for more information on format specifiers.

Redraw Command

The Redraw command can be used only in window mode; it redraws the CodeView screen. This command is seldom necessary, but you might need it if the output of the program being debugged disturbs the CodeView display temporarily.

Keyboard

You cannot redraw the screen using a keyboard command.

Dialog

To redraw the screen with a dialog command, enter a command line with the following syntax:

@

12

Screen Exchange Command

The Screen Exchange command allows you to switch temporarily from the debugging screen to the output screen.

The CodeView debugger uses either screen flipping or screen swapping to store the output and debugging screens. See Chapter 2, "Getting Started," for an explanation of flipping and swapping.

Keyboard

To execute the Screen Exchange command with a keyboard command, press the **F4** key. Press any key when you are ready to return to the debugging screen.

Dialog

To execute the Screen Exchange command from the dialog window, enter a command line with the following syntax:

```
\
```

The output screen appears. Press any key when you are ready to return to the debugging screen.

Search Command

The Search command allows you to search for a regular expression in a source file. The expression being sought is specified either in a dialog box or as an argument to a dialog command. Once you have found an expression, you can also search for the next or previous occurrence of the expression.

Regular expressions are patterns of characters that may match one or many different strings. The use of patterns to match more than one string is similar to the shell method of using wild-card characters in file names.

You can use the Search command without understanding regular expressions. Since text strings are the simplest form of regular expressions, you can simply enter a string of characters as the expression you want to find. For example, you could enter **COUNT** if you wanted to search for the word ''COUNT'' in the source file.

The following characters have special meanings in regular expressions: backslash (\), asterisk (*), left bracket ([), period (.), dollar sign ($), and caret (^). To find strings containing these characters, you must precede the characters with a backslash; this cancels their special meanings.

For example, you would use * to find *x*y*. The periods in the relational operators must also be preceded by a backslash.

The Case Sense selection from the Options menu has no effect on searches for regular expressions.

Note

When you search for the next occurrence of a regular expression, the CodeView debugger searches to the end of the file, and then wraps around and begins again at the start of the file. This can have unexpected results if the expression occurs only once. When you give the command repeatedly, nothing seems to happen. Actually, the debugger is repeatedly wrapping around and finding the same expression each time.

Keyboard

To find a regular expression with a keyboard command, press <ALT>s to open the Search menu, and then press **F** to select Find. A dialog box appears, asking for the regular expression to be found. Type the expression and press the <RETURN> key. The CodeView debugger starts searching at the current cursor position and puts the cursor at the next line containing the regular expression. An error message appears if the expression is not found. If you are in assembly mode, the debugger automatically switches to source mode when the expression is found.

After you have found a regular expression, you can search for the next or previous occurrence of the expression. Press <ALT>s to open the Search menu and then press **N** to select Next or **P** to select Previous. The cursor moves to the next or previous match of the expression.

You can also search the executable code for a label (such as a routine name or an assembly-language label). Press <ALT>s to open the Search menu and then press **L** to select Label. A dialog box appears, asking for the label to be found. Type the label name and press the <RETURN> key. The cursor moves to the line containing the label. This selection differs from other search selections because it searches executable code rather than source code. The CodeView debugger switches to assembly mode, if necessary, to display a label in a library routine or assembly-language module.

Dialog

To find a regular expression using a dialog command, enter a command line with the following syntax:

/[<*regularexpression*>]

If *regularexpression* is given, the CodeView debugger searches the source file for the first line containing the expression. If no argument is given, the debugger searches for the next occurrence of the last regular expression specified.

In window mode, the CodeView debugger starts searching at the current cursor position and puts the cursor at the next line containing the regular expression. In sequential mode, the debugger starts searching at the last source line displayed. It displays the source line in which the expression is found. An error message appears if the expression is not found. If you are in assembly mode, the CodeView debugger automatically switches to source mode when the expression is found.

You cannot search for a label with the dialog version of the Search command, but you can use the View command with the label as an argument for the same effect.

Shell Escape Command

The Shell Escape command allows you to exit from the CodeView debugger to a command shell. You can execute system commands or programs from within the debugger, or you can exit from the debugger to the system while retaining your current debugging context.

Keyboard

To open a shell with a keyboard command, press <ALT>**f** to open the File menu, and then press **D** to select Shell. When you are ready to return to the debugging session, type the command **exit**. The debugging screen appears with the same status it had when you left it.

Dialog

To open a shell using a dialog command, enter a command line with the following syntax:

> ![<*command*>]

If you want to exit to the system and execute several programs or commands, enter the command with no arguments. The standard screen appears. You can run programs or shell commands. When you are ready to return to the debugger, type the command **exit**. The debugging screen appears with the same status it had when you left it.

If you want to execute a program or shell command from within the CodeView debugger, enter the Shell Escape command (**!**) followed by the name of the command or program you want to execute. The output screen appears, and the debugger executes the command or program. When the output from the command or program is finished, the message `Press any key to continue...` appears at the bottom of the screen. Press a key to make the debugging screen reappear with the same status it had when you left it.

Examples

```
>!
```

In the above example, the CodeView debugger saves the current debugging context. The standard screen appears, and you can enter any number of commands. To return to the debugger, enter **exit**.

```
>!ls /tmp
```

In the example above, the command **ls** is executed with the argument */tmp*. The directory listing is followed by a prompt telling you to press any key to return to the CodeView debugging screen.

12

Tab Set Command

The Tab Set command sets the width in spaces that the CodeView debugger fills for each tab character. The default tab is eight spaces. You might want to set a smaller tab size if your source code has so many levels of indentation that source lines extend beyond the edge of the screen. This command has no effect if your source code was written with an editor that indents with spaces rather than with tab characters.

Keyboard

You cannot set the tab size by using a keyboard command.

Dialog

To set the tab size with a dialog command, enter a command line with the following syntax:

#<number>

The *number* is the new number of characters for each tab character. In window mode, the screen is redrawn with the new tab width when you enter the command. In sequential mode, any output of source lines reflect the new tab size.

Example

```
>.
32:                 IF (X(I)) .LE. X(J)) GOTO 301
>#4
>.
32:         IF (X(I)) .LE. X(J)) GOTO 301
>
```

In the example above, the Source Line command (.) is used to show the source line with the default tab width of eight spaces. Next the Tab Set command is used to set the tab width to four spaces. The Source Line command then shows the same line.

Option Command

The Option command allows you to view the state of options in the Option menu (Save Output, Bytes Coded, and Case Sense), and to turn any of the these options on or off.

For each different kind of source module that you debug, there is a different set of default settings. However, the use of the Option command overrides any of these settings.

Keyboard

To view the state of the Options menu with a keyboard command, press <ALT>**o** to open the Options menu. Each option is then displayed. Those options that are turned on have a double arrow immediately to the left. Options that are turned off have no double arrow.

To change one of the Option settings, press the letter key corresponding to the option's mnemonic. This reverses the state of the option. (An option that was on is turned off and vice versa.) You can also reverse an option by moving the highlight down with the arrow key, and then pressing <RETURN> .

Dialog

To view or change options with a dialog command, enter a command line with the following syntax:

O[<*option*> [+ | -]]

In the above display, *option* is one of the following characters: **F**, **B**, **C**, or **3**. If used, there must be no spaces between the character and the **O**. These characters correspond to options as shown below:

Option Command

Command	Correspondence
OF	Save Output option
OB	Bytes-Coded option
OC	Case-Sense option
O	All options

The **O** form of the command (all options) takes no arguments. It simply displays the state of all four options. The other forms of the command (**OF**, **OB**, and **OC**) can be used either with no arguments (in which case they simply display the state of the option) or they can take the argument + or –.

The + argument turns the option on. The – argument turns the option off.

Examples

```
>O
Save Output on
Bytes Coded on
Case Sense off
>OF
Save Output on
>OF-
Save Output off
```

In the example above, the **O** and **OF** commands are used simply to view the current state of an option. The **OF-** command modifies an option and then reports the results of the modification.

The dialog version of the Option command is particularly useful for redirected CodeView commands (which cannot access menus) and for making CodeView startup with certain options. For example, the following shell-level command line brings up CodeView with the Bytes Coded off:

```
CV /c"OB-" test
```

This command line could be put into a shell script for convenient execution.

Redirection Commands

The CodeView debugger provides several options for redirecting commands from or to devices or files. Furthermore, the debugger provides several other commands, which are relevant only when used with redirected files. The redirection commands and related commands are discussed in the following sections.

Keyboard

None of the redirection or related commands can be executed with keyboard commands.

Dialog

The redirection commands are entered with dialog commands, as shown in the following sections.

Redirecting CodeView Input

Syntax

< *devicename*

The Redirected Input command causes the CodeView debugger to read all subsequent command input from a device, such as another terminal or a file.

Examples

```
></dev/tty1a
```

The example above redirects commands from the device (probably a remote terminal) designated as */dev/tty1a* to the CodeView terminal.

```
><infile.txt
```

The example above redirects command input from file *infile.txt* to the CodeView debugger. You might use this command to prepare a CodeView session for someone else to run. You create a text file containing a series of CodeView commands separated by carriage-return-line-feed combinations or semicolons. When you redirect the file, the debugger executes the commands to the end of the file. One way to create such a file is to redirect commands from the CodeView debugger to a file (see the section ''Redirecting CodeView Input and Output'') and then edit the file to eliminate the output and add comments.

Redirecting CodeView Output

Syntax

[T]>[>] *<devicename>*

The Redirected Output command causes the CodeView debugger to write all subsequent command output to a device, such as another terminal, a printer, or a file. The term ''output'' includes not only the output from commands, but the command characters that are echoed as you type them.

The optional **T** indicates that the output should be echoed to the CodeView screen. Normally, you want to use the **T** if you are redirecting output to a file, so that you can see what you are typing. However, if you are redirecting output to another terminal, you may not want to see the output on the CodeView terminal.

The second greater-than symbol (optional) appends the output to an existing file. If you redirect output to an existing file without this symbol, the existing file is replaced.

Examples

```
>>/dev/tty1a
```

In the example above, output is redirected to the device designated as */dev/tty1a* (probably a remote terminal). You might want to enter this command, for example, when you are debugging a graphics program and want CodeView commands to be displayed on a remote terminal while the program display appears on the originating terminal.


```
>T>outfile.txt
.
.
.
>>/dev/tty
.
.
.
```

In the example above, output is redirected to the file *outfile.txt*. You might want to enter this command in order to keep a permanent record of a CodeView session. Note that the optional **T** is used so that the session is echoed to the CodeView screen as well as to the file. After redirecting some commands to a file, output is returned to the console (terminal) with the command *>/dev/tty*.

```
>T>>outfile.txt
```

If, later in the session, you want to redirect more commands to the same file, use the double greater-than symbol, as in the example above, to append the output to the existing file.

Redirecting CodeView Input and Output

Syntax

= *<devicename>*

Redirection Commands

The Redirected Input and Output command causes the CodeView debugger to write all subsequent command output to a device and simultaneously to receive input from the same device. This command is practical only if the device is a remote terminal.

Redirecting input and output works best if you start in sequential mode (using the **-T** option). The CodeView debugger's window interface has little purpose in this situation, since the remote terminal can act only as a sequential (nonwindow) device.

12

Example

```
>=/dev/tty1a
```

In the example above, output and input are redirected to the device designated as */dev/tty1a*. This command would be useful if you wanted to enter debugging commands and see the debugger output on a remote terminal, while entering program commands and viewing program output on the terminal where the debugger is running.

Commands Used with Redirection

The following commands are intended for use when redirecting commands to or from a file. Although they are always available, these commands have little practical use during a normal debugging session.

Command	**Action**
Comment (*)	Displays comment
Delay (:)	Delays execution of commands from a redirected file
Pause (")	Interrupts execution of commands from a redirected file until a key is pressed.

Comment Command

Syntax

**<comment>*

The Comment command is an asterisk (*) followed by text. The CodeView debugger echoes the text of the comment to the screen (or other output device). This command is useful in combination with the redirection commands when saving a commented session, or when writing a commented session that is redirected to the debugger.

Examples

```
>T>output.txt
>* Dump first 20 bytes of screen buffer
>D #B800:0 L 20
B800:0000 54 17 6F 17 20 17 72 17 65 17 74 17 75 17 72 17 T.o. .r.e.t.u.r.
B800:0010 6E 17 20 17                                     n. .
>
```

In the example above, the user is sending a copy of a CodeView session to file *output.txt*. Comments are added to explain the purpose of the command. The text file contains commands, comments, and command output.

```
* Dump first 20 bytes of screen buffer
D #B800:0 L 20
.
.
.
< /dev/tty
```

The example above illustrates another way to use the Comment command. You can put comments into a text file of commands that are executed automatically when you redirect the file into the CodeView debugger. In this example, an editing program was used to create the text file called *input.txt*.

Redirection Commands

```
><input.txt
>* Dump first 20 bytes of screen buffer
>D #B800:0 L 20
B800:0000 54 17 6F 17 20 17 72 17 65 17 74 17 75 17 72 17 T.o. .r.e.t.u.r.
B800:0010 6E 17 20 17                                     n. .
.
.
.
></dev/tty
>
```

When you read the file into the debugger by using the Redirected Input command, you see the comment, the command, and then the output from the command, as in the example above.

Delay Command

Syntax

:

The Delay command interrupts execution of commands from a redirected file and waits about half a second before continuing. You can put multiple Delay commands on a single line to increase the length of the delay. The delay is the same length, regardless of the processing speed of the computer.

Example

```
: ;* That was a short delay...
::::: ;* That was a longer delay...
```

In the example above from a text file that might be redirected into the CodeView debugger, the Delay command is used to slow execution of the redirected file.

Pause Command

Syntax

```
"
```

The Pause command interrupts execution of commands from a redirected file and waits for the user to press a key. Execution of the redirected commands begins as soon as a key is pressed.

Example

```
* Press any key to continue
"
```

In the example above from a text file that might be redirected into the CodeView debugger, a Comment command is used to prompt the user to press a key. The Pause command is then used to halt execution until the user responds.

```
>* Press any key to continue
>"
```

The example above shows the output when the text is redirected into the debugger. The next CodeView prompt does not appear until the user presses a key.

Index

Special Characters

A

B

C

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

SCO UNIX® System V/386

Development System

Macro Assembler's User's Guide

The Santa Cruz Operation, Inc.

SCO Document Number: 6-26-89-6.0/3.2.0

Contents

16 Controlling Program Flow

17 Processing Strings

18 Calculating with a Math Coprocessor

19 Controlling the Processor

A New Features

Introduction

Introduction

Welcome to the Microsoft® Macro Assembler (**masm**). As a part of the Software Development System, **masm** lets you create assembly-language programs and modules.

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

The assembler produces *relocatable object modules* using the Intel Object Module Format (OMF) from assembly-language source files. These object modules can be linked using **ld**, the UNIX link editor, to create executable programs. You can invoke **ld** either directly from the command line, or from high-level-language compilers, such as **cc**, the C compiler. Object modules created with **masm** are compatible with many high-level-language object modules, including those created with the Microsoft BASIC, C, FORTRAN, and Pascal compilers.

To make program development easier, **masm** offers the following standard features:

- It has a full set of macro directives.
- It allows conditional assembly of portions of a source file.
- It supports a wide range of operators for creating complex assembly-time expressions.
- It carries out strict syntax checking of all instruction statements, including strong typing for memory operands.

New Features

Version 5.0 of the assembler has the following major new features:

- All instructions and addressing modes of the 80386 processor and 80387 coprocessor are now supported.
- New segment directives allow simplified segment definitions. These optional directives implement the segment conventions used in Microsoft high-level languages.
- Error messages have been clarified and enhanced.

- The default format for initializing real-number variables has been changed from Microsoft Binary to the more common IEEE (Institute of Electrical and Electronic Engineers, Inc.) format.

Note

In addition to these new features, there are many minor enhancements. If you are updating from a previous version of the Macro Assembler, you should start by reading Appendix A, ''New Features.'' This appendix summarizes new features added for Version 5.0 and discusses compatibility issues.

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 discuss systems programming or advanced techniques. For information of this kind, books on 8086-family assembly-language programming should be consulted.

This manual is divided into three major parts.

- Part 1 is called Using Assembler Programs, and it comprises Chapters 1 and 2.

- Chapters 3-11 make up Part 2, Using Directives.

- The third part, called Using Instructions, comprises Chapters 12-19.

- There are five appendixes at the end of this guide. Appendix A, ''New Features,'' is followed by three appendixes intended to be used as a reference source for assembly-language development. They are Appendix B, ''Instruction Summary,'' Appendix C, ''Directive Summary,'' and Appendix D, ''Segment Names for High-Level Languages.'' The last appendix in this guide is Appendix E, ''Error Messages and Exit Codes.''

Important programming topics and their references include:

- Overview of the program-development process

  Chapter 1, Getting Started, describes the program-development process and gives brief examples of each step.

- Using the assembler

  Part 1, Using Assembler Programs, describes the command lines, options, and output of **masm**.

- Handling errors

  Error messages are described in Appendix E, Error Messages and Exit Codes.

- Overview of the format for assembly-language source code

  Chapter 1, Getting Started, shows examples of assembly-language source files, and Chapter 3, Writing Source Code (in Part 2), discusses basic concepts in a reference format.

- Programming in the assembly language recognized by **masm**

  Part 2 of this document, Using Directives, 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.

- Overview of the architecture of 8086-family processors

  Chapter 12, Understanding 8086-Family Processors (in Part 3), discusses segments, memory use, registers, and other basic features of 8086-family processors.

- Using the instruction sets for the 80x86 microprocessors

  Part 3 of this document, Using Instructions, 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

  Appendix B, ''Instruction Summary,'' provides a complete list of instruction mnemonics arranged in alphabetical order.

- Using the instruction sets of the 80x87 math coprocessors

  Chapter 18, Calculating with a Math Coprocessor, lists the coprocessor instructions and describes how to use them.

- Writing assembly-language routines for high-level languages

  Appendix D, ''Segment Names for High-Level Languages,'' describes the naming conventions used to form assembly-language source files that are compatible with object modules produced by recent Microsoft language compilers.

- Hardware features of your computer

  For some assembly-language tasks, you may need to know about the hardware features of the computers that run your programs. Consult the technical manuals for your computer or one of the many books that describe hardware features.

Notational Conventions

This manual uses the following notational conventions:

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 the screen or by a printer. For example, the following command line is printed in this special typeface: `cc -Foout.o -DTRUE=1 file.c` When discussing this command line in text, items appearing on the command line, such as *out.o*, also appear in the special typeface.
Language elements	Bold type indicates elements of the C language that must appear in source programs as shown. Text that is normally shown in bold type includes operators,

keywords, library functions, commands, options, and preprocessor directives. Examples are shown below:

+=	**#if defined( )**	**int**
if	**-Fa**	**fopen**
main	**sizeof**	

ENVIRONMENT, VARIABLES, and **MACROS**

Bold capital letters are used for environment variables, symbolic constants, and macros.

placeholders

Words in italics are placeholders that you must supply in command-line and option specifications and in the text for types of information. Consider the following option:

-H *number*

Note that *number* is italicized to indicate that it represents a general form for the **-H** option. In an actual command, you would supply a particular number for the placeholder *number*.

Occasionally, italics are also used to emphasize particular words in the text.

Missing code

Vertical ellipses are used in program examples to indicate that a portion of the program is omitted. For instance, in the following excerpt, the ellipses between the statements indicate that intervening program lines occur but are not shown:

```
count = 0;
.
.
.
*pc++;
```

[*optional items*]

Brackets enclose optional fields in command-line and option specifications. Consider the following option specification:

-D*identifier*[=[*string*]]

The placeholder *identifier* indicates that you must supply an identifier when you use the **-D** option. The outer brackets indicate that you are not required to supply an equal sign (=) and a string following the identifier. The inner brackets indicate that you are not required to enter a string following the equal sign, but if you do supply a string, you must also supply the equal sign.

Single brackets are used in C-language array declarations and subscript expressions. For instance, *a[10]* is an example of brackets in a C subscript expression.

Repeating elements...

Horizontal ellipses are used in syntax examples to indicate that more items having the same form may be entered. For example, in the Bourne shell, several paths can be specified in the **PATH** command, as shown in the following syntax:

PATH[=]*path*[;*path*]...

{*choice1*|*choice2*}

Braces and a vertical bar 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 unless all of the items are also enclosed in double square brackets.

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

-W {**0** | **1** | **2** | **3**}

	You can use **-W1**, **-W2**, or **-W3** to display different levels of warning messages or **-W0** to suppress all warning messages.
‘‘Defined terms’’	Quotation marks set off terms defined in the text. For example, the term ‘‘far’’ appears in quotation marks the first time it is defined. Some C constructs require quotation marks. Quotation marks required by the language have the form " " rather than ‘‘ ’’. For example, a C string used in an example would be shown in the following form: `"abc"`
KEY+KEY	Small capital letters are used for the names of keys and key sequences, such as ENTER and CTRL+C. Key sequences to be pressed simultaneously are indicated by the key names in small caps separated by a plus sign (CTRL+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*

This syntax shows that you must first type the program name, **masm**. You can then enter any number of *options*, or none at all. You must enter a *sourcefile*.

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

```
masm test.s
masm -Zi test.s
masm -v -a -p test.s
masm -vap test.s
```

Part 1

Using Assembler Programs

Part 1 of the Programmer's Guide (comprising Chapters 1 and 2) summarizes the process of creating programs from assembly-language source files.

Chapter 1 describes how to set up an efficient system for producing programs. It also provides examples of simple assembly-language source files and a brief summary of each of the utility programs used in program development.

Chapter 2 describes the assembler program, **masm**, in detail.

Chapter 1

Getting Started

Introduction

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

1

System Considerations

Before you start developing assembly-language programs, you need to verify that:

- The current operating system is UNIX System V, release 3.2 or later.

  If the current operating system is not UNIX System V, determine the operating-system version and use the corresponding **masm** manuals.

- The **masm** executable file is located in the **/usr/bin** directory.

  If the **masm** executable file is not located in the **/usr/bin** directory, ask your system administrator for its location.

- You know how to use the 8086, 80286, and 80386 instruction sets.

  To create assembly-language programs, you need to know how to use the 8086, 80286, and 80386 instruction sets. The directives, operands, operators, and expressions of **masm** are explained in this manual.

- Your text editor creates ASCII (American Standard Code for Information Interchange) text files.

  To assemble assembly-language programs, the source file must be in ASCII format. If your text editor does not produce ASCII files, switch to an editor that produces ASCII files.

The Program-Development Cycle

The program-development cycle for assembly language is illustrated in Figure 1-1.

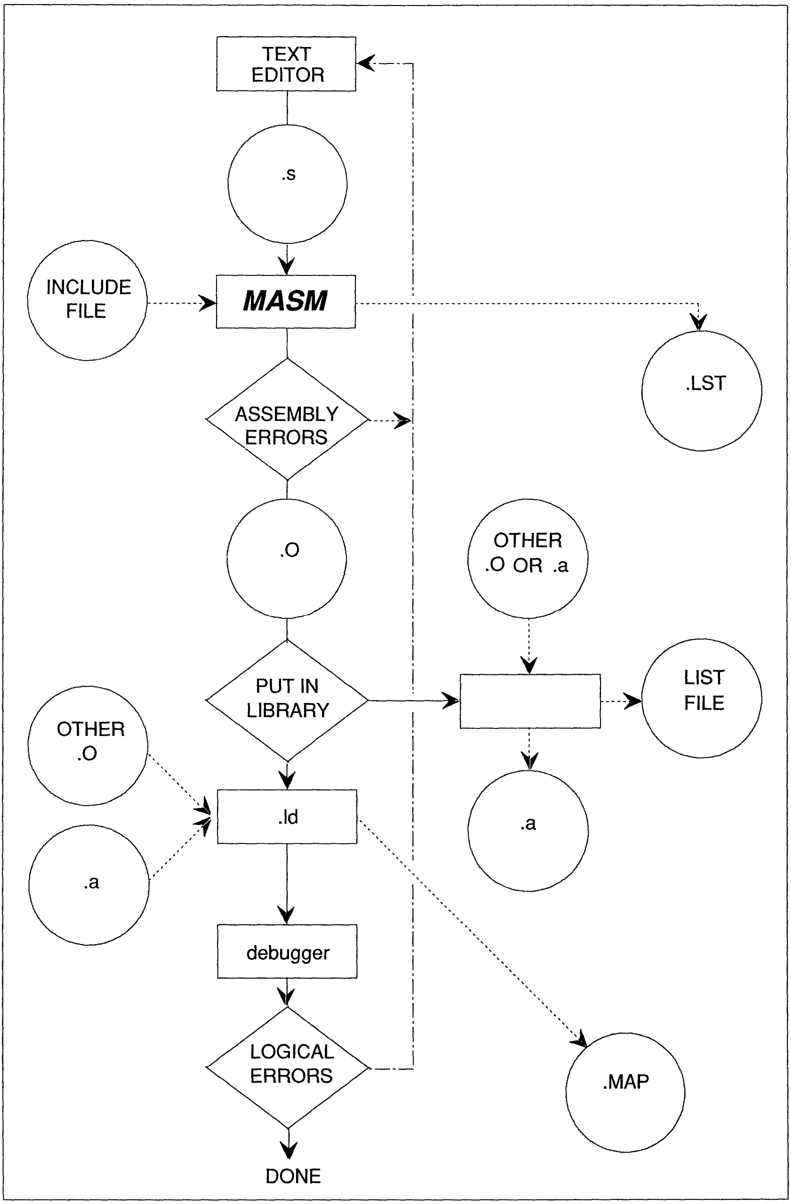


Figure 1-1 The Program Development Cycle

The specific steps for developing a stand-alone assembler program are as follows:

1. Use a text editor to create or modify assembly-language source modules. It is a convention, but not a requirement, to give source modules the **.s** extension. Source modules can be organized in a variety of ways. For instance, you can put all the procedures for a program into one large module, or you can split the procedures between 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. During assembly, **masm** may optionally read in code from include files. If assembly errors are encountered in a module, you must go back to Step 1 and correct the errors before continuing. For each source (**.s**) file, **masm** creates an object file with the default extension **.o**. Optional listing (**.lst**) 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. Use **ld** to combine all the object files and library modules that make up a program into a single executable file. **xld** can be invoked directly from the command line or indirectly from a high-level-language compiler such as the Microsoft C Compiler, **cc**.

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

   - Running the program and studying its input and output
   - Studying source and listing files
   - Using a UNIX System V debugger, such as **adb**(CP)

   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**(CP) with make description files. **make** is most useful for developing complex programs involving numerous source modules.

1

Developing Programs

The following sections describe the steps involved in developing programs. Examples are shown for each step, and the chapters and manuals that describe each topic in detail are cross-referenced.

Writing and Editing Assembly-Language Programs

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

To create assembly-language source files, you need a text editor that is capable of producing ASCII files.

The following example illustrates source code that produces a stand-alone executable program.

Example

```
          .386
          title         hello
          .model        small
          .data
message   db            "Hello, world", 10, 0   ; message to be
                                                ;   written

lmessage  equu        $ - message             ; length of message

extrn     _exit:proc
extrn     _write:proc

          .code
public    _main
_main     proc
          push          ebp
          mov           ebp, esp                ; establish stack
                                                ;   frame

          push          lmessage                ; push length of
                                                ;   message onto the stack

          push          OFFSET message          ;push address of
                                                ;  message onto the stack

          push          1
          call          _write                  ; write(1,message,lmessage)
          add           esp, 6                  ; remove arguments to write()

          push          0
          call          _exit

          leave
_main     endp

          end
```

Note the following points about the source file:

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

2. The string variable *message* is displayed using the write() system call. File descriptor 1 is used to display to the screen.

3. To terminate the program, the exit() system call with an argument of 0 is used. This is the recommended method.

COFF and OMF

This version of UNIX System V can produce object and/or executable files that use either of two different binary file formats: COFF (Common Object File Format) and OMF (Intel Object Module Format). COFF is the most widely used binary file format. OMF files are produced using the **-xenix** option with **cc**(CP) or by using **masm** by itself, from the command line.

The operating system can execute either file format; it does so by reading the file header and acting accordingly. The COFF and OMF formats are described in greater detail in the C compiler documentation.

OMF files can be converted to COFF format by using the **cvtomf** utility. See the man page for more information.

Assembling Source Files Using masm

Source modules are assembled with **masm**. The **masm** command-line syntax is:

masm [*options*] *sourcefile*

Suppose you had an assembly source file called *hello.s*. For the fastest possible assembly, you could start **masm** with the following command line:

```
masm hello.s
```

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

```
masm -v  -Zi hello.s
```

or

```
masm -vZi hello.s
```

The **-v** option instructs **masm** to send additional statistics and error information to the standard output during assembly. The **-Zi** option instructs **masm** to include symbolic and line-number information in the object file.

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

Assembling Source Files Using cc

You can also assemble your source files using the driver program for the C compiler, **cc**. If you follow the convention of using the *.s* extension for your masm source files, **cc** will detect this and automatically use **masm** to assemble your assembly-language source files. You can specify assembly-language source files along with your C-language source files on the same command line.

If you assemble your source files in this manner, the resulting object files use the COFF format used by the AT&T assembler, **as**(CP). If you want object files that use the OMF format used by the Microsoft language tools, use the **-xenix** option on the **cc** command-line. You must also use the **ld** linker options to generate OMF files, rather than the options to create COFF files, when linking the resulting object file. This is described in more detail in the *C User's Guide* manual and in the man page for **cc**(CP).

Chapter 2

Using masm

Introduction

This chapter tells you how to run the **masm** program. It also explains the options that control its behavior and describes the format of the assembly listings **masm** generates.

Running the Assembler

2

Once **masm** has been started from the command line, it attempts to process the source file that has been specified. If errors are encountered, they are output to standard error, and **masm** terminates. If no errors are encountered, **masm** creates an object file. It can also create a listing file if that option is specified.

Assembly Using the Command Line

You can assemble a program source file by entering the **masm** command name and the name of the file you wish to process. The command line has the following syntax:

masm [*options*] *sourcefile*

The *options* can be any combination of the assembler options described in the section, ''Using **masm** Options.'' The option letter or letters must be preceded by a dash (**-**).

The *sourcefile* must be the name of the source file to be assembled. Only one *sourcefile* is recognized on the command line; all other entries on the command line are ignored.

An object file is created to receive the relocatable object code. The object file is given the same name as the *sourcefile*, but the *sourcefile* extension (if any) is replaced with **.o**.

An optional listing file, which receives the assembly listing, is created if the **-l** option is given. The assembly listing shows the assembled code for each source statement and for the names and types of symbols defined in the program. The *sourcefile* extension (if any) is replaced with the extension **.lst**.

All files created during the assembly are written to the current directory.

Using masm Options

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

2

The following options are recognized:

Option	Action
-a	Writes segments in alphabetical order
-b*number*	Sets buffer size
-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	Makes names case sensitive
-Mu	Converts names to uppercase letters
-Mx	Makes public and external names case sensitive
-n	Suppresses tables in listing file
-p	Checks for impure code
-s	Writes segments in source-code order
-t	Suppresses messages for successful assembly
-v	Displays extra statistics to the standard output

-w{0 | 1 | 2}	Sets error-display level
-X	Includes false conditionals in listings
-z	Displays error lines to standard error (set by default)
-Zd	Puts line-number information in the object file
-Zi	Puts symbolic and line-number information in the object file

---

Note

Previous versions of the assembler provided a **-r** option to enable 8087 instructions and real numbers in the IEEE format. Since the current version of the assembler enables 8087 instructions and IEEE format by default, the **-r** option is no longer needed. In the current version, the **-r** option has no effect, but it is still recognized so old **make** files will work. The previous default format, Microsoft Binary, can be specified with the **.MSFLOAT** directive, as described in the section, ''Defining Default Assembly Behavior,'' in Chapter 3.

The following sections describe each of the **masm** options in more detail.

Specifying the Segment-Order Method

The following command-line options are used to control the order in which segments are written to the object file:

Syntax

-s Default
-a

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 segment order written to the object file. 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 the MS-DOS® operating system.

The order of object file segments is only one factor in determining the order in which they will appear in the executable file. The significance of segment order, and ways to control it, are discussed in the sections, ''Setting the Segment-Order Method'' and ''Defining Segment Combinations with Combine Type,'' in Chapter 4.

2

Example

```
masm -a file.s
```

This example creates an object file, *file.o*, 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.

Setting the File-Buffer Size

A buffer larger than your source file lets you do the entire assembly in memory, greatly increasing assembly speed.

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 (1-kilobyte) memory blocks allocated for the buffer. You can set the buffer to any size from 1Kbyte to 63Kbytes. The default size of the buffer is 32Kbytes.

You may not be able to use a large buffer if your computer does not have enough memory. If you receive an error message indicating insufficient memory, decrease the buffer size and try again.

Examples

```
masm -b16 file.s
```

This example decreases the buffer size to 16Kbytes.

```
masm -b63 file.s
```


This example increases the buffer size to 63Kbytes.

Creating a Pass 1 Listing

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.

Syntax

-d

The **-d** option directs **masm** to add a Pass 1 listing to the assembly-listing file, making the assembly listing show the results of both assembler passes.

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. For more information about Pass 1 listings, see the section, ''Reading a Pass 1 Listing.''

Example

```
masm -d file.s
```

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

Defining Assembler Symbols

Initial values of variables or information for conditional assembly can be passed from the **masm** command line with *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 be 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.s
```

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 columns (for more information about the **PAGE** directive, see the section, ''Controlling Page Format in Listings'', in Chapter 11).

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

```
            IF      mode LT 256     ; Use in expression
scrmode     DB      mode            ; Initialize byte variable
            ELSE
scrmode     DW      mode            ; Initialize word variable
            ENDIF
```

2

Creating Code for a Floating-Point Emulator

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 when assembling.

Syntax

-e

The **-e** option directs the assembler to generate data and code in the format expected by coprocessor emulator libraries. An emulator library uses 8088/8086 instructions to emulate the instructions of the 8087, 80287, or 80387 coprocessors. An emulator library can be used if you want your code to take advantage of a math coprocessor, or an emulator library can be used if the machine does not have a coprocessor.

Emulator libraries are only available with high-level-language compilers, including the Microsoft C, BASIC, FORTRAN, and Pascal compilers. The option cannot be used in stand-alone assembler programs unless you write your own emulator library. You cannot simply link with the emulator library from a high-level language, since these libraries require that the compiler start-up code be executed.

To the applications programmer, writing code for the emulator is like writing code for a coprocessor. The instruction sets are the same (except as noted in Chapter 18, ''Calculating with a Math Coprocessor''). However, at run time the coprocessor instructions are used only if there is a coprocessor available on the machine. If there is no coprocessor, the slower code from the emulator library is used instead.

Example

```
masm -e -Mx math.s
cc  calc.c math.o
```

In the first command line, the source file *math.s* is assembled with **masm** by using the **-e** option. Then the C compiler (**cc**) is used to compile the C source file *calc.c* and finally to link the resulting object file (*calc.o*) with *math.o*. The compiler generates emulator code for floating-point instructions. There are similar options for the FORTRAN, BASIC, and Pascal compilers.

Getting Command-Line Help

A quick reference for all the **masm** options is available from the command line.

Syntax

-h

The **-h** (help) option writes the command-line syntax and all the **masm** options to the standard output. You should not give any file names or other options with the **-h** option.

Example

```
masm -h
```

Setting a Search Path for Include Files

When the current source file being assembled uses the **INCLUDE** directive to incorporate other source files, the assembler finds these other files by looking along a *search path*. The **-I** option is used to set search paths for *include* files.

Syntax

-I*path*

You can set as many as 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 the section, ''Using Include Files,'' in Chapter 10.

Example

```
masm -I/usr/lib/io -Imacro file.s
```

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

```
INCLUDE asm.inc
```

In this case, **masm** would search for the file *asm.inc* first along the absolute path */usr/lib/io*, and then in the directory *macro* relative to the current directory. If the file was not in either of these directories, **masm** would then look in the current directory.

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

2

```
INCLUDE /u/me/macro/asm.inc
INCLUDE ../asm.inc
INCLUDE ./asm.inc
```

Specifying Listing Files

When instructed to, **masm** creates an additional file, called a *listing file*, that contains information about how your source code is assembled.

Syntax

-l

The **-l** option directs **masm** to create a listing file. Listing files always have the base name of the source file plus the extension **.lst**. A complete description of listing files is covered in the section, ''Reading Assembly Listings.''

Specifying Case Sensitivity

By default, **masm** is completely case sensitive. The **-Ml** and **-Mx** options are provided for compatibility with MS-DOS, which uses **-Mu** by default.

Syntax

-Ml Default
-Mx
-Mu

The **-Ml** option directs the assembler to make all names case sensitive. The **-Mx** option directs the assembler to make only the public and external names case sensitive. The **-Mu** option directs the assembler to convert all names into uppercase letters.

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 by using the **EXTRN**, **PUBLIC**, or **COMM** directives (see Chapter 7, ''Creating Programs from Multiple Modules'').

If you use the **-Zi** or **-Zd** option (see the section, ''Writing Symbolic Information to the Object File''), the **-Ml**, **-Mx**, and **-Mu** options affect the case of the symbolic data that will be available to a symbolic debugger.

Suppressing Tables in the Listing File

By default, **masm** includes tables of macros, structures, records, segments and groups, and symbols at the end of a listing file. This feature, however, can be turned off.

Syntax

-n

The **-n** option directs the assembler to omit all tables from the end of the listing file. The code portion of the listing file is not changed by the **-n** option.

Example

```
masm -n -l file.s
```

Checking for Impure Code

Code that moves data into memory with a **CS:** override is acceptable in real mode. However, such code may cause problems in protected mode.

Syntax

-p

The **-p** option directs **masm** to check for impure code in the 80286 or 80386 privileged mode. When the **-p** option is in effect, the assembler checks for these situations and generates an error if it encounters them.

Real and privileged modes are explained in Chapter 12, ''Understanding 8086-Family Processors.''

2

Example

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

The example shows a **CS:** override. If assembled with the **-p** option, an error is generated.

Controlling Display of Assembly Statistics

The amount of information **masm** sends to the standard output can be controlled from the command line.

Syntax

-v
-t

The **-v** (verbose) and **-t** (terse) options specify the level of information displayed to the standard output at the end of assembly.

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 standard output, while standard error remains unaffected. This option may be useful in script or make files if you do not want the output cluttered with unnecessary messages.

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

If errors are encountered during assembly, they will be displayed whether these options are given or not. Appendix E, ''Error Messages and Exit Codes,'' describes the messages **masm** displays after assembly.

Setting the Warning Level

During assembly, **masm** provides 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 of the problem and do not wish to take action to remedy it.

For more information on the specific structure and meaning of warning and error messages, see Appendix E of this document, entitled ''Error Messages and Exit Codes''.

Syntax

-w{**0** | **1** | **2**}

The **-w** option sets the assembler warning level. There are 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. A higher warning level adds to the number of warning messages you would have received at a lower warning level. Level 2 includes severe errors, serious warnings, and advisory warnings. If **masm** encounters severe errors during assembly, no object file is produced.

The advisory warnings that indicate potentially inefficient code are

Number	Message
104	`Operand size does not match word size`
105	`Address size does not match word size`
106	`Jump within short distance`

The serious warnings, indications of ambiguous code, are

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, resulting from illegal code, and will terminate all attempts to write an object file.

Listing False Conditionals

Conditional directives that have been evaluated as false are not included in the listing files unless **masm** is told to include them.

Syntax

-X

The **-X** option directs **masm** to copy to the assembly listing all statements forming the body of conditional-assembly blocks 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 11, "Controlling Assembly Output."

The **.LFCOND**, **.SFCOND**, and **.TFCOND** directives can override the effect of the **-X** option, as described in the section, "Controlling Listing of Conditional Blocks," in Chapter 11. The **-X** option does not affect the assembly listing unless you direct the assembler to create an assembly-listing file with the **-l** option.

Example

```
masm -X -l file.s
```

In this example, the listing of false conditionals is turned on when *file.s* is assembled, and the listing file is created. Directives in the source file can override the **-X** option to change the status of false-conditional listing.

Displaying Error Lines on Standard Error

Syntax

-x

The **-x** option directs **masm** to send lines containing errors to standard error. This option is now set by default and the use of the **-x** option on the command line is not necessary.

Writing Symbolic Information to the Object File

Information used by a symbolic debugger is not sent to the object file unless **masm** is instructed to from the command line.

Syntax

-Zi
-Zd

The **-Zi** and **-Zd** options direct **masm** to write symbolic information to the object file. There are two types of symbolic information available: line-number data and symbolic data. The **-Zi** option writes both line-number and symbolic data to the object file.

Line-number data relates each instruction to the source line that created it. Some debuggers need this information for source-level debugging.

Symbolic data specifies a size for each variable or label used in the program. This includes both public and nonpublic labels and variable names. Public symbols are discussed in Chapter 7, ''Creating Programs from Multiple Modules.''

The **-Zd** option writes only line-number information to the object file. It can be used 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.

The option names **-Zi** and **-Zd** are similar to corresponding option names for recent versions of Microsoft compilers.

Reading Assembly Listings

An assembly listing of your source file is created whenever you give the **-l** option on the **masm** command line. 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). Only the types of symbols encountered in the program are listed. For example, if your program has no macros, there will be no macro section in the symbol table.

Reading Code in a Listing

When given the **-l** option, the assembler lists the code generated from the statements of a source file. Each line has the following syntax:

[*offset*] [*code*] *statement*

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 at assembly time. If the value is calculated at link or load 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 appear directly below the statement where the error occurred. For a list of **masm** errors and a discussion of the format in which errors are displayed, refer to Appendix E, "Error Messages and Exit Codes." An example of an error line and message is shown here:

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

The 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 occurred. These lines will seldom be the same.

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
|	80386 size or address prefix

Example

The sample listing shown in this section is produced by using the **-Zl** option. The command line is as follows:

```
masm -l listdemo.s
```

The following is the code portion of the resulting listing.

Example

```
Microsoft (R) Macro Assembler Version 5.00.17 Nov 15 22:09:52 1987
Listing features demo                                    Page     1-1


                                  TITLE  Listing features demo

                                  INCLUDE      asm.mac
                          C
                          C StrAlloc     MACRO name, text
                          C name         DB    &text
                          C              DB    0ah, 0
                          C l&name       EQU   $ - name
                          C              ENDM


 = 0080                           larg   EQU   80h

                                  .MODEL small

                            color RECORD b:1,r:3,i:1=1,f:3=7

                            date  STRUC
 0000   05                  month DB    5
 0001   07                  day   DB    7
 0002   07C3                year  DW    1987
 0004                       date  ENDS

 0000                             .DATA
 0000   0F                  text  color <>
 0001   09                  today date  <9,22,1987>
 0002   16
 0003   07C3

 0005   0064[               buffer dw   100 DUP(?)
          ????
                    ]

                            StrAlloc    ending, "Finished"
 00CD   46 69 6E 69 73 68 65   1 ending      DB     "Finished"
 00D5   0A 00                  1             DB     0ah, 0

                                  EXTRN  _exit:proc
                                  EXTRN  _write:proc
                                  EXTRN  work:proc

 0000                             .CODE
                            PUBLIC _main
 0000                       _main proc
 0000   B8 0063                         mov    ax, 'c'
 0003   26: 8B 0E 0080                  mov    cx, es:larg
 0008   BF 0052                         mov    di, 82
 000B   F2/ AE                          repne  scasb
```

Example (cont.)

```
Microsoft (R) Macro Assembler Version 5.00.17 Nov 15 22:09:52 1987
Listing features demo                                   Page     1-2


 000D  57                        push  di
                                 EXTRN work:NEAR
 000E  E8 0000 E                 call  work
 0011  59                        pop   cx
 0012  6A 33                     push  0c

listdemo.s(40): error A2107: Non-digit in number
 0014  E8 0000 E                 call  _exit
 0017                      _main endp
 0017                            end
```

Reading a Macro Table

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

Example

```
Macros:

                 N a m e                    Lines

StrAlloc . . . . . . . . . . . .              3
```

Reading a Structure and Record Table

All structures and records declared in the source file are given at the end of the listing file. The names are listed alphabetically. Each name is followed by all the fields of that particular record or structure, in the order in which they are declared. All values are hexadecimal.

Example

```
Structures and Records:

            N a m e                 Width   # fields
                                     or      or
                                    Shift   Width    Mask    Initial

color . . . . . . . . . . . . . .   0008    0004
  b . . . . . . . . . . . . . . .   0007    0001     0080    0000
  r . . . . . . . . . . . . . . .   0004    0003     0070    0000
  i . . . . . . . . . . . . . . .   0003    0001     0008    0008
  f . . . . . . . . . . . . . . .   0000    0003     0007    0007
date . . . . . . . . . . . . . . .  0004    0003
  month . . . . . . . . . . . . .   0000
  day . . . . . . . . . . . . . .   0001
  year . . . . . . . . . . . . . .  0002
```

There are five columns of information in a structure and record table. They are organized as follows:

Heading	**Meaning**
`N a m e`	This is the name of the structure, record, or the fields therein.
`Width or Shift`	If the entry in this column follows the name of a structure (*COLOR*, in the example), then it refers to the width of that structure in bytes. If the entry follows the name of a field within that structure, then it refers to the shift, or offset, of that field (in bytes). The entries for records, and fields within records, are analogous, except that the values are in bits instead of bytes.
`# fields or Width`	In this column, entries that follow the name of a structure or record are the number of fields within that structure or record. The entry that follows the name of a field within a structure is the width of that field in bits.
`Mask`	This column contains the maximum value of the named record field. This value assumes that all other fields in the record are set to 0.

`Initial` This column contains the initial value, if any, of the named record field. This value assumes that all other fields in the record are set to 0.

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                 Length   Align   Combine Class

DGROUP . . . . . . . . . . . . . .  GROUP
  _DATA  . . . . . . . . . . . . .  00D7     WORD    PUBLIC  'DATA'
_TEXT  . . . . . . . . . . . . . .  0017     WORD    PUBLIC  'CODE'
```

The ''Name'' column lists the names of all segments and groups. Segment and group names are given in alphabetical order, except for segments that belong to a group. Names of segments belonging to a group are placed under the group name in the order in which they were added to the group.

The ''Length'' 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 the section, ''Defining Full Segments,'' in Chapter 4.

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

2

```
Symbols:

                N a m e              Type    Value Attr

b  . . . . . . . . . . . . . .              0007
buffer . . . . . . . . . . . .  L WORD  0005  _DATA Length = 0064

ending . . . . . . . . . . . .  L BYTE  00CD  _DATA

f  . . . . . . . . . . . . . .              0000

i  . . . . . . . . . . . . . .              0003

larg . . . . . . . . . . . . .  NUMBER  0080
lending  . . . . . . . . . . .  NUMBER  000A

r  . . . . . . . . . . . . . .              0004

text . . . . . . . . . . . . .  L BYTE  0000  _DATA
today  . . . . . . . . . . . .  L DWORD       0001   _DATA

work . . . . . . . . . . . . .  L NEAR  0000  _DATA External

@CodeSize  . . . . . . . . . .  TEXT  0
@DataSize  . . . . . . . . . .  TEXT  0
Microsoft (R) Macro Assembler Version 5.00.17 Nov 15 22:09:52 1987
Listing features demo                                   Symbols-2


@code  . . . . . . . . . . . . TEXT  _TEXT
@fileName  . . . . . . . . . . TEXT  listdemo.s
_exit  . . . . . . . . . . . . L NEAR 0000  _DATA External
_main  . . . . . . . . . . . . N PROC 0000  _TEXT Global Length=0017
_write . . . . . . . . . . . . L NEAR 0000  _DATA External
```

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 *type*	An ''L'' before a type refers to a label to that type, such as *L NEAR* (a near label), *L BYTE* (a byte label), etc.
N PROC	A near procedure label
F PROC	A far procedure label
NUMBER	An absolute label
ALIAS	An alias for another symbol
OPCODE	An equate for an instruction opcode
TEXT	A text equate
BYTE	One byte
WORD	One word (two bytes)
DWORD	Doubleword (four bytes)
FWORD	Farword (six bytes)
QWORD	Quadword (eight bytes)
TBYTE	Ten bytes
number	Length in bytes of a structure variable

The length of a multiple-element variable, such as an array or string, is the length of a single element, not the length of the entire variable. For example, string variables are always shown as *L BYTE.*

The ''Value'' column shows the symbol's value if the symbol represents an absolute value defined with an **EQU** or equal-sign (=) directive. 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**, **PUBLIC**, or **COMM** directives. The scope can be **EXTERNAL**, **GLOBAL**, or **COMMUNAL**. The code length (in hexadecimal) is given only for procedures. The "Attr" column is blank if the symbol has no attribute.

The text equates, shown at the end of the sample table, are defined automatically when you use simplified segment directives (see the section, "Understanding Memory Models", in Chapter 4).

Reading Assembly Statistics

Data on the assembly, including the number of lines and symbols processed and the errors or warnings encountered, are shown at the end of the listing. For further information on errors and warnings, see Appendix E, "Error Messages and Exit Codes."

Example

```
   48 Source  Lines
   52 Total   Lines
   53 Symbols

45570 + 310654 Bytes symbol space free

    0 Warning Errors
    1 Severe  Errors
```

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 shows 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.s(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. This error is displayed because **masm** 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:

```
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 three 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.

Part 2

Using Directives

Part 2 of this manual (Chapters 3-11) describes the directives and operators recognized by the Macro Assembler. Directives tell you how to generate code and data at assembly time. Operators tell you how to combine operands to form assembly-language expressions.

Chapter 3 introduces basic concepts of the assembly language recognized by the Macro Assembler. Topics covered include symbols, constants, statement syntax, and processor directives.

Chapters 4-7 explain the different directives and operators. The material is organized topically, with related directives discussed together. Operators and expressions are discussed specifically in Chapter 8.

Chapter 9 describes how to use directives to assemble code conditionally. This chapter covers two types of conditional directives: conditional-assembly directives and conditional-error directives.

Chapter 10 explains how to use equates and macros to make the assembly process more efficient.

Chapter 11 describes how to control the way **masm** reports assembly results.

Chapter 3

Writing Source Code

Introduction

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 using **ld** to form executable files.

Source files are made up of assembly-language statements. Statements are in turn made up of mnemonics, operands, and comments. This chapter describes how to write assembly-language statements. Symbol names and constants are explained. It also tells you how to start and end assembly-language source files.

Writing Assembly-Language Statements

A statement is a combination of 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.

Syntax

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

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

Field	**Purpose**
name	Labels the statement so that the statement can be accessed by name in other statements
operation	Defines the action of the statement
operands	Defines the data to be operated on 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 letters. 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 (except the comment field) must be separated from other fields by a space or tab character. This is the only structure limitation imposed by **masm**. For example, the following code is legal:

```
        .386
        title   hello
        .model  small

        .data
message db      "Hello, world", 10, 0  ; message to be written
lmessage equ    - message              ; length of message

extrn   _exit:proc
extrn   _write:proc

        .code
public  _main
_main   proc
        push    bp
        mov     bp, sp                 ; establish stack frame

        push    lmessage               ; push length of message
                                       ;   onto the stack

        push    OFFSET message         ; push address of
                                       ;   message onto the stack

        push    1
        call    _write                 ; write(1,message,lmessage)
        add     sp, 6                  ; remove arguments to write()

        push    0
        call    _exit

        leave
_main   endp

        end
```

3

However, the code is much easier to interpret if each field is assigned a specified tab position and a standard convention is used for capitalization. The example program in Chapter 1, ''Getting Started,'' is the same as the example above except for the conventions used.

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 give directions to the assembler. They specify the manner in which the assembler is to generate object code at assembly time. Part 2, ''Using Directives,'' describes the directives recognized by the assembler. Directives are also discussed in Part 3, ''Using Instructions.''

Instructions give directions 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."

Operands define the data that is used by directives and instructions. They can be made up of symbols, constants, expressions, and registers. The sections, "Assigning Names to Symbols" and "Constants," discuss symbol names and constants. Operands, expressions, and registers are discussed throughout the manual, but particularly in Chapter 8, "Using Operands and Expressions," and Chapter 13, "Using Addressing Modes."

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 nonblank character after the **COMMENT** directive. The *text* includes the comments up to and including the line containing the next occurrence of the delimiter.

Example

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

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 (both uppercase and lowercase), digits, and special characters. The Macro Assembler recognizes the following character set:

A-Z a-z 0-9

? @ _ $: . [ ] ( ) < > { } + - / *

& % ! ' ~ / \ = # ^ ; , ' "

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

- A name can have any combination of uppercase and lowercase letters. Case sensitivity is retained by the assembler, unless the **-Mu** or **-Mx** options are used, as shown in the section, ''Specifying Case Sensitivity,'' in Chapter 2.
- Digits may be used within a name, but not as the first character.
- A name can be given any number of characters, but only the first 31 are significant. All other characters are ignored.
- The following characters may be used at the beginning of a name or within a name: underscore (_), question mark (**?**), dollar sign (**$**), and at sign (**@**).
- The period (**.**) is an operator and cannot be used within a name, but it can be used as the first character of a name.
- A name may not be the same as any reserved name. Note that two special characters, the question mark (**?**) and the dollar sign (**$**), are reserved names and therefore can't stand alone as symbol 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. All uppercase and lowercase letter combinations of these names are treated as the same name.

Table 3.1 lists names that are always reserved by the assembler. Using any of these names for a symbol results in an error.

Table 3.1

Reserved Names

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

In addition to the names in Table 3.1, instruction mnemonics and register names are considered reserved names. These vary depending on the processor directives given in the source file. For example, the register name **EAX** is a reserved word with the **.386** directive but not with the **.286** directive. The section, "Defining Default Assembly Behavior," describes processor directives. Instruction mnemonics for each processor are listed in Appendix B, ''Instruction Summary.'' Register names are listed in the section, ''Using Register Operands,'' in Chapter 13.

Constants

Constants can be used in source files to specify numbers or strings that are set or initialized at assembly time. Four types of constant values are recognized : integers, packed binary coded decimals, real numbers, and strings.

3

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 and equates, or 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 the section, "Packed Binary Coded Decimal Constants."

Integer constants can be specified in binary, octal, decimal, or hexadecimal values. Table 3.2 shows the legal digits for each of these radixes. For the hexadecimal radix, the digits can be either uppercase or lowercase letters.

Table 3.2

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 by using radix specifiers, or a default radix can be defined globally with the **.RADIX** directive.

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 letters; sample code in this manual uses lowercase letters.

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 letters. Sample code in this manual uses uppercase letters.

If no radix is given, the assembler interprets the integer by 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    00001111b +  17o + 0Fh + 15d ; 4 * 15
```

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 2-16. It defines whether the numbers are binary, octal, decimal, hexadecimal, or numbers of some other base.

Numbers given in *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** directive. Initial values for real-number variables declared with these directives are always evaluated as decimal unless a radix specifier is appended. Also, the **.RADIX** directive does not affect the optional radix specifiers, **B** and **D**, used with integer numbers. When the letters **B** or **D** appear at the end of any integer, they are always considered to be a radix specifier even if the current 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 as 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
```

Packed 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 10-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 the section, ''Binary Coded Decimal Variables,'' in Chapter 5, for complete information on storage of binary coded decimals.

Examples

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

Real-Number Constants

A real number is a number consisting of an integer part, a fractional part, and an exponent. Real numbers are usually 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 the section, ''Real-Number Variables,'' in Chapter 5, for an explanation of how real numbers are encoded.

You can specify the encoded format directly using hexadecimal digits (0-9 or A-F). The number must begin with a decimal digit (0-9) and cannot be signed. It must be followed by the real-number designator (**R**). This designator is used the same as a radix designator except 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.)

Note

Real numbers will be encoded differently depending upon whether you use the **.MSFLOAT** directive. By default, real numbers are encoded in the IEEE format. This is a change from previous versions, which assembled real numbers by default in the Microsoft Binary format. The **.MSFLOAT** directive overrides the default and specifies Microsoft Binary format. See the section, ''Real-Number Variables,'' in Chapter 5, for a description of these formats.

3

Example

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

            ; Assumes .MSFLOAT
mbshort     DD      81000000r               ; 1.0 as Microsoft Binary short
mblong      DQ      8100000000000000r       ; 1.0 as Microsoft Binary long

            ; Assumes default IEEE format
ieeeshort   DD      3F800000r               ; 1.0 as IEEE short
ieeelong    DQ      3FF0000000000000r       ; 1.0 as IEEE long

            ; The same regardless of processor directives
temporary   DT      3FFF8000000000000000r ; 1.0 as 10-byte temporary real
```

String Constants

A string constant consists of one or more ASCII characters enclosed in single 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'
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.
```

3

Defining Default Assembly Behavior

Since the assembler processes a source-code file sequentially, any directives that define the behavior of the assembler for sections of code or for the entire source file must come before the sections affected by the directive.

There are three types of directives that may define behavior for the assembly:

3

1. The **.MODEL** directive defines the memory model.

2. Processor directives define the processor and coprocessor.

3. The **.MSFLOAT** directive and the coprocessor directives define how floating-point variables are encoded.

These directives are optional. If you do not use them, **masm** makes default assumptions. However, if you do use them, you must put them before any statements that will be affected by them.

The **.MSFLOAT** and **.MODEL** directives affect the entire assembly and can only occur once in the source file. Normally they should be placed at the beginning of the source file.

The **.MODEL** directive is part of the new system of simplified segment directives implemented in Version 5.0. It is explained in the section, "Defining the Memory Model," in Chapter 4.

The **.MSFLOAT** directive disables all coprocessor instructions and specifies that initialized real-number variables be encoded in the Microsoft Binary format. Without this directive, initialized real-number variables are encoded in the IEEE format. This is a change from previous versions of the assembler, which used Microsoft Binary format by default and required a coprocessor directive or the **-r** option to specify IEEE format. **.MSFLOAT** must be used for programs that require real-number data in the Microsoft Binary format. The section, "Real-Number Variables," in Chapter 5, describes real-number data formats and the factors to consider in choosing a format.

Processor and coprocessor directives define the instruction set that is recognized by **masm**. They are listed and explained below:

Directive	Description
.8086	The **.8086** directive enables assembly of instructions for the 8086 and 8088 processors and the 8087 coprocessor. It disables assembly of the instructions unique to the 80186, 80286, and 80386 processors. This is the default mode and is used if no instruction set directive is specified. Using the default instruction set ensures that your program can be used on all 8086-family processors. However, if you choose this directive, your program will not take advantage of the more powerful instructions available on more advanced processors.
.186	The **.186** directive enables assembly of the 8086 processor instructions, 8087 coprocessor instructions, and the additional instructions for the 80186 processor.
.286	The **.286** directive enables assembly of the 8086 instructions plus the additional nonprivileged instructions of the 80286 processor. It also enables 80287 coprocessor instructions. If privileged instructions were previously enabled, the **.286** directive disables them. This directive should be used for programs that will be executed only by an 80286, or 80386 processor. For compatibility with previous versions of **masm**, the **.286C** directive is also available. It is equivalent to the **.286** directive.
.286P	This directive is equivalent to the **.286** directive except that it also enables the privileged instructions of the 80286 processor. This does not mean that the directive is required if the program will run in protected mode; it only means that the directive is required if the program uses the instructions that initiate and manage privileged-mode processes. These instructions (see the section, ''Controlling Protected-Mode Processes,'' in Chapter 19) are normally used only by systems programmers.

.386 The **.386** directive enables assembly of the 8086 and the nonprivileged instructions of the 80286 and 80386 processors. It also enables 80387 coprocessor instructions. If privileged instructions were previously enabled, this directive disables them.

This directive should be used for programs that will be executed only by an 80386 processor.

.386P This directive is equivalent to the **.386** directive except that it also enables the privileged instructions of the 80386 processor.

.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 the IEEE format for encoding floating-point variables.

This is the default mode and is used if no coprocessor directive is specified. This directive should be used for programs that must run with either the 8087, 80287, or 80387 coprocessors.

.287 The **.287** directive enables assembly of instructions for the 8087 floating-point coprocessor and the additional instructions for the 80287. It also specifies the IEEE format for encoding floating-point variables.

Coprocessor instructions are 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. See the section, ''Coordinating Memory Access,'' in Chapter 18, for an explanation.

.387 The **.387** directive enables assembly of instructions for the 8087 and 80287 floating-point coprocessors and the additional instructions and addressing modes for the 80387. It also specifies the IEEE format for encoding floating-point variables.

If you do not specify any processor directives, **masm** uses the following defaults:

- 8086/8088 processor instruction set
- 8087 coprocessor instruction set
- IEEE format for floating-point variables

Normally the processor and coprocessor directives can be used at the start of the source file to define the instruction sets for the entire assembly. However, it is possible to use different processor directives at different points in the source file to change assumptions for a section of code. For instance, you might have processor-specific code in different parts of the same source file. You can also turn privileged instructions on and off or allow unusual combinations of the processor and coprocessor.

There are two limitations on changing the processor or coprocessor:

1. The directives must be given outside segments. You must end the current segment, give the processor directive, and then open another segment. See the section, "Using Predefined Equates," in Chapter 4, for an example of changing the processor directives with simplified segment directives.

2. You can specify a lower-level coprocessor with a higher-level coprocessor, but an error message will be generated if you try to specify a lower-level processor with a higher-level coprocessor.

The coprocessor directives have the opposite effect of the **.MSFLOAT** directive. **.MSFLOAT** turns off coprocessor instruction sets and enables the Microsoft Binary format for floating-point variables. Any coprocessor instruction turns on the specified coprocessor instruction set and enables IEEE format for floating-point variables.

Examples

```
; .MSFLOAT affects the whole source file
          .MSFLOAT
          .8087              ; Ignored

; Legal - use 80386 and 80287
          .386
          .287

; Illegal - can't use 8086 with 80287
          .8086
          .287

; Turn privileged mode on and off
          .286P
          .
          .
          .
          .286
```

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 begins when the program is loaded.

Syntax

END [*startaddress*]

Any statements following the **END** directive are ignored by the assembler. For instance, 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 the section, ''Initializing the CS and IP Registers,'' in Chapter 4.

Chapter 4

Defining Segment Structure

Introduction

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 12, "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 by using simplified segment directives or 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. Simplified segment directives automatically define the segment structure required when combining assembler modules with modules prepared with Microsoft high-level languages.

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

This chapter describes both methods. If you choose to use simplified segment directives, you will probably not need to read about full segment definitions.

Simplified Segment Definitions

Version 5.0 of **masm** implements a new simplified system for declaring segments. By default, the simplified segment directives use the segment names and conventions followed 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 in which 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 that are consistent with Microsoft conventions are generated automatically. You can learn more about the **ASSUME** and **GROUP** directives in the sections, "Full Segment Definitions" and "Defining Segment Groups." However, for most programs you do not need to understand these directives. You simply use the simplified segment directives in the format shown in the examples.

Understanding Memory Models

To use simplified segment directives, you must declare a memory model for your program. The memory model specifies 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 find out what memory models it can use.

The most commonly used memory models are as follows:

Model	Description
Tiny	All data and code fits in a single segment. 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. All 386 C programs are ''small model'' in the sense that all the data and code each fit into a segment. However, on a 386, the segment size is so large that this ceases to be an issue.
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 (although 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 (although 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 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.

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

Specifying MS-DOS Segment Order

4

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

Syntax

DOSSEG

Using the **DOSSEG** directive enables you to maintain a consistent, logical segment order without actually defining segments in that order in your source file. Without this directive, the final segment order of the executable file depends on a variety of factors, such as segment order, class name, and order of linking. These factors are described in the section, "Full Segment Definitions."

Since segment order is not crucial to the proper functioning of most stand-alone assembler programs, you can simply use the **DOSSEG** directive and ignore the whole issue of segment order.

Note

Using the **DOSSEG** directive (or the **-dosseg** linker option) has two side effects. The linker generates symbols called **_end** and **_edata**. You should not use these names in programs that contain the **DOSSEG** directive. Also, the linker increases the offset of the first byte of the code segment by 16 bytes in small and compact models. This is to give proper alignment to executable files created with Microsoft compilers.

If you want to use the MS-DOS segment-order convention in stand-alone assembler programs, you should use the **DOSSEG** argument in the main module. Modules called from the main module need not use the **DOSSEG** directive.

You do not need to use the **DOSSEG** directive for modules called from Microsoft high-level languages, since the compiler already defines MS-DOS segment order.

Under the MS-DOS segment-order convention, segments have the following order:

1. All segment names having the class name **CODE**
2. Any segments that do not have class name **CODE** and are not part of the group **DGROUP**
3. Segments that are part of **DGROUP**, in the following order:
    1. Any segments of class **BEGDATA** (this class name is reserved for Microsoft use)
    2. Any segments not of class **BEGDATA**, **BSS**, or **STACK**
    3. Segments of class **BSS**
    4. Segments of class **STACK**

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 MS-DOS order.

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 the same for large and huge models, but the **@DataSize** equate (explained in the section, "Using Predefined Equates") 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 of the memory models described in the section, "Understanding Memory Models." 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 is generated.

Example 1

```
.MODEL  small
```

This statement defines default segments for small-model programs and creates the **ASSUME** and **GROUP** statements used by small-model programs. The segments are automatically ordered according to the Microsoft convention. The example statements 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 programs and creates the **ASSUME** and **GROUP** statements used by large-model programs. It does not automatically order segments according to the Microsoft convention. The example 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 Only

If you use the **.386** directive before the **.MODEL** directive, the segment definitions defines 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.

Defining Simplified Segments

The **.CODE**, **.DATA**, **.DATA?**, **.FARDATA**, **.FARDATA?**, **.CONST**, and **.STACK** directives indicate 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 the section, "Default Segment Names," for more information.

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 with a default size of one kilobyte.

Stacks are defined by the compiler or interpreter for modules linked with a main module from a high-level language.

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 allowing multiple code segments (medium and large). *Name* will be ignored if given with small or compact models.

Uninitialized data is any variable declared by 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 uninitialized data in any data segment.

Constant data is data that must be declared in a data segment but is not subject to change at run time. Use of this segment is optional for stand-alone assembler programs. If you are writing assembler routines to be called from a high-level language, you can use the **.CONST** directive to declare strings, real numbers, and other constant data that must be allocated as data.

4

Data in segments defined with the **.STACK**, **.CONST**, **.DATA** or **.DATA?** directives is placed in a group called **DGROUP**. Data in segments defined with the **.FARDATA** or **.FARDATA?** directives is not placed in any group. For more information on segment groups, see the section, ''Defining Segment Groups.'' When initializing the **DS** register to access data in a group-associated segment, the value of **DGROUP** should be loaded into **DS**. For information on initializing data segments, see the section, ''Initializing the DS Register.''

Example 1

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

This code uses simplified segment directives for a small-model, stand-alone assembler program. Notice that initialized data, uninitialized data, and a string constant are all defined in the same data segment. See the section, ''Default Segment Names,'' for an equivalent version that uses full segment definitions.

Example 2

```
            .MODEL  LARGE
            .FARDATA?
fuarray     DW      10 DUP (?)         ; Far uninitialized data
            .CONST
string      DB      "This is a string" ; String constant
            .DATA
niarray     DB      100 DUP (5)        ; Near initialized data
            .FARDATA
            EXTRN   xvariable:FAR
fiarray     DW      100 DUP (10)       ; Far initialized data
            .CODE   ACTION
            EXTRN   xprocedure:PROC
task        PROC
            .
            .
            .
            ret
task        ENDP
            END
```


This example uses simplified segment directives to create a module that might be called from a large-model, high-level-language program. Notice that different types of data are put in different segments to conform to Microsoft compiler conventions. See the section, ''Default Segment Names,'' for an equivalent version using full segment definitions.

Using Predefined Equates

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 as follows:

Name	**Value**
@CurSeg	This name has the segment name of the current segment. This value may be convenient for **ASSUME** statements, segment overrides, or other cases in which you need to access the current segment. It can also be used to end a segment, as shown:

```
@CurSeg ENDS     ; End current segment
       .286     ; Must be outside segment
       .CODE    ; Restart segment
```

@fileName

This value represents the base name of the current source file. For example, if the current source file is *task.s*, 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
```

4

@CodeSize and @DataSize

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

```
IF    @DataSize
les   bx,pointer          ; Load far pointer
mov   ax,es:WORD PTR [bx]
ELSE
mov   bx,WORD PTR pointer ; Load near pointer
mov   ax,WORD PTR [bx]
ENDIF
```

Segment equates

For each of the primary segment directives, there is a corresponding equate with the same name, except that the equate starts with an at sign (@) but the directive starts with a period. For example, the *@code* equate represents the segment name defined by the **.CODE** directive. Similarly, *@fardata* represents the **.FARDATA** segment name and *@fardata?* represents the **.FARDATA?** segment name. The *@data* equate represents the group name shared by all the near data segments. It can be used to access the segments created by the **.DATA**, **.DATA?**, **.CONST**, and **.STACK** segments.

These equates can be used in **ASSUME** statements and at any other time a segment must be referred to by name, for example:

```
ASSUME es:@fardata ; Assume ES to far data
                   ;   (.MODEL handles DS)
mov    ax,@data    ; Initialize near to DS
mov    ds,ax
mov    ax,@fardata ; Initialize far to ES
mov    es,ax
```

Note

Although predefined equates are part of the simplified segment system, the *@CurSeg* and *@fileName* equates are also available when using full segment definitions.

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:

- 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 and **FAR** for medium, large, and huge models. See the section, ''Procedure Labels,'' in Chapter 5, 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 following statement loads the address of *test1* relative to **DGROUP**:

```
mov     ax,OFFSET test1
```

The next statement loads the address of *test2* relative to the segment defined by the **.FARDATA** directive:

```
mov     ax,OFFSET test2
```

For more information on groups, see the section, "Defining Segment Groups."

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, some programmers may wish to know the actual names assigned to all segments.

Table 4.1 shows the default segment names created by each directive.

Table 4.1

Default Segments and Types for Standard Memory Models

Model	Directive	Name	Align	Combine	Class	Group
Small	.CODE	_TEXT	WORD	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	WORD	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	WORD	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	WORD	PUBLIC	'CODE'	
or huge	.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 models. 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. However, the group and assume statements generated by the **.MODEL** directive are not shown in listing files. For a program that uses all possible segments, group statements equivalent to the following would be generated:

```
DGROUP      GROUP _DATA,CONST,_BSS,STACK
```

For small and compact models, the following would be generated:

```
ASSUME  cs:_TEXT,ds:DGROUP,ss:DGROUP
```

For medium, large, and huge models, the following statement is given:

```
ASSUME  cs:name_TEXT,ds:DGROUP,ss:DGROUP
```

80386 Only

If the **.386** directive is used, the default align type for all segments is **DWORD**.

Example 1

```
            EXTRN   xvariable:WORD
            EXTRN   xprocedure:NEAR
DGROUP      GROUP   _DATA,_BSS
            ASSUME  cs:_TEXT,ds:DGROUP,ss:DGROUP
_TEXT       SEGMENT WORD PUBLIC 'CODE'
start:      mov     ax,DGROUP
            mov     ds,ax
            .
            .
            .
_TEXT       ENDS
_DATA       SEGMENT WORD PUBLIC 'DATA'
ivariable   DB      5
iarray      DW      50 DUP (5)
string      DB      "This is a string"
uarray      DW      50 DUP (?)
_DATA       ENDS
STACK       SEGMENT PARA STACK 'STACK'
            DB      100h DUP (?)
STACK       ENDS
            END     start
```

This example is equivalent to Example 1 in the section, "Defining Simplified Segments." The external variables are declared at the start of the source code in this example. With simplified segment directives, they can be declared in the segment in which they are used.

Example 2

```
DGROUP      GROUP   _DATA,CONST,STACK
            ASSUME  cs:TASK_TEXT,ds:FAR_DATA,ss:STACK
            EXTRN   xprocedure:FAR
            EXTRN   xvariable:FAR
FAR_BSS     SEGMENT PARA 'FAR_DATA'
fuarray     DW      10 DUP (?)          ; Far uninitialized data
FAR_BSS     ENDS
CONST       SEGMENT WORD PUBLIC 'CONST'
string      DB      "This is a string" ; String constant
CONST       ENDS
_DATA       SEGMENT WORD PUBLIC 'DATA'
niarray     DB      100 DUP (5)         ; Near initialized data
_DATA       ENDS
FAR_DATA    SEGMENT WORD 'FAR_DATA'
fiarray     DW      100 DUP (10)
FAR_DATA    ENDS
TASK_TEXT   SEGMENT WORD PUBLIC 'CODE
task        PROC    FAR
            .
            .
            .
            ret
task        ENDP
TASK_TEXT   ENDS
            END
```

This example is equivalent to Example 2 in the section, "Defining Simplified Segments." Notice that the segment order is the same in both versions. The segment order shown here is written to the object file, but it is different in the executable file. The segment order specified by the compiler overrides the segment order in the module object file.

Full Segment Definitions

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

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 are written in the order in which they appear in the source code. If the alphabetical method is specified, segments are written in the alphabetical order of their segment names.

4

The default is sequential. If no segment-order directive or option is given, segments are ordered sequentially. The segment-order method is only one factor in determining the final order of segments in memory. The **DOSSEG** directive (see the section, "Specifying MS-DOS Segment Order") and class type (see the section, "Controlling Segment Structure with Class Type") can also affect segment order.

The ordering method can be set by 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 the section, "Specifying the Segment-Order Method"), in Chapter 2. 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 are 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 modules, the compiler automatically sets the segment order.

Example 1

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

Example 2

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

Defining Full 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*] [*combine*] [*use*] ['*class*']
statements
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*, *combine*, *use*, 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 might try using the simplified segment directives discussed in the section, ''Simplified Segment Definitions.''

Note

Don't confuse the **PAGE** align type and the **PUBLIC** combine type 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.

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	Uses the next available byte address.
WORD	Uses the next available word address (2 bytes per word).
DWORD	Uses the next available doubleword address (4 bytes per doubleword); the **DWORD** align type is normally used in 32-bit segments with the 80386 processor.
PARA	Uses the next available paragraph address (16 bytes per paragraph) .
PAGE	Uses the next available page address (256 bytes per page).

If no *align* type is given, **PARA** is used by default (except with the 80386 processor).

The linker uses the alignment information to determine the relative start address for each segment.

Align types are illustrated in Figure 4.1, in the section, "Defining Segment Combinations with Combine Type," below.

Setting Segment Word Size with Use Type

80386 Only

The *use* type specifies the segment word size on the 80386 processor. Segment word size is the default operand and address size of a segment.

The *use* type can be **USE16** or **USE32**. These types are only relevant if you have enabled 80386 instructions and addressing modes with the **.386** directive. The assembler generates an error if you specify *use* type when the 80386 processor is not enabled.

With the 80286 and other 16-bit processors, the segment word 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, the segment word size is 32 bits by default when the **.386** directive is used.

The effect of addressing modes is changed by the word size you specify for the code segment. For more information on 80386 addressing modes, see the section, "80386 Indirect Memory Operands," in Chapter 13. The meaning of the **WORD** and **DWORD** type specifiers is not changed by the *use* type. **WORD** always indicates 16 bits and **DWORD** always indicates 32 bits regardless of the current segment word size.

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. For the most part, it is used only by systems programmers.

Example 1

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

Example 2

```
; 32-bit segment
            .386
_TEXT       SEGMENT DWORD USE32 PUBLIC 'CODE'
            .
            .
            .
_TEXT       ENDS
```

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 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 length of the segment. The stack segment of your program should normally use the **STACK** type, since this automatically initializes the **SS** register, as described in the section, "Initializing the SS and SP Registers." If you create a stack segment and do not use the **STACK** type, you must give instructions to initialize the **SS** and **SP** registers.
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 linker treats **MEMORY** segments exactly the same as **PUBLIC** segments. You are allowed to use **MEMORY** type

even though **ld** does not recognize a separate **MEMORY** type. This feature is compatible 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 locations defined by hardware. The linker will not generate any code or data for **AT** segments, but existing code or data can be accessed by name if it is given a label in an **AT** segment. The section, ''Setting the Location Counter,'' in Chapter 5, shows an example of a segment with **AT** combine type. 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.

4

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 types are given for an initial segment definition, 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, **ld** displays a warning message. You can ignore this message if you have a specific reason for not declaring a stack segment.

Example

The following source-code shell illustrates one way in which the *combine* and *align* types can be used. Figure 4-1 shows the way **ld** would load the sample program into memory.

```
            NAME module_1

ASEG        SEGMENT WORD PUBLIC 'CODE'
start:      .
            .
            .
ASEG        ENDS

BSEG        SEGMENT WORD COMMON 'DATA'
            .
            .
            .
BSEG        ENDS

BSEG        SEGMENT PARA STACK 'STACK'
            .
            .
            .
CSEG        ENDS

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

            NAME module_2

ASEG        SEGMENT WORD PUBLIC 'CODE'
            .
            .
            .
ASEG        ENDS

BSEG        SEGMENT WORD COMMON 'DATA'
            .
            .
            .
BSEG        ENDS
            END
```

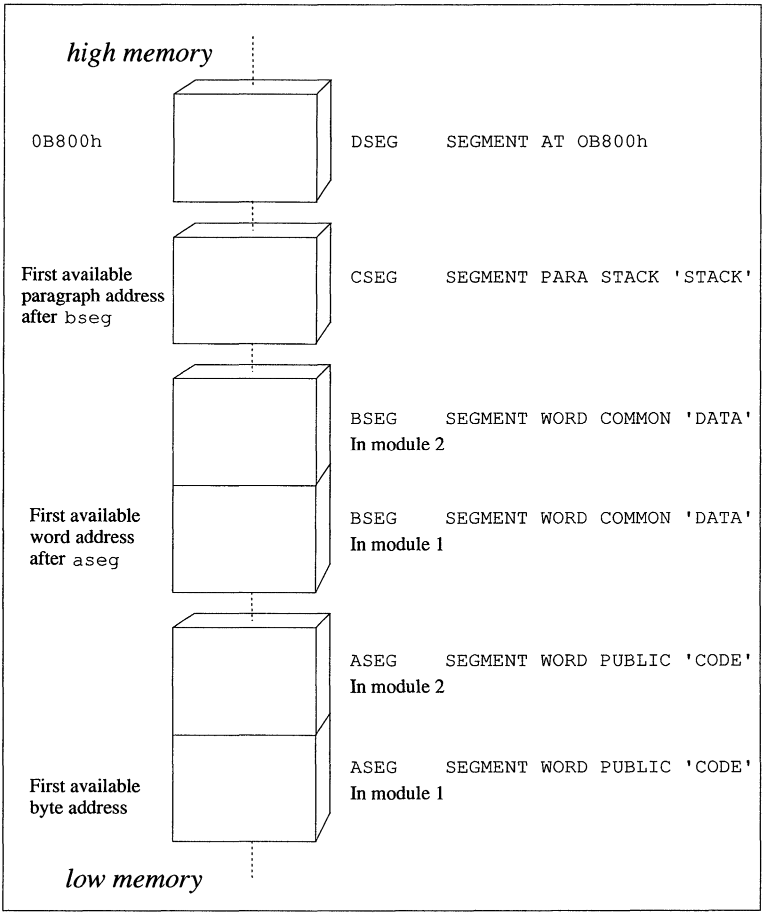


Figure 4-1 Segment Structure with Combine and Align Types

Controlling Segment Structure with Class Type

Class type is a means of associating segments that have different names, but similar purposes. It can be used to control segment order and to identify the code segment.

The *class* name must be enclosed in single quotation marks (').

All segments belong to a class. Segments for which no class name is explicitly stated have the null class name. Because **ld** 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.

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

Class type is one of two factors that control the final order of segments in an executable file. The other factor is 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 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 the **.ALPHA** directive was used. In the example above, 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* are placed together because they both have class type *SEG_1*. The final order in memory is *A_SEG, C_SEG, B_SEG*.

Since **ld** processes modules in the order it receives them on the command line, you may not always be able to easily specify the order 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. In this case, **ld** will not put the segments in the proper order because it first loads the segments encountered in the first module.

You can avoid this problem by starting your program with dummy segment definitions in the order 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 WORD 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
```

Once a segment has been defined, you do not need to specify the align, combine, use, and class types on subsequent definitions. For example, if your code defined dummy segments as shown above, you could define an actual data segment with the following statements:

```
_DATA        SEGMENT
             .
             .
             .
_DATA        ENDS
```

Defining Segment Groups

A group is a collection of segments 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 want them all to be accessible from a single, common segment register at run time.

Syntax

name **GROUP** *segment* [,*segment*]...

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 *segment* can be any previously defined segment or a **SEG** expression (see the section, ''SEG Operator'', in Chapter 8).

Segments can be added to a group one at a time. For example, you can define and add segments to a group one by one. This is a new feature of Version 5.0. 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 bytes.

Note

When the **MODEL** directive is used, the offset of a group-relative segment refers to the ending address of the segment, not the beginning. For example, the expression *OFFSET STACK* evaluates to the end of the stack segment.

Group names can be used with the **ASSUME** directive (discussed in the section, ''Associating Segments with Registers'') and as an operand prefix with the segment-override operator (discussed in the section, ''Segment-Override Operator'', in Chapter 8).

Example

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

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

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

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

Figure 4-2 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 *asym* and *csym* have offsets from the beginning of the group, which is also the beginning of *ASEG*. The offset of *bsym* is from the beginning of *BSEG*, since it is not part of the group. This sample illustrates the way **ld** organizes segments in a group. It is not intended as a typical use of a group.

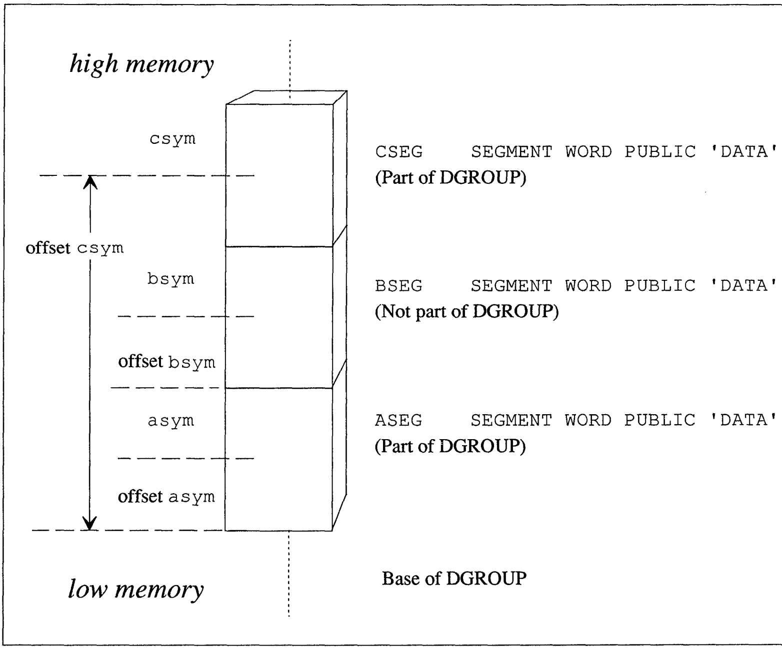


Figure 4-2 Segment Structure with Groups

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; **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 register 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 the section, ''Initializing Segment Registers.''

4

Syntax

ASSUME *segmentregister*:*name* [,*segmentregister*:*name*]...
ASSUME *segmentregister*:**NOTHING**
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 automatically assume that if the default segment is *segmentregister*, then the label or variable is 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 processor). 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**
- A **SEG** expression (see the section, ''SEG Operator'', in Chapter 8)
- A string equate that evaluates to a segment or group name (but not a string equate that evaluates to a **SEG** expression)

The keyword **NOTHING** cancels the current segment selection. For example, the statement **ASSUME NOTHING** cancels all register selections made by previous **ASSUME** statements.

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 the section, ''Segment-Override Operator'', in Chapter 8). The segment-override operator is more convenient for one-time overrides, whereas the **ASSUME** directive may be more convenient if previous assumptions must be overridden for a sequence of instructions.

Example

```
            .MODEL   large        ; DS automatically assumed to @data
            .STACK   100h
            .DATA
d1          DW       7
            .FARDATA
d2          DW       9

            .CODE
start:      mov      ax,@data     ; Initialize near data
            mov      ds,ax
            mov      ax,@fardata ; Initialize far data
            mov      es,ax
            .
            .
            .

; Method 1 for series of instructions that need override
; Use segment override for each statement

            mov      ax,es:d2
            .
            .
            .
            mov      es:d2,bx

; Method 2 for series of instructions that need override
; Use ASSUME at beginning of series of instructions
            ASSUME   es:@fardata
            mov      cx,d2
            .
            .
            .
            mov      d2,dx
```

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 from 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.

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** register is normally initialized to 0. You can change the initial value of the **IP** register by using the **ORG** directive (see the section, ''Setting the Location Counter'', in Chapter 5) just before the *startaddress* label.

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 is assumed. Neither **masm** nor **ld** will generate an error message, but your program will probably start execution at the wrong address.

Example

```
; Module 1
            .CODE
start:      .                  ; First executable instruction
            .
            .
            EXTRN   task:NEAR
            call    task
            .
            .
            .
            END     start      ; Starting address defined in main module

; Module 2
            PUBLIC  task
            .CODE
task        PROC
            .
            .
            .
task        ENDP
            END                ; No starting address in secondary module
```

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

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 for 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 convention and segment order, the address loaded into the **DS** register is not a segment address but the address of **DGROUP**, as shown in Example 2. With simplified segment directives, the address of **DGROUP** is represented by the predefined equate **@data**.

Example 2

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

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 size of the stack segment. Thus **SS:SP** initially points to the end of the stack.

If you use a stack segment with combine type **STACK**, initialization of **SS** and **SP** is automatic. The stack is automatically set up in this way with the simplified segment directives.

However, you can initialize or reinitialize the stack segment directly by changing the values of **SS** and **SP**. Since hardware interrupts use the same stack as the program, you should turn off hardware interrupts while changing the stack. Most 8086-family processors do this automatically, but early versions of the 8088 processor do not.

Example

```
          .MODEL  small
          .STACK  100h                 ; Initialize "STACK"
          .DATA
          .
          .
          .
          .CODE
start:    mov     ax,@data             ; Load segment location
          mov     ds,ax                ;   into DS register
          mov     ss,ax                ; Load same value as DS into SS
          mov     sp,OFFSET STACK      ; Give SP new stack size
          .
          .
          .
```

This example reinitializes **SS** so that it has the same value as **DS**, and adjusts **SP** to reflect the new stack offset. Microsoft high-level-language compilers do this so that stack variables in near procedures can be accessed relative to either **SS** or **DS**.

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.

Example

```
ASSUME  es:@fardata        ; Tell the assembler
mov     ax,@fardata        ; Tell the processor
mov     es,ax
```

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.

Nesting of segments makes it possible to mix segment definitions in programs that use simplified segment directives for most segment definitions. When a full segment definition is given, the new segment is nested in the simplified segment in which it is defined.

Nesting Segments

Example 1

```
; Macro to print message to standard output
; Uses full segment definitions - segments nested

         .286

extrn    _write:proc

message MACRO     text
         LOCAL     symbol, lsymbol
_DATA    segment   word public 'DATA'
symbol   db        &text
         db        10
lsymbol  db        0
_DATA    ends
         push      offset lsymbol - offset symbol
         push      offset symbol
         push      1
         call      _write
         add       sp, 6
         endm

_TEXT    segment   byte public 'CODE'
         assume    cs:_TEXT, ds:_DATA, ss:_DATA
public   _main
_main    proc      near
         push      bp
         mov       bp, sp
         message   "Please insert disk"
         message   "This is the second string"
         leave
         ret
_main    endp
_TEXT    ends
         end
```

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

Example 2

```
; Macro to print message to standard output
; Uses simplified segment directives - segments not nested

        .286
        .MODEL      SMALL

extrn   _write:proc

message MACRO       text
        LOCAL       symbol, lsymbol
        .DATA
symbol  db          &text
        db          10
lsymbol db          0

        .CODE
        push        offset lsymbol - offset symbol
        push        offset symbol
        push        1
        call        _write
        add         sp, 6
        endm

        .CODE
public  _main
_main   proc        near
        push        bp
        mov         bp, sp
        message     "Please insert disk"
        message     "This is the second string"
        leave
        ret
_main   endp
_TEXT   ends
        end
```

Although Example 2 has the same practical 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, and then assembly of the code segment is restarted.

Chapter 5

Defining Labels and Variables

Introduction

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 4, ''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 offset 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 you how to assign labels and most kinds of variables. (Multifield variables such as structures and records are discussed in Chapter 6, ''Using Structures and Records.'') The chapter also discusses related directives, including those that control the location counter directly.

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 as follows, with the number of bytes specified by each:

Specifier	Number of Bytes
BYTE	1
WORD	2
DWORD	4
FWORD	6
QWORD	8
TBYTE	10

5

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

The type specifiers that give the distance of a label are as follows:

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 the section, ''Simplified Segment Definitions''), in Chapter 4, the default type is near for small and compact models or far for medium, large, and huge models.

Directives that use type specifiers include **LABEL**, **PROC**, **EXTRN**, and **COMM**. Operators that use type specifiers include **PTR** and **THIS**.

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.

Near Code Labels

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

Syntax

name:

5

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 that the assembler currently associates with the **CS** register. The **ASSUME** directive is used to associate a segment with a segment register (see the section, ''Associating Segments with Registers''), in Chapter 4.

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 7, ''Creating Programs from Multiple Modules.''

Example

```
            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:
```

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.

5

Syntax

> *label* **PROC** [**NEAR**|**FAR**]
> *statements*
> **RET** [*constant*]
> *label* **ENDP**

The *label* assigns a symbol to the procedure. The distance can be **NEAR** or **FAR**. Any **RET** instructions within the procedure automatically have the same distance (**NEAR** or **FAR**) as the procedure. Procedures and the **RET** instruction are discussed in more detail in the section, ''Using Procedures,'' in Chapter 16.

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. If it does not find an **ENDP** directive to match each **PROC** directive, **masm** generates an error message.

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—that is, **NEAR** for small and compact models or **FAR** for medium, large, and huge 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 an address in 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 7, ''Creating Programs from Multiple Modules.''

Example

```
            call   task        ; Call procedure
            .
            .
            .
task        PROC   NEAR        ; Start of procedure
            .
            .
            .
            ret
task        ENDP               ; End of procedure
```

5

Code Labels Defined with the LABEL Directive

The **LABEL** directive provides an alternative method of defining code labels.

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**. **PROC** means **NEAR** or **FAR**, depending on the default memory model. You can use the **LABEL** directive to define a second entry point into a procedure. **FAR** code labels can also be the destination of far jumps or of far calls that use the **RETF** instruction (see the section, ''Defining Procedures'', in Chapter 16).

Example

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

5

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.

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 as follows:

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 symbol indicating that the value of the variable is undefined. You can define multiple values by using multiple initializers separated by commas, or by using the **DUP** operator, as explained in the section, "Arrays and Buffers."

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

Integer Variables

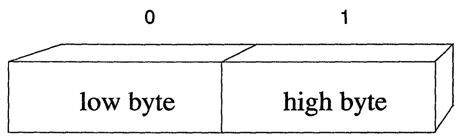
When defining an integer variable, you can specify an initial value as an integer constant or as a constant expression. If you specify an initial value too large for the specified variable, **masm** generates an error.

Integer values for all sizes except 10-byte variables are stored in the complement format of the binary two. They can be interpreted as either signed or unsigned numbers. For instance, 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 then to use the appropriate instructions to handle the value correctly.

The following is a list of the directives for defining integer variables along with the sizes of integers 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 32,767. The bytes of a word integer are stored in the following format:



Word

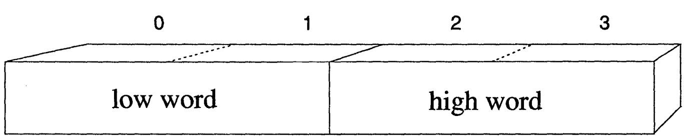
Note that in assembler listings and in many debuggers the bytes of a word are shown in the opposite order—high byte first—since this is the way most people think of numbers.

Word values can be used directly in 8086-family instructions. They can also be loaded, used in calculations, and stored with 8087-family instructions.

5

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. The words of a doubleword integer are stored in the following format:



Doubleword

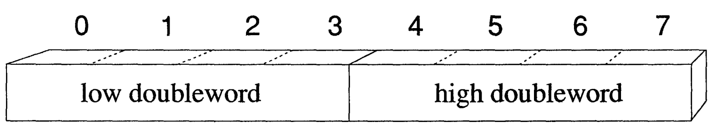
These 32-bit values (called long integers) can be loaded, used in calculations, and stored with 8087-family instructions. Some calculations can be done on these numbers directly with 16-bit 8086-family processors; others involve an indirect method of doing calculations on each word separately (see the section, "Adding"), in Chapter 15. These long integers can be used directly in calculations with the 80386 processor.

DF (farwords)

Allocates 6-byte (48-bit) integers. These values are normally only used as pointer variables on the 80386 processor (see the section, "Pointer Variables", below).

DQ (quadwords)

Allocates 64-bit integers. The doublewords of a quadword integer are stored in the following format:



Quadword

5

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, but others require an indirect method of doing calculations on each doubleword separately (see the section, "Adding"), in Chapter 15.

DT

Allocates 10-byte (80-bit) integers if the **D** radix specifier is used. By default, **DT** allocates packed BCD (binary coded decimal) numbers, as described in the section, "Binary Coded Decimal Variables," below. If you define binary 10-byte integers, you must write your own routines to use routines in calculations.

Example

```
integer     DB     16            ; Initialize byte to 16
expression  DW     4*3           ; Initialize word to 12
empty       DQ     ?             ; Allocate uninitialized quadword 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
```

Binary Coded Decimal Variables

Binary coded decimals (BCD) 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. They can be defined with the **DB** directive. For example, an unpacked BCD number could be defined and initialized as shown here:

5

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

Whether least-significant digits can come either first or last, depends on how you write the calculation routines that handle the numbers. Calculations with unpacked BCD numbers are discussed in the section, "Unpacked BCD Numbers," in Chapter 15.

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:

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

The 8087-family coprocessors can do fast calculations with packed BCD numbers, as described in Chapter 18, "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 the section, "Packed BCD Numbers," in Chapter 15, for details.

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 definitions 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 (four-character strings on the 80386) 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 is filled with zeros.

Examples

```
function9   DB     'Hello',10,'$'

asciiz      DB     "/u/me/asm/test.s",0 ; Use as ASCIIZ string

message     DB     "Enter file name: "
l_message   EQU    $-message
a_message   EQU    OFFSET message

str1        DB     "ab"                ; Stored as 61 62
str2        DD     "ab"                ; Stored as 62 61 00 00
str3        DD     "a"                 ; Stored as 61 00 00 00
```

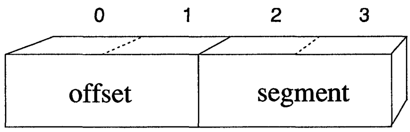
Pointer Variables

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 by 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 when the segment portion of the address may be outside the current segment. The segment and offset of a far pointer are stored in the following format:



Far Pointer

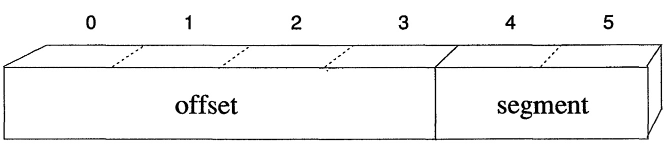
Examples

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

5

80386 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, and the segment portion consists of 16 bits. Therefore a near pointer is 32 bits (a doubleword), and a far pointer is 48 bits (a farword). The segment and offset of a 32-bit-mode far pointer are stored in the following format:



Far Pointer in 32-Bit Mode

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
```

Real-Number Variables

Real numbers must be stored in binary format. However, when initializing variables, you can specify decimal or hexadecimal constants and let the assembler automatically encode them into their binary equivalents. There are two different binary formats for real numbers that **masm** can use: IEEE or Microsoft Binary. You can specify the format by using directives (IEEE is the default).

This section tells you how to initialize real-number variables, describes the two binary formats, 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 as follows, along with the sizes of the numbers they can allocate:

Directive	Size
DD	Allocates short (32-bit) real numbers in either the IEEE or Microsoft Binary format.
DQ	Allocates long (64-bit) real numbers in either the IEEE or Microsoft Binary format.
DT	Allocates temporary or 10-byte (80-bit) real numbers. The format of these numbers is similar to the IEEE format. They are always encoded the same regardless of the real-number format. Their size is

nonstandard and incompatible with Microsoft high-level languages. Temporary-real format is provided for those who want to initialize 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; they can also use the values in calculations and store the results back to memory, as explained in Chapter 18, ''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 format
elong     DQ     3F41AA4C6F445B7Ar   ; 5.391E-4 encoded in IEEE format
```

5

The real-number designator (**R**) used to specify encoded numbers is explained in the section, ''Real-Number Constants,'' in Chapter 3.

Selecting a Real-Number Format

There are two different formats that **masm** can encode four- and eight-byte real numbers into: IEEE and Microsoft Binary. Your choice depends on the type of program you are writing. The four primary alternatives are as follows:

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 modules that will be called from such a language, your program should use the IEEE format. All versions of the C, FORTRAN, and Pascal compilers sold by Microsoft use the IEEE format.

3. If you are writing a module that will be called from Microsoft Part 1, ''Using Assembler Programs 286 BASIC, your program should use the Microsoft Binary format.

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; the Microsoft Binary format may be necessary for compatibility with existing source code.

Note

When you interface assembly-language modules with high-level languages, the real-number format only matters if you initialize real-number variables in the assembly module. If your assembly module does not use real numbers, or if all real numbers are initialized in the high-level-language module, the real-number format does not make any difference.

By default, **masm** assembles real-number data in the IEEE format. This is a change from previous versions of the assembler, which used the Microsoft Binary format by default. If you wish to use the Microsoft Binary format, you must put the **.MSFLOAT** directive at the start of your source file before initializing any real-number variables.

Real-Number Encoding

The IEEE format for encoding four- and eight-byte real numbers is illustrated in Figure 5-1.

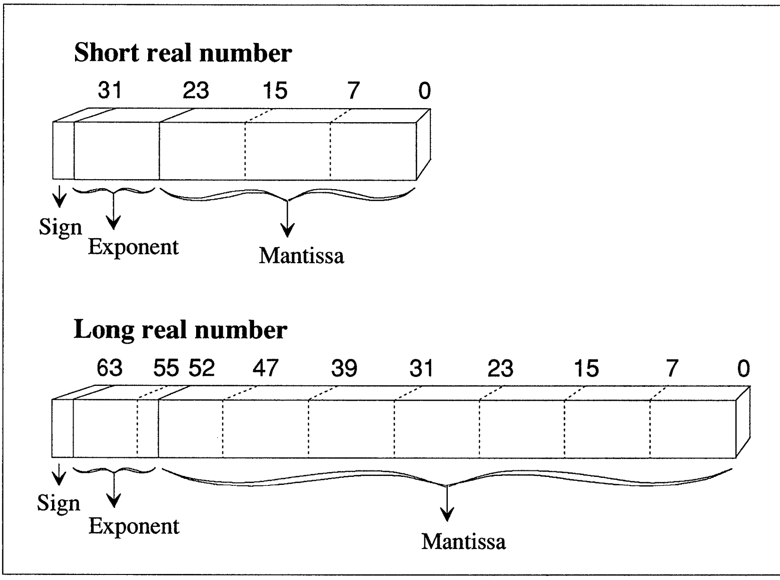


Figure 5-1 Encoding for Real Numbers in IEEE Format

The following list describes the parts of the real numbers:

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 number or 11 bits for long real number).

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 real numbers and 52 bits for long real numbers.

The Microsoft Binary format for encoding real numbers is illustrated in Figure 5-2.

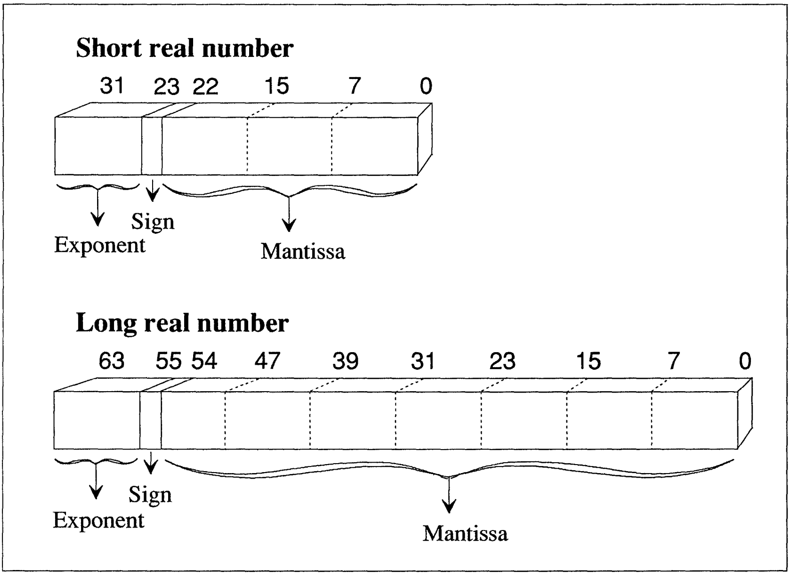


Figure 5-2 Encoding for Real Numbers in Microsoft Binary Format

The three parts of real numbers are:

1. Biased exponent (8 bits) in the high-address byte. The bias is 81h for short real numbers and 401h for long real numbers.

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 real numbers and 55 bits for long real numbers.

Also supported is 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 temporary-real format for encoding real numbers is illustrated in Figure 5-3.

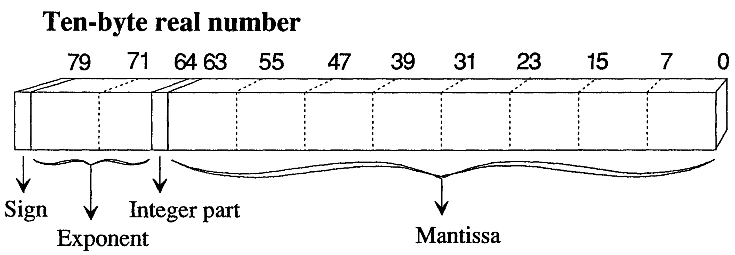


Figure 5-3 Encoding for Real Numbers in Temporary-Real Format

The four 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. The integer part 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 integer part of the mantissa. This differs from the 4- and 8-byte formats, in which the integer part is implicit.

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 undefined symbol (**?**) if there is no initial value.

Multiple initial values must be separated by commas. If multiple values are specified within the parentheses, the sequence of values is allocated *count* times. For example, the statement

```
        DB      5 DUP ("Text ")
```

allocates the string *"Text "* five times for a total of 20 bytes.

5

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 ("I am here ")        ; 320 byte buffer with
                                                 ;  signature for debugging
three_d     DD      5 DUP (5 DUP (5 DUP (0)))    ; 125 doublewords
                                                 ;  initialized to 0
```

Note

Sometimes **masm** will generate different object code when the **DUP** operator is used rather than when multiple values are given. For example, the statement

```
test1       DB      ?,?,?,?,?  ; Indeterminate
```

is "indeterminate." It causes **masm** to write five zero-value bytes to the object file. The statement

```
test2       DB      5 DUP (?)  ; Undefined
```

is "undefined." 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.

In most cases, the distinction between indeterminate and undefined definitions is trivial. The linker adjusts 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 varies depending on link order. If the module with the indeterminate value is linked last, the 0 initialized for it overrides 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 definition, the explicit value is always used in the executable file regardless of link order.

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
```

Setting the Location Counter

The location counter is the value **masm** maintains to keep track of the 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 offset specified 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

5

The value of the location counter, represented by the dollar sign ($), can be used in *expression*, as described in the section, "Using the Location Counter," in Chapter 8.

Example

```
; Labeling absolute addresses

STUFF       SEGMENT AT 0        ; Segment has constant value 0
            ORG    410h         ; Offset has constant value 410h
equipment   LABEL  WORD         ; Value at 0000:0410 labeled "equipment"
            ORG    417h         ; Offset has constant value 417h
keyboard    LABEL  WORD         ; Value at 0000:0417 labeled "keyboard"
STUFF       ENDS

            .CODE
            .
            .
            .
            ASSUME  ds:STUFF    ; Tell the assembler
            mov     ax,STUFF    ; Tell the processor
            mov     ds,ax

            mov     dx,equipment
            mov     keyboard,ax
```

The example illustrates one way of assigning symbolic names to absolute addresses. This technique is not possible under protected-mode operating systems.

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** directive aligns on the next byte that is a multiple of *number*. The *number* must 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.

5

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 give 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 do two memory fetches if the data is not word aligned. Similarly, using **ALIGN 4** can speed some operations with a 80386 processor, since the processor can fetch four bytes at a time if the data is doubleword aligned.

Note

The **ALIGN** directive is a new feature of Version 5.0 of the Macro Assembler. In previous versions, data could be word aligned by using the **EVEN** directive, but other alignments could not be specified.

The **EVEN** directive should not be used in segments with **BYTE** align type. Similarly, the *number* specified with the **ALIGN** directive should be at least equal to the size of the align type of the segment where the directive is given.

Example

```
            .MODEL  small
            .STACK  100h
            .DATA
            .
            .
            .
            ALIGN   4                   ; For faster data access
stuff       DW      66,124,573,99,75
            .
            .
            .
            ALIGN   4                 ; For faster data access
evenstuff   DW      ?,?,?,?,?
            .CODE
start:      mov     ax,@data          ; Load segment location
            mov     ds,ax             ;   into DS
            mov     es,ax             ;   and ES registers

            mov     cx,5              ; Load count
            mov     si,OFFSET stuff   ; Point to source
            mov     di,OFFSET evenstuff;  and destination
            ALIGN   4               ; Align for faster loop access
mloop:      lodsw                   ; Load a word
            inc     ax              ; Make it even by incrementing
            and     ax,NOT 1        ;   and turning off first bit
            stosw                   ; Store
            loop    mloop           ; Again
```

In this example, the words at *stuff* and *evenstuff* are forced to doubleword boundaries. This makes access to the data faster with processors that have either a 32-bit or 16-bit data bus. Without this alignment, the initial data might start on an odd boundary and the processor would have to fetch half of each word at a time with a 16-bit data bus or half of each doubleword with a 32-bit data bus.

Similarly, the alignment in the code segment speeds up repeated access to the code at the start of the loop. The sample code sacrifices program size in order to achieve significant speed improvements on the 80386 and more moderate improvements on the 8086 and 80286. There is no speed advantage on the 8088.

Chapter 6

Using Structures and Records

Introduction

The Macro Assembler can define and use two kinds of multifield variables: structures and records.

Structures are templates for data objects made up of smaller data objects. A structure can be used to define structure variables, which are made up of smaller variables called fields. Fields within a structure can be different sizes, and each can be accessed individually.

Records are templates for data objects whose bits can be described as groups of bits called fields. A record can be used to define record variables. Each bit field in a record variable can be used separately in constant operands or expressions. The processor cannot access bits individually at run time, but bit fields can be used with logical bit instructions to change bits indirectly.

This chapter describes structures and records and tells how to use them.

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 two 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.

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 structure and field names.

6

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 the section, "Defining and Initializing Data," in Chapter 5. Default initial values may be declared individually or with the **DUP** operator.

The names given to fields must be unique within the source file where they are declared. 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 declaration can contain field declarations and comments. Starting with Version 5.0 of the Macro Assembler, conditional-assembly statements are allowed in structure declarations. 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
```

Within the sample structure *student*, the fields *id*, *sname*, and *scores* have the offset values 0, 2, and 24, respectively.

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 by using the **STRUC** and **ENDS** directives.

An *initialvalue* can be given for each field in the structure. Its 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 *initial-values* are given for more than one field, the values must be separated by commas.

If the **DUP** operator (see the section, ''Arrays and Buffers''), in Chapter 5, is used to initialize multiple structure variables, only the angle brackets and initial values, if given, need to be enclosed in parentheses. For example, you can define an array of structure variables as shown here:

```
war          date     365 DUP (<,,1940>)
```

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 the section, ''Declaring Structure Types'':

```
s1         student <>            ; Uses default values of type


s2         student <1467,"White, Robert D.",>
                                 ; Override default values of first two
                                 ;   fields--use default value of third

sarray     student 100 DUP (<>)  ; 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 generate an error. The *endmark* field can be overridden by any byte value.

Using Structure Operands

Like other variables, structure variables can be accessed by name. Fields within structure variables can also be accessed by 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 the section, ''Structure-Field-Name Operator,'' in Chapter 8.

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].month          ; Use as indirect operand
            .
            .
            .
```

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 two 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.

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).

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** *field* [,*field*...]

The *recordname* is the name of the record type to be used when creating the record. The *field* declares the name, width, and initial value for the field.

The syntax for each *field* is shown below:

Syntax

fieldname:*width*[=*expression*]

The *fieldname* 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 *field* 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 eight bits, then the assembler uses two bytes. Otherwise, only one byte is used.

80386 Only

Records can be up to 32 bits in width when the 80386 processor is enabled with **.386**. 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.

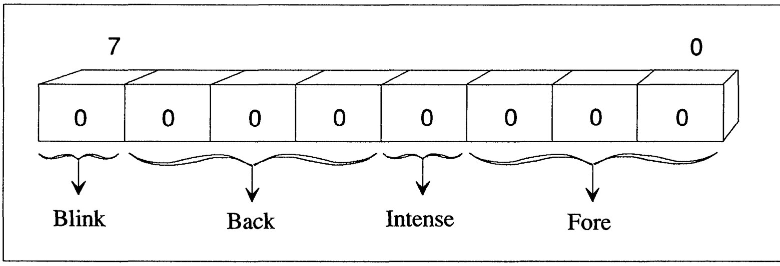
If *expression* is given, it declares the initial value for the field. An error message is generated if an initial value is too large for the width of its field. If the field is at least seven bits wide, you can use an ASCII character for *expression*. The expression must not contain a forward reference 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 are 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:

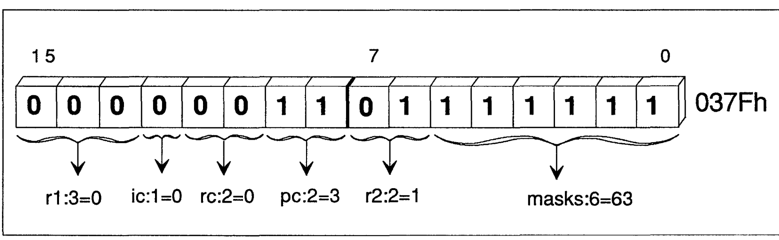


Since no initial values are given, all bits are set to 0. Note that this is only a template maintained by the assembler. No data are created.

Example 2

```
cw          RECORD   r1:3=0,ic:1=0,rc:2=0,pc:2=3,r2:2=1,masks:6=63
```

Example 2 creates a record type *cw* having six fields. Each record declared by using this type occupies 16 bits of memory. The following bit diagram shows the contents of the record type:



Default values are given for each field. They can be used when data is declared for the record.

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.

Records

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 by 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 the section, ''Arrays and Buffers''), in Chapter 5, is used to initialize multiple record variables, only the angle brackets and initial values, if given, need to be enclosed in parentheses. For example, you can define an array of record variables as shown here:

```
xmas          color    50 DUP (<1,2,0,4>)
```

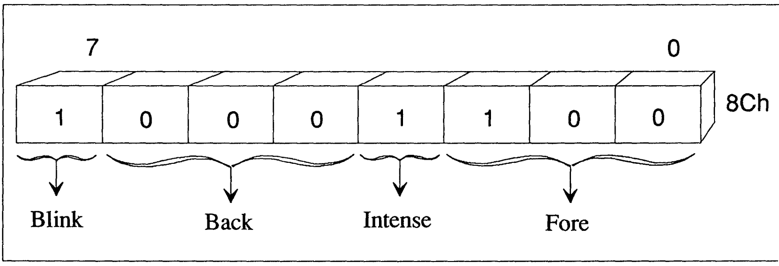
You do not have to initialize all fields in a record. If an initial value is left blank, the assembler automatically uses the default initial value of the field. This is declared by the record type. If there is no default value, each bit in the field is cleared.

The sections, ''Using Record Operands and Record Variables,'' and ''Record Operators,'' illustrate ways to use record data after it has been declared.

Example 1

```
color   RECORD  blink:1,back:3,intense:1,fore:3 ; Record declaration
warning color   <1,0,1,4>                       ; Record definition
```

Example 1 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. The initial values would override the default record values, had any been given in the declaration. The contents of the record variable are:



Example 2

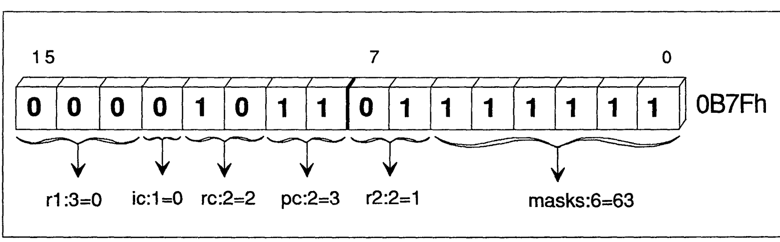
```
color  RECORD  blink:1,back:3,intense:1,fore:3 ; Record declaration
colors color   16 DUP (<>)                      ; Record declaration
```

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

```
cw        RECORD   r1:3=0,ic:1=0,rc:2=0,pc:2=3,r2:2=1,masks:6=63
newcw     cw       <,,2,,,>
```

Example 3 creates a variable named *newcw* with type *cw*. The default values set in the type declaration are used for all fields except the *pc* field. This field is set to 2. The contents of the variable are:



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; a record variable is a value stored in memory. A record operand can be used with the following syntax:

Syntax

recordname <[[*value*][,*value*]...]>

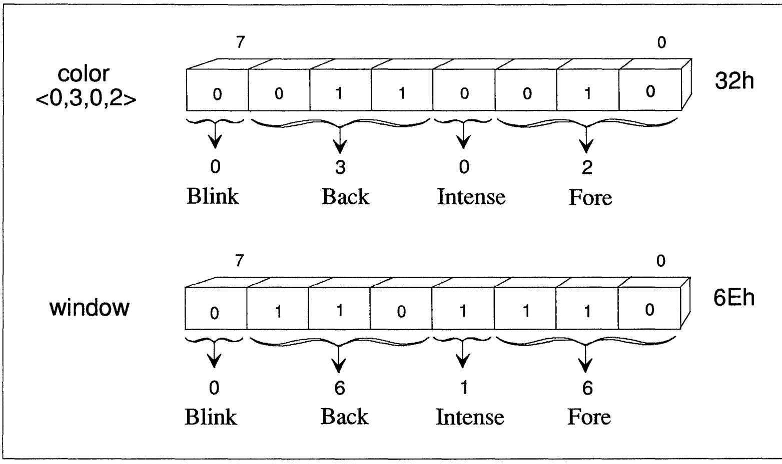
The *recordname* must be the name of a record type declared in the source file. The optional *value* is the value of a field in the record. If more than one *value* is given, each value must be separated by a comma. Values can include expressions or symbols that evaluate to constants. The enclosing angle 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 as follows:



Record Operators

The **WIDTH** and **MASK** operators are used exclusively with records to return constant values representing different aspects of previously declared records.

6

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 intense     ; Toggle      XOR 0000 1000
                                      ; "intense"       ---------
                                      ;                 1000 0001
```

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; the value of the field is the starting position (from the right) of the field.

Example

```
            .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" = 3
wfore       EQU     WIDTH fore    ; "wfore"    = 3   "fore"    = 0
wcolor      EQU     WIDTH color   ; "wcolor"   = 8

prompt      color   <1,5,1,1>

            .CODE
            .
            .
            .
            IF      (WIDTH color) GE 8 ; If color is 16 bit, load
            mov     ax,prompt          ;   into 16-bit register
            ELSE                       ; else
            mov     al,prompt          ;   load into low 8-bit register
            xor     ah,ah              ;   and clear high 8-bit register
            ENDIF
```

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.

6

Record-field operands are often used with the **WIDTH** and **MASK** operators, as described in ‘‘The MASK Operator’’ and ‘‘The WIDTH Operator’’ in this Chapter.

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 in which record fields can be used as operands and in expressions.

6

Chapter 7

Creating Programs from Multiple Modules

Introduction

Most medium and large assembly-language programs 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 you 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. Symbols must be declared global if they must be accessed from modules other than the one in which they are defined.

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—meaning that the symbol is both public and external. The **PUBLIC**, **EXTRN**, and **COMM** directives are used to declare symbols public, external, and communal, respectively.

Note

The term ''local'' 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, depending on the language). 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 the section, ''Using Local Variables,'' in Chapter 16.

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—are resolved to a single address.

Public symbol names are also used by some symbolic debuggers to associate addresses with symbols.

Syntax

PUBLIC *name* [,*name*]...

The *name* must be the name of a variable, label, or numeric equate defined within the current source file. **PUBLIC** declarations can be placed anywhere in the source file. Equate names, if given, can only represent 1- or 2-byte integer or string values. Text macros (or text equates) cannot be declared public.

80386 Only

Equate names on the 80386 processor can represent 1-, 2-, or 4-byte integer values or string values.

7

Note

Although absolute symbols can be declared public, aliases for public symbols should be avoided, since they may decrease the efficiency of the linker. For example, the following statements would increase processing time for the linker:

```
        PUBLIC  lines       ; Declare absolute symbol public
lines   EQU     rows        ; Declare alias for lines
rows    EQU     25          ; Assign value to alias
```

In addition, the symbol made public is *rows*, not *lines*.

Example

```
            PUBLIC  true,status,first,clear
            .MODEL  small
true        EQU     -1
            .DATA
status      DB      1
            .CODE
            .
            .
            .
first       LABEL   FAR
clear       PROC
            .
            .
            .
clear       ENDP
```

Declaring Symbols External

If a symbol undeclared in a module 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, even though the symbol is not in the current module. The assembler assumes that the symbol occurs in another module. However, the symbol must actually exist and must be declared public in some module. Otherwise, the linker generates 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 the section, ''Using Equates'', in Chapter 10).

The **PROC** type represents the default type for a procedure. For programs that use simplified segment directives, the type of an external symbol declared with **PROC** will be near for small or compact model, or far for medium, large, or huge model. The section, ''Defining the Memory Model,'' in Chapter 4, tells you 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 assumes a default segment for the item, based on where the **EXTRN** directive is placed in the source code. Placement of **EXTRN** directives should follow these rules:

- **NEAR** code labels (such as procedures) must be declared in the code segment from which they are accessed.

- **FAR** code labels can be declared anywhere in the source code. It may be convenient to declare them in the code segment from which they are accessed if the label may be **FAR** in one context or **NEAR** in another.

- Data must be declared in the segment in which it occurs. This may require that you define a dummy data segment for the external declaration.

- Absolute symbols can be declared anywhere in the source code.

Example 1

```
            EXTRN   max:ABS,act:FAR     ; Constant or FAR label anywhere
            .MODEL  small
            .STACK  100h
            .DATA
            EXTRN   nvar:BYTE           ; NEAR variable in near data
            .FARDATA
            EXTRN   fvar:WORD           ; FAR variable in far data

            .CODE
            EXTRN   task:PROC           ; PROC or NEAR in near code
start:      mov     ax,@data            ; Load segment
            mov     ds,ax               ;   into DS
            ASSUME  es:@fardata         ; Tell assembler
            mov     ax,@fardata         ; Tell processor that ES
            mov     es,ax               ;   has far data segment
            .
            .
            .
            mov     ah,nvar             ; Load external NEAR variable
            mov     bx,fvar             ; Load external FAR variable
            mov     cx,max              ; Load external constant
            call    task                ; Call procedure (NEAR or FAR)
            jmp     act                 ; Jump to FAR label

            END     start
```

Example 1 shows how each type of external symbol could be declared and used in a small-model program that uses simplified segment directives. Notice the use of the **PROC** type specifier to make the external-procedure memory model independent. The jump and its external declaration are written so that they will be **FAR** regardless of the memory model. Using these techniques, you can change the memory model without breaking code.

Example 2

```
            EXTRN   max:ABS,act:FAR      ; Constant or FAR label anywhere
STACK       SEGMENT PARA STACK 'STACK'
            DB      100h DUP (?)
STACK       ENDS
_DATA       SEGMENT WORD PUBLIC 'DATA'
            EXTRN   nvar:BYTE            ; NEAR variable in near data
_DATA       ENDS
FAR_DATA    SEGMENT PARA 'FAR_DATA'
            EXTRN   fvar:WORD            ; FAR variable in far data
FAR_DATA    ENDS

DGROUP      GROUP   _DATA,STACK
_TEXT       SEGMENT BYTE PUBLIC 'CODE'
            EXTRN   task:NEAR            ; NEAR procedure in near code
            ASSUME  cs:_TEXT,ds:DGROUP,
                    ss:DGROUP

start:      mov     ax,DGROUP            ; Load segment
            mov     ds,ax                ;   into DS
            ASSUME  es:FAR_DATA          ; Tell assembler
            mov     ax,FAR_DATA          ; Tell processor that ES
            mov     es,ax                ;   has far data segment
            .
            .
            .
            mov     ah,nvar              ; Load external NEAR variable
            mov     bx,fvar              ; Load external FAR variable
            mov     cx,max               ; Load external constant

            call    task                 ; Call NEAR procedure

            jmp     act                  ; Jump to FAR label

_TEXT       ENDS
            END     start
```

7

Example 2 shows a fragment similar to the one in Example 1, but with full segment definitions. Notice that the types of code labels must be declared specifically. If you wanted to change the memory model, you would have to specifically change each external declaration and each call or jump.

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. Execution then returns to the address after the call in the *hello* module.

Here is the *hello* module:

```
          .286
          TITLE    hello

          .MODEL   SMALL

          .DATA
public    message, lmessage
message   DB       "Hello, world", 10
lmessage  EQU      - message

         .CODE

EXTRN    display:PROC            ; declare in near code segment
EXTRN    _exit:PROC              ; system call provided in system
                                 ; library, libc.a

PUBLIC   _main
_main:   call     display        ; call other module

         call     _exit          ; xenix system call

         END
```

7

Next, the *display* module:

```
        .286
        TITLE    display

        .MODEL   SMALL
        .DATA
EXTRN    lmessage:ABS             ; declare anywhere
EXTRN    message:BYTE             ; declare in near data segment

        .CODE

EXTRN    _write:PROC              ; system call provided in
                                  ; system library, libc.a
PUBLIC   display
display  PROC
         push     lmessage
         push     offset message
         push     0
         call     _write          ; xenix system call
         add      sp, 6
         ret
display           ENDP
         END
```

The sample program is a variation of the *hello.s* program used in the example in Chapter 1, "Getting Started," except that it uses an external procedure to display to the standard output. Notice that all symbols defined in one module but used in another are declared **PUBLIC** in the defining module and declared **EXTRN** in the using module.

For instance, *message* and *lmessage* are declared **PUBLIC** in *hello* and declared **EXTRN** in *display*. The procedure *display* is declared **EXTRN** in *hello* and **PUBLIC** in *display*.

7

To create an executable file for these modules, assemble each module separately, as in the following command lines:

```
masm hello.s
masm display.s
```

Then link the two modules:

```
xld display.o hello.o
```

The result is the executable file *hello*.

For each source module, **masm** writes a module name to the object file. The module name is used by some debuggers and by the linker when it displays error messages. Starting with Version 5.0, the module name is always the base name of the source module file. With previous versions, the module name could be specified with the **NAME** or **TITLE** directive.

For compatibility, **masm** recognizes the **NAME** directive. However, **NAME** has no effect. Arguments to the directive are ignored.

Declaring Symbols Communal

Communal variables are uninitialized variables that are both public and external. They are often declared in include files.

If a variable must be used by several assembly routines, you can declare the variable communal in an include file, and then include the file in each of the assembly routines. Although the variable is declared in each source module, it exists at only one address. Using a communal variable in an include file and including it in several source modules is an alternative to defining the variable and declaring it public in one source module and then declaring it external in other modules.

If a variable is declared communal in one module and public in another, the public declaration takes precedence and the communal declaration has the same effect as an external declaration.

Syntax

> **COMM** *definition*[,*definition*]...

Each *definition* has the following syntax:

> [**NEAR** | **FAR**] *label*:*size*[:*count*]

A communal variable can be **NEAR** or **FAR**. If neither is specified, the type will be that of the default memory model. If you use simplified segment directives, the default type is **NEAR** for small and medium models, or **FAR** for compact, large, and huge models. If you use full segment definitions, the default type is **NEAR**.

7

The *label* is the name of the variable. The *size* can be **BYTE**, **WORD**, **DWORD**, **QWORD**, or **TBYTE**. The *count* is the number of elements. If no *count* is given, one element is assumed. Multiple variables can be defined with one **COMM** statement by separating each variable with a comma.

Note

C variables declared outside functions (except static variables) are communal unless explicitly initialized; they are the same as assembly-language communal variables. If you are writing assembly-language modules for C, you can declare the same communal variables in C include files and in **masm** include files.

Because **masm** cannot tell whether a communal variable has been used in another module, allocation of communal variables is handled by the linker. As a result, communal variables have the following limitations that other variables declared in assembly language do not have:

- Communal variables cannot be initialized. Under Part 1, "Using Assembler Programs, initial values are guaranteed to be 0. The variables can be used for data that are not given a value until run time, such as file buffers.

- Communal variables are not guaranteed to be allocated in the sequence in which they are declared. Assembly-language techniques that depend on the sequence and position in which data is defined should not be used with communal variables. For example, the following statements do not work:

```
        COMM    buffer:WORD:128
lbuffer EQU     $ - buffer ; "lbuffer" won't have desired value

bbuffer LABEL   BYTE       ; "bbuffer" won't have desired address
        COMM    wbuffer:WORD:128
```

- Placement of communal declarations follows the same rules as external declarations. They must be declared inside a data segment. Examples of near and far communal variables are as follows:

```
COMM    NEAR nbuffer:BYTE:30
COMM    FAR fbuffer:WORD:40
```

- Communal variables are allocated in segments that are part of the Microsoft segment conventions. You cannot override the default to place communal variables in other segments.

  Near communal variables are placed in a segment called **c_common**, which is part of **DGROUP**. This group is created and initialized automatically if you use simplified segment directives.

If you use full segment directives, you must create a group called **DGROUP** and use the **ASSUME** directive to associate it with the **DS** register.

Far communal variables are placed in a segment called **FAR_BSS**. This segment 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 (with word size) is called *fcomvar*, its segment can be initialized with the following lines:

```
ASSUME  ds:SEG comvar       ; Tell the assembler
mov     ax,SEG comvar       ; Tell the processor
mov     ds,ax
mov     bx,comvar           ; Use the variable
```

Example 1

```
IF      @DataSize
.FARDATA
ELSE
.DATA
ENDIF
COMM    var:WORD, buffer:BYTE:10
```

Example 1 creates two communal variables. The first is a word variable called *var*. The second is a 10-byte array called *buffer*. Both have the default size associated with the memory model of the program in which they are used.

Example 2

```
        EXTRN   _read:PROC

        .DATA
        COMM    temp:BYTE:128

asciiz  MACRO   address          ; name of address for string
        LOCAL   ok
        push    128              ; maximum length
        push    OFFSET temp
        push    0                ; standard input
        call    _read            ; xenix system call
        add     sp, 6
        or      ax, ax
        jge     ok
        xor     ax, ax
ok:
        mov     bx, ax           ; length of string
        mov     temp[bx], 0      ; overwrite CR with NULL
address EQU     OFFSET temp
        ENDM
```

Example 2 shows an include file that declares a buffer for temporary data. The buffer is then used in a macro in the same include file.

The following is an example of how the macro could be used in a source file:

Example

```
            .286
            .MODEL          SMALL

            INCLUDE         communal.inc

            .DATA
message     DB              "Enter file name: ", 0
lmessage    EQU             - message

            .CODE
            EXTRN           _open:PROC
            EXTRN           _write:PROC

PUBLIC      _main
_main       PROC
            push            bp
            mov             bp, sp

            push            lmessage
            push            OFFSET message
            push            1
            call            _write    ; write(1, message, lmessage)
            add             sp, 6     ; clear stack

            asciiz          place     ; get file name and
                                      ;   return address as "place"

            push            0         ; see <sys/fcntl.h>
            push            place
            call            _open     ; open(place, 0)
            add             sp, 4     ; clear stack

            leave
            ret
_main       ENDP
            end
```

Note that once the macro is written, the user does not need to know the name of the temporary buffer or how it is used in the macro.

Chapter 8

Using Operands and Expressions

Introduction

Operands are the arguments that define values to be acted on by instructions or directives. Operands can be constants, variables, expressions, or keywords, 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 you are writing source code.

This chapter discusses operands, expressions, and operators as they are evaluated at assembly time. See Chapter 13, "Using Addressing Modes," for a discussion of the addressing modes that can be used to calculate operand values at run time. This chapter also discusses the location-counter operand, forward references, and strong typing of operands.

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 13, "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. This is because the contents may change at run time and are therefore not 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.

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.

It is important to understand the difference between operators and instructions. Operators handle calculations of constant values that are known at assembly time. Instructions handle calculations of values that may not be known until run time. For example, the addition operator (+) handles assembly-time addition, while the **ADD** and **ADC** instructions handle run-time addition.

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 ''Arrays and Buffers'', in Chapter 5) the record operators (section ''Using Record-Field Operands'', in Chapter 6) and the macro operators (section ''Using Macro Operators'', in Chapter 10).

Calculation Operators

Common arithmetic operators are provided by **masm**, as well as several other operators for adding, shifting, or doing bit manipulations. The sections below describe operators that can be used for doing numeric calculations.

Note

> Constant values used with calculation operators are extended to 33 bits before the calculations are done. This rule applies regardless of the processor used. Exceptions are noted to this rule.

Arithmetic Operators

A variety of arithmetic operators for common mathematical operations are recognized. Table 8.1 lists the arithmetic operators.

Table 8.1

Arithmetic Operators

Operator	Syntax	Meaning
+	%+<expression>	Positive (unary)
-	-<expression>	Negative (unary)
*	<expression1>*<expression2>	Multiplication
/	<expression1>/<expression2>	Integer division
MOD	<expression1>MOD<expression2>	Remainder (modulus)
+	<expression1>+<expression2>	Addition
-	<expression1>-<expression2>	Subtraction

For all arithmetic operators except the addition operator (+) and the subtraction operator (-), the expressions operated on must be integer constants.

The addition and subtraction 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 subtraction 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 will be 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 the section, "Operator Precedence."

Example 1

```
intgr        =       14  *  3              ; = 42
intgr        =       intgr /  4            ; 42 / 4 = 10
intgr        =       intgr  MOD  4         ; 10 mod 4 = 2
intgr        =       intgr +  4            ; 2 + 4 = 6
intgr        =       intgr -  3            ; 6 - 3 = 3
intgr        =       -intgr - 8            ; -3 - 8 = -11
intgr        =       -intgr - intgr        ; 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 = 0FBh
const        EQU     b - a                 ; const = 101h - 100h = 1
```

Example 2 illustrates arithmetic operators used in memory expressions. Note that *mem1* and *mem2* are memory addresses relative to the segment they are defined in, while *const* is equal to the constant 1.

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. For more information, see the section, "Structures," in Chapter 6.

8

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 today      ; Load structure variable address
            inc     [bx].year            ; Use in indirect memory operand
```

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. The section, "Indirect Memory Operands," in Chapter 13, explains the use 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        ; Load into 8th element of string
mov       ax,table[bx][di]
mov       ax,table[bx+di]
mov       ax,[table+bx+di]
mov       ax,[table][bx][di]
```

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.

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 (32 on the 80386 processor), the result is 0.

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. The assembler can tell the difference between instructions and operands from context.

Examples

```
mov     ax,01110111b SHL 3 ; Load 01110111000b
mov     ah,01110111b SHR 3 ; Load 01110b
```

Bitwise Logical Operators

The bitwise operators perform logical operations on each bit of an expression. The expressions must resolve to constant values. Table 8.2 lists the logical operators and their meanings.

Table 8.2

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

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, immediate, or memory values at run time. The assembler can tell the difference between instructions and operands from context.

Note

Although calculations on expressions using the **AND**, **OR**, and **XOR** operators are done using 33-bit numbers, the results are truncated to 32 bits. Calculations on expressions using the **NOT** operator are truncated to 16 bits (except on the 80386).

Examples

```
mov     ax,NOT 11110000b              ; Load 1111111100001111b
mov     ah,NOT 11110000b              ; Load 00001111b
mov     ah,01010101b AND 11110000b    ; Load 01010000b
mov     ah,01010101b OR  11110000b    ; Load 11110101b
mov     ah,01010101b XOR 11110000b    ; Load 10100101b
```

Relational Operators

The relational operators compare two expressions 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. Relational operators are typically used with conditional directives. Table 8.3 lists the operators and the values they return if the specified condition is satisfied.

Table 8.3

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

8

Note

The **EQ** and **NE** operators treat their arguments as 32-bit numbers. Numbers specified with the 32nd bit set are considered negative. For example, the expression *-1 EQ OFFFFFFFFh* is true, but the expression *-1 NE OFFFFFFFFh* is false.

The **LT,LE,GT**, and operators treat their arguments as 33-bit numbers, in which the 33rd bit specifies the sign. For example, *OFFFFFFFFh* is 4,294,967,295, not -1. The expression *1 GT -1* is true, but the expression *1 GT OFFFFFFFFh* is false.

Examples

```
mov     ax,4 EQ 3  ; Load false ( 0)
mov     ax,4 NE 3  ; Load true  (-1)
mov     ax,4 LT 3  ; Load false ( 0)
mov     ax,4 LE 3  ; Load false ( 0)
mov     ax,4 GT 3  ; Load true  (-1)
mov     ax,4 GE 3  ; Load true  (-1)
```

Segment-Override Operator

The segment-override operator (**:**) forces the address of a variable or label to be computed relative to a specific segment.

Syntax

segment:expression

The *segment* can be specified in several ways. It can be one of the segment registers: **CS**, **DS**, **SS**, or **ES** (or **FS** or **GS** on the 80386). It can also be a segment or group name. In this case, the name must have been previously defined with a **SEGMENT** or **GROUP** directive and assigned to a segment register with an **ASSUME** directive. The expression can be a constant, expression, or a **SEG** expression. For more information on the **SEG** operator, see the section, "SEG Operator," in this chapter.

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]* generates an error.

Examples

```
mov     ax,ss:[bx+4]         ; Override default assume (DS)
mov     al,es:082h           ; Load from ES

ASSUME  ds:FAR_DATA          ; Tell the assembler and
mov     bx,FAR_DATA:count    ;   load from a far segment
```

As shown in the last two statements, a segment override with a segment name is not enough if no segment register is assumed for the segment name. You must use the **ASSUME** statement to assign a segment register, as explained in the section, ''Associating Segments with Registers,'' in Chapter 4.

Type Operators

This section describes the assembler operators that specify or analyze the types of memory operands and other expressions.

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**, **QWORD**, or **TBYTE** for memory operands. It can be **NEAR**, **FAR**, or **PROC** for labels.

The **PTR** operator is typically used with forward references to define explicitly what size or distance a reference has. If it is not used, the assembler assumes a default size or distance for the reference. See the section, ''Using Forward References,'' 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. The **PTR** operator is required for **FAR** calls and jumps to forward-referenced labels.

Example

```
            .DATA
stuff       DD      ?
buffer      DB      20 DUP (?)

            .CODE
            .
            .
            .
            call    FAR PTR task            ; Call a far procedure
            jmp     FAR PTR place           ; Jump 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
```

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. For information on using the **SHORT** operator with jump instructions, see the section, "Forward References to Labels."

Example

```
               jmp     again                 ; Jump 128 bytes or more
               .
               .
               .
               jmp     SHORT again           ; Jump less than 128 bytes
               .
               .
               .
again:
```

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**, **QWORD**, 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:
check4       PROC    NEAR
check4       ENDP
```

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 eight bits of *expression*; the **LOW** operator returns the low-order eight bits. The *expression* must evaluate to a constant. You cannot use the **HIGH** and **LOW** operators on the contents of a memory operand since the contents may change at run time.

Examples

```
stuff       EQU     0ABCDh
            mov     ah,HIGH stuff       ; Load 0ABh
            mov     al,LOW stuff        ; Load 0CDh
```

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.

Example

```
         .DATA
var      DB      ?
         .CODE
         .
         .
         .
         mov     ax,SEG var       ; Get address of segment
                                  ; where variable is declared
         ASSUME  ds:SEG var       ; Assume segment of variable
```

OFFSET Operator

The **OFFSET** operator returns the offset address of an expression.

Syntax

OFFSET *expression*

The *expression* can be any label, variable, or other direct memory operand. Constant expressions return meaningless values. The value returned by the **OFFSET** operand is an immediate (constant) operand.

If simplified segment directives are given, the returned value varies. If the item is declared in a near data segment, the returned value is the number of bytes between the item and the beginning of its group (normally **DGROUP**). If the item is declared in a far segment, the returned value is the number of bytes between the item and the beginning of the 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 in the first segment in *DGROUP*.

Examples

```
            .DATA
string      DB      ''This is it.''
            .CODE
            .
            .
            .
            mov     dx,OFFSET string    ; Load offset of variable
```

.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 8.4. Only bits 0, 1, 5, and 7 are affected. Other bits are always undefined.

Table 8.4

.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 in which different kinds of arguments may need to be handled differently.

Example

```
            EXTRN   -printf:PROC
display     MACRO   string
            IFE     ((.TYPE string) AND 02h)
            IF2
            %OUT    Argument must be a variable
            ENDIF
            ENDIF
            push    OFFSET string
            call    _printf
            add     sp,2
            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 message is generated.

TYPE Operator

The **TYPE** operator returns a number that represents 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 for **NEAR** labels and 0FFFEh for **FAR** labels. If *expression* is a constant, the operator returns 0.

The returned value can be used to specify the type for a **PTR** operator.

Example

```
         .DATA
var      DW   ?
array    DD   10 DUP (?)
str      DB   "This is a test"
         .CODE
         .
         .
         .
         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   ; Jump is near or far,
         .                           ; depending on memory model
         .
         .
room     LABEL   PROC
```

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 is 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 is returned. If the variable was not declared with the **DUP** operator, the value returned is always 1.

Example

```
array     DD     100 DUP(0FFFFFFh)
table     DW     100 DUP(1,10 DUP(?))
string    DB     'This is a string'
var       DT     ?
larray    EQU    LENGTH array       ; 100 - number 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
```

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 is 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 - bytes in variable
           .
           .
           .
           mov    cx,SIZE array     ; Load number of bytes
again:     .                        ; Perform some operation on
           .                        ;   each byte
           .
           loop   again
```

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 8.5. Operators on the same line have equal precedence.

Table 8.5

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
```

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 change it 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     $+5         ; If ax >= bx, continue
            jmp     longjump    ;   else if ax < bx, go to "longjump"
            .                   ; This is "$+5"
            .
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, and the second part shows how to code the same jump when the label is more than 128 bytes away.

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 and 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 more slowly than source code with no forward references.

In addition, the assembler may make incorrect assumptions that it cannot correct, or corrects at a cost in program efficiency.

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). It is assumed that *target* is a near label, and **masm** assembles the number of bytes necessary to specify a near label: one byte for the instruction and two bytes for the operand.

If on the second pass the assembler learns that *target* is a short label, it will need only two bytes: one for the instruction and one for the operand. However, it will not be able to change its previous assembly and the three-byte version of the assembly will stand. If the assembler learns that *target* is a far label, it will need five 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 makes the code smaller and slightly faster. If the assembler has to use the three-byte form when the two-byte form would be acceptable, 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 the section, ''Setting the Warning Level,'' in Chapter 2.) 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** generates an error message. You can either remove the **SHORT** operator, or try to reorganize your program to reduce the distance.

If a far jump to a forward-referenced label is required, you must 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:                                 ; In different segment
```

If the type of a label has been established earlier in the source code with an **EXTRN** directive, the type does not need to be specified in the jump statement.

80386 Only

If the 80386 processor is enabled, jumps with forward references have different limitations. One difference is that conditional jumps can be either short or near. With previous processors, all conditional jumps were short. For 32-bit segments, the number of bytes generated for near and far jumps is greater in order to handle the larger addresses in the operand.

Example 1

```
.MODEL  large               ; Model comes first, so use
.386                        ;   16-bit segments
.CODE
.
.
.
jmp     SHORT place         ; Short unconditional jump - 2 bytes
jne     SHORT place         ; Short conditional jump - 2 bytes
jmp     place               ; Near unconditional jump - 3 bytes
jne     place               ; Near conditional jump - 4 bytes
jmp     FAR PTR place       ; Far unconditional jump - 5 bytes
```

Example 2

```
.386                        ; .386 comes first, so use
.MODEL  large               ;   32-bit segments
.CODE
.
.
.
jmp     SHORT place         ; Short unconditional jump - 2 bytes
jne     SHORT place         ; Short conditional jump - 2 bytes
jmp     place               ; Near unconditional jump - 5 bytes
jne     place               ; Near conditional jump - 6 bytes
jmp     FAR PTR place       ; Far unconditional jump - 7 bytes
```

Forward References to Variables

When **masm** encounters code referencing variables that have not yet been defined in Pass 1, it makes assumptions about the segment where the variable will be defined. If on Pass 2 the assumptions turn out to be wrong, an error will occur.

These problems usually occur with complex segment structures that do not follow the Microsoft segment conventions. The problems never appear if simplified segment directives are used.

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 been defined earlier in the source code, you must use the segment-override operator to specify the segment.

The situation is different if neither the variable nor the segment in which it is defined has been defined earlier in the source code. In this case, you must assign the segment to a group earlier in the source code, then **masm** will know about the existence of the segment even though it has not yet been defined.

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 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

8

```
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 that 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 in these examples. You should not ignore warnings about type mismatches unless you are sure you understand how your code will be assembled.

Note

Some assemblers 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 type 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 9

Assembling Conditionally

Introduction

The Macro Assembler provides two types of conditional directives, conditional-assembly and conditional-error 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 assembly 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 10, ''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 the section, ''Using Macro Operators,'' in Chapter 10.

Using Conditional-Assembly Directives

The conditional-assembly directives include the following:

IF	**IFDEF**	**IFNB**
IF1	**IFDIF**	**IFNDEF**
IF2	**IFE**	**ENDIF**
IFB	**IFIDN**	**ELSE**

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*
statements
[**ELSE**
statements]
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 are assembled. The statements that follow the **ELSE** directive are assembled only if the **IF** statement 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. A nested **ELSE** directive always belongs to the nearest preceding **IF** statement that does not have its own **ELSE**.

9

Testing Expressions with IF and IFE

The **IF** and **IFE** directives test the value of an expression and grant assembly based on the result.

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      debug GT 20
push    debug
call    adebug
ELSE
call    bdebug
ENDIF
```

In this example, a different debug routine will be called, depending on the value of *debug*.

Testing the Pass with IF1 and IF2

The **IF1** and **IF2** directives test the current assembly pass and grant assembly only on the pass specified by the directive. Multiple passes of the assembler are discussed in the section, "Reading a Pass 1 Listing," in Chapter 2.

Syntax

IF1
IF2

The **IF1** directive grants assembly only on Pass 1. **IF2** grants assembly only on Pass 2. The directives take no arguments.

Macros usually 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                   ; Define on first pass only
dostuff   MACRO     argument
          .
          .
          .
          ENDM
          ENDIF
```

Testing Symbol Definition with IFDEF and IFNDEF

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

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

In this example, *buff* is allocated only if *buffer* has been previously defined.

One way to use this conditional block is to leave *buffer* undefined in the source file and define it if needed by using the **-D***symbol* option (see the section, ''Defining Assembler Symbols'', in Chapter 2) when you start **masm**. For example, if the conditional block is in *test.s*, you could start the assembler with the following command line:

```
masm -Dbuffer=1024 test.s
```

The command line would define the symbol *buffer*; as a result, the conditional assemble would allocate *buff*. However, if you didn't need *buff*, you could use the following command line:

```
masm test.s
```

Verifying Macro Parameters with IFB and IFNB

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 whether 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. Angle brackets (< >) are required.

Example

```
Write   MACRO   buffer,bytes,descriptor
        IFNB    <descriptor>
        mov     bx,descriptor   ; (1=standard output, 2=standard error)
        ELSE
        mov     bx,1            ; default standard output
        ENDIF
        push    bytes           ; number of bytes to write
        push    OFFSET buffer   ; address of buffer to write to
        push    descriptor      ; stdout
        call    _write          ; xenix call
        add     sp,6            ; clear stack
        ENDM
```

In this example, a default value is used if no value is specified for the third macro argument.

9

Comparing Macro Arguments with IFIDN and IFDIF

The **IFIDN** and **IFDIF** directives compare two macro arguments and grant assembly based on the result.

Syntax

IFIDN[I] *<argument1>,<argument2>*
IFDIF[I] *<argument1>,<argument2>*

These directives are always used inside macros, and they always test whether real arguments passed for two specified 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 names, numbers, or expressions. They must be enclosed in angle brackets and separated by a comma.

The optional **I** at the end of the directive name specifies that the directive is case insensitive. Arguments that are spelled the same will be evaluated the same, regardless of case. This is a new feature starting with Version 5.0. If the **I** is not given, the directive is case sensitive.

Example

```
divide8     MACRO    numerator,denominator
            IFDIFI   <numerator>,<al>    ;; If numerator isn't AL
            mov      al,numerator        ;;   make it AL
            ENDIF
            xor      ah,ah
            div      denominator
            ENDM
```

In this example, a macro uses the **IFDIFI** directive to check one of the arguments and take a different action, depending on the text of the string. The sample macro could be enhanced further by checking for other values that would require adjustment (such as a denominator passed in **AL** or passed in **AH**).

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 error messages they produce are listed in Table 9.1.

Table 9.1

Conditional-Error Directives

Directive	#	Message
.ERR1	87	`Forced error - pass1`
.ERR2	88	`Forced error - pass2`
.ERR	89	`Forced error`
.ERRE	90	`Forced error - expression true (0)`
.ERRNZ	91	`Forced error - expression false (not 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%[%**I**%]%	96	`Forced error - strings identical`
.ERRDIF%[%**I**%]%	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.

Generating Unconditional Errors with .ERR, .ERR1, and .ERR2

The **.ERR**, **.ERR1**, and **.ERR2** directives force an error where the directives occur in the source file. The error is generated unconditionally when the directive is encountered, but the directives 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. The **.ERR1** and **.ERR2** directives force the error only on their respective passes. The **.ERR1** directive appears only on standard output or in the listing file if you use the **-d** option to request a Pass 1 listing (as described in the section, ''Creating a Pass 1 Listing'', in Chapter 2).

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
          %OUT dos or xenix must be defined
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. You could use **.ERR1** or **.ERR2** to check if you want the error to be generated only on the corresponding pass.

Testing Expressions with .ERRE or .ERRNZ

9

The **.ERRE** and **.ERRNZ** directives test the value of an expression and conditionally generate an error based on the result.

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.

Verifying Symbol Definition with .ERRDEF and .ERRNDEF

The **.ERRDEF** and **.ERRNDEF** directives test whether or not a symbol is defined and conditionally generate an error based 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.

9

Example

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

In this example, the **.ERRNDEF** directive at the beginning of the conditional block makes sure that a symbol being tested in the block actually exists.

Testing for Macro Parameters with .ERRB and .ERRNB

The **.ERRB** and **.ERRNB** directives test whether 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 whether 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. 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 **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.**

Comparing Macro Arguments with .ERRIDN and .ERRDIF

The **.ERRIDN** and **.ERRDIF** directives compare two macro arguments and conditionally generate an error based on the result.

Syntax

> **.ERRIDN**[**I**] *<argument1>,<argument2>*
> **.ERRDIF**[**I**] *<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** directive generates an error if the arguments are different. The arguments can be names, numbers, or expressions. They must be enclosed in angle brackets and separated by a comma.

The optional **I** at the end of the directive name specifies that the directive is case insensitive. Arguments that are spelled the same will be evaluated the same regardless of case. This is a new feature starting with Version 5.0. If the **I** is not given, the directive is case sensitive.

Example

```
addem       MACRO     ad1,ad2,sum
            .ERRIDNI <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 **.ERRIDNI** directive is used to protect against passing the **AX** register as the second parameter, since this would cause the macro to fail.

Chapter 10

Using Equates, Macros, and Repeat Blocks

Introduction

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 a 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. Assembly may be slightly slower for a program that uses macros and equates extensively than for the same program written without them. However, the program without macros and equates usually takes longer to write and is more difficult to maintain.

Using Equates

The equate directives enable you to use symbols that represent numeric or string constants. There are three kinds of equates that **masm** recognizes:

1. Redefinable numeric equates
2. Nonredefinable numeric equates
3. String equates (also called text macros)

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 constant expression, a one- or two-character string constant (four-character on the 80386 processor), or an expression that evaluates to an address. The *name* must be either a unique name or a name previously defined by using the equal-sign (=) directive.

Note

Redefinable equates must be assigned numeric values. String constants longer than two characters cannot be used.

Example

```
counter     =       0               ; Initialize counter
array       LABEL   BYTE            ; Label array of increasing numbers
            REPT    100             ; Repeat 100 times
            DB      counter         ; Initialize number
counter     =       counter + 1     ; Increment counter
            ENDM
```

This example redefines equates inside a repeat block to declare an array initialized to increasing values from 0 to 100. The equal-sign directive is used to increment the *counter* symbol for each loop. See the section, ''Defining Repeat Blocks,'' for more information on repeat blocks.

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 numeric 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*. Once a numeric equate has been defined with the **EQU** directive, it cannot be redefined. Attempting to do so generates an error.

Note

String constants can also be defined with the **EQU** directive, but the syntax is different, as described in the section, ''String Equates.''

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
line        EQU     row             ; Alias for "row"

            .DATA
buffer      DW      screenful

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

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 equal-sign (=) directive.

An alias is a special kind of string equate. It is a symbol that is equated to another symbol or keyword.

Note

The use of angle brackets to force string evaluation is a new feature of Version 5.0 of the Macro Assembler. Previous versions 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 alone that evaluate correctly, 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.

Example

```
; String equate definitions
pi          EQU     <3.1415>            ; String constant "3.1415"
prompt      EQU     <'Type Name: '>     ; String constant "'Type Name: '"
WPT         EQU     <WORD PTR>          ; String constant for "WORD PTR"
parm1       EQU     <[bp+4]>            ; String constant for "[bp+4]"

; Use of string equates
            .DATA
message     DB      prompt              ; Allocate string "Type Name: "
pie         DQ      pi                  ; Allocate real number 3.1415

            .CODE
            .
            .
            .
            inc     WPT parm1           ; Increment word value of
                                        ;   argument passed on stack
```

Using Macros

Macros enable you to assign a symbolic name to a block of source statements, and then to 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 associated with a macro name, it replaces that text with the text of the statements in the macro definition. Similarly, the text of 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 the macro. Macros and equates are often kept in a separate file and made available to the program through an **INCLUDE** directive (see the section, ''Using Include Files'') at the start of the source code.

Note

> Since most macros only need to be expanded once, you can increase efficiency by processing them only during a single pass of the assembler. You can do this by enclosing the macros (or an **INCLUDE** statement that calls them) in a conditional block using the **IF1** directive. Any macros that use the **EXTRN** or **PUBLIC** statements should be processed on Pass 1 rather than Pass 2 to increase linker efficiency.

Often a task can be done by using either a macro or procedure. For example, the *addup* procedure shown in the section, ''Passing Arguments on the Stack,'' in Chapter 16, does the same thing as the *addup* macro in the section, ''Defining Macros.'' 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 section below tells how to define and call macros. Repeat blocks, a special form of macro for doing repeated operations, are discussed separately in the section, ''Defining Repeat Blocks.''

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 of the macro block.

Syntax

> *name* **MACRO** [*parameter* [,*parameter*]...]
> *statements*
> **ENDM**

The *name* must be unique and a valid symbol name. It can be used later in the source file to invoke the macro.

The *parameters* (sometimes called 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 parameter, you must separate them with commas, spaces, or tabs. Commas can always be used as separators; spaces and tabs may cause ambiguity if the arguments are expressions.

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 statement 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 the section, ''Using Recursive, Nested, and Redefined Macros.''

The statements in a macro are assembled 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 the sections, ''Using Local Symbols'' and ''Exiting from a Macro.'' Macro operators can also be used in macro definitions, as described in the section, ''Using Macro Operators.''

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.

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.

When assembling macros, **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 then take appropriate action to adjust to different kinds of arguments. See Chapter 9, ''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 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 the section, ''Controlling Listing of Macros''), in Chapter 11.

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 7, ''Creating Programs from Multiple Modules,'' or to variables that are defined to be local to a procedure, as described in the section, ''Using Local Variables,'' in Chapter 16. ''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, *localname* can be used in any statement within the macro definition.

A new actual name for *localname* is created 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 a macro; 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, an error will be generated.

Example

```
power       MACRO   factor,exponent     ;; Use for unsigned only
            LOCAL   again,gotzero       ;; Declare symbols for macro
            xor     dx,dx               ;; Clear DX
            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 local names *again* and *gotzero* as labels to be used within the *power* macro.

These local 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.

Exiting from a Macro

Normally, **masm** processes all the statements in a macro definition and then continues with the next statement after the macro call. However, you can use the **EXITM** directive to tell the assembler to terminate macro expansion before all the statements in the macro have been assembled.

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** 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

```
allocate     MACRO    times         ; Macro definition
x            =        0
             REPT     times         ;; Repeat up to 256 times
             IF       x GT 0FFh     ;; Is 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 allocates a variable amount of data, but no more than 255 bytes. The macro contains an **IF** directive that checks the expression *x - 0FFh*. When the value of this expression is true (x-255 = 0), the **EXITM** directive is processed and expansion of the macro stops.

Defining Repeat Blocks

Repeat blocks are a special form of macro that allows you to create blocks of repeated statements. They differ from macros in that they are not named, and thus cannot be called. However, like macros, they can have 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.

Repeat blocks are processed at assembly time and should not be confused with the **REP** instruction, which causes string instructions to be repeated at run time, as explained in Chapter 17, ''Processing Strings.''

Three different kinds of repeat blocks can be defined by using the **REPT**, **IRP**, and **IRPC** directives. The difference between them is in how the number of repetitions is specified.

The REPT Directive

The **REPT** directive is used to create repeat blocks in which the number of repetitions is specified with a numeric argument.

Syntax

> **REPT** *expression*
> *statements*
> **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

```
alphabet    LABEL   BYTE
x           =       0          ;; Initialize
            REPT    26         ;; Specify 26 repetitions
            DB      'A' + x    ;; Allocate ASCII code for letter
x           =       x + 1      ;; Increment
            ENDM
```

This example repeats the equal-sign (=) and **DB** directives to initialize ASCII values for each uppercase letter of the alphabet.

The IRP Directive

The **IRP** directive is used to create repeat blocks in which the number of repetitions, as well as parameters for each repetition, are specified in a list of arguments.

Syntax

> **IRP** *parameter*,<*argument*[,*argument*]...>
> *statements*
> **ENDM**

The assembler *statements* inside the block are repeated once for each *argument* in the list enclosed by angle brackets (< >). The *parameter* is a name for a placeholder to be replaced by the current argument. Each argument can be text, such as a symbol, string, or numeric constant. Any number of arguments can be given. If multiple arguments are given, they must be separated by commas. The angle brackets (< >) around the argument list are required. The *parameter* 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 *parameter* 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

```
numbers         LABEL   BYTE
                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, allocating 10 bytes for each number in the list. The resulting statements create 100 bytes of data, starting with 10 zeros, followed by 10 ones, and so on.

The IRPC Directive

The **IRPC** directive is used to create repeat blocks in which the number of repetitions, as well as arguments for each repetition, is specified in a string.

Syntax

> **IRPC** *parameter,string*
> *statements*
> **ENDM**

The assembler *statements* inside the block are repeated as many times as there are characters in *string*. The *parameter* 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 *parameter* 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 *parameter* in these statements.

Example 1

```
ten             LABEL   BYTE
                IRPC    x,0123456789
                DB      x
                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 *letter* will be treated as an argument rather than a string. See the section, "Substitute Operator," for more information.

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.

Substitute Operator

The substitute operator (**&**) forces **masm** to replace a parameter with its corresponding actual argument value.

Syntax

&parameter

The substitute 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

```
            PUBLIC  err5
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 dummy parameter is replaced as many times as there are numbers 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
```

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 even if it contains commas, spaces, or tabs. The literal-text 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.

One set of angle brackets is removed by **masm** 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

```
        work    1,2,3,4,5           ; Passes five parameters
                                    ;   to "work"

        work    <1,2,3,4,5>         ; Passes one five-element
                                    ;   parameter to "work"
```

Note

When the **IRP** directive is used inside a macro definition and when the argument list of the **IRP** directive is also a parameter of the macro, you must use the literal-text operator (< >) 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:

```
init        MACRO   x
            IRP     y,<x>
            DB      y
            ENDM
            ENDM
```

If this macro is called with

```
init    <0,1,2,3,4,5,6,7,8,9>
```

the macro removes the angle brackets from the parameter so that it 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
```

Literal-Character Operator

The literal-character operator (!) forces the assembler to treat a specified character literally rather than as a symbol.

Syntax

!*character*

The literal-character operator is used with special characters such as the semicolon or ampersand when meaning of the special character must be suppressed. Using the literal-character operator is the same as enclosing a single character in brackets. For example, *!!* is the same as <*!*>.

Example

```
errgen          MACRO     y,x
                PUBLIC    err&y
err&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.

Expression Operator

The expression operator (%) causes the assembler to treat the argument following the operator as an expression.

Syntax

%text

The expression's value is computed and **masm** replaces *text* with the result. The expression can be either a numeric expression or a text equate. Handling text equates with this operator is a new feature in Version 5.0. Previous versions handled numeric expressions only. If there are additional arguments after an argument that uses the expression operator, the additional arguments must be preceded by a comma, not a space or tab.

The expression operator is typically used in macro calls when the programmer needs to pass the result of an expression rather than the actual expression to a macro.

Example

```
printe      MACRO   exp,val
            IF2                         ;; On pass 2 only
            %OUT    exp = val           ;; Display expression and result
            ENDIF                       ;;   to standard output
            ENDM

sym1        EQU     100
sym2        EQU     200
msg         EQU     <"Hello, World.">

            printe  <sym1 + sym2>,%(sym1 + sym2)
            printe  msg,%msg
```

In the first macro call, the text literal *sym1 + sym2* is passed to the parameter *exp*, and the result of the expression is passed to the parameter *val*. In the second macro call, the equate name *msg* is passed to the parameter *exp*, and the text of the equate is passed to the parameter *val*. As a result, **masm** displays the following messages:

```
sym1 + sym2 = 300
msg = "Hello, World."
```

The **%OUT** directive, which sends a message to the standard output, is described in the section, ''Sending Messages to Standard Output'', in Chapter 11; the **IF2** directive is described in the section, ''Testing the Pass with IF1 and IF2 Directives,'' in Chapter 9.

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 the section, ''Controlling Page Breaks,'' in Chapter 11.

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.

Using Recursion

Macro definitions can be recursive: that is, they can call themselves. Using recursive macros is one way of doing repeated operations. The macro does a task, and 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 no parameters are left to push. A variable number of parameters (up to six) can be given.

Nesting Macro Definitions

One macro can define another. Nested definitions are not processed 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, 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 4
            REPT    rotates
            opname  operand,1          ;; One at a time is faster
            ENDM                       ;;   for 4 or less on 8088/8086
            ELSE
            mov     cl,rotates         ;; Using CL is faster
            opname  operand,cl         ;;   for more than 4 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 the macro names are identical except for the instruction. For example, the macro for the **SHR** instruction is called *shrs*; the macro for the **ROL** instruction is called *rols*. If you want to enhance the macros by doing more parameter checking, you can modify the original macro. Doing so will change the created macros automatically. This macro uses the substitute operator, as described in the section, "Substitute Operator."

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 <4 + 2 * 7 - 3 MOD 4>
```

The two sample macros enable you to print the result of a complex expression to the standard output by using the **%OUT** directive, even though that directive expects text rather than an expression (see the section, "Sending Messages to Standard Output"), in Chapter 11. 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 (4 + 2 * 7 - 3 MOD 4) has the value: 15
```

You could get the same output by 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. the section, "Literal-Text Operator," explains why.

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 statements or comments between the **ENDM** directive of the nested redefinition and the **ENDM** directive of the original macro.

Example

```
EXTRN   _read:PROC

getasciiz MACRO
        .DATA
max     DW      80
actual  DW      ?
tmpstr  DB      80 DUP(?)
        .CODE
        push    max
        push    OFFSET tmpstr
        push    0               ;; standard input
        call    _read
        add     sp, 6
        mov     actual, ax
getasciiz MACRO
        push    max
        push    OFFSET tmpstr
        push    0               ;; standard input
        call    _read
        add     sp, 6
        mov     actual, ax
        ENDM
        ENDM
```

This macro allocates data space the first time it is called, and then redefines itself so that it doesn't try to reallocate the data on subsequent calls.

Avoiding Inadvertent Substitutions

All parameters are replaced when they occur with the corresponding argument, even if the substitution 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. You will not be warned about using reserved names as macro parameters.

You will be given a warning if you use a reserved name as a macro name. You can ignore the warning, but 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. Your *ADD* macro takes its place.

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.

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 or relative file specification, including drive or directory name.

If a complete or relative 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 the section, ''Setting a Search Path for Include Files,'' in Chapter 2, for more information on the **-I** option.

2. The current directory is searched for the include file.

Nested **INCLUDE** directives are allowed, and **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 MS-DOS 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.mac                     ; File name only; use with -I

INCLUDE /usr/jons/include/stdio.mac  ; Complete file specification

INCLUDE masm_inc\define.inc           ; Partial path name in MS-DOS format
```

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.

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 needed only 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.

The **PURGE** directive works by redefining the macro to a null string. Therefore, calling a purged macro does not cause an error. The macro name is simply ignored.

Example

```
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 11

Controlling Assembly Output

Introduction

There are two ways that the Macro Assembler can communicate results of an assembly to the user: it can write information to a listing or object file, or it can display messages to the standard output.

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.

1

Sending Messages to Standard Output

The **%OUT** directive instructs the assembler to display text to the standard output device. This device is normally the screen, but you can also redirect the output to a file or some other device.

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 for control when the directive is processed. Macros that enable you to output the value of expressions are shown in the section, ''Nesting Macro Calls,'' in Chapter 10.

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.

Controlling Page Format in Listings

There are several directives provided for controlling the page format of listings. These directives include the following:

Directive	Action
TITLE	Sets title for listings
SUBTTL	Sets title for sections in listings
PAGE	Sets page length and width, and controls page and section breaks

Setting the Listing Title

The **TITLE** directive specifies a title to be used on each page of assembly listings.

Syntax

TITLE *text*

The *text* can be any combination of characters up to 60 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.

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
```

Setting the Listing Subtitle

1

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 in length. The subtitle is printed flush left on the third line of the 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, which creates a new section (see the section, "Controlling Page Breaks").

Example

```
SUBTTL Point Plotting Procedure
PAGE    +
```

The example above creates a section title and 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
```

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 of 10-255 lines. The default page length is 50 lines. The *width* must be in the range of 60-132 characters. 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. For an example, see the section, ''Setting the Listing Subtitle.''

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.

Controlling the Contents of Listings

Several directives are provided for controlling what text will be shown in listings. The directives that control the contents of listings are shown below:

Directive	Action
.LIST	Lists statements in program listing
.XLIST	Suppresses listing of statements
.LFCOND	Lists false-conditional blocks in program listing
.SFCOND	Suppresses false-conditional listing
.TFCOND	Toggles false-conditional listing
.LALL	Includes macro expansions in program listing
.SALL	Suppresses listing of macro expansions
.XALL	Excludes comments from macro listing

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
.
.
.
```

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** option 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 the section, ''Listing False Conditionals,'' in Chapter 2.

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.

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 instance, 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
                               ; Normal comment
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
= 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 is never listed in macro expansions. Normal comments are listed only with the **.LALL** directive.

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
.
.
```

Part 3

Using Instructions

Part 3 of this manual (Chapters 12-19, Appendixes A-E) explains how to use instructions in assembly-language source code. Instructions define the code that will be executed by the processor at run time.

Chapters 12 and 13 describe overall concepts that apply to all instructions. Chapter 12 summarizes the 8086-family of microprocessors; it explains protection modes, tells how the processors address memory, and describes registers. Chapter 13 explains the addressing modes that can be used with instruction operands.

Chapters 14-19 describe the instructions themselves. The material is organized topically, with related instructions discussed together. The 8087-family coprocessors and their instructions are explained in Chapter 18.

Appendix A describes the new features included in Version 5.0 of **masm**. This appendix covers improvements and additions to **masm**, as well as compatibility issues.

Appendix B lists the syntax of each instruction recognized by **masm** and the instruction-set directives. This appendix also includes mnemonics for various instruction sets.

Appendix C summarizes **masm** directives, including concise functional descriptions.

Appendix D describes the naming conventions used to form assembly-language source files that are compatible with existing object modules. Several Microsoft compilers use the conventions listed in this appendix.

Appendix E lists and explains status messages, error messages, and exit codes generated by **masm**.

Chapter 12

Understanding 8086-Family Processors

Introduction

This chapter introduces the 8086-family of processors. It describes their segmented-memory structure and their registers. Differences between the chips in the family are also covered.

Using the 8086-Family Processors

The Intel Corporation manufactures the group of processors referred to in this manual as the 8086-family processors. The UNIX System V and MS-DOS operating systems are designed to work under these processors and to take advantage of their features. The processors have several features in common, as follows:

- Memory is organized by using a segmented architecture.
- The instruction set is upwardly compatible—all features available in the early versions of the processor are also available in the newer versions, but the new versions contain additional features not supported in the old versions.
- The register set is also upwardly compatible.

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 one megabyte of memory. Addresses specified in assembly language correspond to physical memory addresses. The 8088 uses an 8-bit data bus, and 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 you word-align it by using the **EVEN** or **ALIGN** directives (see the section, ''Aligning Data''), in Chapter 5.

80186 — This processor is identical to the 8086 except that new instructions have been added and some old instructions have been 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 and speed 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 can be run concurrently. Memory used by each process can be protected from 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. Under protected-mode operating systems, the processor allocates and manages memory dynamically. Additional privileged instructions for initializing protected mode and controlling multiple processes are available.

80386 — This is both a 16-bit and a 32-bit processor. It is fully compatible with the 80286; but at the system level, it implements many new features, including virtual memory, multiple 8086 processes, and addressing for up to four gigabytes of memory. This manual does not explain how to use these features.

For the applications programmer, 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 are summarized in the section, "Using the 80386 Processor."

8087, 80287, and 80387 — These are math coprocessors that work concurrently with the 8086-family processors. They do mathematical calculations faster and more accurately than can be done with the 8086-family processors. Although there are performance and technical differences between the three coprocessors, the main difference to the applications programmer is that the 80287 and 80387 can operate in protected mode. The 80387 also has several new instructions.

2

Real and Protected Modes

Protected mode is the multiple-process mode used in Part 1, ''Using Assembler Programs/286, and UNIX System V. It is also used in OS/2, the multitasking version of MS-DOS. Real mode is the single-process mode used in current versions of MS-DOS.

To the applications programmer, there is little difference between assembly-language programming in real or protected mode. Processes are managed at the system level by the operating system. The applications programmer does not deal with processes 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:

- In protected mode, up to 16 megabytes of memory can be addressed (compared to one megabyte in real 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 protected mode, segment registers contain segment selectors rather than actual segment values. The selectors must come from the operating system. They cannot be calculated by the program. Programming techniques that attempt to calculate segment values or address memory directly will not work.

- Certain instructions that can be used normally in real mode are privileged instructions in protected-mode operating systems. These include **STI**, **CLI**, **IN**, and **OUT**. These instructions are still available at privilege levels normally used only by systems programmers.

Protected-mode operating systems, such as UNIX System V and OS/2, provide extended functions for doing the kinds of tasks that are currently done by using the previously described restricted practices.

12

Segmented Addresses

When used in real mode, 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 one megabyte (0FFFFFh) in real mode or 16 megabytes (0FFFFFFh) 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 the *segment:offset* format. For example, an address with a segment of 053C2h and an offset of 0107Ah would be represented as *53C2:107A*. This method of specifying addresses can be used directly in most debuggers, but it is not legal in assembler source code.

In real mode, the address *53C2:107A* represents a physical 20-bit address. This address can be calculated by multiplying the *segment* portion of the address by 16 (10h), and then adding the *offset* portion, as shown below:

53C20h	Segment times 10h
+ 107Ah	Offset
54C9Ah	Physical address

In protected mode, the address *53C2:107A* represents a movable address. The segment portion of the address is a selector 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 Only

The 80386 processor supports 48-bit addresses consisting of a 16-bit segment selector and a 32-bit offset. This enables the processor to access addresses of up to four gigabytes per segment in protected mode. The processor can also run in modes 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 4, ''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 3, ''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.

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 data 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 12-1 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 this section.

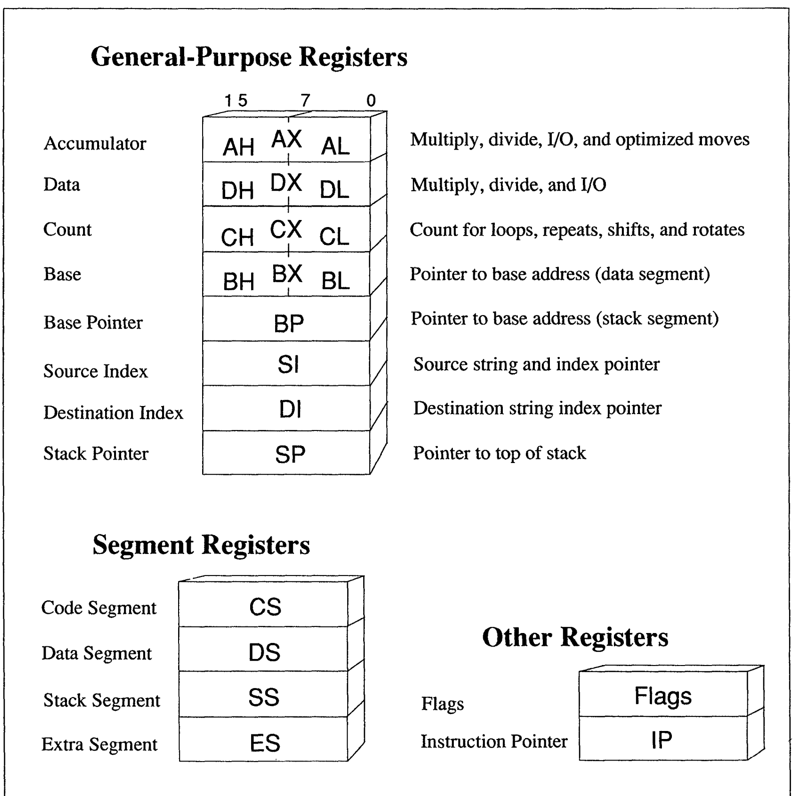


Figure 12-1 Register for 8088-80286 Processors

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 extended registers begin with the letter **E**. For example, the 32-bit version of **AX** is **EAX**. The 80386 also has two additional segment registers, **FS** and **GS**. Figure 12-2 shows the extended registers of the 80386.

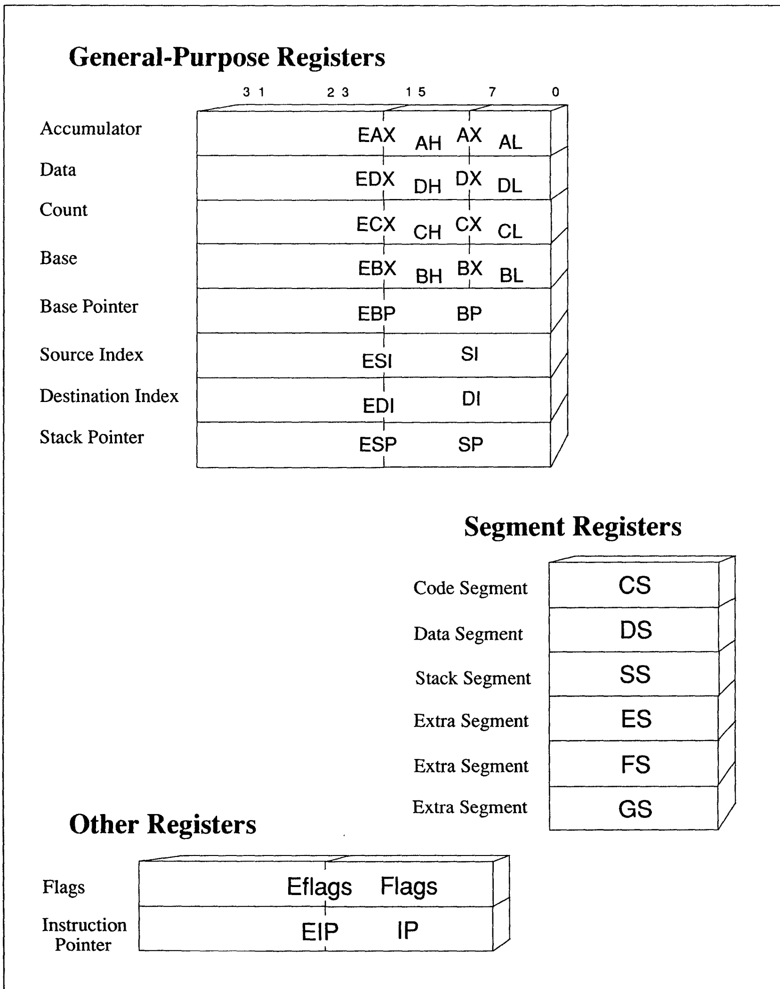


Figure 12-2 Extended Registers of 80386 Processor

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 of memory reserved for storing temporary data. For information on using stacks, see the section, "Transferring Data to and from the Stack," in Chapter 14.
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 segments, **FS** and **GS**.

General-Purpose Registers

The **AX**, **DX**, **CX**, **BX**, **BP**, **SI**, and **DI** 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**, **DX**, **CX**, and **BX**, can be accessed as two 8-bit registers or as a single 16-bit register. The **AH**, **DH**, **CH**, and **BH** registers represent the high-order 8 bits of the corresponding registers. Similarly, **AL**, **DL**, **CL**, and **BL** represent the low-order 8 bits of the registers. All the general registers can be extended to 32 bits on the 80386 by appending the letter **E**—**EAX**, **EDX**, **ECX**, and so on.

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. When dividing a doubleword value, **DX** holds the upper word of the dividend before the operation and the remainder afterward. When multiplying word values, **DX** holds the upper word of the doubleword result. In I/O operations to and from ports, **DX** holds the number of the port to be accessed.
CX	The **CX** (Count) register must be 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. This register can also be used for temporary data storage.
BX	The **BX** (Base) register can be used as a pointer. For instance, it can point to the base of a data object (see the section, "Indirect Memory Operands"), in Chapter 13. This register can also be used for temporary data storage.
BP	The **BP** (Base Pointer) register can be used for general data storage. It is more often used as a pointer. For instance, it is often used to point to the base of a stack frame. The conventions for passing arguments to

procedures have a specific use for **BP** as described in the section, ''Passing Arguments on the Stack,'' in Chapter 16. The **SS** register is assumed as the segment register in operations using **BP**.

SI The **SI** (Source Index) register can be used as a pointer or for general data storage. It is often used for pointing to (indexing) an item within a data object. With string instructions, **SI** is used to point to bytes or words within a source string.

DI The **DI** (Destination Index) register can be used as a pointer or for general data storage. It is often used for pointing to (indexing) an item within a data object. With string instructions, **DI** is used to point to bytes or words within a destination string.

Other Registers

The 8086-family processors have two additional registers whose values are changed automatically by the processor.

Register	Description
SP	The **SP** (Stack Pointer) register points to the current location within the stack segment. Pushing a value onto the stack decreases the value of **SP** by two; popping from the stack increases the value of **SP** by two. Call instructions store the calling address on the stack and decrease **SP** accordingly; return instructions get the stored address and increase **SP**. With 80386 32-bit segments, **SP** is increased or decreased by four instead of two. The sections, ''Using the Stack'', in Chapter 14, and ''Passing Arguments on the Stack,'' in Chapter 16, discuss operation of the stack in more detail. **SP** is technically a general-purpose register that could be used in calculations or for temporary data storage. However, it should generally be used only for stack operations.
IP	The **IP** (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, instructions that control program

flow (such as calls, jumps, loops, and interrupts) automatically change the instruction pointer.

The Flags Register

The flags register is a 16-bit register made up of bits that control various instructions and reflect the current status of the processor. In the 80386 processor, 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. The extend flags register of the 80386 is sometimes called eflags.

Figure 12-3 shows the bits of the 32-bit flags register for the 8088 - 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.

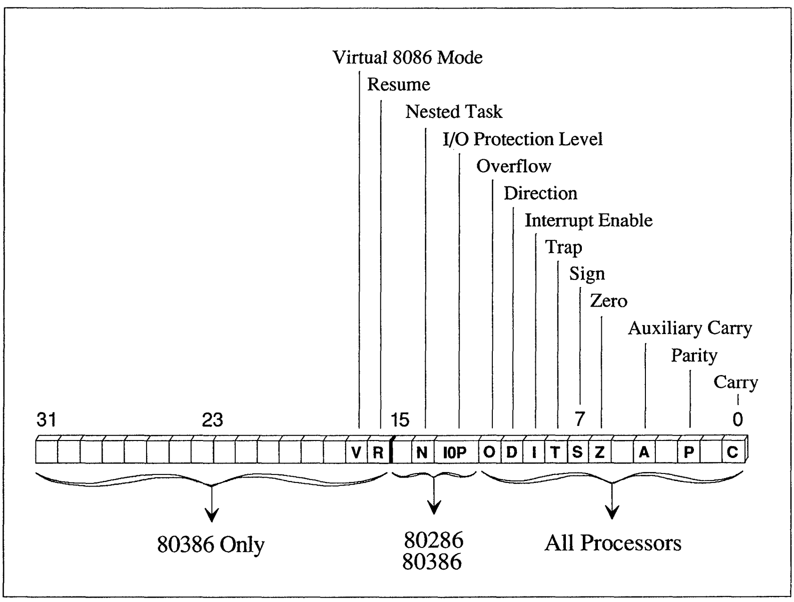


Figure 12-3 Flags for 8088-80386 Processors

The 13 flags common to all 8086-family processors are summarized below, starting with the low-order flags. In these descriptions, the term ''set'' means the bit value is 1, and ''cleared'' means the bit value is 0.

Flag	Description
Carry	Is set if an operation generates a carry to or a borrow from a destination operand.
Parity	Is set if the low-order bits of the result of an operation contain an even number of set bits.
Auxiliary Carry	Is set if an operation generates a carry to or a borrow from the low-order four bits of an operand. This flag is used for binary coded decimal arithmetic.
Zero	Is set if the result of an operation is 0.
Sign	Equal to the high-order bit of the result of an operation (0 is positive, 1 is negative).
Trap	If set, the processor generates a single-step interrupt after each instruction. A 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	Is 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 not described in this manual.
Nested Task	Controls chaining of interrupted and called tasks. Controlling tasks in protected mode is a systems task not described in this manual.

Resume	If set, debug exceptions are temporarily disabled. Using 80386 debug exceptions is a systems task 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 not described in this manual.

8087-Family Registers

The 8087-family processors use a stack-based architecture to access up to eight 80-bit registers. For information on using 8087-family registers and instructions, see Chapter 18, ''Calculating with a Math Coprocessor.'' The format of real numbers used by coprocessors is explained in the section, ''Real-Number Variables'', in Chapter 5.

12

Using the 80386 Processor

Applications programmers can use some 80386 enhancements. 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 the integer-multiply instruction (**IMUL**); conversion instructions (**CWDE** and **CDQ**); string instructions (**CMPSD**, **LODSD**, **MOVSD**, **SCASD**, **STOSD**, **INSD**, **OUTSD**); and 32-bit stack enhancements (**PUSHAD**, **POPAD**, **PUSHFD**, **POPFD**, and **IRETD**).

- You can use 32-bit registers for calculations. For instance, you can add and subtract doubleword integers without using multiple registers, and you can do some multiplication and division operations 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 any general-purpose 32-bit register to be a pointer in an indirect memory operand. If you use this technique, you must make sure that 32-bit registers used as pointers actually contain valid 16-bit addresses.

Chapter 13

Using Addressing Modes

Introduction

Instruction operands can be given in different forms called addressing modes. Addressing modes tell the processor how to calculate the actual value of an operand at run time.

The three kinds of addressing modes are immediate, register, and memory operands. Memory operands are further broken into two groups, direct and indirect memory operands.

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 their instruction mnemonic the same, **masm** may actually assemble 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.

Instructions that take two or more operands always work right to left. The right operand is the source operand. It specifies data that will be used, but not changed, in the operation. The left operand is the destination operand. It specifies the data that will be operated on and possibly changed by the instruction.

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 (four characters on the 80386) cannot be immediate data. They must be stored in memory before they can be processed by instructions.

Many instructions permit immediate data in the source (right) operand and either memory or register data in the destination (left) operand. 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
            or      bx,00100100b
            in      al,43h
            cmp     cx,200

; Only operand is immediate
            ret     6
            int     21h
```

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 the section, ''Indirect Memory Operands.''

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 13.1 can be used in register-direct mode.

Table 13.1

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, pointer, and index[1]	**EAX**	**EBX**	**ECX**	**EDX**
16-bit pointer and index	**SP**	**BP**	**SI**	**DI**
32-bit general, 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 the section, ''Using 8086-Family Registers,'' in Chapter 12. Limitations on register use for specific instructions are discussed in sections on the specific instructions throughout Part 3, ''Using Instructions.''

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
```

13

Using Memory Operands

Many instructions can work on data in memory. When a memory operand is given, the processor must calculate the address of the data to be processed. This address is called the "effective address." Calculation of the effective address depends on how the operand is specified, as explained below.

Note

Memory-to-memory operations are never allowed. These operations must be done indirectly by moving one of the memory values into a register before processing it.

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 load 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.

The effective address is always relative to a segment register. The default segment register is **DS** for direct memory operands, but the default segment can be overridden with the segment-override operator (**:**), as explained in the section, "Segment-Override Operator," in Chapter 8.

Direct memory operands are often specified as constant expressions by using the index operator. For example, the operand *table[4]* refers to the byte having an offset four bytes from the address of *table*. This expression is equivalent to *table+4*.

Example

```
          .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.

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 (immediate) value of the operand 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
```

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, only four registers can be used in indirect mode (see the section, ''80386 Indirect Memory Operands,'' for information on 80386 enhancements). **BX** and **BP** are called base registers; **DI** and **SI** are called index registers. The distinction between base and index registers is not always important. In many contexts, any of these registers can be thought of as the base or the index. In any case, an attempt to use any register other than these four in a statement that accesses memory indirectly results in an error.

You can use the base and index registers separately or in pairs, with or without specifying a displacement. A displacement can be either a constant or a direct memory. Several displacements can be given, but they are all added into a single displacement at assembly time. For example, in the statement

```
mov     ax,table[bx][di]+6
```

both *table* and *6* are displacements. To get the total displacement, **masm** calculates the actual offset of *table* and the offset at 6.

3

The modes in which registers can be used to specify indirect memory operands are shown in Table 13.2.

Table 13.2

Indirect Addressing Modes

Mode	Syntax	Description
Register indirect	**[BX]** **[BP]** **[DI]**	Effective address is contents of register
Based or indexed	**[BX]***disp* *displacement***[BP]** *displacement***[DI]** *displacement***[SI]**	Effective address is contents of register and *displacement*
Based indexed	**[BX][DI]** **[BP][DI]** **[BX][SI]** **[BP][SI]**	Effective address is contents of base register and contents of index register
Based indexed with displacement	*displacement***[BX][DI]** *displacement***[BP][DI]** *displacement***[BP][SI]**	Effective address is contents of base register and contents of index registers and *displacement*

Register-indirect operands are typically used to point to a memory address within a segment. Based and indexed operands are used to point to a memory address relative to a table, a one-dimensional array, or a structure. Operands with multiple indexes 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 are used in specific situations, as explained in Chapter 17, "Processing Strings." With other instructions, base and index registers can often be used interchangeably, depending on which registers are available.

When calculating the effective address of an indirect operand, the processor uses **DS** as the default segment register if **BX** is used as a base register, or if no base register is specified. If **BP** is used anywhere in the operand, the default segment register is **SS**. The default segment can be overridden with the segment-override operator (**:**), as explained in the section, "Segment-Override Operator," in Chapter 8, on the segment-override operator.

A common syntax for indirect memory operands is each register put within index operators ([ ]). The register or registers must always be within brackets, but a variety of alternate syntaxes is 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
mov     ax,[bx][di]+table
mov     ax,table[di][bx]
```

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
```

Use of the index operator is explained in more detail in Chapter 8.

When an index or displacement points into an array, it must be scaled for the size of elements in the array. On all processors except the 80386, scaling must be done in separate statements (see the section, "80386 Indirect Memory Operands," for information on 80386 scaling). The scaling factor is 1 for bytes (no scaling necessary), 2 for words, 4 for doublewords, and 8 for quadwords. Since scaling factors (other than for bytes) are multiples of 2, they can usually be calculated quickly with the **SHL** instruction, as shown below:

```
shl     di,1        ; Scale DI for words (DI *2)

shl     di,1        ; Scale DI for doublewords (DI*4)
shl     di,1

shl     di,1        ; Scale DI for quadwords (DI*8)
shl     di,1
shl     di,1
```

Use of the **SHL** instruction for multiplication is described in more detail in the section, ''Multiplying and Dividing by Constants,'' in Chapter 15.

3

Example 1

```
        add     dx,[bx]             ; Add the word contents of DS:BX
                                    ;   to the contents of DX
        mov     dl,[bp+6]           ; Load the byte contents
                                    ;   of SS:BP+6 into DL
        sub     dx,12[bx]           ; Subtract the word contents of
                                    ;   DS:12+BX 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     BYTE PTR [bx][si]   ; Decrement the byte
                                    ;   at DS:BX+SI
        cmp     cx,here[bp][si]     ; Compare the contents of CX
                                    ;   to the contents of SS:here+BP+SI
        push    place[bx][di]+2     ; Save the contents of
                                    ;   DS:place+BX+DI+2 on the stack
        call    cs:table[bx]        ; Call the routine pointed to
                                    ;   by the contents of CS:table+bx
```

The statements in Example 1 illustrate how the various instructions can be used with indirect memory operands.

Example 2

```
scrnbuff    EQU     0B800h              ; CGA screen buffer (actual
                                        ;   value is hardware dependent)
            mov     ax,scrnbuff         ; Load address of screen buffer
            mov     es,ax               ;   into ES

            mov     ax,4                ; Push column 4 as third argument
            push    ax
            mov     ax,6                ; Push row 6 as second argument
            push    ax
            mov     ax,"z"              ; Push "z" as first argument
            push    ax
            call    show                ; Call the procedure
            add     sp,6                ; Restore stack
            .
            .
            .
show        PROC    NEAR
            push    bp                  ; Save BP
            mov     bp,sp               ;   and set up stack frame
            push    si                  ; Save SI (so procedure could
                                        ;   be called from C)

            mov     si,[bp+8]           ; Load column
            dec     si                  ; Adjust for zero
            shl     si,1                ; Scale for 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 and BP
            pop     bp
            ret                         ; Return
show        ENDP
```

Example 2 illustrates two uses of indirect memory operands. Arguments are pushed onto the stack before calling a procedure. When the procedure is called, the arguments 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; 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).

The procedure follows the calling conventions of C and could be called directly from that language. Note that **SI** is saved and restored because the C compiler requires that it not be changed by a procedure.

Example 2 works on any processor. The section, "80386 Indirect Memory Operands," shows an enhanced version that uses 80386 instructions and addressing modes.

80386 Indirect Memory Operands

13

Instructions for the 80386 can be given in two modes, 16 bit and 32 bit. Understanding these modes is important, since indirect memory operands are different in each mode.

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, an offset address can be up to four 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 **BX**, **BP**, **DI**, and **SI** registers can be used in indirect memory operands. With the 80386, any general-purpose 32-bit register can be used. The same register can even be used as both the base and the index. Several examples are shown below:

```
add   edx, [eax]                ; Add double
mov   dl, [esp+10]              ; Add byte from stack
dec   WORD PTR [edx][eax]       ; Decrement word
cmp   cx,array[eax][eax]        ; Compare word from array
jmp   table[ecx]                ; Jump into pointer table
```

- The index register can have a scaling factor of 1, 2, 4, or 8. Any register except **ESP** can be the index register and can have a scaling factor. The scaling factor is specified by using the multiplication operator (*) adjacent to the register.

Scaling can be used to index into arrays with different sizes of elements. For example, the scaling factor is 1 for byte arrays (no scaling needed), 2 for word arrays, 4 for doubleword arrays, and 8 for quadword arrays. There is no performance penalty for using a scaling factor. Scaling is illustrated in the following examples:

```
mov   eax,darray[edx*4]      ; Load double of double array
mov   eax,[esi*8][edi]       ; Load double of quad array
mov   ax,wtbl[ecx+2][edx*2]  ; Load word of word array
```

- The default segment register is **SS** if the base register is **EBP** or **ESP**; it is **DS** for all other the base registers. If two registers are used, only one can have a scaling factor and it is defined to be the index register. The other register is the base. If scaling is not used, the first register is the base. If one register is used, it is the base, regardless of scaling. The following examples illustrate how to determine the base register:

```
mov   eax,[edx][ebp*4] ; EDX base (not scaled) - DS segment
mov   eax,[edx*1][ebp] ; EBP base (not scaled) - SS segment
mov   eax,[edx][ebp]   ; EDX base (first) - DS segment
mov   eax,[ebp][edx]   ; EBP base (first) - SS segment
mov   eax,[ebp*2]      ; EBP base (only) - SS segment
```

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 for 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 segment. 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 moved into a 16-bit segment may cause problems. If **EAX** contains a 16-bit value (the top half of the 32-bit register is 0), then the statement works. However, if the top half of the **EAX** register is not 0, the processor may generate an error.

Warning

It is possible to use both 16-bit and 32-bit modes in the same program by defining separate code segments 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 only in systems programming and is beyond the scope of this manual.

13

Example

```
            .MODEL   small                  ; .MODEL precedes .386
            .386                            ;   to make 16-bit segments

scrnbuff    EQU      0B800h                 ; CGA screen buffer (actual
                                            ;   value is hardware dependent)
            .CODE
            .
            .
            .
            mov      ax,scrnbuff            ; Load address of screen buffer
            mov      es,ax                  ;   into ES

            push     4                      ; Push column 4 as third argument
            push     6                      ; Push line 6 as second argument
            push     "z"                    ; Push "z" as first argument
            call     show                   ; Call the procedure
            add      sp,6                   ; Restore stack
            .
            .
            .
show        PROC     NEAR

            movzx    ebx,WORD PTR [esp+6];  Load column
            dec      ebx                    ; Adjust for zero
            movzx    eax,WORD PTR [esp+4];  Load row
            dec      eax                    ; Adjust for zero
            imul     eax,160                ; Multiply 160 bytes per line

            mov      dl, [esp+2]            ; Load character
            mov      es:[eax][ebx*2],dl     ; Put character in buffer

            ret                             ; Return
show        ENDP
```

This example is the same as the one in the section, ''Indirect Memory Operands,'' except that it uses enhanced 80386 instructions and addressing modes to make the code shorter and more efficient. Note the following differences:

- Since **ESP** can be used as a base register, stack registers can be accessed directly without the stack setup required by previous processors. This assumes that **ESP** does not change inside the procedure.

- Values are loaded and zero-extended in one step by using the **MOVZX** instruction (see the section, ''Moving and Extending Values''), in Chapter 14.

- **EBX** is used with scaling. In the previous example, scaling had to be done with a separate instruction.

- **EAX** and **EBX** are used instead of **BX** and **SI**. This saves some register swapping, since **EAX** can be used both for the result of the multiplication operation and as a base register.

- Immediate operands are used with the **PUSH** and **IMUL** instructions (described in the sections, ''Pushing and Popping,'' in Chapter 14, and ''Multiplying,'' in Chapter 16, respectively). These enhancements were implemented with the 80186 processor, but they are rarely used since most programs have to be able to run on the 8088 and 8086. Since 80836 programs can never run on the earlier processors, there is no reason not to use enhanced 80186 instructions.

Chapter 14

Loading, Storing, and Moving Data

Introduction

The 8086-family processors provide several instructions for loading, storing, or moving various kinds of data. Among the types of transferable data are variables, pointers, and flags. Data can be moved to and from registers, memory, ports, and the stack. This chapter explains the instructions for moving data from one location to another.

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.

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 operands both contain the same value. The old value in the destination operand is destroyed.

4

Syntax

MOV {*register* | *memory*},{*register* | *memory* | *immediate*}

Example 1

```
mov     ax,7          ; Immediate to register
mov     mem,7         ; Immediate to memory direct
mov     mem[bx],7     ; Immediate to memory indirect

mov     mem,ds        ; Segment register to memory
mov     mem,ax        ; Register to memory direct
mov     mem[bx],ax    ; Register to memory indirect

mov     ax,mem        ; Memory direct to register
mov     ax,mem[bx]    ; Memory indirect to register
mov     ds,mem        ; Memory to segment register

mov     ax,bx         ; Register to register
mov     ds,ax         ; General register to segment register
mov     ax,ds         ; Segment register to general register
```

The statements in Example 1 illustrate each type of memory move that can be done with a single instruction. Example 2 illustrates several common types of moves that require two instructions.

Example 2

```
; Move immediate to segment register
        mov     ax,DGROUP   ; Load immediate to general register
        mov     ds,ax       ; Store general register to segment register

; Move memory to memory
        mov     ax,mem1     ; Load memory to general register
        mov     mem2,ax     ; Store general register to memory

; Move segment register to segment register
        mov     ax,ds       ; Load segment register to general register
        mov     es,ax       ; Store general register to segment register
```

Exchanging Data

The **XCHG** (Exchange) instruction exchanges the data in the source and destination operands. Data can be exchanged between registers or between registers and memory.

Syntax

XCHG {*register* | *memory*},{*register* | *memory*}

Examples

```
xchg    ax,bx       ; Put AX in BX and BX in AX
xchg    memory,ax   ; Put "memory" in AX and AX in "memory"
```

Looking Up Data

The **XLAT** (Translate) instruction is used to load data from a table in memory. The instruction is useful for translating bytes from one coding system to another.

Syntax

XLAT[B] [[*segment*:]*memory*]

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. The operand need not be given except when specifying a segment override.

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** register.

Note

> For compatibility with Intel 80386 mnemonics, **masm** recognizes **XLATB** as a synonym for **XLAT**. In the Intel syntax, **XLAT** requires an operand; **XLATB** does not allow one. An operand is never required by **masm**, but one is always allowed.

Transferring Flags

The 8086-family processors provide instructions for loading and storing flags in the **AH** register.

Syntax

> **LAHF**
> **SAHF**

The status of the lower byte of the flags register can be saved to the **AH** register with **LAHF** and then later restored with **SAHF**. If you need to save and restore the entire flags register, use **PUSHF** and **POPF**, as described in the section, ''Saving Flags on the Stack.''

SAHF is often used with a coprocessor to transfer coprocessor control flags to processor control flags. The section, ''Controlling Program Flow,'' in Chapter 18, explains and illustrates this technique.

Converting between Data Sizes

Since moving data between registers of different sizes is illegal, you must take special steps if you need to extend a register value to a larger register or register pair.

The procedure is different for signed and unsigned values. The processor cannot tell the difference between signed and unsigned numbers; the programmer has to understand this difference and program accordingly.

Extending Signed Values

The **CBW** (Convert Byte to Word) and **CWD** (Convert Word to Doubleword) instructions are provided to sign-extend values. Sign-extending means copying the sign bit of the unextended operand to all bits of the extended operand.

Syntax

CBW
CWD

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. Both instructions work only on values in the accumulator register.

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
```

80386 Only

The 80386 processor provides additional conversion instructions for 32-bit signed values.

Syntax

CWDE
CDQ

The **CWDE** (Convert Word to Doubleword Extended) instruction converts a signed 16-bit value in **AX** to a signed 32-bit signed value in **EAX**. The **CDQ** (Convert Doubleword to Quadword) instruction converts a 32-bit signed value in **EAX** to a signed 64-bit value in the **EDX:EAX** register pair.

Example 2

```
            .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)
            cdq                 ; Convert to 64-bit -5
                                ;   (FFFFFFFF:FFFFFFFBh) in EDX:EAX
```

Extending Unsigned Values

To extend unsigned numbers, set the value of the upper register to 0.

Example

```
                .DATA
mem8            DB      251
mem16           DB      251
                .CODE
                .
                .
                .
                mov     al,mem8      ; Load 251 (FBh) from 8-bit memory
                xor     ah,ah        ; Zero upper half (AH)

                mov     ax,mem16     ; Load 251 (FBh) from 16-bit memory
                xor     dx,dx        ; Zero upper half (DX)
```


Moving and Extending Values

80386 Only

The 80386 processor provides instructions that move and extend a value to a larger data size in a single step. The same thing can be done in two steps with earlier processors, 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.

Converting between Data Sizes

Example

```
; Enhanced 80386 instructions

          movzx   dx,bl       ; Load unsigned 8-bit value into
                              ;   16-bit register and zero extend

; Equivalent to these 80286 instructions

          mov     dl,bl       ; Load 8-bit unsigned value
          xor     dh,dh       ; Clear the top of register

; Enhanced 80386 instructions

          movsx   dx,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 to AL
          cbw                 ; Sign extend to AX
          mov     dx,ax       ; Copy to 16-bit register
```

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.

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 register.

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 have the same effect, 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 Only

Scaling of indirect memory operands gives the **LEA** instruction some interesting side effects with the 80386 processor. (Scaling is explained in the section, "80386 Indirect Memory Operands,") in Chapter 13. 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 by using the **MUL** instruction.

```
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 by using shift instructions, as described in the section, "Multiplying and Dividing by Constants," in Chapter 15.

Loading Far Pointers

The **LDS** and **LES** instructions load far pointers.

Syntax

14

LDS *register,memory*
LES *register,memory*

The memory address being pointed to is specified in the source operand, and the register where the offset will be stored is specified in the destination operand.

The address must be stored in memory with the offset in the upper word and the segment in the lower word. 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 17, "Processing Strings."

Example

```
            .DATA
string      DB      "This is a string."
fpstring    DD      string              ; Far pointer to string
pointers    DD      100 DUP (?)
            .CODE
            .
            .
            .
            les     di,fpstring         ; Put address in ES:DI pair
            lds     si,pointers[bx]     ; Put address in DS:SI pair
```

80386 Only

The 80386 processor has additional instructions for loading far pointers. These instructions are exactly like **LDS** and **LES**, except for the segment register in which they put the segment address.

Syntax

> **LSS** *register,memory*
> **LFS** *register,memory*
> **LGS** *register,memory*

The **LSS**, **LFS**, and **LGS** instructions load the segment address into **SS**, **FS**, and **GS**, respectively.

Example

```
            .386                         ; .386 first for 32-bit mode
            .MODEL  large
            .DATA
string      DB      "This is a string."
fpstring    DF      string               ; Far pointer to string
            .CODE
            .
            .
            .
            lgs     edi,fpstring         ; Put address in GS:EDI pair
```

Transferring Data to and from the Stack

A stack is an area of memory for storing temporary data. Unlike other segments in which data is stored starting from low memory, data on the stack is stored in reverse order starting from high memory.

Initially, the stack is an uninitialized segment of a finite size. As data is added to the stack at run time, the stack grows downward from high memory to low memory. When items are removed from the stack, it shrinks upward from low memory to high memory.

The stack has several purposes in 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 the sections, ''Using Procedures'' and ''Using Interrupts'', in Chapter 16). You can also use the **PUSH** and **POP** instructions and their variations to store values on the stack.

Pushing and Popping

In 8086-family processors, the **SP** (stack pointer) register 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.

The values pointed to by the **BP** and **SP** registers are relative to the stack segment (**SS** register). The **BP** register is often used to point to the base of a frame of reference (a stack frame) within the stack.

Syntax

> **PUSH** {*register* | *memory*}
> **POP** {*register* | *memory*}
> **PUSH** *immediate* (80186-80386 only)

The **PUSH** instruction is used to store a two-byte 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** register 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. (In 80386 32-bit segments, four-byte values are pushed and **ESP** changes in multiples of four.)

Note

> 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 in **SP** after the push operation. The same statement under the 80186, 80286, or 80386 processor pushes the word in **SP** before the push operation.

Figure 14-1 illustrates how pushes and pops change the **SP** register. Notice 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 now unknown and may be overwritten the next time the stack is used.

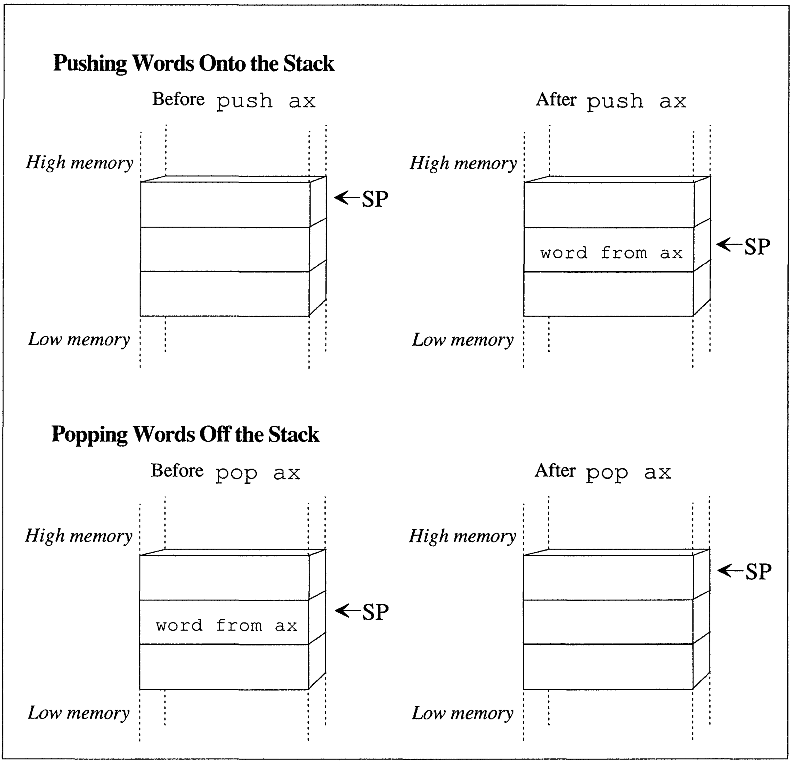


Figure 14-1 Stack Status after Pushes and Pops

The **PUSH** and **POP** instructions are almost always used in pairs. Words are popped off the stack in reverse order from the order in which they are pushed onto the stack. You should normally do the same number of pops as pushes to return the stack to its original position. However, it is possible to return the stack to its original position by adding the correct number of words from the **SP** register.

Values on the stack can be accessed by 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
.
.
.
add     sp,6            ; Restore stack pointer
                        ;   two bytes per push
```


80186/286/386 Only

Starting with the 80186, the **PUSH** instruction can be given with an immediate operand. For example, the following statement is legal on the 80186, 80286, and 80386 processors:

```
push    7           ; 3 clocks on 80286
```

This statement is faster than the following equivalent statements, which are required on the 8088 or 8086:

```
mov     ax,7        ; 2 clocks on 80286
push    ax          ; 3 clocks on 80286
```

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 four-byte value. A warning message will be generated if you try to push a 16-bit value in a 32-bit segment or a 32-bit value in a 16-bit segment.

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 onto 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 the section, "Passing Arguments on the Stack," in Chapter 16.

Another common use of the stack is 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 original count is popped back into **CX**.

4

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
```

Saving Flags on the Stack

Flags can 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. They can also be used within a procedure to save and restore the flag status of the caller.

Example

```
pushf
call    systask
popf
```


80386 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**.

Saving All Registers on the Stack

80186/286/386 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** instructions is significantly faster and takes fewer bytes of code than pushing and popping each register individually.

Transferring Data to and from the Stack

The registers are pushed in the following order: **AX**, **CX**, **DX**, **BX**, **SP**, **BP**, **SI**, and **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 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**.

14

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 can be used for low-level communication with devices such as disks, the video display, or the keyboard. The **OUT** instruction is used to send data to a port; the **IN** instruction receives data from a port.

Syntax

> **IN** *accumulator*,{*portnumber* | **DX**}
> **OUT** {*portnumber* | **DX**},*accumulator*

When using the **IN** and **OUT** instructions, the number of the port can either be an 8-bit immediate value or the **DX** register. You must use **DX** for ports with a number higher than 256. The value to be received from the port must be in the accumulator register (**AX** for word values or **AL** for byte values).

When using the **IN** instruction, the number of the port is given as the source operand and the value to be sent to the port is the destination operand. When using the **OUT** instruction, the number of the port is given as the destination operand and the value to be sent to the port is the source operand.

In applications programming, most communication with hardware is done with system 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.

Note

Under Part 1, ''Using Assembler Programs and other protected-mode operating systems, **IN** and **OUT** are privileged instructions and can only be used in privileged mode.

Transferring Data to and from Ports

80186/286/386 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 other string instructions. They are discussed in the section, ''Transferring Strings to and from Ports,'' in Chapter 17.

14

Chapter 15

Doing Arithmetic and Bit Manipulations

Introduction

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 tells you how to use the instructions that do calculations on numbers and bits.

Adding

The **ADD**, **ADC**, and **INC** instructions are used for adding and incrementing values.

Syntax

ADD {*register* | *memory*},{*register* | *memory* | *immediate*}
ADC {*register* | *memory*},{*register* | *memory* | *immediate*}
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 the section, ''Calculating with Binary Coded Decimals.''

5

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 incremented. The value is treated as an unsigned integer, so the carry flag is not updated for signed carries.

The **ADD** instruction adds values given in source and destination operands. The destination can be either a register or a memory operand. Its contents will be destroyed by the operation. The source operand can be an immediate, memory, or register operand. 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 must be the same size.

An addition operation can be interpreted as addition of either signed numbers or unsigned numbers. It is the programmer's responsibility to decide how the addition should be interpreted and to 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.

There are two ways to take action on an overflow: you can use the **JO** or **JNO** instruction to direct program flow to or around instructions that handle the overflow (see the section, "Testing Bits and Jumping," in Chapter 16). You can also use the **INTO** instruction to trigger the overflow interrupt (interrupt 4) if the overflow flag is set.

Example

```
mem8        .DATA
            DB      39
            .CODE
            .
            .
            .                 ;                       unsigned  signed
            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) or **INTO** (Interrupt 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.

Adding Values in Multiple Registers

The **ADC** (Add with Carry) instruction makes it possible to add numbers larger than can be held in a single register.

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.

Adding

When adding numbers in multiple registers, the carry flag should be ignored for the least-significant portion, but taken into account for the more-significant portion. This can be done by using the **ADD** instruction for the least-significant portion and the **ADC** instruction for more-significant portions.

You can add and carry repeatedly inside a loop for calculations that require more than two registers. Use the **ADC** instruction in each iteration, but 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. You could also do the first add outside the loop.

Example

```
          .DATA
mem32     DD      316423
          .CODE
          .
          .
          .
          mov     ax,43981              ; Load immediate      43981
          xor     dx,dx                 ;   into DX:AX
          add     ax,WORD PTR mem32[0]  ; Add to both       + 316423
          adc     dx,WORD PTR mem32[2]  ;   memory words      ------
                                        ; Result in DX:AX    360404
```

15

Subtracting

The **SUB**, **SBB**, **DEC**, and **NEG** instructions are used for subtracting and decrementing values.

Syntax

> **SUB** {*register* | *memory*},{*register* | *memory* | *immediate*}
> **SBB** {*register* | *memory*},{*register* | *memory* | *immediate*}
> **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 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 used to do calculations on BCD numbers, as described in the section, "Calculating with Binary Coded Decimals."

15

Subtracting Values Directly

The **SUB** and **DEC** instructions are used for subtracting from values in registers or memory. A related instruction, **NEG** (Negate), reverses the sign of a number.

The **DEC** instruction takes a single register or memory operand. The value of the operand is decremented. The value is treated as an unsigned integer, so the carry flag is not updated for signed borrows.

The **NEG** instruction takes a single register or memory operand. The sign of the value of the operand is reversed. The **NEG** instruction should only be used on signed numbers.

The **SUB** instruction subtracts the values given in the source operand from the value of the destination operand. The destination can be either a register or a 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 must be the same size.

A subtraction operation can be interpreted as subtraction of either signed numbers or of unsigned numbers. It is the programmer's responsibility to decide how the subtraction should be interpreted and to 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.

Example

```
        .DATA
mem8    DB      122
        .CODE
        .
        .
        .                ;                       signed  unsigned
        mov     al,95    ; Load register             95      95
        dec     al       ; Decrement             -    1  -    1
        sub     al,23    ; Subtract immediate    -   23  -   23
                         ;                         ----    ----
                         ;                           71      71
        sub     al,mem8  ; Subtract memory       - 122   - 122
                         ;                         ----    ----
                         ;                         - 51    205+sign

        mov     ah,119   ; Load register            119
        sub     al,ah    ;   and subtract         -- 51
                         ;                         ----
                         ;                           86+overflow
```

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.

Subtracting with Values in Multiple Registers

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 in multiple registers, the carry flag should be ignored for the least-significant portion, but taken into account for the most-significant portion. This can be done by using the **SUB** instruction for the least-significant portion and the **SBB** instruction for the most-significant portions.

You can subtract and borrow repeatedly inside a loop for calculations that require more than two registers. Use the **SBB** instruction in each iteration, but 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. You could also do the first subtraction outside the loop.

Example

```
         .DATA
mem32a   DD      316423
mem32b   DD      156739
         .CODE
         .
         .
         .
         mov     ax,WORD PTR mem32a[0]  ; Load mem32        316423
         mov     dx,WORD PTR mem32a[2]  ;   into DX:AX
         sub     ax,WORD PTR mem32b[0]  ; Subtract low      156739
         sbb     dx,WORD PTR mem32b[2]  ;   then high       ------
                                        ; Result in DX:AX   159684
```

Multiplying

The **MUL** and **IMUL** instructions are used to multiply numbers. The **MUL** instruction should be used for unsigned numbers; 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. Its contents 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.

Similarly, 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 **DX**, the least-significant bits will be in **AX**, 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 slower operations on 8086-family processors (especially the 8086 and 8088). Multiplying by certain common constants is often faster when done by shifting bits (see the section, "Multiplying and Dividing by Constants") or by using 80386 scaling (see the section, "Loading Near Pointers", in Chapter 14).

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
```

80186/286/386 Only

Starting with the 80186, the **IMUL** instruction has two additional syntaxes that allow for 16-bit multiples 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 operand and a word register as the destination operand. The product appears in the destination operand. The 16-bit product 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 operand must be a 16-bit register operand, the second a 16-bit memory operand, and the third 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
```

80386 Only

On the 80386, the **IMUL** instruction has an additional instruction that allows multiplication of a register value by a register or memory value.

Syntax

IMUL *register*,{*register* | *memory*}

15

The destination can be any 16-bit or 32-bit register. The source must be the same size as the destination.

Examples

```
imul    dx,ax       ; Multiply DX times AX
imul    ax,[bx]     ; Multiply AX by the value pointed to by BX
```

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; 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. The contents of this register will be destroyed by the operation. Specify the dividing number (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 least significant bits go in **AX**. The contents of 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 by using the **EDX**:**EAX** register pair.)

To divide a 16-bit number by a 16-bit number, you must first sign-extend or zero-extend (see the section, ''Converting between Data Sizes'', in Chapter 14) the dividend to 32 bits; then divide 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 terminates. To solve this problem, determine the value of the divisor before division occurred. If the value of the divisor is invalid, go to an error routine. For more information on interrupts, see the section, ''Using Interrupts,'' in Chapter 16.

Note

Division is one of the slower operations on 8086-family processors (especially the 8086 and 8088). Dividing by common constants that are powers of two is often faster when done by shifting bits, as described in the section, "Multiplying and Dividing by Constants."

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
```

Calculating with Binary Coded Decimals

The 8086-family processors provide several instructions for adjusting BCD numbers. The BCD format is seldom used for applications programming in assembly language. Programmers who wish to use BCD numbers usually use a high-level language. 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.

Unpacked BCD Numbers

Unpacked BCD numbers are made up of bytes containing a single decimal digit in the lower four 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:

```
mov     ax,9    ; Load 9
mov     bx,3    ;   and 3 as unpacked BCD
add     al,bl   ; Add 09h and 03h to get 0Ch
aaa             ; Adjust 0Ch in AL to 02h,
                ;   increment AH to 01h, set carry
                ; Result 12 unpacked BCD in AX
```

AAS Adjusts 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:

```
mov     ax,103h ; Load 13
mov     bx,4    ;   and 4 as unpacked BCD
sub     al,bl   ; Subtract 4 from 3 to get FFh (-1)
aas             ; Adjust 0FFh in AL to 9,
                ;   decrement AH to 0, set carry
                ; Result 9 unpacked BCD in AX
```

AAM Adjusts 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:

```
mov     ax,903h ; Load 9 and 3 as unpacked BCD
mul     ah      ; Multiply 9 and 3 to get 1Bh
aam             ; Adjust 1Bh in AL
                ;   to get 27 unpacked BCD in AX
```

15

AAD Adjusts 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 by 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:

```
mov     ax,205h ; Load 25
mov     bl,2    ;   and 2 as unpacked BCD
aad             ; Adjust 0205h in AX
                ;   to get 19h in AX
div     bl      ; Divide by 2 to get
                ;   quotient 0Ch in AL
                ;   remainder 1 in AH
aam             ; Adjust 0Ch in AL
                ;   to 12 unpacked BCD in AX
                ;   (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 their use for processing unpacked BCD numbers, the ASCII adjust instructions can be used in routines that convert between different number bases.

Packed BCD Numbers

Packed BCD numbers are made up of bytes containing two decimal digits: one in the upper four bits and one in the lower four 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.

Calculating with Binary Coded Decimals

Unlike the ASCII adjust instructions, the decimal adjust instructions never affect **AH**. The auxiliary carry flag is set if the digit in the lower four bits carries to or borrows from the digit in the upper four bits. The carry flag is set if the digit in the upper four bits needs to carry to or borrow from another byte.

The decimal adjust instructions are described below:

Instruction	**Description**
DAA	Adjusts after an addition operation. For example, to add 88 and 33, put 88 in **AL** and 33 in **BL** in packed BCD format. Then use the following lines to add them:

```
mov     ax,8833h;Load 88 and 33 as packed BCD
add     al,ah    ; Add 88 and 33 to get 0BBh
daa              ; Adjust 0BBh to 121 packed BCD:
                ;   1 in carry and 21 in AL
```

DAS	Adjusts after a subtraction operation. For example, to subtract 38 from 83, put 83 in **AL** and 38 in **BL** in packed BCD format. Then use the following lines to subtract them:

```
mov     ax,3883h;Load 83 and 38 as packed BCD
sub     al,ah    ; Subtract 38 from 83 to get 04Bh
das              ; Adjust 04Bh to 45 packed BCD:
                ;   0 in carry and 45 in AL
```

Multidigit BCD numbers are usually processed in loops. Each byte is processed and adjusted in turn.

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 15.1 shows a truth table for the logical operations.

Table 15.1

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 syntax of the **AND**, **OR**, and **XOR** instructions is the same. The only difference is the operation performed. For all instructions, the target value to be changed by the operation is placed in one operand. A mask showing the positions of bits to 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.

AND Operations

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*},{*register* | *memory* | *immediate*}

15

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 1

```
        mov     ax,035h         ; Load value                  00110101
        and     ax,0FBh         ; Mask off bit 2          AND 11111011
                                ;                             --------
                                ; Value is now 31h            00110001
        and     ax,0F8h         ; Mask off bits 2,1,0     AND 11111000
                                ;                             --------
                                ; Value is now 30h            00110000
```

Example 2

```
ans         db      ?
            mov     al,ans
            and     al,11011111b ; Convert to uppercase by clearing bit 5
            cmp     al,'Y'       ; Is it Y?
            je      yes          ; If so, do Yes stuff
            .                    ;   else do No stuff
            .
yes:        .
```

Example 2 illustrates how to use the **AND** instruction to convert a character to uppercase. If the character is already uppercase, the **AND** instruction has no effect, since bit 5 is always clear in uppercase letters. If the character is lowercase, clearing bit 5 converts it to uppercase.

OR Operations

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.

15

Syntax

OR {*register* | *memory*},{*register* | *memory* | *immediate*}

The **OR** instruction can be used to set 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 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
        mov     ax,035h     ; Move value to register        00110101
        or      ax,08h      ; Mask on bit 3              OR 00001000
                            ;                               --------
                            ; Value is now 3Dh              00111101
        or      ax,07h      ; 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
```

XOR Operations

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.

5

Syntax

XOR {*register* | *memory*},{*register* | *memory* | *immediate*}

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,08h      ; Mask on bit 3             XOR 00001000
                            ;                               --------
                            ; Value is now 3Dh              00111101
        xor     ax,07h      ; 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.

NOT Operations

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. Use the **NOT** instruction if the value of the mask is not known until run time; use the NOT operator (see the section, ''Bitwise Logical Operators'', in Chapter 8) if the mask is a constant.

Example

```
             .DATA
masker       DB       00010000b  ; Value may change at run time
             .CODE
             .
             .
             .
             mov      ax,0D743h  ; Load 0D7h to AH; 43h to AL  01000011
             or       al,masker  ; Turn on bit 4 in AL      OR 00010000
                                 ;                             --------
                                 ; Result is 53h               01010011

             not      masker     ; Reverse sense of mask       11101111
             and      ah,masker  ; Turn off bit 4 in AH    AND 11010111
                                 ;                             --------
                                 ; Result is 0C7h              11000111
```

15

Scanning for Set Bits

80386 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 0.

Syntax

> **BSF** *register*,{*register* | *memory*}
> **BSR** *register*,{*register* | *memory*}

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
            push    ds                  ; Load segment of bitfield
            pop     es                  ;   into ES
            mov     cx,widfield         ; Load maximum count
            xor     eax,eax             ; Set search value to 0
            mov     di,OFFSET bitfield  ; Load bitfield address
            repe    scasd               ; Find first nonzero bit
            jecxz   none                ; If none found, get out
            sub     di,4                ; Point back to doubleword
            mov     eax,[di]            ; Else load first nonzero
            bsr     ecx,eax             ; Find first set bit
            .                           ; ECX now contains bit position
            .                           ; DI points to doubleword
none:       .
```

Scanning for Set Bits

This example scans a large bit field. 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 17, ''Processing Strings,'' for more information on the string instructions used in this example.

15

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 of the operand 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 value of the 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 15-1 illustrates the eight variations of shift and rotate instructions for 8-bit operands. Notice that **SHL** and **SAL** are exactly the same.

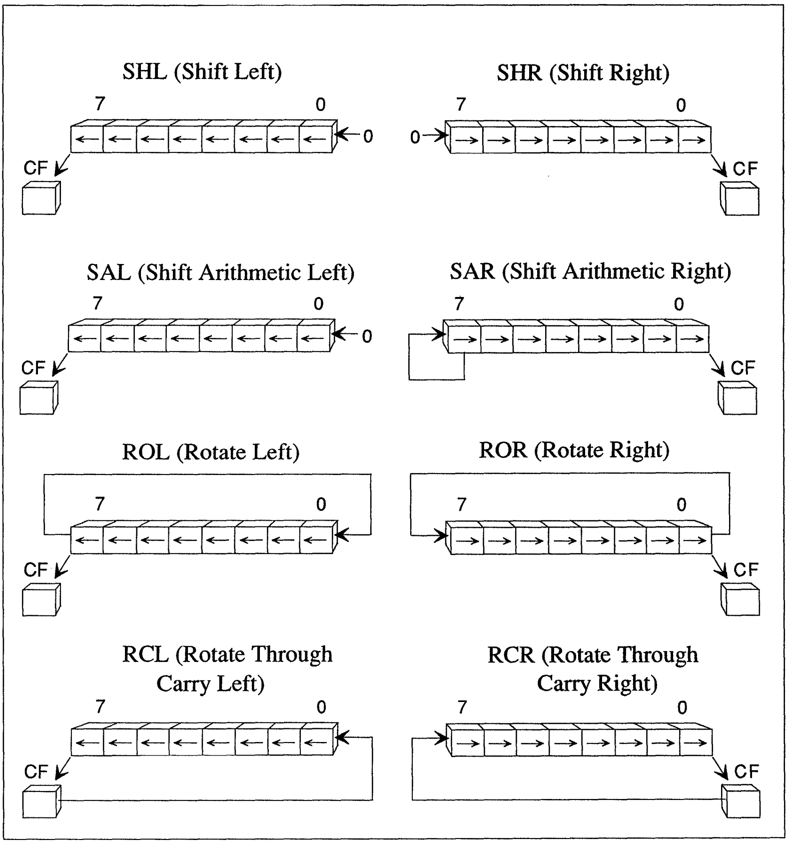


Figure 15-1 Shifts and Rotates

15

Syntax

> **SHL** {*register* | *memory*},{**CL** | **1**}
> **SHR** {*register* | *memory*},{**CL** | **1**}
> **SAL** {*register* | *memory*},{**CL** | **1**}
> **SAR** {*register* | *memory*},{**CL** | **1**}
> **ROL** {*register* | *memory*},{**CL** | **1**}
> **ROR** {*register* | *memory*},{**CL** | **1**}
> **RCL** {*register* | *memory*},{**CL** | **1**}
> **RCR** {*register* | *memory*},{**CL** | **1**}

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.

Note

80186/286/386 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, as shown below:

```
        shr     bx,4            ; 9 clocks, 3 bytes on 80286
```

The following statements are equivalent if the program must run on the 8088 or 8086:

```
        mov     cl,4            ; 2 clocks, 3 bytes on 80286
        shr     bx,cl           ; 9 clocks, 2 bytes on 80286
                                ;11 clocks, 5 bytes
```

Multiplying and Dividing by Constants

Shifting right by one has the effect of dividing by two; 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 four, shifting left three times multiplies by eight, and so on.

SHR is used to divide unsigned numbers. **SAR** can be used to divide signed numbers, but **SAR** rounds negative numbers down—**IDIV** always rounds up. Code that divides by using **SAR** must adjust for this difference. Multiplication by shifting is the same for signed and unsigned numbers, so either **SAL** or **SHL** can be used. Both instructions do the same operation.

Since the multiply and divide instructions are the slowest on the 8088 and 8086 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 statements take 4 clocks:

```
        xor     ah,ah           ; Clear AH
        shl     ax,1            ; Multiply byte in AL by 2
```

The following statements have the same effect, but take between 74 and 81 clocks on the 8088 or 8086:

```
        mov     bl,2            ; Multiply byte in AL by 2
        mul     bl
```

The same statements take 15 clocks on the 80286 or between 11 and 16 clocks on the 80386.

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      ; Factor must be unsigned
            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    ; Dividend must be unsigned
            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
```

Moving Bits to the Least-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, and then shifting them into the least-significant positions. After the arithmetic operation is done, the bits are shifted back to the original position and merged with the original bits by using **OR**. For an example of this operation, see the section, ''Defining and Redefining Interrupt Routines,'' in Chapter 16.

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

```
                .DATA
masker          DB      00000010b   ; Mask that may change at run time
                .CODE
                .
                .
                .
                mov     cl,2        ; Rotate two at a time
                mov     bl,57h      ; Load value to be changed     01010111b
                rol     masker,cl   ; Rotate two to left           00001000b
                or      bl,masker   ; Turn on masked values        ---------
                                    ; New value is 05Fh            01011111b
                rol     masker,cl   ; Rotate two more              00100000b
                or      bl,masker   ; Turn on masked values        ---------
                                    ; New value is 07Fh            01111111b
```

This technique is useful only if the mask value is unknown until run time.

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
              .
              .                              ; Divide 32-bit unsigned by 16
              .
              mov      cx,4                  ; Shift right 4        500000
again:        shr      WORD PTR mem32[2],1   ; Shift into carry  DIV    16
              rcr      WORD PTR mem32[0],1   ; Rotate carry in      ------
              loop     again                 ;                       31250
```

Shifting Multiple Bits

80386 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* | *memory*},*register*,{**CL** | *immediate*}
> **SHLD** {*register* | *memory*},*register*,{**CL** | *immediate*}

These instructions take three operands. The first (leftmost) contains the value to be shifted. It must be a 16-bit or 32-bit register or memory operand. 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,9C00h ; Load    BX=                  10011100 00000000
shld    ax,bx,7  ; Shift 7    01111001 0        <- 7
                 ;                  1001110 <- 7
                 ;            -----------------
                 ;         AX=01111001 01001110 (794Eh)
```

Chapter 16

Controlling Program Flow

Introduction

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 you how to use these instructions and how to test conditions for the instructions that change program flow conditionally.

16

Jumping

Jumps are the most direct method of changing program control from one location 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.

Jumps can be short, near, or far. Near and short jumps are handled automatically, though **masm** may not always generate the most efficient code if the label being jumped to is a forward reference. The size and control of jumps is discussed in the section, ''Forward References to Labels,'' in Chapter 8.

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 short, near, or far. For more information on specifying the distance for conditional jumps, see the section, ''Forward References to Labels,'' in Chapter 8.

If a conditional jump must be greater than 128 bytes, the construction must be reorganized (except on the 80386). This can be done by reversing the sense of the conditional jump and adding an unconditional 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
```

An unconditional jump 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 GOTO** statements, or Pascal **case** statements.

16

Example 2

```
            .CODE
            .
            .
            .
            jmp     process         ; Jump over data
ctl_tbl     LABEL   WORD            ;   (required in overlay procedures)
            DW      extended        ; Null key (extended code)
            DW      ctrla           ; Address of CONTROL-A key routine
            DW      ctrlb           ; Address of CONTROL-B key routine
process:    .                      ; Get a key into AL
            .
            .
            cbw                     ; Convert AL to AX
            mov     bx,ax           ; Copy
            shl     bx,1            ; Convert to address

            jmp     ctl_tbl[bx]     ; Jump to key routine

extended:   .                      ; Get second key of extended
            .
            .                       ; Use another jump table
            .                       ;   for extended 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. Notice that the jump table is placed in the code segment. This technique is optional in stand-alone assembler programs, but it may be required for procedures called from some languages.

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, and then jump if the condition is true or continue if it is false.

Syntax

J*condition label*

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. For information on arranging longer conditional jumps, see the section, "Jumping Unconditionally."

80386 Only

Conditional jumps to forward references are near by default under the 80386 processor. But you can use the **SHORT** operator to specify short jumps. For information specifying the size of jumps, see the section, "Forward References to Labels," in Chapter 8.

Conditional-jump instructions (except **JCXZ**) use the status of one or more flags as their condition. 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.

Comparing and Jumping

The **CMP** instruction is specifically designed to test for conditional jumps. It does not change the destination operand, so it can be used to compare two values without changing either of them. Instructions that change operands (such as **SUB** or **AND**) can also be used to test conditions.

The **CMP** instruction compares two operands and sets flags based on the result. It is used to test the following relationships: equal; not equal; greater than; less than; greater than or equal; or less than or equal.

Syntax

CMP {*register* | *memory*},{*register* | *memory* | *immediate*}

The destination operand can be memory or register. The source operand can be immediate, memory, or register. However, they cannot both be memory operands.

Jumping

The jump instructions that can be used with **CMP** are made up of mnemonic letters combined to indicate the type of jump. The letters are shown below:

Letter	Meaning
J	Jump
G	Greater than (for signed comparisons)
L	Less than (for signed comparisons)
A	Above (for unsigned comparisons)
B	Below (for unsigned comparisons)
E	Equal
N	Not

The mnemonic names always refer to the relationship that the first operand of the **CMP** instruction has to the second operand of the **CMP** instruction. For instance, **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.

6

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 by using the following syntax:

CMP *value1,value2*
J*relationship truelabel*
.
.
.
truelabel:

Table 16.1 lists conditional-jump instructions for each *relationship* and shows the flags that are tested in order to see if *relationship* is true.

Table 16.1

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	≠	**JNE**	**ZF=0**	**JNE**	**ZF=0**
Greater than	>	**JG** or **JNLE**	**ZF=0** and **SF=OF**	**JA** or **JNBE**	**CF=0** and **ZF=0**
Less than or equal	≤	**JLE** or **JNG**	**ZF=1** or **SF≠OF**	**JBE** or **JNA**	**CF=1** or **ZF=1**
Less than	<	**JL** or **JNGE**	**SF≠OF**	**JB** or **JNAE**	**CF=1**
Greater than or equal	≥	**JGE** or **JNL**	**SF=OF**	**JAE** or **JNB**	**CF=0**

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     further     ; Finished
less:     mov     dx,30       ; Then set DX to 30
further:
```

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 -20, then make DX 20, else make DX 30

            cmp     cx,-20      ; If signed CX is smaller than -20
            jnl     notless     ;    else do stuff at "notless"
            mov     dx,30       ; Then set DX to 30
            jmp     continue    ; Finished
notless:    mov     dx,20       ; Else set DX to 20
continue:
```

The then-if-else format shown in Example 3 is often more efficient. Do the work for the most likely case, and then compare for the opposite condition. If the condition is true, you are finished.

Example 3

```
; DX is 20, unless CX is less than -20, then make DX 30

            mov     dx,20       ; DX is 20
            cmp     cx,-20      ; If signed CX is greater than -20
            jge     greatequ    ;    Then done
            mov     dx,30       ; Else set DX to 30
greatequ:
```

This example avoids the unconditional jump used in Examples 1 and 2 and thus is faster even if the less likely condition is true.

Jumping Based on Flag Status

The **CMP** instruction is the most mnemonic way to set the flags 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 on relationships of operands. Some of these instructions have the same effect as instructions listed in Table 16.1.

Instruction	**Action**
JO	Jumps if the overflow flag is set
JNO	Jumps if the overflow flag is clear
JC	Jumps if the carry flag is set (same as **JB**)

JNC	Jumps if the carry flag is clear (same as **JAE**)
JZ	Jumps if the zero flag is set (same as **JE**)
JNZ	Jumps if the zero flag is clear (same as **JNE**)
JS	Jumps if the sign flag is set
JNS	Jumps if the sign flag is clear
JP	Jumps if the parity flag is set
JNP	Jumps if the parity flag is clear
JPE	Jumps if parity is even (parity flag set)
JPO	Jumps if parity is odd (parity flag clear)
JCXZ	Jumps if **CX** is 0

Notice that the **JCXZ** is the only conditional jump based on the condition of a register (**CX**) rather than flags. Since **JCXZ** is usually used with loop instructions, it is discussed in more detail in the section, ''Setting Bytes Conditionally.''

Example 1

```
            add     ax,bx       ; Add two values
            jo      overflow    ; If value too large, adjust
            .
            .
            .
overflow:                       ; Adjustment routine here
```

Example 2

```
         sub    ax,dx      ; Subtract
         jnz    go_on      ; If the result is not zero, continue
         call   zhandler   ;    else do special case
go_on:
```

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*},{*register* | *memory* | *immediate*}

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 the same as the **AND** instruction, except that neither operand is changed. If the result of the operation is 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 the same and can be used if you find them more mnemonic.

Example

```
            .DATA
bits        DB      ?
            .CODE
            .
            .
            .
; If bit 2 or bit 4 is set, then call taska

                                ; Assume "bits" is 0D3h       11010011
            test    bits,10100b; If 2 or 4 is set         AND 00010100
            jz      go_on       ; Else continue               --------
            call    taska       ;    Then call taska          00010000
go_on:                          ; Jump not taken
            .
            .
            .
; If bits 2 and 4 are clear, then call taskb

                                ; Assume "bits" is 0E9h       11101001
            test    bits,10100b; If 2 and 4 are clear     AND 00010100
            jnz     next        ; Else continue               --------
            call    taskb       ;    Then call taskb          00000000
next:                           ; Jump not taken
```

Testing and Setting Bits

80386 Only

The 80386 processor has bit test and set instructions. 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 **BX** points to a bit field and **CX** contains 4 in the following statements:

```
bt      [bx],cx      ; Put bit 4 of bit field
                     ;   pointed to by BX in carry
jc      somewhere    ; Jump if carry set
```

The same thing could be done less efficiently on other 8086-family processors with the following statements:

```
mov     ax,[bx]     ; Load value pointed to by BX
shr     ax,cl       ; Shift bit 4 to first position
test    ax,1        ; See if bit is set
jnz     somewhere   ; Jump if it is
```

This instruction is only useful if the source operand is not known until run time. If the source operand is a constant, the **TEST** instruction (see the section, ''Testing Bits and Jumping'', in this chapter) is more efficient.

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 **BX** points to a bit field and **CX** contains 4 in the following statements:

```
btc     [bx],cx     ; Put bit 4 of bit field in carry
                    ;   and toggle bit 4
jc      somewhere   ; Jump if carry set
```

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 **BX** points to a bit field and **CX** contains 4 in the following statements:

```
btr     [bx],cx     ; Put bit 4 of bit field in carry
                    ;   and clear bit 4
jc      somewhere   ; Jump if carry set
```

BTS 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 **BX** points to a bit field and **CX** contains 4 in the following statements:

```
bts     [bx],cx    ; Put bit 4 of bit field in carry
                   ;   and set bit 4
jc      somewhere  ; Jump if carry was set
```

Example

```
            .DATA
flag        RECORD  a:3=0,b:2=0,c:1=0,d:2=0,e:1=0,f:1=0
error       flag    <>
            .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

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 *label*
LOOPE *label*
LOOPZ *label*
LOOPNE *label*
LOOPNZ *label*
JCXZ *label*

The **LOOP** instruction is used for loops with a set number of iterations. For example, it can be used in constructions similar to the "for" loops of BASIC, C, and Pascal, and 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 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, C, and Pascal; the "repeat" loops of Pascal; and 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 do not want a maximum count.

The **JCXZ** instruction (and its 32-bit 80386 extension, **JECXZ**) are often used in loop structures. For example, it may be used in loops that check a condition at the start of the loop rather than at the end. Unlike the loop instruction, **JCXZ** does not decrement **CX**, so the programmer must use another statement to decrement the count.

80386 Only

Unlike conditional-jump instructions, which can jump to either a near or a short label under the 80386, the loop instructions, **JCXZ** instruction, and **JECXZ** instruction always jump to a short label.

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
            cmp     cx,0
            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 often required for loops that test several conditions.

If the counter in **CX** is variable because of 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 decremented to -1 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

```
; 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
```

16

Setting Bytes Conditionally

80386 Only

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. For more information on how flags are tested for conditional jumps, see the section, ''Jumping Unconditionally.''

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 operand is set to 1. Otherwise the operand is set to 0.

Conditional-set instructions are usually preceded 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
amount      DW      ?           ; Size variable to be set at run time
            .CODE
            .
            .                   ; Size is set
            .
; bigflag = amount > 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 make the code compatible with such compilers, you should negate the value after setting it. For example, add the following line to the previous example:

```
        neg     bigflag     ; Negate result
```

This statement would be necessary for BASIC, since the expression *BIGFLAG=SIZE>1000* evaluates to -1. It would not be necessary for C, since the expression *bigflag=size>1000* evaluates to 1.

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 a program without using the same code in each place. Assembly-language procedures are comparable to functions in C; subprograms, functions, and subroutines in BASIC; procedures and functions in Pascal; or routines and functions 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 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 the section, ''Defining Procedures.''

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.

16

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 operand. 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 the section, ''Comparing and Jumping'', in this chapter.

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. You must give the type of far calls to forward-referenced labels using the **FAR** type specifier and the **PTR** operator. For example, use the following statement to make a far call to a label that has not been earlier defined or declared external in the source code:

```
call    FAR PTR task
```

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**]
statements
RET [*constant*]
label **ENDP**

Procedures are normally defined by 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** instruction automatically matches the size defined by the **PROC** directive.

16

Syntax 2

label:
statements
RETN [*constant*]

Syntax 3

label **LABEL FAR**
statements
RETF [*constant*]

Starting with Version 5.0 of the Macro Assembler, the **RET** instruction can be extended to **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) allows a constant 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 shown in the example in the section, "Using Local Variables."

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 be meaningless, but the stack status will be incorrect, causing the program to fail.

Passing Arguments on the Stack

Procedure arguments can be passed in various ways. For example, values can be passed to a procedure in registers or in variables. However, the most common method of passing arguments is to use the stack. Microsoft languages have a specific convention for doing this.

The arguments are pushed onto the stack before the call. After the call, the procedure retrieves and processes them. At the end of the procedure, the stack is adjusted to account for the arguments.

Although the same basic method is used for all Microsoft high-level languages, the details vary. For instance, in some languages, pointers to the arguments are passed to the procedure; 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. For details on calling conventions for each Microsoft language, see Appendix D, ''Segment Names for High-Level Languages.''

Example

```
; 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
            pop     bp          ; Restore BP
            ret                 ; Return result in AX
addup       ENDP
```

The example shows one method of passing arguments to a procedure. This method is similar to the way procedures are called in C. Figure 16-1 shows the stack condition at key points in the process.

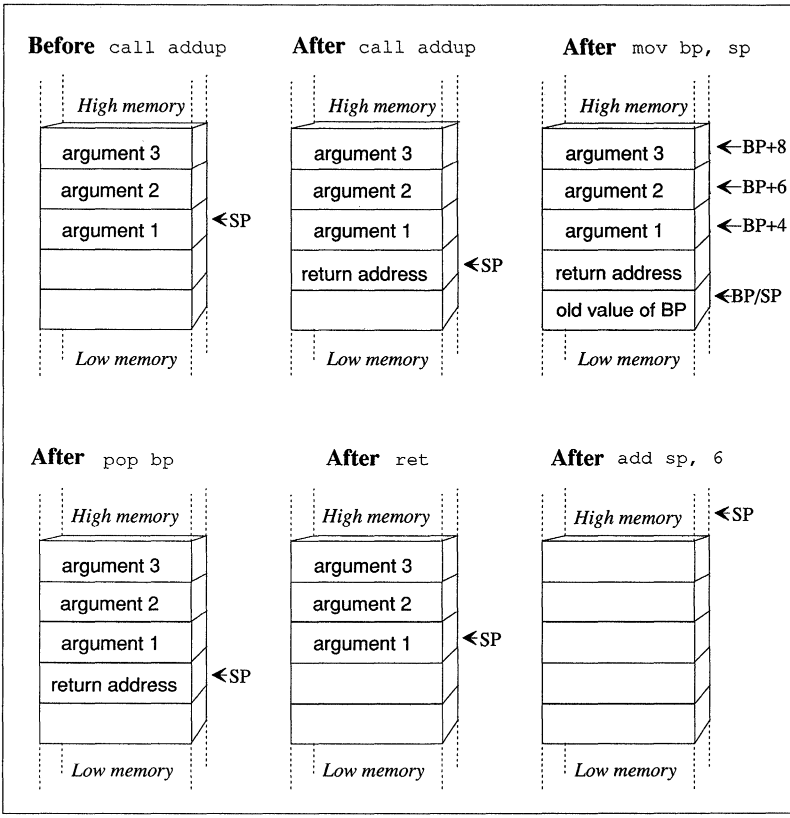


Figure 16-1 Procedure Arguments on the Stack

Using Local Variables

In high-level languages, local variables are variables known only within a procedure. In Microsoft languages, these variables are usually 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 7, "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 its position in the stack. At the end of the procedure, the stack pointer is restored to restore the memory used by local variables.

Example

```
          push    ax          ; Push one argument
          call    task        ; Call
          .
          .
          .
arg       EQU     <[bp+4]>    ; Name for argument
loc       EQU     <[bp-2]>    ; Name for local 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
          .
          .
          .
          mov     loc,3       ; Initialize local variable
          add     ax,loc      ; Add local variable to AX
          sub     arg,ax      ; Subtract local from argument
          .                   ; Use "loc" and "arg" in other operations
          .
          .
          mov     sp,bp       ; Adjust for stack variable
          pop     bp          ; Restore base
          ret                 ; Return result in AX
task      ENDP
```

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. Notice that the instruction *mov sp,bp* is given at the end to restore the original value of **SP**. The statement is only required if the value of **SP** is changed inside the procedure (usually by allocating local variables). The argument passed to the procedure is returned with the **RET** instruction. Contrast this to the example in the section, ''Passing Arguments on the Stack,'' in which the calling code adjusts for the argument. Figure 16.2 shows the state of the stack at key points in the process.

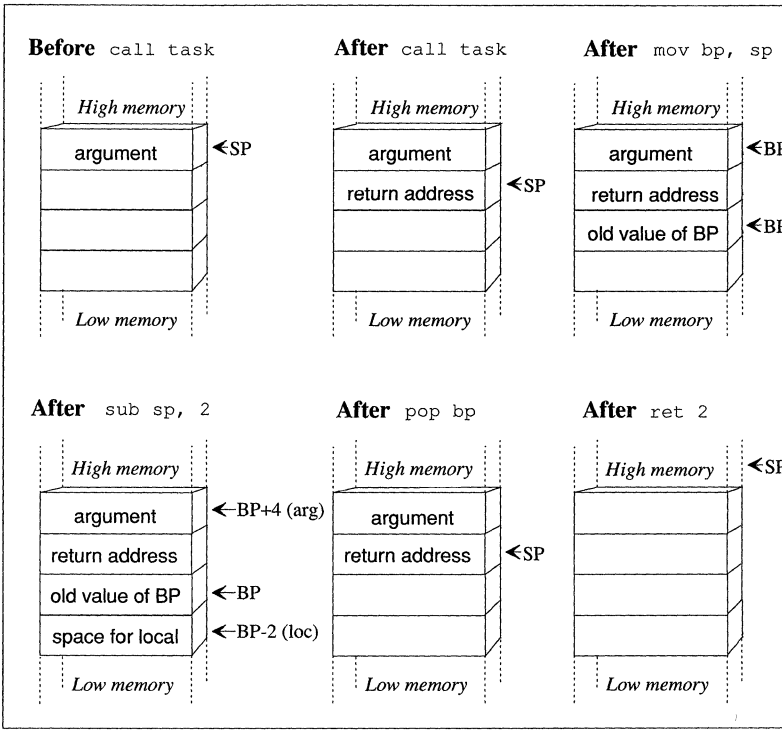


Figure 16-2 Local Variables on the Stack

Setting Up Stack Frames

80186/286/386 Only

Starting with the 80186 processor, the **ENTER** and **LEAVE** instructions are provided for setting up a stack frame. These instructions do the same thing as the multiple instructions at the start and end of procedures in the Microsoft calling conventions (see the example in the section, "Passing Arguments on the Stack").

Syntax

> **ENTER** *framesize,nestinglevel*
> *statements*
> **LEAVE**

The **ENTER** instruction takes two constant operands. The *framesize* (a 16-bit constant) specifies how many bytes to reserve for local variables. The *nestinglevel* (an 8-bit constant) specifies the level at which the procedure is nested. This operand should always be 0 when writing procedures for BASIC, C, and FORTRAN. The *nestinglevel* can be greater than 0 with 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 by restoring **BP** and **SP** to their values before the procedure call.

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 same 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.

Using Interrupts

Interrupts are a special form of routines that are called by number instead of by address. 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 by the processor, the BIOS, or the operating system.

The programmer can call existing interrupts with the **INT** instruction. Interrupt routines can also be defined or redefined to be called later. For example, an interrupt routine that is called automatically by a hardware device can be redefined so that its action is different.

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 the instruction is called, the processor takes the following six 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 four bytes (two segment and two offset) for each interrupt. Thus the address of an interrupt routine can be found by multiplying the number of the interrupt by four.

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 16.3 shows the status of the stack immediately after the **INT** instruction has been executed.

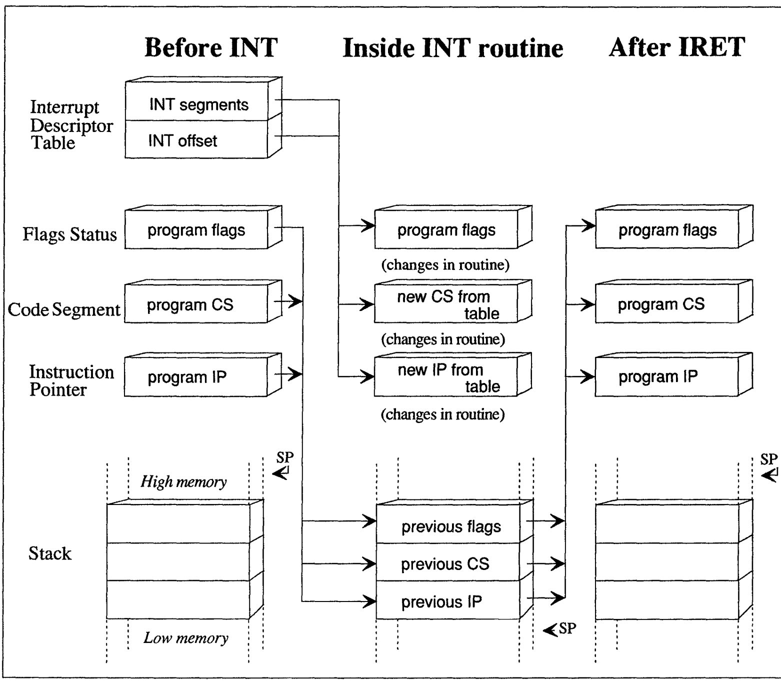


Figure 16-3 Operation of Interrupts

The **INTO** (Interrupt on Overflow) instruction is a variation of the **INT** instruction. It calls interrupt 04h if called when the overflow flag is set. By default, interrupt 4 sends a **SIGSEGV** to the process. Using **INTO** is an alternative to using **JO** (Jump on Overflow) to jump to an overflow routine. The section, "Defining and Redefining Interrupt Routines," gives an example of this.

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. Interrupts received while interrupt processing was turned off by **CLI** are saved and executed when **STI** turns interrupts back on.

Defining and Redefining Interrupt Routines

You can write your own interrupt routines, either to replace an existing routine or to use an undefined interrupt number.

Syntax

> *label* **PROC FAR**
> *statements*
> **IRET**
> *label* **ENDP**

An interrupt routine can be written like a procedure by using the **PROC** and **ENDP** directives. The only differences are that the routine should always be defined as far and the routine should be terminated by an **IRET** instruction instead of a **RET** instruction.

Interrupt routines can be part of device drivers. Writing interrupt routines is usually a systems task.

80386 Only

The **INT** instruction automatically pushes 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 to form **IRETD**.

Checking Memory Ranges

80186/286/386 Only

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. **BOUND** is a conditional interrupt instruction like **INTO**. If the condition is not met (the index is out of range), an interrupt 5 is executed.

Syntax

BOUND *register16,memory32*
BOUND *register32,memory64* (80386 Only)

To use it for this purpose, the starting and ending values of the array must be stored as 16-bit values in the low and high words of a doubleword memory operand. This operand is given as the source operand. The index value to be checked is given as the destination operand. If the index value is out of range, the instruction issues interrupt 5. This means that the operating system or the program must provide an interrupt routine for interrupt 5. Part 1, "Using Assembler Programs does not provide an interrupt routine for interrupt 5, so you must write your own. For more information, see the section, "Using Interrupts."

16

Example

```
        .DATA
bottom  EQU     0
top     EQU     19
dbounds LABEL   DWORD           ; Allocate boundaries
wbounds DW      bottom,top      ;   initialized to bounds
array   DB      top+1 DUP (?)   ; Allocate array
        .CODE
        .
        .
        .                       ; Assume index in DI
        bound   di,dbounds      ; Check to see if it is in range
                                ;   if out of range, interrupt 5
        mov     dx,array[di]    ; If in range, use it
```

Checking Memory Ranges

80386 Only

The 80386 can optionally check larger arrays. The destination operand can be a 32-bit register and the source can be a 64-bit memory operand containing 32-bit starting and ending values.

16

Chapter 17

Processing Strings

Introduction

The 8086-family processors have a full set of instructions for manipulating strings. In the discussion of 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 double-words on the 80386).

The following instructions are provided for 8086-family string functions:

Instruction	Description
MOVS	Moves string from one location to another
SCAS	Scans string for specified values
CMPS	Compares values in one string with values in another
LODS	Loads values from a string to accumulator register
STOS	Stores 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 instructions 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 and then tells you how to use each instruction.

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 five 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.

2. Load the number of iterations for the string instruction into the **CX** register. For instance, if you want to process a 100-byte string, load 100. If a string instruction will be terminated conditionally, load the maximum number of iterations that can 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 source (shown in Table 17.1 below). 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 17.1 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*[ES:[*destination*,]][[*segmentregister*:]*source*]

The string instruction can be given with the source and/or destination as operands. 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 is 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 only)

The letter **B** or **W** appended to the string instruction indicates bytes or words; the letter **D** indicates doublewords on the 80386. With a letter appended to a string instruction, no operand is allowed.

For instance, **MOVS** can be given with byte operands to move bytes or with word operands to move words. As an alternative, **MOVSB** can be given with no operands to move bytes or **MOVSW** can be given with no operands to move words.

Note

Instructions that specify the size in the name never accept operands. Therefore, the following statement is illegal:

```
lodsb   es:0              ; Illegal - no operand allowed
```

Instead, the statement must be coded as shown below:

```
lods    BYTE PTR es:0     ; Legal - use type specifier
```

If a repeat prefix is used, it can be one of the following instructions:

Instruction	Description
REP	Repeats for a specified number of iterations. The number is given in **CX**.
REPE or **REPZ**	Repeats while equal. The maximum number of iterations should be specified in **CX**.
REPNE or **REPNZ**	Repeats 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. The prefixes ending with **E** are used in syntax listings and tables in the rest of this chapter.

Table 17.1 lists each string instruction with the type of repeat prefix it uses and whether the instruction works on a source, a destination, or both.

Table 17.1

Requirements for String Instructions

Instruction	Repeat Prefix	Source/Destination	Register Pair
MOVS	**REP**	Both	**DS:SI, ES:DI**
SCAS	**REPE/REPNE**	Destination	**ES:DI**
CMPS	**REPE/REPNE**	Both	**ES:DI, DS:SI**
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. Checks the **CX** registers and exits from the string instruction if **CX** is 0.

2. Performs the string operation once.

3. Increases **SI** and/or **DI** if the direction flag is cleared. Decreases **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. Decrements **CX** (no flags are modified).

5. If the string instruction is **SCAS** or **CMPS**, checks the zero flag and exits if the repeat condition is false—that is, if the flag is set with **REPE** or **REPZ** or if it is clear with **REPNE** or **REPNZ**.

6. Goes to the next iteration (step 1).

Although string instructions (except **LODS**) are most often 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, you must decrement the **CX** register and set up a loop for the repeated action.

Note

Although you can use a segment override on the source operand, a segment override combined with a repeat prefix can cause problems in certain situations on all processors except the 80386. If an interrupt occurs during the string operation, the segment override is lost and the rest of the string operation processes incorrectly. Segment overrides can be used safely when interrupts are turned off, when a string instruction is used without a segment override, or when a 80386 processor is used.

Moving Strings

The **MOVS** instruction is used to move data from one area of memory to another.

Syntax

> **[REP MOVS [ES:]***destination*,[*segmentregister*:]*source*
> [REP] **MOVSB**
> **[REP] MOVSW**
> **[REP] MOVSD** (80386 only)

To move the data, load the count and the source and destination addresses into the appropriate registers, as discussed in the section, "Setting Up String Operations." 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,@data           ; 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
          mov     di,OFFSET destin   ; Load address of destination
          rep     movsb              ; Move 100 bytes
```

Example 1 shows how to move a string by 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     al,es:[si]         ; Get a byte from source
         mov     [di],al            ; Put it in destination
         inc     si                 ; Increment source pointer
         inc     di                 ; Increment destination pointer
         loop    repeat             ; Do it again
```

Both examples illustrate how to move byte strings in a small-model program in which **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 starting address of its segment into **ES**. 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
```

Searching Strings

The **SCAS** instruction is used to scan a string for a specified value.

Syntax

> [**REPE** | **REPNE**] **SCAS** [**ES:**]*destination*
> [**REPE** | **REPNE**] **SCASB**
> [**REPE** | **REPNE**] **SCASW**
> [**REPE** | **REPNE**] **SCASD** (80386 only)

SCAS and its variations work only 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** with 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 the value immediately after the first occurrence. You can decrement **DI** to make it point to the first matching value.

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              ; Length of string
pstring     DD      string                ; Far pointer to string
            .CODE
            .
            .
            .
            cld                           ; Work upward
            mov     cx,lstring            ; Load length of string
            les     di,pstring            ; Load address of string
            mov     al,'z'                ; Load character to find
            repne   scasb                 ; Search
            jcxz    notfound              ; CX is 0 if not found
            .                             ; ES:DI points to character
            .                             ;   after first 'z'
            .
notfound:                                 ; Special case for not found
```

This example assumes that **ES** is not the same as **DS**, but that the address of the string is stored in a pointer variable. The **LES** instruction is used to load the far address of the string into **ES:DI**.

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 [segment register:]*source*,**[ES:]**,destination
[REPE | REPNE] CMPSB
[REPE | REPNE] CMPSW
[REPE | REPNE] CMPSD (80386 only)

The count and the addresses of the strings are loaded into registers, as described in the section, "Setting Up String Operations." Either string can be considered the destination or source string unless a segment override is used. Notice that unlike other instructions, **CMPS** requires the source to be on the left.

The **CMPS** instruction works by comparing in turn each value pointed to by **DI** with the value pointed to by **SI**. If the values are the same, the zero flag is set. Thus the instruction makes sense only 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 after the first match in the respective strings. You can decrement **DI** or **SI** to point to the match.

17

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 first 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 in which 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,@data           ; Load data segment
             mov     ds,ax              ;   into DS
             mov     ax,@fardata        ; Load far data segment
             mov     es,ax              ;   into ES
             .
             .
             .
             cld                        ; Work upward
             mov     cx,lstring         ; Load length of string
             mov     si,OFFSET string1  ; Load offset of string1
             mov     di,OFFSET string2  ; Load offset of string2
             repe    cmpsb              ; Compare
             jcxz    allmatch           ; CX is 0 if no nonmatch
             dec     si                 ; Adjust to point to nonmatch
             dec     di                 ;   in each string
             .
             .
allmatch:    .                          ; Special case for all match
```

This example assumes that the strings are in different segments. Both segments must be initialized to the appropriate segment register.

Filling Strings

The **STOS** instruction is used to store a specified value in each position of a string.

Syntax

> [**REP**] **STOS** [**ES:**]*destination*
> [**REP**] **STOSB**
> [**REP**] **STOSW**
> [**REP**] STOSD (80386 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 the section, "Setting Up String Operations." 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.

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 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 move a value repeatedly 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

```
        .DATA
stuff   DB      0,1,2,3,4,5,6,7,8,9
        .CODE
        .
        .
        .
        cld                             ; Work upward
        mov     cx,10                   ; Load length
        mov     si,OFFSET stuff         ; Load offset of source
get:    lodsb                           ; Get a character
        add     al,48                   ; Convert to ASCII
        mov     dl,al                   ; Move to DL
```

This example loads, processes, and displays each byte in a string of bytes.

Transferring Strings to and from Ports

80186/286/386 Only

The **INS** instruction reads a string from a port to memory, and the **OUTS** instruction writes a string from memory to a port.

Syntax

OUTS DX,[*segmentregister*:]*source*
OUTSB
OUTSW
OUTSD (80386 only)

INS [ES:]*destination*,**DX**
INSB
INSW
INSD (80386 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 instance, 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
```

Note

Under Part 1, ''Using Assembler Programs and other protected-mode operating systems, **IN** and **OUT** are privileged instructions and can only be used in privileged mode.

17

Chapter 18

Calculating with a Math Coprocessor

Introduction

The 8087-family coprocessors are used to do fast mathematical calculations. When used with real numbers, packed BCD numbers, or long integers, they 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 taken is from an applications standpoint. Features that would be used by systems programmers (such the flags used when writing exception handlers) are not explained. This 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 will not need to use a feature unless you understand the mathematics involved. For example, you need to understand logarithms to use the **FYL2X** and **FYL2XP1** instructions.

Coprocessor Architecture

The math coprocessor works simultaneously with the main processor. However, since the coprocessor cannot handle device input or output, most data originates in the main processor.

The main processor and the coprocessor have their own registers, which are completely separate and inaccessible to the other. They exchange data through memory, since memory is available to both.

Ordinarily you follow these three steps when using the coprocessor:

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 is explained in the section, "Transferring Data."

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; register 7 is the bottom. The syntax for specifying registers is shown below:

18

ST[(*number*)]

The *number* must be a digit between 0 and 7. If *number* is omitted, register 0 (top of stack) is assumed.

All coprocessor data are stored in registers in the temporary-real format. This is the 10-byte IEEE format described in the section, "Real-Number Variables", in Chapter 5. The registers and the register format are shown in Figure 18-1.

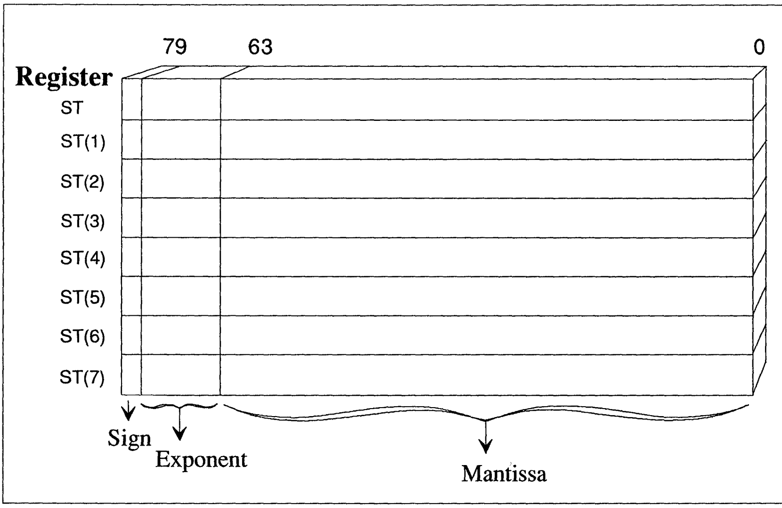


Figure 18-1 Coprocessor Data Registers

Internally, all calculations are done on numbers of the same type. Since temporary-real numbers have the greatest precision, lower-precision 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.

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. Errors and exceptions are reported in the status-word register. However, the coprocessor already has a default system for handling exceptions. Applications programmers can usually accept the defaults. Systems programmers may want to use the status-word and control-word registers when writing exception handlers, but such problems are beyond the scope of this manual.

Figure 18-2 shows the overall layout of the control registers, including the control word, status word, tag word, instruction pointer, and operand pointer. The format of each of 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-word register. These bits are explained in the section, "Controlling Program Flow."

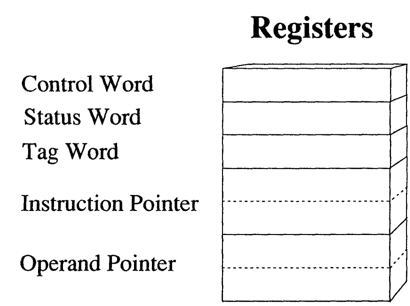


Figure 18-2 Coprocessor Control Registers

Emulation

If you have a Microsoft high-level language that supports floating-point emulation, you can write assembly-language procedures that use the emulator library when called from the high-level language. First write the procedure by using coprocessor instructions, then assemble it using the **-e** option, and finally link it with your high-level-language modules. When compiling modules, use the compiler options that specify emulation.

Some coprocessor instructions are not emulated by Microsoft emulation libraries. How unemulated instructions vary depends on the language and version. If you use a coprocessor instruction that is not emulated, the program will generate a run-time error when it tries to execute the unemulated instruction. You cannot use a Microsoft emulation library with stand-alone assembler programs, since the library depends on the compiler start-up code.

For information on the **-e** option, see the section, ''Creating Code for a Floating-Point Emulator,'' in Chapter 2. For information on writing assembly-language procedures for high-level languages, see Appendix D, ''Segment Names for High-Level Languages.''

18

Using Coprocessor Instructions

Coprocessor instructions are readily recognizable because, unlike all 8086-family instruction mnemonics, they 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 the section, "Loading Constants"), 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. Table 18.1 lists the variations of coprocessor instructions along with the syntax for each.

Table 18.1

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.

Using Implied Operands in the Classical-Stack Form

The classical-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 the section, ''Doing Arithmetic Calculations''), 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 the section, ''Transferring Data to and from Registers''). 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.

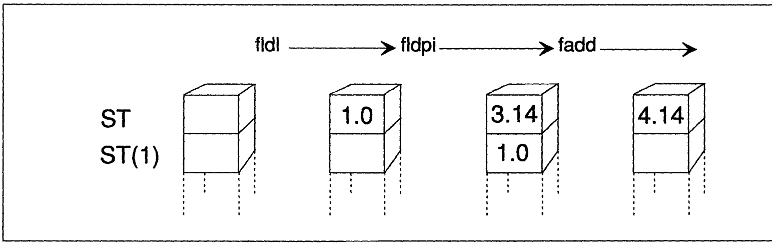
Note

> 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 into first position
fldpi           ; Push pi into first position
fadd            ; Add pi and 1 and pop
```

The status of the register stack after each instruction is shown below:



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 the section, "Transferring Data to and from Registers"). 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.

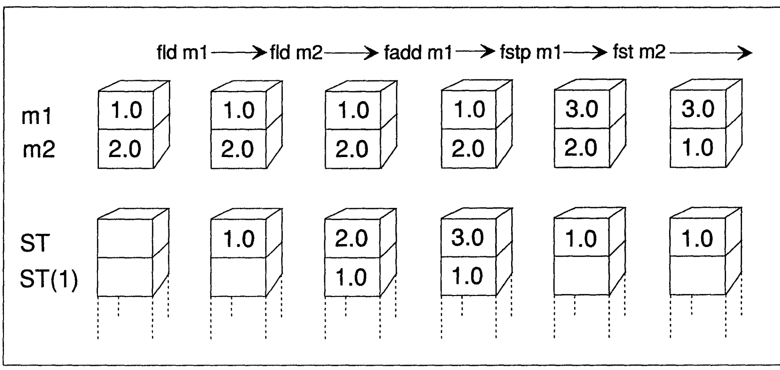
18

Memory operands can also be used in calculation instructions that operate on two values (see the section, "Doing Arithmetic Calculations"). 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 into first position
              fld      m2          ; Push m2 into first position
              fadd     m1          ; Add m2 to first position
              fstp     m1          ; Pop first position into m1
              fst      m2          ; Copy first position to m2
```

The status of the register stack and the memory locations used in the instructions is shown below:



Specifying Operands in the Register Form

The register form treats coprocessor registers as traditional registers. Registers are specified the same as 8086-family instructions with two register operands. The only limitation is that one of the two registers must be the stack top (**ST**).

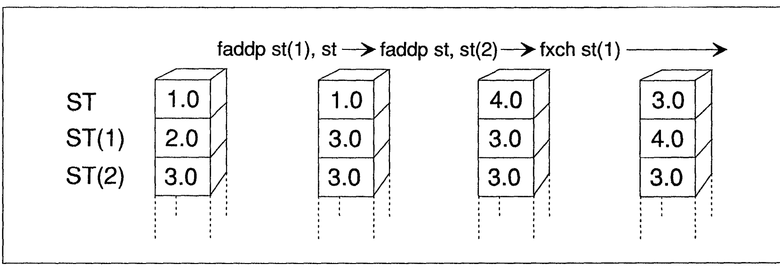
In the register form, operands are specified by name. The second operand is the source; it is not affected by the operation. The first operand is the destination; its value is replaced with the result of the operation. The stack position of the operands does not change.

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 position to first -
                   ; result goes in second position
fadd    st,st(2)   ;Add first position to second -
                   ; result goes in first position
fxch    st(1)      ;Exchange first and second positions
```

The status of the register stack if the registers were previously initialized to 1.0, 2.0, and 3.0 is shown below:



Specifying Operands in the Register-Pop Form

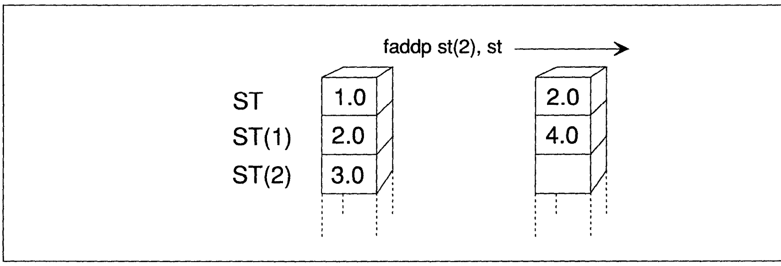
The register-pop form treats coprocessor registers as a modified stack. This form has some of the aspects of both a stack and registers. The destination register can be specified by name, but the source register must always be the stack top.

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 register-pop form is only used for instructions that do calculations on two values.

Example

```
faddp   st(2),st     ; Add first and third positions and pop -
                     ;   first position destroyed
                     ;   third moves to second and holds result
```

The status of the register stack if the registers were already initialized to 1.0, 2.0, and 3.0 is shown below:



Coordinating Memory Access

Problems of coordinating memory access 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.

In the first case, if a processor instruction is given and then 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.

Coprocessor Differences

To synchronize operations between the 8088 or 8086 processor and the 8087 coprocessor, each 8087 instruction must be preceded by a **WAIT** instruction. This is not necessary for the 80287 or 80387. If you use the **.8087** directive, **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.

In the second case, if a coprocessor instruction that accesses memory is followed by a processor instruction attempting to access the same memory location, memory access is not automatically synchronized. For instance, if you store a coprocessor register to a variable and 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 finished rather than the value stored by the coprocessor. Use the **WAIT** or **FWAIT** instruction (they are mnemonics for the same instruction) to ensure that the coprocessor finishes before the processor begins.

Example

```
; 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]

; Processor instruction first - No wait needed
          mov     WORD PTR mem32,ax   ; Load memory
          mov     WORD PTR mem32[2],dx
          fild    mem32               ; Load to register
```

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

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, a 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 take up 10 bytes.

The adjustments between formats are made automatically. Notice that floating-point numbers must be stored in the IEEE format, not in the Microsoft Binary format. Data is automatically stored correctly by default. It is stored incorrectly and the coprocessor instructions disabled if you use the **.MSFLOAT** directive. Data formats for real numbers are explained in the section, ''Real-Number Variables'', in Chapter 5.

Data are transferred to stack registers by using load commands. These push data onto the stack from memory or coprocessor registers. Data are removed by using store commands. Some store commands pop data off the register stack into memory or coprocessor registers, whereas others simply copy the data without changing it on the stack.

Real Transfers

The following instructions are available for transferring real numbers:

Syntax	Description
FLD *mem*	Pushes a copy of *mem* into **ST**. The source must a 4-, 8-, or 10-byte memory operand. It is automatically converted to the temporary-real format.
FLD ST(*num*)	Pushes a copy of the specified register into **ST**.
FST *mem*	Copies **ST** to *mem* without affecting the register stack. The destination can be a 4- or 8-byte memory operand. It is automatically converted from temporary-real format to short real or long real format, depending on the size of the operand. It cannot be converted to the 10-byte-real format.
FST ST(*num*)	Copies **ST** to the specified register. The current value of the specified register is replaced.
FSTP *mem*	Pops a copy of **ST** into *mem*. The destination can be a 4-, 8-, or 10-byte memory operand. It is automatically converted from temporary-real format to the appropriate real-number format, depending on the size of the operand.
FSTP ST(*num*)	Pops **ST** into the specified register. The current value of the specified register is replaced.
FXCH [ST(*num*)]	Exchanges the value in **ST** with the value in **ST**(*num*). If no operand is specified, **ST(0)** and **ST(1)** are exchanged.

Transferring Data

Integer Transfers

The following instructions are available for transferring binary integers:

Syntax	Description
FILD *mem*	Pushes a copy of *mem* into **ST**. The source must be a 2-, 4-, or 8-byte integer memory operand. It is interpreted as an integer and converted to temporary-real format.
FIST *mem*	Copies **ST** to *mem*. 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 a quadword integer.
FISTP *mem*	Pops **ST** into *mem*. 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 integer, depending on the size of the operand.

Packed BCD Transfers

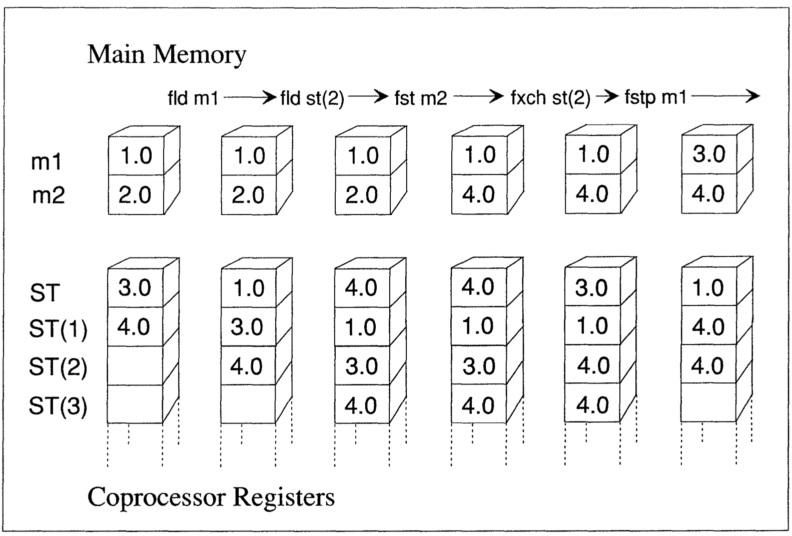
The following instructions are available for transferring BCD integers:

Syntax	Description
FBLD *mem*	Pushes a copy of *mem* into **ST**. 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 *mem*	Pops **ST** into *mem*. The destination must be a 10-byte memory operand. The value is rounded to an integer if necessary, and converted to a packed BCD value.

Example 1

```
fld     m1              ; Push m1 into first item
fld     st(2)           ; Push third item into first
fst     m2              ; Copy first item to m2
fxch    st(2)           ; Exchange first and third items
fstp    m1              ; Pop first item into m1
```

With the assumption that registers **ST** and **ST(1)** were previously initialized to 3.0 and 4.0, the status of the register stack is shown below:



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,si           ; 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.

Loading Constants

Constants cannot be given as operands and loaded directly into coprocessor registers. You must allocate memory and initialize the variable to a constant value. The variable can then be loaded by using one of the load instructions described in the section, ‘‘Transferring Data to and from Registers.’’

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 into **ST**
FLD1	Pushes 1 into **ST**
FLDPI	Pushes the value of pi into **ST**
FLDL2E	Pushes the value of $\log_2 e$ into **ST**
FLDL2T	Pushes $\log_2 10$ into **ST**
FLDLG2	Pushes $\log_{10} 2$ into **ST**
FLDLN2	Pushes $\log_e 2$ **ST**

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 seldom need to load anything other than the status word.

All the control-transfer instructions take a single memory operand. Load instructions use the memory operand as the destination; store instructions 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.

Transferring Data

Syntax	Description
FLDCW *mem2byte*	Loads control word
F[N]STCW *mem2byte*	Stores control word
F[N]STSW *mem2byte*	Stores status word
FLENV *mem14byte*	Loads environment
F[N]STENV *mem14byte*	Stores environment
FRSTOR *mem94byte*	Restores state
F[N]SAVE *mem94byte*	Saves state

80287/387 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, as shown below:

```
fstsw   ax
```

80387 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** instruction stores 18 bytes instead of 14.

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 the section, "Using Coprocessor Instructions."

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 do not 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

The following instructions add the source and destination and put the result in the destination:

Syntax	Description
FADD	Classical-stack form. Adds **ST** and **ST(1)** and pops the result into **ST**. Both operands are destroyed.
FADD ST(*num***),ST**	Register form with stack top as source. Adds the two register values and replaces **ST(***num***)** with the result.
FADD ST,ST(*num***)**	Register form with stack top as destination. Adds the two register values and replaces **ST** with the result.
FADD *mem*	Real-memory form. Adds a real number in *mem* to **ST**. The result replaces **ST**.
FIADD *mem*	Integer-memory form. Adds an integer in *mem* to **ST**. The result replaces **ST**.
FADDP ST(*num***),ST**	Register-pop form. Adds the two register values and pops the result into **ST(***num***)**. Both operands are destroyed.

Normal Subtraction

The following 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. Subtracts **ST** from **ST(1)** and pops the result into **ST**. Both operands are destroyed.
FSUB ST(*num***),ST**	Register form with stack top as source. Subtracts **ST** from **ST(***num***)** and replaces **ST(***num***)** with the result.
FSUB ST,ST(*num***)**	Register form with stack top as destination. Subtracts **ST(***num***)** from **ST** and replaces **ST** with the result.
FSUB *mem*	Real-memory form. Subtracts the real number in *mem* from **ST**. The result replaces **ST**.
FISUB *mem*	Integer-memory form. Subtracts the integer in *mem* from **ST**. The result replaces **ST**.
FSUBP ST(*num***),ST**	Register-pop form. Subtracts **ST** from **ST(***num***)** and pops the result into **ST(***num***)**. Both operands are destroyed.

Reversed Subtraction

The following instructions subtract the destination from the source and put the difference in the destination. Thus the number subtracted is replaced by the result.

18

Syntax	Description
FSUBR	Classical-stack form. Subtracts **ST(1)** from **ST** and pops the result into **ST**. Both operands are destroyed.
FSUBR ST(*num***),ST**	Register form with stack top as source. Subtracts **ST(***num***)** from **ST** and replaces **ST(***num***)** with the result.

FSUBR ST,ST(*num*)	Register form with stack top as destination. Subtracts **ST** from **ST(*num*)** and replaces **ST** with the result.
FSUBR *mem*	Real-memory form. Subtracts **ST** from the real number in *mem*. The result replaces **ST**.
FISUBR *mem*	Integer-memory form. Subtracts **ST** from the integer in *mem*. The result replaces **ST**.
FSUBRP ST(*num*),ST	Register-pop form. Subtracts **ST(*num*)** from **ST** and pops the result into **ST(*num*)**. Both operands are destroyed.

Multiplication

The following instructions multiply the source and destination and put the product in the destination:

Syntax	Description
FMUL	Classical-stack form. Multiplies **ST** by **ST(1)** and pops the result into **ST**. Both operands are destroyed.
FMUL ST(*num*),ST	Register form with stack top as source. Multiplies the two register values and replaces **ST(*num*)** with the result.
FMUL ST,ST(*num*)	Register form with stack top as destination. Multiplies the two register values and replaces **ST** with the result.
FMUL *mem*	Real-memory form. Multiplies a real number in *mem* by **ST**. The result replaces **ST**.
FIMUL *mem*	Integer-memory form. Multiplies an integer in *mem* by **ST**. The result replaces **ST**.
FMULP ST(*num*),ST	Register-pop form. Multiplies the two register values and pops the result into **ST(*num*)**. Both operands are destroyed.

Normal Division

The following 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. Divides **ST(1)** by **ST** and pops the result into **ST**. Both operands are destroyed.
FDIV ST(*num*),ST	Register form with stack top as source. Divides **ST(*num*)** by **ST** and replaces **ST(*num*)** with the result.
FDIV ST,ST(*num*)	Register form with stack top as destination. Divides **ST** by **ST(*num*)** and replaces **ST** with the result.
FDIV *mem*	Real-memory form. Divides **ST** by the real number in *mem*. The result replaces **ST**.
FIDIV *mem*	Integer-memory form. Divides **ST** by the integer in *mem*. The result replaces **ST**.
FDIVP ST(*num*),ST	Register-pop form. Divides **ST(*num*)** by **ST** and pops the result into **ST(*num*)**. Both operands are destroyed.

Reversed Division

The following instructions divide the source by the destination and put the quotient in the destination. Thus the divisor is replaced by the quotient.

18

Syntax	Description
FDIVR	Classical-stack form. Divides **ST** by **ST(1)** and pops the result into **ST**. Both operands are destroyed.
FDIVR ST(*num*),ST	Register form with stack top as source. Divides **ST** by **ST(*num*)** and replaces **ST(*num*)** with the result.

FDIVR ST,ST(*num*)	Register form with stack top as destination. Divides **ST**(*num*) by **ST** and replaces **ST** with the result.
FDIVR *mem*	Real-memory form. Divides the real number in *mem* by **ST**. The result replaces **ST**.
FIDIVR *mem*	Integer-memory form. Divides the integer in *mem* by **ST**. The result replaces **ST**.
FDIVRP ST(*num*),**ST**	Register-pop form. Divides **ST** by **ST**(*num*) and pops the result into **ST**(*num*). Both operands are destroyed.

Other Operations

The following 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 **ST** to positive.
FCHS	Reverses the sign of **ST**.
FRNDINT	Rounds the **ST** to an integer.
FSQRT	Replaces the contents of **ST** 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 Only

The 80387 has a new instruction called **FPREM1**. Its effect is similar to that of **FPREM**, but it conforms to the IEEE standard.

Example

```
        .DATA
a       DD      3.0
b       DD      7.0
c       DD      2.0
posx    DD      0.0
negx    DD      0.0

        .CODE
        .
        .
        .
; Solve quadratic equation - no error checking

        fld1                    ; Get constants 2 and 4
        fadd    st,st           ; 2 at bottom
        fld     st              ; Copy it
        fmul    a               ; = 2a

        fmul    st(1),st        ; = 4a
        fxch                    ; Exchange
        fmul    c               ; = 4ac

        fld     b               ; Load b
        fmul    st,st           ; = b^2
        fsubr                   ; = b^2 - 4ac
                                ; Negative value here produces error
        fsqrt                   ; = square root(b^2 - 4ac)
        fld     b               ; Load b
        fchs                    ; Make it negative
        fxch                    ; Exchange
        fld     st              ; Copy square root
        fadd    st,st(2)        ; Plus version = -b + root((b^2 - 4ac)
        fxch                    ; Exchange
        fsubp   st(2),st        ; Minus version = -b - root((b^2 - 4ac)

        fdiv    st,st(2)        ; Divide plus version
        fstp    posx            ; Store it
        fdivr                   ; Divide minus version
        fstp    negx            ; Store it
```

This example solves quadratic equations. It does no error checking and fails for some values because it attempts to find the square root of a negative number. You could enhance the code by using the **FTST** instruction (see the section, ''Comparing Operands to Control Program Flow'') to check for a negative number or 0 just before the square root is calculated. If *b* squared minus *4ac* is negative or 0, the code can jump to routines that handle special cases for no solution or one solution, respectively.

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:

```
fstsw   mem16       ; Store status word in memory
fwait               ; Make sure coprocessor is done
mov     ax,mem16    ; Move to AX
sahf                ; Store upper word in flags
```

As noted in the section, "Transferring Control Data," you can save several steps by loading the status word directly to **AX** on the 80287 and 80387.

Figure 18.3 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 cannot count on them being unchanged after the operation.

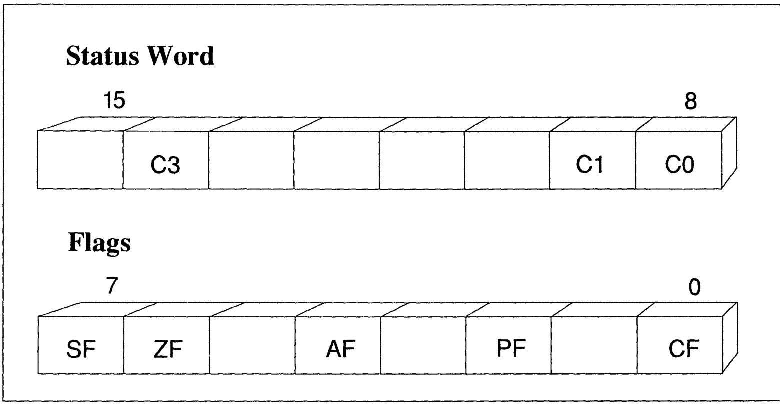


Figure 18-3 Coprocessor and Processor Control Flags

See the section, ''Jumping Conditionally,'' in Chapter 16, for more information on using conditional-jump instructions based on flag status.

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 implied as **ST(1)**.

The compare instructions affect the **C3**, **C2**, and **C0** control flags. The **C1** flag is not affected. Table 18.2 below shows the flags set for each possible result of a comparison or test.

18

Table 18.2

Control-Flag Settings after Compare or Test

After FCOM	After FTEST	C3	C2	C0
ST > *source*	**ST** is positive	0	0	0
ST < *source*	**ST** is negative	0	0	1
ST = *source*	**ST** is 0	1	0	0
Not comparable	**ST** is NAN or projective infinity	1	1	1

Variations on the compare instructions 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 compare instructions are listed below.

Compare

These instructions compare the stack top to the source. The source and destination are unaffected by the comparison.

Syntax	Description
FCOM	Compares **ST** to **ST(1)**.
FCOM ST(*num***)**	Compares **ST** to **ST(***num***)**.
FCOM *mem*	Compares **ST** to *mem*. The memory operand can be a four- or eight-byte real number.
FICOM *mem*	Compares **ST** to *mem*. The memory operand can be a two- or four-byte integer.
FTST	Compares the **ST** to 0. The control registers will be affected as if **ST** had been compared to 0 in **ST(1)**. Table 18.2 above shows the possible results.

Compare and Pop

These instructions compare the stack top to the source, and then pop the stack. Thus the destination is destroyed by the comparison.

Syntax	Description
FCOMP	Compares **ST** to **ST(1)** and pops **ST** off the register stack.
FCOMP ST(*num*)	Compares **ST** to **ST(*num*)** and pops **ST** off the register stack.
FCOMP *mem*	Compares **ST** to *mem* and pops **ST** off the register stack. The operand can be a four- or eight-byte real number.
FICOMP *mem*	Compares **ST** to *mem* and pops **ST** off the register stack. The operand can be a two- or four-byte integer.
FCOMPP	Compares **ST** to **ST(1)**, and then pops the stack twice. Both the source and destination are destroyed by the comparison.

80387 Only

Unordered compare instructions are available with the 80387. The **FUCOM**, **FUCOMP**, and **FUCOMPP** instructions are like **FCOM**, **FCOMP**, and **FCOMPP** except that the unordered versions do not cause invalid operation exceptions if one of the operands is a quiet NAN (not a number). Exceptions and NANs are beyond the scope of this manual and are not explained here. See Intel coprocessor reference books for more information.

Example

```
            IFDEF   c287
            .287
            ENDIF
            .DATA
down        DD      10.35       ; Sides of a rectangle
across      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
            fwait               ; Wait for coprocessor
            mov     ax,status   ; Memory to register
            ELSE
            fstsw   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
```

18

Notice how conditional blocks are used to enhance 80287 code. If you define the symbol *c287* from the command line by using the **-D***symbol* option (see the section, ‘‘Defining Assembler Symbols’’, in Chapter 2), the code is smaller and faster, but does not run on an 8087.

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.

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.

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** when computing the log of a number 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 result of 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 result of the **FPATAN** instruction can be used to calculate other inverse trigonometric functions, including arcsine and arccosine.

80387 Only

The following 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 cosine of the original stack-top value. The value in **ST(1)** is the sine of the original stack-top value. One of the values is pushed so that the former value in **ST(1)** is in **ST(2)**.

Controlling the Coprocessor

Additional instructions are available for controlling various aspects of the coprocessor. With the exception of **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 ensures that no register values from previous programs affect your program. Placing it at the end ensures 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. It 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 specified 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.

8

8087 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 coprocessors permit these instructions, but ignore them. Applications programmers will not normally need these instructions. Systems programmers should avoid using them so that their programs are portable to all coprocessors.

80287/387 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. For information on segment selectors, see the section, ''Segmented Addresses,'' in Chapter 12.

Either the **.286P** or **.386P** directive must be given before the **FSETPM** instruction can be used. Protected-mode operating systems normally set protected mode automatically. Therefore, you need this instruction only if you are writing control software.

Chapter 19

Controlling the Processor

Introduction

The 8086-family processors provide instructions for processor control. Some of these instructions are available on all processors; others 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 in detail, in the sections below.

Controlling Timing and Alignment

The **NOP** instruction does nothing but take up time and space. It works by exchanging the **AX** register with itself. The **NOP** instruction 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 take different lengths of time on different machines.

NOP instructions are automatically inserted for padding when you use the **ALIGN** or **EVEN** directive (see the section, "Aligning Data", in Chapter 5) to align data or code on a given boundary. The assembler automatically inserts **NOP** instructions for alignment.

Controlling the Processor

The **WAIT**, **ESC**, **LOCK**, and **HLT** instructions control different aspects of the processor.

These instructions can be used to control processes handled by external coprocessors. The 8087-family coprocessors are the coprocessors most commonly used with 8086-family processors, but 8086-based machines can work with other coprocessors if they have 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 change data being worked on by the processor.
WAIT	Instructs the processor to do nothing until it receives a signal that a coprocessor has finished with a task being performed at the same time. For information on using **WAIT** or its coprocessor equivalent, **FWAIT**, with the 8087-family coprocessors, see the section, "Coordinating Memory Access," in Chapter 18.
ESC	Provides an instruction and possibly a memory operand for use by a coprocessor. **ESC** instructions are automatically inserted when required for use with 8087-family coprocessors.
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.

Controlling Protected-Mode Processes

80286/386 Only

Protected mode is available starting with the 80286 processors. This mode is generally initiated and controlled by the operating system. Under Part 1, ''Using Assembler Programs and OS/2, applications programmers do not need to use protected-mode instructions. Process control is managed through system calls.

The instructions that control protected mode are privileged and can only be used if the **.286P** or **.386P** directives have been given. These instructions are generally needed only for operating systems and other control software. Some privileged-mode instructions use internal registers of the 80286 or 80386 processors. Instructions are provided for loading values from these registers into memory where the values can be modified. Other instructions can then be used to store the values back to the special registers.

The privileged-mode instructions are listed below:

Instruction	Description
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	Clears task-switched flag
VERR	Verifies read access
VERW	Verifies write access

Controlling the 80386

80386 Only

The 80386 processor can use all the privileged-mode instructions of the 80286, 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 32-bit registers or from them.

Examples

```
mov     eax,cr0              ; Load CR0 into EAX
mov     cr3,ecx              ; Store ECX in CR3
```

19

Appendix A

New Features

Introduction

Version 5.0 of the Macro Assembler (**masm**) has many significant new features. This appendix describes these features and tells you where they are documented.

Enhancements to masm

This version of **masm** has several important enhancements. The following sections summarize new options, directives, instructions, and other features.

80386 Support

The **masm** program now supports the 80386 instruction set and addressing modes. The 80386 processor is a superset of other 8086-family processors. Most new features of the 80386 are simply 32-bit extensions of 16-bit features, and are used in much the same way as the 16-bit registers. However, some features of the 80386 processor are significantly different. (The 80386 registers are explained in the section, ''Using 8086-Family Registers,'' in Chapter 12.)

Throughout this manual, the heading ''80386 Only'' indicates sections describing 80386 enhancements. Areas of particular importance include the following:

- the **.386** directive for initializing the 80386 (see the section, ''Defining Default Assembly Behavior'', in Chapter 3)
- the **USE32** and **USE16** segment types for setting the segment word size (see the section, ''Setting Segment Word Size with Use Type,'' in Chapter 4)
- indirect addressing modes (see the section, ''80386 Indirect Memory Operands'', in Chapter 13)

The 80386 processor and the 80387 coprocessor have some new instructions that are unique, and unrelated to any 16-bit instructions. These are listed in Table A.1.

Table A.1

80386 and 80387 Instructions

Name	Mnemonic	Reference
Bit Scan Forward	**BSF**	Chapter 15
Bit Scan Reverse	**BSR**	Chapter 15
Bit Test	**BT**	Chapter 16
Bit Test and Complement	**BTC**	Chapter 16
Bit Test and Reset	**BTR**	Chapter 16
Bit Test and Set	**BTS**	Chapter 16
Move with Sign Extend	**MOVSX**	Chapter 14
Move with Zero Extend	**MOVZX**	Chapter 14
Set Byte on Condition	**SET***condition*	Chapter 16
Double Precision Shift Left	**SHLD**	Chapter 15
Double Precision Shift Right	**SHRD**	Chapter 15
Move to/from Special Registers	**MOV**	Chapter 17
Sine	**FSIN**	Chapter 18
Cosine	**FCOS**	Chapter 18
Sine Cosine	**FSINCOS**	Chapter 18
IEEE Partial Remainder	**FPREM1**	Chapter 18
Unordered Compare Real	**FUCOM**	Chapter 18
Unordered Compare Real and Pop	**FUCOMP**	Chapter 18
Unordered Compare Real and Pop Twice	**FUCOMPP**	Chapter 18

Segment Simplification

A new system of defining segments is available in **masm** Version 5.0. The simplified segment directives use the Microsoft naming conventions and allow segments to be defined easily and consistently. However, this segment definition system 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. For more information about segment simplification, see the section, ''Simplified Segment Definitions.''

A new **DOSSEG** directive enables you to specify MS-DOS segment order in the source file. For more information on this feature, see the section, ''Specifying MS-DOS Segment Order.''

Performance Improvements

The **masm** program's performance has been enhanced through faster assembly and larger symbol space:

1. For most source files, Version 5.0 of the assembler is significantly faster than previous versions. The degree of improvement varies, depending on the relative amounts of code and data in the source file, and on the complexity of expressions used.

2. Symbol space is now limited only by the amount of system memory available to your machine.

Enhanced Error Handling

Error handling has been enhanced from previous versions 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 with the **-w** option. Type-checking errors are now serious warnings rather than severe errors. See the section, ''Setting the Warning Level.''

- During assembly, messages are output to standard output. In Version 4.0 they were sent to standard error.

New Options

The following command-line options have been added to Version 5.0:

Option	Description
-w0|1|2]	Sets the warning level to determine what type of messages will be displayed: severe errors, serious warnings, or advisory warnings. For more information about warning levels, see the section, ''Setting the Warning Level.''
-Zd and **-Zi**	Sends debugging information for symbolic debuggers to the object file. The **-Zd** option outputs line-number information, whereas the **-Zi**

option outputs both line-number and type information. These options are described in the section, "Writing Symbolic Information to the Object File."

-h Displays the **masm** command line and options, as explained in the section, "Creating Code for a Floating-Point Emulator."

-D*sym*[=*val*] Allows definition of a symbol from the command line. This is an enhancement of a current option. For more information, see the section, "Defining Assembler Symbols."

In addition, **.ALPHA** and **.SEQ** directives have been added to **masm**. These directives have the same effect as the **-a** and **-s** options. These directives are described in the section, "Setting the Segment-Order Method."

String Equates

String equates have been enhanced for easier use. By enclosing the argument to the **EQU** directive in angle brackets, you can ensure that the argument is evaluated as a string equate rather than as an expression. For examples, see the section, "String Equates," in Chapter 10.

The expression operator (%) can now be used with macro arguments that are text macros as well as with arguments that are expressions. This feature is described in the section, "Expression Operator," in Chapter 10.

RETF and RETN Instructions

Version 5.0 makes two new instructions available, **RETF** (Return Far) and **RETN** (Return Near). These instructions let you define procedures without using the **PROC** and **ENDP** directives. The section, "Defining Procedures," in Chapter 16, explains these instructions.

Communal Variables

You can now declare *communal variables*. These uninitialized global data items can be used in include files, and are compatible with variables declared in C include files. For details, see the section, "Using Multiple Modules."

Flexible Structure Definitions

Structure definitions can now include conditional-assembly statements, thus enabling more flexible structures. For more information, see the section, ''Declaring Structure Types.''

Compatibility with Assemblers and Compilers

If you are upgrading from a previous version of the Microsoft Macro Assembler, you may need to make some adjustments before assembling source code developed with previous versions.

Previous versions (pre-5.0) of **masm** assembled initialized real-number variables in the Microsoft Binary format by default. Version 5.0 assembles initialized real-number variables in the IEEE format. If you have source modules that expect Microsoft Binary format, you must modify them by placing the **.MSFLOAT** directive at the start of the module, before the first variable is initialized.

In previous versions of **masm**, the following default conditions were recognized:

- 8086 instructions enabled
- math coprocessor instructions disabled
- real numbers assembled in Microsoft Binary format

In these earlier versions, the **-r** option, the **.8087** directive, or the **.287** directive was required to enable coprocessor instructions and to achieve IEEE format for real numbers.

Version 5.0 recognizes the following default conditions:

- 8086 and 8087 instructions enabled
- real numbers assembled in IEEE format

Although the **-r** option is no longer used, it is recognized and ignored by 5.0 so that existing makefiles work without modification.

Some early versions of **masm** 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. Version 5.0 solves this incompatibility by turning 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 given specifically. The section, "Strong Typing for Memory Operands," describes strict type checking and how to modify source code that was developed without this type-checking feature.

Appendix B

Instruction Summary

Introduction

The Macro Assembler is capable of assembling instructions for the 8086, 80186, 80286, and 80386 microprocessors and the 8087 and 80287 floating-point coprocessors. It will assemble any program written for an 8086, 80186, 80286, or 80386 microprocessor environment as long as the program uses the instruction syntax described in this appendix.

By default, **masm** recognizes 8086 and 8087 instructions only. If a source program contains 80186, 80286, 80287, or 80387 instructions, one or more instruction-set directives must be used in the source file to enable assembly of the instructions. The following sections list the syntax of all instructions recognized by **masm** and the instruction-set directives.

Table B.1 explains the abbreviations used in the 8086, 8087, 80186, 80286, 80287, 80386, and 80387 syntax descriptions:

Table B.1

Syntax-Description Abbreviations

Symbol	Meaning
accum	accumulator: AX, or AL
reg	byte or word register byte: AL, AH, BL, BH, CL, CH, DL, DH word: AX, BX, CX, DX, SI, DI, BP, SP dword: EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
segreg	segment register: CS, DS, SS, ES, FS, GS
r/m	general operand: register, memory address, indexed operand, based operand, or based-indexed operand
immed	8-, 16-, or 32-bit immediate value: constant or symbol
mem	memory operand: label, variable, or symbol
label	instruction label

8086 Instruction Mnemonics

The 8086 instructions are listed below. All 8086 instructions are assembled by default.

B

Table B.2

8086 Instruction Mnemonics

Syntax	Action
AAA	ASCII adjust for addition
AAD	ASCII adjust for division
AAM	ASCII adjust for multiplication
AAS	ASCII adjust for subtraction
ADC *accum, immed*	Add immediate with carry to accumulator
ADC *r/m, immed*	Add immediate with carry to operand
ADC *r/m, reg*	Add register with carry to operand
ADC *reg, r/m*	Add operand with carry to register
ADD *accum, immed*	Add immediate to accumulator
ADD *r/m, immed*	Add immediate to operand
ADD *r/m, reg*	Add register to operand
ADD *reg, r/m*	Add operand to register
AND *accum, immed*	Bitwise AND immediate with accumulator
AND *r/m, immed*	Bitwise AND immediate with operand
AND *r/m, reg*	Bitwise AND register with operand
AND *reg, r/m*	Bitwise AND operand with register
CALL *label*	Execute instruction at label
CALL *r/m*	Execute instruction indirect
CBW	Convert byte to word
CLC	Clear carry flag
CLD	Clear direction flag
CLI	Clear interrupt flag
CMC	Complement carry flag
CMP *accum, immed*	Compare immediate with accumulator

(*Continued on next page.*)

8086 Instruction Mnemonics *(Continued)*

Syntax	Action
CMP *r/m, immed*	Compare immediate with operand
CMP *r/m, reg*	Compare register with operand
CMP *reg, r/m*	Compare operand with register
CMPS *src, dest*	Compare strings
CMPSB	Compare strings byte for byte
CMPSW	Compare strings word for word
CWD	Convert word to doubleword
DAA	Decimal adjust for addition
DAS	Decimal adjust for subtraction
DEC *r/m*	Decrement operand
DEC *reg*	Decrement 16-bit register
DIV *r/m*	Divide accumulator by operand
ESC *immed, r/m*	Escape with 16-bit immediate and operand
HLT	Halt processor
IDIV *r/m*	Integer divide accumulator by operand
IMUL *r/m*	Integer multiply accumulator by operand
IN *accum, immed*	Input from port (8-bit immediate)
IN *accum*, DX	Input from port given by DX
INC *r/m*	Increment operand
INC *reg*	Increment 16-bit register
INT 3	Execute software interrupt 3 (encoded as one byte)
INT *immed*	Execute software interrupt 0 through 255
INTO	Interrupt on overflow
IRET	Return from interrupt
JA *label*	Jump on above
JAE *label*	Jump on above or equal
JB *label*	Jump on below
JBE *label*	Jump on below or equal
JC *label*	Jump on carry
JCXZ *label*	Jump on CX zero

(Continued on next page.)

8086 Instruction Mnemonics (*Continued*)

Syntax	Action
JE *label*	Jump on equal
JG *label*	Jump on greater
JGE *label*	Jump on greater or equal
JL *label*	Jump on less
JLE *label*	Jump on less or equal
JMP *label*	Jump to instruction at label
JMP *r/m*	Jump to instruction indirect
JNA *label*	Jump on not above
JNAE *label*	Jump on not above or equal
JNB *label*	Jump on not below
JNBE *label*	Jump on not below or equal
JNC *label*	Jump on no carry
JNE *label*	Jump on not equal
JNG *label*	Jump on not greater
JNGE *label*	Jump on not greater or equal
JNL *label*	Jump on not less
JNLE *label*	Jump on not less or equal
JNO *label*	Jump on not overflow
JNP *label*	Jump on not parity
JNS *label*	Jump on not sign
JNZ *label*	Jump on not zero
JO *label*	Jump on overflow
JP *label*	Jump on parity
JPE *label*	Jump on parity even
JPO *label*	Jump on parity odd
JS *label*	Jump on sign
JZ *label*	Jump on zero
LAHF	Load AH with flags
LDS *r/m*	Load operand into DS

(*Continued on next page.*)

8086 Instruction Mnemonics (*Continued*)

Syntax	Action
LEA *r/m*	Load effective address of operand
LES *r/m*	Load operand into ES
LOCK	Lock bus
LODS *src*	Load string
LODSB	Load byte from string into AL
LODSW	Load word from string into AX
LOOP *label*	Loop
LOOPE *label*	Loop while equal
LOOPNE *label*	Loop while not equal
LOOPNZ *label*	Loop while not zero
LOOPZ *label*	Loop while zero
MOV *accum, mem*	Move memory to accumulator
MOV *mem, accum*	Move accumulator to memory
MOV *r/m, immed*	Move immediate to operand
MOV *r/m, reg*	Move register to operand
MOV *r/m, segreg*	Move segment register to operand
MOV *reg, immed*	Move immediate to register
MOV *reg, r/m*	Move operand to register
MOV *segreg, r/m*	Move operand to segment register
MOVS *dest, src*	Move string
MOVSB	Move string byte by byte
MOVSW	Move string word by word
MUL *r/m*	Multiply accumulator by operand
NEG *r/m*	Negate operand
NOP	No operation
NOT *r/m*	Invert operand bits
OR *accum, immed*	Bitwise OR immediate with accumulator
OR *r/m, immed*	Bitwise OR immediate with operand
OR *r/m, reg*	Bitwise OR register with operand

(*Continued on next page.*)

B

8086 Instruction Mnemonics *(Continued)*

Syntax	Action
OR *reg, r/m*	Bitwise OR operand with register
OUT DX, *accum*	Output to port given by DX
OUT *immed, accum*	Output to port (8-bit immediate)
POP *r/m*	Pop 16-bit operand
POP *reg*	Pop 16-bit register from stack
POP *segreg*	Pop segment register
POPF	Pop flags
PUSH *r/m*	Push 16-bit operand
PUSH *reg*	Push 16-bit register onto stack
PUSH *segreg*	Push segment register
PUSHF	Push flags
RCL *r/m*, 1	Rotate left through carry by 1 bit
RCL *r/m*, CL	Rotate left through carry by CL
RCR *r/m*, 1	Rotate right through carry by 1 bit
RCR *r/m*, CL	Rotate right through carry by CL
REPE	Repeat if equal
REPNE	Repeat if not equal
REPNZ	Repeat if not zero
REPZ	Repeat if zero
RET [*immed*]	Return after popping bytes from stack
ROL *r/m*, 1	Rotate left by 1 bit
ROL *r/m*, CL	Rotate left by CL
ROR *r/m*, 1	Rotate right by 1 bit
ROR *r/m*, CL	Rotate right by CL
SAHF	Store AH in flags
SAL *r/m*, 1	Shift arithmetic left by 1 bit
SAL *r/m*, CL	Shift arithmetic left by CL
SAR *r/m*, 1	Shift arithmetic right by 1 bit
SAR *r/m*, CL	Shift arithmetic right by CL
SBB *accum, immed*	Subtract immediate and carry flag

(Continued on next page.)

8086 Instruction Mnemonics (*Continued*)

Syntax	**Action**
SBB *r/m, immed*	Subtract immediate and carry flag
SBB *r/m, reg*	Subtract register and carry flag
SBB *reg, r/m*	Subtract operand and carry flag
SCAS *dest*	Scan string
SCASB	Scan string for byte in AL
SCASW	Scan string for word in AX
SHL *r/m*, 1	Shift left by 1 bit
SHL *r/m*, CL	Shift left by CL
SHR *r/m*, 1	Shift right by 1 bit
SHR *r/m*, CL	Shift right by CL
STC	Set carry flag
STD	Set direction flag
STI	Set interrupt flag
STOS *dest*	Store string
STOSB	Store byte in AL at string
STOSW	Store word in AX at string
SUB *accum, immed*	Subtract immediate from accumulator
SUB *r/m, immed*	Subtract immediate from operand
SUB *r/m, reg*	Subtract register from operand
SUB *reg, r/m*	Subtract operand from register
TEST *accum, immed*	Compare immediate bits with accumulator
TEST *r/m, immed*	Compare immediate bits with operand
TEST *r/m, reg*	Compare register bits with operand
TEST *reg, r/m*	Compare operand bits with register
WAIT	Wait
XCHG *accum, reg*	Exchange accumulator with register
XCHG *r/m, reg*	Exchange operand with register
XCHG *reg, accum*	Exchange register with accumulator
XCHG *reg, r/m*	Exchange register with operand
XLAT *mem*	Translate

(*Continued on next page.*)

8086 Instruction Mnemonics *(Continued)*

Syntax	Action
XOR *accum, immed*	Bitwise XOR immediate with accumulator
XOR *r/m, immed*	Bitwise XOR immediate with operand
XOR *r/m, reg*	Bitwise XOR register with operand
XOR *reg, r/m*	Bitwise XOR operand with register

The string instructions (CMPS, LODS, MOVS, SCAS, and STOS) use the DS, SI, ES, and DI registers to compute operand locations. Source operands are assumed to be at DS:[SI]; destination operands at ES:[DI]. The operand type (BYTE or WORD) is defined by the instruction mnemonic. For example, CMPSB specifies BYTE operands and CMPSW specifies WORD operands. For the CMPS, LODS, MOVS, SCAS, and STOS instructions, the *src* and *dest* operands are dummy operands that define the operand type only. The offsets associated with these operands are not used. The *src* operand can also be used to specify a segment override. The ES register for the destination operand cannot be overridden.

Examples

```
CMPS  WORD ptr string, WORD ptr ES:0
LODS  BYTE ptr string
mov   BYTE ptr ES:0,  BYTE ptr string
```

The REP, REPE, REPNE, REPNZ, and REPZ instructions provide ways to repeatedly execute a string instruction for a given count or while a given condition is true. If a repeat instruction immediately precedes a string instruction (both instructions must be on the same line), the instructions are repeated until the specified repeat condition is false or the CX register is equal to zero. The repeat instruction decrements CX by one for each execution.

Example

```
mov   CX, 10
REP      SCASB
```

8087 Instruction Mnemonics

The 8087 instructions are listed below. All 8087 instructions are assembled by default.

Table B.3

8087 Instruction Mnemonics

Syntax	Action
F2XM1	Calculate 2^x-1
FABS	Take absolute value of top of stack
FADD	Add real
FADD *mem*	Add real from memory
FADD ST, ST(*i*)	Add real from stack
FADD ST(*i*), ST	Add real to stack
FADDP ST(*i*), ST	Add real and pop stack
FBLD *mem*	Load 10-byte packed decimal on stack
FBSTP *mem*	Store 10-byte packed decimal and pop
FCHS	Change sign on the top stack element
FCLEX	Clear exceptions after WAIT
FCOM	Compare real
FCOM ST	Compare real with top of stack
FCOM ST(*i*)	Compare real with stack
FCOMP	Compare real and pop stack
FCOMP ST	Compare real with top of stack and pop
FCOMP ST(*i*)	Compare real with stack and pop stack
FCOMPP	Compare real and pop stack twice
FDECSTP	Decrement stack pointer
FDISI	Disable interrupts after WAIT
FDIV	Divide real

(*Continued on next page.*)

8087 Instruction Mnemonics *(Continued)*

Syntax	Action
FDIV *mem*	Divide real from memory
FDIV ST, ST(*i*)	Divide real from stack
FDIV ST(*i*), ST	Divide real in stack
FDIVP ST(*i*), ST	Divide real and pop stack
FDIVR	Reversed real divide
FDIVR *mem*	Reverse real divide from memory
FDIVR ST, ST(*i*)	Reverse real divide from stack
FDIVR ST(*i*), ST	Reverse real divide in stack
FDIVRP ST(*i*), ST	Reversed real divide and pop stack twice
FENI	Enable interrupts after WAIT
FFREE	Free stack element
FFREE ST	Free top of stack element
FFREE ST(*i*)	Free *i*th stack element
FIADD *mem*	Add 2- or 4-byte integer
FICOM *mem*	2- or 4-byte integer compare
FICOMP *mem*	2- or 4-byte integer compare and pop stack
FIDIV *mem*	2- or 4-byte integer divide
FIDIVR *mem*	Reversed 2- or 4-byte integer divide
FILD *mem*	Load 2-, 4-, or 8-byte integer on stack
FIMUL *mem*	Multiply 2- or 4-byte integer
FINCSTP	Increment stack pointer
FINIT	Initialize processor after WAIT
FIST *mem*	Store 2- or 4-byte integer
FISTP *mem*	Store 2-, 4-, or 8-byte integer and pop stack
FISUB *mem*	2- or 4-byte integer subtract
FISUBR *mem*	Reversed 2- or 4-byte integer subtract
FLD *mem*	Load 4-, 8-, or 10-byte real on stack
FLD1	Load +1.0 onto top of stack
FLDCW *mem*	Load control word
FLDENV *mem*	Load 8087 environment (14 bytes)

(Continued on next page.)

8087 Instruction Mnemonics (*Continued*)

Syntax	Action
FLDL2E	Load $\log_2 e$ onto top of stack
FLDL2T	Load $\log_2 10$ onto top of stack
FLDLG2	Load $\log_{10} 2$ onto top of stack
FLDLN2	Load $\log_e 2$ onto top of stack
FLDPI	Load pi onto top of stack
FLDZ	Load +0.0 onto top of stack
FMUL	Multiply real
FMUL *mem*	Multiply real from memory
FMUL ST, ST(*i*)	Multiply real from stack
FMUL ST(*i*), ST	Multiply real to stack
FMULP ST(*i*), ST	Multiply real and pop stack
FNCLEX	Clear exceptions with no WAIT
FNDISI	Disable interrupts with no WAIT
FNENI	Enable interrupts with no WAIT
FNINIT	Initialize processor with no WAIT
FNOP	No operation
FNSAVE *mem*	Save 8087 state (94 bytes) with no WAIT
FNSTCW *mem*	Store control word with no WAIT
FNSTENV *mem*	Store 8087 environment with no WAIT
FNSTSW *mem*	Store 8087 status word with no WAIT
FPATAN	Calculate partial arctangent
FPREM	Calculate partial remainder
PFPTAN	Calculate partial tangent
FRNDINT	Round to integer
FRSTOR *mem*	Restore 8087 state (94 bytes)
FSAVE *mem*	Save 8087 state (94 bytes) after WAIT
FSCALE	Scale
FSQRT	Square root
FST	Store real
FST ST	Store real from top of stack

(*Continued on next page.*)

8087 Instruction Mnemonics *(Continued)*

Syntax	Action
FST ST(*i*)	Store real from stack
FSTCW *mem*	Store control word with WAIT
FSTENV *mem*	Store 8087 environment after WAIT
FSTP *mem*	Store 4-, 8-, or 10-byte real and pop stack
FSTSW *mem*	Store 8087 status word after WAIT
FSUB	Subtract real
FSUB *mem*	Subtract real from memory
FSUB ST, ST(*i*)	Subtract real from stack
FSUB ST(*i*), ST	Subtract real to stack
FSUBP ST(*i*), ST	Subtract real and pop stack
FSUBR	Reversed real subtract
FSUBR *mem*	Reversed real subtract from memory
FSUBR ST, ST(*i*)	Reversed real subtract from stack
FSUBR ST(*i*), ST	Reversed real subtract in stack
FSUBRP ST(*i*), ST	Reversed real subtract and pop stack
FTST	Test top of stack
FWAIT	Wait for last 8087 operation to complete
FXAM	Examine top of stack element
FXCH	Exchange contents of stack elements
FFREE ST	Exchange top of stack element
FFREE ST(*i*)	Exchange top of stack and *i*th element
FXTRACT	Extract exponent and significant
FYL2X	Calculate Y $\log_2$x
FYL2PI	Calculate Y $\log_2$(x+1)

80186 Instruction Mnemonics

The 80186 instruction set consists of all 8086 instructions plus the following instructions. The **.186** directive must be placed at the beginning of the source file to enable these instructions.

Table B.4

80186 Instruction Mnemonics

Syntax	Action
BOUND *reg, mem*	Detect value out of range
ENTER *immed16, immed8*	Enter procedure
IMUL *immed, reg*	Integer multiply immediate byte into word register
IMUL *r/m, immed*	Integer multiply operand by immediate word/byte
INS *mem*, DX	Input string from port DX
INSB *mem*, DX	Input byte string from port DX
INSW *mem*, DX	Input word string from port DX
LEAVE	Leave procedure
OUTS DX, *mem*	Output byte/word/string to port DX
OUTSB DX, *mem*	Output byte string to port DX
OUTSW DX, *mem*	Output word string to port DX
POPA	Pop all registers
PUSH *immed*	Push immediate word/byte
PUSHA	Push all registers
RCL *r/m, immed*	Rotate left through carry immediate
RCR *r/m, immed*	Rotate
ROL *r/m, immed*	Rotate left immediate
ROL *r/m, immed*	Rotate right immediate
SAL *r/m, immed*	Shift arithmetic left immediate

(*Continued on next page.*)

80186 Instruction Mnemonics *(Continued)*

Syntax	Action
SAR *r/m*, *immed*	Shift arithmetic right immediate
SHL *r/m*, *immed*	Shift left immediate
SHR *r/m*, *immed*	Shift right immediate

80286 Nonprotected Instruction Mnemonics

The 80286 nonprotected instruction set consists of all 8086 instructions plus the following instructions. The **.286** directive must be placed at the beginning of the source file to enable these instructions.

Table B.5

80286 Nonprotected Instruction Mnemonics

Syntax	Action
BOUND *reg, mem*	Detect value out of range
ENTER *immed16, immed8*	Enter procedure
IMUL *immed, reg*	Integer multiply immediate byte into word register
IMUL *r/m, immed*	Integer multiply operand by immediate word/byte
INS *mem*, DX	Input string from port DX
INSB *mem*, DX	Input byte string from port DX
INSW *mem*, DX	Input word string from port DX
LEAVE	Leave procedure
OUTS DX, *mem*	Output byte/word/string to port DX
OUTSB DX, *mem*	Output byte string to port DX
OUTSW DX, *mem*	Output word string to port DX
POPA	Pop all registers
PUSH *immed*	Push immediate word/byte
PUSHA	Push all registers
RCL *r/m, immed*	Rotate left through carry immediate
RCR *r/m, immed*	Rotate right through carry immediate
ROL *r/m, immed*	Rotate left immediate
ROL *r/m, immed*	Rotate right immediate
SAL *r/m, immed*	Shift arithmetic left immediate
SAR *r/m, immed*	Shift arithmetic right immediate
SHL *r/m, immed*	Shift left immediate
SHR *r/m, immed*	Shift right immediate

80286 Protected Instruction Mnemonics

B

The 80286 protected instruction set consists of all 8086"and 80286 nonprotected instructions plus the following instructions. The **.286P** directive must be placed at the beginning of the source file to enable these instructions.

Table B.6

80286 Protected Instruction Mnemonics

Syntax	Action
ARPL *mem, reg*	Adjust requested privilege level
LAR *reg, mem*	Load access rights
LSL *reg, mem*	Load segment limit
SGDT *mem*	Store global-descriptor table (8 bytes)
SIDT *mem*	Store interrupt-descriptor table (8 bytes)
SLDT *mem*	Store local-descriptor table
SMSW *mem*	Store machine-status word
STR *mem*	Store task register
VERR *mem*	Verify read access
VERW *mem*	Verify write access

80287 Instruction Mnemonics

The 80287 instruction set consists of all 8087 instructions plus the following instructions. The **.287** directive must be used to enable these instructions.

Table B.7

80287 Instruction Mnemonics

Syntax	Action
FSETPM	Set protected mode
FSTSW AX	Store status word in AX (wait)
FNSTSW AX	Store status word in AX (no wait)

80386 Nonprotected Instruction Mnemonics

The 80386 nonprotected instruction set consists of all 8086 and 80286 nonprotected instructions plus the following instructions. The **.386** directive must be placed at the beginning of the source file to enable these instructions.

Table B.8

80386 Nonprotected Instruction Mnemonics

Syntax	Action
BT *reg, reg*	Bit test
BT *mem, reg*	Bit test
BT *reg, immed*	Bit test
BT *mem, immed*	Bit test
BT *mem*	Bit test
BTC *reg, reg*	Bit test and complement
BTC *mem, reg*	Bit test and complement
BTC *reg, immed*	Bit test and complement
BTC *mem, immed*	Bit test and complement
BTC *mem*	Bit test and complement
BTR *reg, reg*	Bit test and reset
BTR *mem, reg*	Bit test and reset
BTR *reg, immed*	Bit test and reset
BTR *mem, immed*	Bit test and reset
BTR *mem*	Bit test and reset
BTS *reg, reg*	Bit test and set
BTS *mem, reg*	Bit test and set
BTS *reg, immed*	Bit test and set
BTS *mem, immed*	Bit test and set
BTS *mem*	Bit test and set
CDQ	Convert doubleword in EAX to quadword in EAX:EDX

(*Continued on next page.*)

80386 Nonprotected Instruction Mnemonics (*Continued*)

Syntax	Action
CMPSD	String compare doubleword
CWDE	Convert word in AX, doubleword in EAX
IMUL *r/m*	Uncharacterized multiply
IMUL *reg, r/m*	Uncharacterized multiply
IMUL *reg, r/m, immed*	Uncharacterized multiply
IMUL *reg, immed*	Uncharacterized multiply
INSD	String input doubleword
IRETD	Return from an 80386 32-bit mode far interrupt
JA	Jump on above
JAE	Jump on above or equal
JB	Jump on below
JBE	Jump on below or equal
JC	Jump on carry
JE	Jump on equal
JG	Jump on greater
JGE	Jump on greater or equal
JL	Jump on less
JNA	Jump on not above
JNA	Jump on not above or equal
JNB	Jump on not below
JNBE	Jump on not below or equal
JNC	Jump on no carry
JNE	Jump on not equal
JNG	Jump on not greater
JNGE	Jump on not greater or equal
JNL	Jump on not less
JNLE	Jump on not less or equal
JNO	Jump on not overflow
JNP	Jump on not parity
JNS	Jump on not sign

(*Continued on next page.*)

B

80386 Nonprotected Instruction Mnemonics *(Continued)*

Syntax	Action
LFS *reg, mem*	Load reg and FS with far pointer
LGS *reg, mem*	Load reg and GS with far pointer
LODSD *mem*	Load string doubleword
LSS	Load reg and SS with far pointer
MOVSD	String move doubleword
MOVSX *reg, r/m*	Sign extend
MOVZX *reg, r/m*	Zero extend
OUTSD	Output string doubleword
POP FS/GS	Pop 80386 segment register
POPFD	Pop doubleword flags
POPAD	Pop all doubleword registers
PUSH FS/GS	Push 80386 segment register
PUSHAD	Push all doubleword registers
PUSHFD	Push doubleword flags
SCASD	Scan string doubleword
SETA *r/m*	Set byte if above
SETAE *r/m*	Set byte if above or equal
SETB *r/m*	Set byte if below
SETBE *r/m*	Set byte if below or equal
SETC *r/m*	Set byte if carry
SETE *r/m*	Set byte if equal
SETG *r/m*	Set byte if greater
SETGE *r/m*	Set byte if greater or equal
SETL *r/m*	Set byte if less
SETLE *r/m*	Set byte if less or equal
SETNA *r/m*	Set byte if not above
SETNAE *r/m*	Set byte if not above or equal
SETNB *r/m*	Set byte if not below
SETNBE *r/m*	Set byte if not below or equal

(Continued on next page.)

80386 Nonprotected Instruction Mnemonics (*Continued*)

Syntax	**Action**
SETNC *r/m*	Set byte if not carry
SETNE *r/m*	Set byte if not equal
SETNG *r/m*	Set byte if greater
SETNGE *r/m*	Set byte if not greater or equal
SETNL *r/m*	Set byte if not less
SETNLE *r/m*	Set byte if not less or equal
SETNO *r/m*	Set byte if not overflow
SETNP *r/m*	Set byte if not parity
SETNS *r/m*	Set byte if not sign
SETNZ *r/m*	Set byte if not zero
SETO *r/m*	Set byte if overflow
SETP *r/m*	Set byte if parity
SETPE *r/m*	Set byte if parity even
SETPO *r/m*	Set byte if parity odd
SETS *r/m*	Set byte if sign
SETZ *r/m*	Set byte if zero
SHLD *reg/mem,reg,imm/cl*	Shift double-precision left
SHRD *reg/mem,reg,imm/cl*	Shift double-precision right
STOSD *mem*	Store string doubleword

80386 Protected Instruction Mnemonics

B

The 80386 protected instruction set consists of all 8086 instructions and 80286 protected instructions plus the following instructions. The **.386P** directive must be placed at the beginning of the source file to enable these instructions.

Table B.9

80386 Protected Instruction Mnemonics

Syntax	Action
CLTS	Clear task switched flag
HLT	Halt processor
LGDT *mem*	Load global-descriptor table (8 bytes)
LIDT *mem*	Load interrupt-descriptor table (8 bytes)
LLDT *mem*	Load local-descriptor table
LMSW *mem*	Load machine-status word
LTR *mem*	Load task register
MOV *creg,creg*	Move to or from *creg*
MOV *dreg,dreg*	Move to or from *dreg*
MOV *treg,treg*	Move to or from *treg*
MOV *creg,reg*	Move to or from *creg*
MOV *dreg,reg*	Move to or from *dreg*
MOV *treg,reg*	Move to or from *treg*

80387 Instruction Mnemonics

The 80387 instruction set consists of all 80287 instructions plus the following instructions. The **.387** directive must be used to enable these instructions.

Table B.10

80387 Instruction Mnemonics

Syntax	Action
FCOS	Cosine
FPRIM1	Partial remainder (IEEE compatible)
FSIN	Sine
FSINCOS	Sine and cosine
FUCOM	Unordered compare
FUCOMP	Unordered compare and pop stack
FUCOMPP	Unordered compare and pop stack twice

Appendix C

Directive Summary

Introduction

Directives give the assembler directions and information about input and output, memory organization, conditional assembly, listing and cross-reference control, and definitions. Table C.1 shows the directives.

C

Table C.1

Directives

.186	ASSUME	ENDS	IFNB	PUBLIC
.286	COMMENT	EQU	IFNDEF	.RADIX
.286C	.CREF	EVEN	INCLUDE	RECORD
.286P	DB	EXTRN	LABEL	.SALL
.287	DD	GROUP	.LALL	SEGMENT
.386	DF	IF	.LFCOND	.SFCOND
.386C	DQ	IF1	.LIST	STRUC
.386P	DT	IF2	NAME	SUBTTL
.387	DW	IFB	ORG	.TFCOND
.8086	ELSE	IFDEF	%OUT	TITLE
.8087	END	IFDIF	PAGE	.XALL
=	ENDIF	IFE	.PRIV	.XCREF
ALIGN	ENDP	IFIDN	PROC	.XLIST

Any combination of upper- and lowercase letters can be used when giving directive names in a source file.

The following is a complete list of directive syntax and function:

Table C.2

Directive Syntax and Function

Directive	Action
.186	Enables assembly of 80186 instruction set.
.286	Enables assembly of 80286 nonprotected instruction set.
.286C	Enables assembly of 80286 nonprotected instruction set.

(*Continued on next page.*)

Directive Syntax and Function *(Continued)*

Directive	Action

.286P	Enables assembly of 80286 protected instruction set and is equivalent to the following sequence:
	`.286`
	`.PRIV`

.287	Enables assembly of 80287 instruction set.
.386	Enables assembly of 80386 nonprotected instruction set and sets the default segment wordsize to 4 bytes.
.386C	Enables assembly of 80386 nonprotected instruction set and sets the default segment wordsize to 4 bytes.

.386P	Enables assembly of 80386 protected instruction set and is equivalent to the following sequence:
	`.386`
	`.PRIV`

.387	Enables assembly of 80387 instruction set.
.8086	Enables assembly of 8086 instruction set.
.8087	Enables assembly of 8087 instruction set.
name = *expression*	Assigns the numeric value of *expression* to *name*.
ALIGN *size*	Aligns the segment word size to *size* bytes. The *size* argument must be a power of 2.
ASSUME *segmentregister* : *segmentname*,,,	Selects the given *segmentregister* to be the default segment register for all symbols in the named segment or group. If *segmentname* is **NOTHING**, no register is selected.

(Continued on next page.)

Directive Syntax and Function (*Continued*)

Directive	Action
COMMENT *delimiter text delimiter*	Treats all *text* between the given pair of *delimiter* delimiters as a comment.
.CREF	Restores listing of symbols in the cross-reference listing file.
[*name*] DB *initialvalue*,,,	Allocates and initializes a byte (8 bits) of storage for each *initialvalue*.
[*name*] DD *initialvalue*,,,	Allocates and initializes a doubleword (4 bytes) of storage for each given *initialvalue*.
[*name*] DF *initialvalue*,,,	Allocates and initializes 6 bytes of storage for each given *initialvalue*.
[*name*] DQ *initialvalue*,,,	Allocates and initializes a quadword (8 bytes) of storage for each given *initialvalue*.
[*name*] DT *initialvalue*,,,	Allocates and initializes 10 bytes of storage for each given *initialvalue*.
[*name*] DW *initialvalue*,,,	Allocates and initializes a word (2 bytes) of storage for each given *initialvalue*.
ELSE	Marks the beginning of an alternate block within a conditional block.
END [*expression*]	Marks the end of the module and optionally sets the program entry point to *expression*.
ENDIF	Terminates a conditional block.
name ENDP	Marks the end of a procedure definition.
name ENDS	Marks the end of a segment or structure type definition.
name EQU *expression*	Assigns the *expression* to the given *name*.

(*Continued on next page.*)

Directive Syntax and Function (*Continued*)

Directive	Action
EVEN	If necessary, increments the location counter to an even value and generates one **NOP** instruction (90h).
EXTRN *name* : *type*,,,	Defines an external variable, label, or symbol named *name* whose type is *type*.
name GROUP *segmentname*,,,	Associates a group *name* with one or more segments.
IF *expression*	Grants assembly if the *expression* is nonzero (true).
IF1	Grants assembly on Pass 1 only.
IF2	Grants assembly on Pass 2 only.
IFB < *argument* >	Grants assembly if the *argument* is blank.
IFDEF *name*	Grants assembly if *name* is a previously defined label, variable, or symbol.
IFDIF < *argument1* >, < *argument2* >	Grants assembly if the arguments are different.
IFE *expression*	Grants assembly if the *expression* is 0 (false).
IFIDN < *argument1* >, < *argument2* >	Grants assembly if the arguments are identical.
IFNB < *argument* >	Grants assembly if the *argument* is not blank.
IFNDEF *name*	Grants assembly if *name* has not yet been defined.
INCLUDE *filename*	Inserts source code from the source file given by *filename* into the current source file during assembly.

(*Continued on next page.*

Directive Syntax and Function (*Continued*)

Directive	Action
name LABEL *type*	Creates a new variable or label by assigning the current location-counter value and the given *type* to *name*.
.LALL	Lists all statements in a macro.
.LFCOND	Restores the listing of conditional blocks.
.LIST	Restores the listing of statements in the program listing.
NAME *modulename*	Sets the name of the current module to *modulename*.
ORG *expression*	Sets the location counter to *expression*.
%OUT *text*	Displays *text* at the user's terminal.
PAGE *length* , *width*	Sets the line length and character width of the program listing.
PAGE +	Increments section page numbering.
PAGE	Generates a page break in the listing.
.PRIV	Enables the protected-mode instruction set. Use with either the **.286** or **.386** directive.
name PROC *type*	Marks the beginning of a procedure definition.
PUBLIC *name*,,,	Makes the variable, label, or absolute symbol given by *name* available to all other modules in the program.
.RADIX *expression*	Sets to *expression* the input radix for numbers in the source file.
recordname RECORD *fieldname* : *width* [= *exp*],,,	Defines a record type for an 8- or 16-bit record that contains one or more fields.

(*Continued on next page.*)

Directive Syntax and Function *(Continued)*

Directive	Action
.SALL	Suppresses listing of all macro expansions.
name SEGMENT *align combine class*	Marks the beginning of a program segment *name* having segment attributes *align*, *combine*, and *class*.
.SFCOND	Suppresses listing of any subsequent conditional blocks whose **IF** condition is false.
name STRUC	Marks the beginning of a type definition for a structure.
SUBTTL *text*	Defines the listing subtitle.
.TFCOND	Sets the default mode for listing of conditional blocks.
TITLE *text*	Defines the program-listing title.
.XALL	Lists only those macro statements that generate code or data.
.XCREF *name*,,,	Suppresses the listing of symbols in the cross-reference-listing file.
.XLIST	Suppresses listing of subsequent source lines to the program listing.

Appendix D

Segment Names for High-Level Languages

Introduction

This appendix describes the naming conventions used to form assembly-language source files that are compatible with object modules produced by recent Microsoft language compilers. Compilers that use these conventions include the following:

- Microsoft C Version 3.0 or later
- Microsoft Pascal Version 3.3 or later
- Microsoft FORTRAN Version 3.3 or later

High-level-language modules have the following four predefined segment types:

Type	Contents
_TEXT	Program code
_DATA	Program data
_BSS	Uninitialized space (blank static storage)
_CONST	Constant data

Any assembly-language source file to be assembled and linked to a high-level-language module must use these segments. Segments are covered in Chapter 4, "Defining Segment Structure."

High-level-language modules must be one of three different memory-model types when integrated with 8086 or 80286 code:

Type	Contents
Small	Single code and data segments
Medium	Multiple code segments with a single data segment
Large	Multiple code and data segments

High-level-language modules must be one of two different memory-model types when integrated with 80386 code:

Type	Contents
Pure-Text Small	Text and data in separate segments
Mixed	Code located in one segment and procedures or data located in another segment

For more information on memory models, see the sections, ''Understanding Memory Models'' and ''Defining the Memory Model,'' in Chapter 4.

D

Text Segments

Syntax

*name*_TEXT SEGMENT BYTE PUBLIC 'CODE'
statements
*name*_TEXT ENDS

A text segment defines a module's program code. It contains *statements* that define instructions and data within the segment. A text segment must have the name *name*_TEXT, where *name* can be any valid name.

A segment can contain any combination of instructions and data statements. These statements must appear in an order that creates a valid program. All instructions and data addresses in a text segment are relative to the **CS** segment register. Therefore, the following statement must appear at the beginning of the segment:

```
ASSUME CS: name_TEXT
```

This statement ensures that each label and variable declared in the segment will be associated with the **CS** segment register (this is covered in the section, "Associating Segments with Registers", in Chapter 4).

Text segments must have **BYTE** alignment and **PUBLIC** combination type, and must have the class name **CODE**. These directives define loading instructions that are passed to the linker. Although other segment attributes are available, they should not be used. (For a complete description of the attributes, see Chapter 4, "Defining Segment Structure.")

For small-model programs, only one text segment is allowed. The segment must not exceed 64K in 8086 or 80286 code, or 4 gigabytes in 80386 code. All procedure and statement labels must have **NEAR** type.

Example

```
_TEXT segment     BYTE PUBLIC 'CODE'
      assume cs:_TEXT
_main proc near
      .
      .
      .
_main endp
_TEXT ends
```

Near Data Segments

Syntax

DGROUP *group*_**DATA**
ASSUME ds:DGROUP
_DATA SEGMENT WORD PUBLIC 'DATA'
statements
_DATA ENDS

A "near" data segment contains initialized data that is in the segment pointed to by the **DS** segment register when the program starts execution. The segment is "near" because all data in the segment is accessible without giving an explicit segment value. All programs have exactly one near data segment.

D

A near data segment's name must be **_DATA**. The segment can contain any combination of data *statements* defining variables to be used by the program. The segment must not exceed 64K in 8086 or 80286 code or 4 gigabytes in 80386 code. All data addresses in the segment are relative to the predefined group **DGROUP**. Therefore, the following statements must appear at the beginning of the segment:

```
DGROUP      group_DATA
            ASSUME ds: DGROUP
```

These statements ensure that each variable declared in the data segment will be associated with the **DS** segment register and **DGROUP**. For more information, see the section, "Associating Segments with Registers," in Chapter 4.

Near data segments must be **WORD** aligned in 8086 or 80286 code, and **DWORD** aligned in 80386 code. They must also have **PUBLIC** combination type, and they must have the class name **DATA**. These directives define loading instructions that are passed to the linker. Although other segment attributes are available, they must not be used. For a complete description of the attributes, see Chapter 4, "Defining Segment Structure."

Example

```
DGROUP      group _DATA
      assume ds:DGROUP

_DATA segment     word public 'DATA'
count dw    0
array dw    10 dup(1)
string      db    "Type CANCEL then press RETURN", 0ah, 0
_DATA ends
```

Far Data Segments

Syntax

*name*_**DATA SEGMENT WORD PUBLIC 'FAR_DATA'**
statements
*name*_**DATA ENDS**

A ''far'' data segment contains data that is not pointed to by the **DS** segment register when the program starts execution. To access data in a far data segment, an explicit segment value must be given.

A far data segment's name must be *name*_**DATA**, where *name* can be any valid name. The name of the first variable declared in the segment is recommended. The segment can contain any combination of data *statements* defining variables to be used by the program. The segment must not exceed 64K in 8086 or 80286 code or 4 gigabytes in 80386 code. All data addresses in the segment are relative to the **ES** segment register. When accessing a variable in a far data segment, the **ES** register must be set to the appropriate segment value. Also, the segment-override operator (**:**) must be used with the variable's name. For further information, see the sections, ''Segment-Override Operator'', in Chapter 8, and ''Using Memory Operands,'' in Chapter 14.

Far data segments must be **WORD** aligned, must have **PUBLIC** combination type, and must have the class name **FAR_DATA.** These directives define loading instructions that are passed to the linker. For a complete description of the attributes, see Chapter 4, ''Defining Segment Structure.''

Example

```
array_DATA   segment     word public 'far_DATA'
array        dw    0
             dw    1
             dw    2
             dw    4
table        dw    1600 dup(?)
array_DATA   ends
```

BSS Segments

Syntax

DGROUP *group*_**BSS**
ASSUME ds:DGROUP
_BSS SEGMENT WORD PUBLIC 'BSS'
statements
_BSS ENDS

A **BSS** segment defines uninitialized data space. A **BSS** segment's name must be **_BSS**. The segment can contain any combination of data *statements* defining variables to be used by the program. The segment must not exceed 64K in 8086 or 80286 code or 4 gigabytes in 80386 code. All data addresses in the segment are relative to the predefined group **DGROUP**. Therefore, the following statements must appear at the beginning of the segment:

```
DGROUP     group_BSS
        ASSUME ds:DGROUP
```

These statements ensure that each variable declared in the **BSS** segment will be associated with the **DS** segment register and **DGROUP**. For more information, see the section, "Associating Segments with Registers," in Chapter 4.

The group name **DGROUP** must not be defined in more than one **GROUP** directive in a source file. If a source file contains both a **DATA** and a **BSS** segment, the **DGROUP** directive should be used:

```
DGROUP  group  _DATA, _BSS
```

A **BSS** segment must be **WORD** aligned, must have **PUBLIC** combination type, and must have the class name **BSS**. These directives define loading instructions that are passed to the linker. Although other segment attributes are available, they must not be used.

BSS Segments

Example

```
DGROUP      group _BSS
ASSUME      ds:DGROUP

_BSS  segment     word public 'BSS'
count dw    ?
array dw    10 dup(?)
string      db    30 dup(?)
_BSS  ends
```

D

Constant Segments

Syntax

DGROUP *group*_**CONST**
ASSUME ds:DGROUP
CONST SEGMENT WORD PUBLIC 'CONST'
statements
CONST ENDS

A constant segment defines constant data that will not change during program execution.

The constant segment's name must be **CONST**. The segment can contain any combination of data *statements* defining constants to be used by the program. The segment must not exceed 64K in 8086 or 80286 code or 4 gigabytes in 80386 code. All data addresses in the segment are relative to the predefined group **DGROUP**. Therefore, the following statements must appear at the beginning of the segment:

```
DGROUP        group_CONST
              ASSUME ds:DGROUP
```

These statements ensure that each variable declared in the constant segment will be associated with the **DS** segment register and **DGROUP**. For more information, see the section, "Associating Segments with Registers," in Chapter 4. The group name **DGROUP** must not be defined in more than one **GROUP** directive in a source file. If a source file contains a **DATA**, **BSS**, or **CONST** segment, the **DGROUP** directive should be used:

```
DGROUP        group_DATA, _BSS, CONST
```

A constant segment must be **WORD** aligned, must have **PUBLIC** combination type, and must have the class name **CONST**. These directives define loading instructions that are passed to the linker. Although other segment attributes are available, they must not be used.

In the following example, the constant segment receives the segment values of two far data segments: *ARRAY_DATA* and *MESSAGE_DATA*. These data segments must be defined elsewhere in the module.

Constant Segments

Example

```
DGROUP      group CONST
      ASSUME ds:DGROUP

CONST segment     word public 'CONST'
seg1  dw    ARRAY_DATA
seg2  dw    MESSAGE_DATA
CONST ends
```

D

Appendix E

Error Messages and Exit Codes

Introduction

This appendix lists and explains the messages and exit codes that can be generated by **masm**.

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.

Messages and Exit Codes from masm

The assembler can display several kinds of messages as well as output an exit code; the kind of exit code output depends on the error, if any, encountered during the assembly.

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. The second number indicates far symbol space. This is equal to the size of **masm**, the size of **masm** buffers, and the amount of available memory less near data space. Most symbols go into far space. When far space is exhausted, additional symbols go into near space. Using both far and near space causes a decrease in speed of assembly.

You can use the **-v** option to direct **masm** to display additional 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 standard output after assembly.

Numbered Assembler Messages

The assembler displays messages on the standard error (stderr) whenever it encounters an error while processing a source file. It also displays a warning message whenever it encounters questionable syntax. Messages that can be associated with a particular line of code are numbered. General errors related to the entire assembly rather than to a particular line are unnumbered. (For more information, see the section, ''Unnumbered Error Messages.'')

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* is 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 indicates the warning level. The number is 2 for severe errors, 4 for serious warnings, and 5 for advisory warnings. The next three digits are the error number. For example, severe error 38 is shown as *A2038*.

The *message* is a descriptive line describing the error.

Messages from **masm** 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.

`0 Block nesting error`

Nested procedures, segments, structures, macros, or repeat blocks were 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`

Sufficient information to define a statement has been received on a line, but additional characters were also provided. This may indicate that you provided too many arguments.

`2 Internal error - Register already defined` *symbol*

Note the conditions when the error occurs and contact your software distributor.

`3 Unknown type specifier`

An invalid type specifier was used to give the size of a label or external declaration. For instance, **BYTE** or **NEAR** might have been misspelled.

`4 Redefinition of symbol`

A symbol was defined in two places with different types. This error occurs during Pass 1 on the second declaration of the symbol.

`5 Symbol is multidefined:`

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. The format of Pass 1 listings is discussed in the section, ''Reading a Pass 1 Listing,'' in Chapter 2.

`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 Must be in conditional block`

An **ENDIF** or **ELSE** was 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 is ignored if the references are resolved on the second pass.

`10 Syntax error`

A statement did not match any recognizable assembler syntax. Because **masm** tries to be specific, this error only occurs if the statement bears no resemblance to any legal statement.

`11 Type illegal in context`

The type specifier was given an unacceptable size. For example, a procedure was defined 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 during 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 instance, a text equate cannot be declared public. The section, ''Declaring Symbols Public,'' in Chapter 7, explains public declarations.

`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 Reserved word used as symbol:` *name*

An assembler keyword was used as a symbol. This is a warning, not an error, and can be ignored if you wish. However, the keyword is no longer available for its original purpose. For example, if you name a macro *add*, it replaces the **ADD** instruction.

17 `Forward reference illegal`

A symbol was referenced before it was defined on 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 `Operand must be register:` *operand*

A register was expected as an operand, but a symbol or constant was supplied.

20 `Operand must be segment or group`

A segment or group name was expected, but some other kind of operand was given. For instance, 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 `Operand must be type specifier`

An operand was expected to be a type specifier, such as **NEAR** or **FAR**, but some other kind of operand was received.

23 `Symbol 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 the section, "Full Segment Definitions," in Chapter 4, for information on defining segments.

25 `Improper 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 the section, "Full Segment Definitions," in Chapter 4.

26 `Reference to multidefined symbol`

An instruction referenced a symbol defined in more than one place.

27 `Operand expected`

An operand was expected, but an operator was received.

28 `Operator expected`

An operator was expected, but an operand was received.

29 `Division by 0 or overflow`

An expression resulted in division by 0 or in a number too large to be represented.

30 `Negative shift count`

An 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, this warning is generated by the following code:

```
string      DB      "This is a test"
            .
            .
            .
            mov     ax,string[4]
```

Since this is a warning rather than an error, **masm** attempts to generate code based on its best guess of the intended result. If one of the operands is a register, the register size overrides the size of the other operand. In the example, the word size of **AX** overrides 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 the section, "Declaring Symbols External," in Chapter 7, for information about correct declaration and use of external symbols.

34 `Operand must be record or field name`
An operand was expected to be a record name or record-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]
```

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 instance, two memory operands cannot be added in an expression. Valid uses of the addition operator are explained in the section, "Arithmetic Operators," in Chapter 8.

40 `Operands must be in same segment, or one must be cc`
The subtraction operator was used incorrectly. For instance, a memory operand in the code segment cannot be subtracted from a memory operand in the data segment. Valid uses of the subtraction operator are explained in Chapter 8.

42 `Constant 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 instance, a constant operand cannot have a segment. See the section, "SEG Operator," in Chapter 8, 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 `Multiple base registers`
More than one base register was used in an operand. For example, the following line is illegal:

```
mov     ax,[bx+bp]
```

47 `Multiple index registers`
More than one index register was used in an operand. For example, the following line is illegal:

```
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, the following line is illegal:

```
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 out of range`
A value was too large for its context. 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 was used to represent a code address outside the code segment assigned with the **ASSUME** statement. This usually indicates a call or jump to a label outside the current code segment.

`52 Improper operand type:` *symbol*

An illegal operand was given for a particular context. For example

```
mov     mem1,mem2
```

is illegal if both operands are memory operands.

`53 Jump out of range by` *number* `bytes`

A conditional jump was not within the required range. For all except the 80386 processor, the range is 128 bytes backward or 127 bytes forward from the start of the instruction following the jump instruction. For the 80386, the default range is from -32,768 to 32,767. 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 the section, ''Forward References to Labels,'' in Chapter 8.

`55 Illegal register value`

A register was specified with an illegal syntax. For example, you cannot access a stack variable with the following:

```
mov     ax,bp+4
```

The correct syntax (as explained in the section, ''Passing Arguments on the Stack'', in Chapter 16) is shown below:

```
mov     ax,[bp+4]
```

`56 Immediate mode illegal`

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 operand`

The size of an operand is illegal with the specified instruction. For instance, 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 Macro Assembler. Usually you can solve the problem by specifying the size of the item with the **PTR** operator, as explained in the section, ''Strong Typing for Memory Operands,'' in Chapter 8.

58 `Byte register illegal`

A byte register was used in a context where a word register (or 32-bit register on the 80386) is required. For example, *push al* is illegal; use *push ax* instead.

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 accumulator register`

A register other than **AL**, **AX**, or **EAX** was supplied in a context where only the accumulator register is acceptable. For instance, 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 the section, "Associating Segments with Registers," in Chapter 4, for information on the **ASSUME** directive.

65 `Label cannot have segment override`

A segment override was used incorrectly. See the section, "Segment-Override Operator," in Chapter 8, for examples of valid uses of the segment override operator.

66 `Must have instruction after prefix`

A repeat prefix such as **REP**, **REPE**, or **REPNE** was given without specifying the instruction to repeat.

67 `Cannot override ES for destination`

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 the section, "Associating Segments with Registers," in Chapter 4, for information on the **ASSUME** directive.

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 with byte alignment`
The **EVEN** or **ALIGN** directive was used in a segment that is byte aligned. The section, "Aligning Data," in Chapter 5, explains the **EVEN** and **ALIGN** directives.

`71 Forward reference needs override or FAR`
A call or jump attempts to access a far label that was not declared far earlier in the source code. You can use the **PTR** operator to specify far calls and jumps, as shown below:

```
call    FAR PTR task
jmp     FAR PTR location
```

`72 Illegal value for DUP count`
The count value specified for a **DUP** operator did not evaluate to a constant integer greater than 0.

`73 Symbol is already external`
A symbol that had already been declared external was later defined locally. The section, "Declaring Symbols External," in Chapter 7, describes external declarations.

`74 DUP nesting too deep`
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 the section, "Arrays and Buffers," in Chapter 5.

`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 Angle 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'0
```

The following definitions are correct:

```
stest  test  <4,,'c',0> ; Three initial values, one blank
ttest  test  <>         ; No initial values
```

`78 Directive illegal in structure`

A statement within a structure definition was not one of the following: a data definition using define directives such as **DB** or **DW**, a comment preceded by a semicolon, or a conditional-assembly directive.

`79 Override with DUP 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 instance, if a field is initialized in the structure definition with the **DUP** operator, it cannot be overridden in a declaration. See the note in the section, ''Defining Structure Variables,'' in Chapter 6.

`83 Circular chain of EQU aliases`

An alias declared with the **EQU** directive 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 produced an opcode that the coprocessor emulator does not support. Since the emulator library is not supplied with the Macro Assembler, this error can only occur if you are linking assembler routines with code from a high-level-language compiler that uses 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 or data was used outside all segment blocks. Instructions and data declarations must be in segments, but directives that specify assembler behavior without generating 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 true (0)`

An error was forced with the **.ERRE** directive.

`91 Forced error - expression false (not 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 Wrong length for override value`

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 two bytes; the structure declaration provided only room for one byte.

`99 Line too long expanding symbol:` *symbol*

An equate defined with the **EQU** directive was so long that expanding it caused the assembler's internal buffers to overflow. This message may indicate a recursive text macro.

`100 Impure memory reference`

Data was stored into the code segment when the **-p** option and privileged instructions (enabled with **.286P** or **.386P**) were in effect. An example of storing data in 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 real mode, but can cause problems in privileged mode.

`101 Missing data; zero assumed`

An operand is missing from a statement, as shown below:

```
mov     ax,
```

Since some programmers use this syntax purposely, the message is a warning. It is assumed that 0 was intended and **masm** assembles the following code:

```
mov     ax,0
```

`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 in privileged mode. 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 `Align must be power of 2`

A number that is not a power of two was used with the **ALIGN** directive. The directive is explained in the section, ‘‘Aligning Data,’’ in Chapter 5.

104 `Jump within short distance`

A **JMP** instruction was used to jump to a short label (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** uses a short form of the **JMP** instruction (2 bytes) rather than the long form (3 bytes with 16-bit segments or 5 bytes with 32-bit segments). You can make your code slightly more efficient by using the **SHORT** operator to specify that a jump is short rather than near. For example, using the **SHORT** operator in the following example saves 1 byte of code:

```
            jmp     SHORT there
            .
            .
there:      .                       ; Less than 127 bytes
```

Using the **SHORT** operator with forward references to code labels is explained in the section, ‘‘SHORT Operator’’, in Chapter 8. With the 80386 processor, this message also applies to conditional jumps, which can be either short (2 bytes) or near (4 bytes).

105 `Expected` *element*

An element such as a punctuation mark or operator was omitted. For instance, if you omit the comma between source and destination operands, the message *Expected comma* is generated.

106 `Line too long`

A source line was longer than 128 characters, the maximum allowed by **masm**.

107 `Illegal digit in number`

A constant number contained a digit that is not allowed in the current radix.

`108 Empty string not allowed`

A statement used an empty string. For example, the following definition is illegal:

```
null        DB      ""
```

In many languages an empty string represents ASCII character 0. In assembly language, you must give the value 0, as shown below:

```
null        DB      0
```

`109 Missing operand`

The instruction or directive requires more operands than were provided.

`110 Open parenthesis or bracket`

Only one parenthesis or bracket was given in a statement that requires opening and closing parentheses or brackets.

`111 Directive must be in macro`

A directive that is expected only in macro definitions was used outside a macro.

`112 Unexpected end of line`

A line ended before a complete statement was formed. More information is expected, but **masm** cannot identify what information is missing.

`113 Cannot change processor in segment`

A processor directive was encountered within a segment. Processor directives must be given before the first segment directive or between segments. If you want to change the processor in the middle of the segment, you must close the current segment, give the processor directive, and then start another segment.

`114 Operand size does not match segment word 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 a questionable practice in a 32-bit segment:

```
mov   ax,OFFSET nearlabel ; Load near (32-bit) label
```

The following statement is a 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.

`115 Address size does not match segment word 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 a questionable practice in a 32-bit segment:

```
mov  eax,[si]   ; Load value pointed to by 16-bit pointer
```

The following statement is a 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.

E

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.

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.

`End of file encountered on input file`

`Include file` *filename* `not found`

`Read error on standard input`

`Unable to access input file:` *filename*

`Unable to open cref file:` *filename*

`Unable to open input file:` *filename*

`Unable to open listing file:` *filename*

`Unable to open object file:` *filename*

`Write error on cross-reference file`

`Write error on listing file`

`Write error on object file`

Command-Line Errors

Any of the following errors may occur if you give an invalid command line when starting **masm**.

`Buffer size expected after B option`

`Error defining symbol "`*name*`"from command line`

`Extra file name ignored`

`Line invalid, start again`

`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.

`Internal error - Problem with expression analyzer`
Note the conditions when the error occurs and contact your software distributor.

`Internal unknown error`
This error may indicate that the internal error table has been corrupted and **masm** cannot figure out what the error is. Note the conditions when the error occurs and contact your software distributor.

The following errors indicate a problem with memory allocation or some other assembler problem not related to a specific source line.

`Number of open conditionals: <number>`
: Conditional-assembly directives (starting with **IF**) were given without corresponding **ENDIF** directives.

`Open procedures`
: A **PROC** directive was given without a corresponding **ENDP** directive.

`Open segments`
: A segment was defined, but never terminated with an **ENDS** directive. This error does not occur with simplified segment directives.

`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.

You can solve this problem in several ways. First, try assembling with no listing file. If this works, you can reassemble by specifying a null object file to get a listing file. You can also rewrite the source file to require less symbol space. Techniques for reducing symbol space include minimizing use of macros, equates, and structures; 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 last use.

Exit Codes from masm

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.

Index

Special Characters

Numerical Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

U

V

W

X

Z